1 Soluzioni primo compitino AA 2003-04

1.1 Primo esercizio

Si dimostri la verità o la falsità di ciascuna delle seguenti affermazioni

(a) $\frac{1}{2}n + lg\ n^2 + \sqrt{n}$ *è nella classe* $\Theta(n)$

VERO. Verifichiamo che esistono tre costanti c_1, c_2, n_0 positive tali che per ogni $n \geq n_0$:

$$c_1 n \leq \frac{1}{2}n + \lg\ n^2 + \sqrt{n} \leq c_2 n$$

ovvero, per $n > 0$,

$$c_1 \leq \frac{1}{2} + \frac{2\lg\ n}{n} + \frac{\sqrt{n}}{n} \leq c_2.$$

Poiché, per $n \geq 2$, $\frac{2\lg\ n}{n} + \frac{\sqrt{n}}{n} \leq 2$ le due diseguaglianze sono verificate scegliendo $c_1 = \frac{1}{2}$, $c_2 = 3$ ed $n_0 = 2$.

(b) $n^2 + nlg\ n$ *è nella classe* $\Theta(n)$

FALSO. Dimostriamo che non possono esistere tre costanti c_1, c_2, n_0 positive tali che per ogni $n \geq n_0$:

$$c_1 n \leq n^2 + n\lg\ n \leq c_2 n.$$

In particolare dimostriamo che non esistono c_2 ed n_0 con le proprietà richieste. Infatti, per ogni valore di c_2 si ha $n\lg\ n > c_2 n$ per ogni $n > 2^{c_2}$.

(c) *La ricorrenza* $T(n) = 3T(\frac{n}{4}) + nlg\ n$ *individua una funzione nella classe* $\Theta(nlg\ n)$

VERO. Possiamo utilizzare il Master Theorem. Poiché $n\lg\ n = \Omega(n^{\log_4 3+\epsilon})$, dove $\epsilon = 1 - \log_4 3$, ricadiamo nel caso 3. Per poter affermare questo si deve anche verificare che sia soddisfatta la condizione di regolarità:

$$\forall n \geq n_0 \quad 3\frac{n}{4}\ \lg\ (\frac{n}{4}) \leq cn\lg\ n$$

per un qualche $c < 1$ e $n_0 > 0$. Poiché per ogni $n \geq 1$

$$\frac{3}{4}n\ \lg\ (\frac{n}{4}) \leq \frac{3}{4}n\ \lg\ n$$

è sufficiente prendere proprio $c = \frac{3}{4}$ ed $n_0 = 1$. Allora per il terzo caso del Master Theorem $T(n) = \Theta(n\lg\ n)$.

(d) *La ricorrenza* $T(n) = 3T(\frac{n}{2}) + lg\, n$ *individua una funzione nella classe* $O(n^2)$

VERO. Possiamo utilizzare il Master Theorem. Poiché $\lg n = O(n^{\log_2 3-\epsilon})$, dove $\epsilon = \log_2 3 - 1$, possiamo applicare il primo caso del Master Theorem ed ottenere $T(n) = \Theta(n^{\log_2 3})$ e quindi $T(n) = O(n^2)$.

1.2 Secondo esercizio

Date le seguenti procedure A *e* B*, si determini la complessità asintotica della procedura* A(n) *su input* $n \in N$

A(n)

```
1  s ← 0                          1
2  for i ← 1 to n                 n + 1
3        do s ← s + B(i)          ∑_{i=1}^{n} T_B(i)
4  return s                       1
```

B(m)

```
1  s ← 0                          1
2  for i ← 1 to m                 m + 1
3        do s ← s + i             m
4  return s                       1
```

Allora $T_B(i) = \Theta(i)$ e la complessità di A è data da:

$$\sum_{i=1}^{n} T_B(i) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

1.3 Terzo esercizio

(a) *Scelta una rappresentazione per le liste, si descriva un algoritmo che data una lista* L *ed un intero* k *modifichi* L *eliminando tutti gli elementi con chiave minore di* k*.*

Per la lista singola:

LIST-DELETE-SMALLER(L, k)

```
1  x ← head[L]
2  while x ≠ NIL
3        do y ← next[x]
4           if key[x] < k
5              then LIST-DELETE(L, x)
6           x ← y
```

Per l'implementazione con lista doppia il codice non cambia; per l'implementazione con lista circolare invece è sufficiente cambiare NIL in $nil[L]$.

(b) *Si dimostri la correttezza dell'algoritmo proposto utilizzando un opportuno invariante.*

Invariante: nessun nuovo elemento è stato inserito nella lista L; tutti gli elementi con chiave maggiore o uguale a k originariamente in L solo ancora in L; tutti gli elementi prima di x hanno chiave maggiore o uguale a k;

Inizializzazione: quando $x = head[L]$ la lista non è stata ancora modificata e non ci sono elementi prima di x, per cui l'invariante è soddisfatta;

Mantenimento: all'inizio del ciclo **while** per l'invariante tutti gli elementi prima di x hanno chiave maggiore o uguale a k; ci sono due casi:

- l'elemento in x ha chiave minore di k: questo viene eliminato dalla lista e x passa al prossimo elemento; allora gli elementi prima di x nella lista L sono gli stessi elementi che c'erano all'inizio del ciclo; poiché questi per l'invariante erano originariamente nella lista l'invariante è mantenuta;
- l'elemento in x ha chiave maggiore o uguale a k: x passa al prossimo elemento; gli elementi che precedono x sono quelli che c'erano all'inizio del ciclo più quest'elemento con chiave maggiore o uguale a k; inoltre quest'elemento è un elemento della lista originaria; allora l'invariante è mantenuta;

Terminazione: $x =$ NIL, per cui è posizionato dopo l'ultimo elemento; allora tutti gli elementi prima di x sono la lista intera L che contiene ora solo gli elementi originari maggiori o uguali a k.

(c) *Si discuta la complessità dell'algoritmo proposto e si dica se e come questa varia al variare della rappresentazione scelta.*

La complessità varia nel seguente modo:

- lista semplice: $\Theta(n^2)$ (la LIST-DELETE è $\Theta(n)$)
- lista doppia: $\Theta(n)$ (la LIST-DELETE è $\Theta(1)$)
- lista circolare doppia: $\Theta(n)$ (la LIST-DELETE è $\Theta(1)$)

1.4 Quarto esercizio

Si sviluppi un algoritmo di tipo divide-et-impera per calcolare la somma degli elementi di un array $A[1 \dots n]$ e se ne valuti la complessità.

SUM-R(A, p, r)

```
1  if p > r
2     then return 0
3          q ← ⌊(p + r)/2⌋
4          return SUM-R(A, p, q − 1) + A[q] + SUM-R(A, q + 1, r)
```

La complessità è definita dalla seguente equazione ricorsiva:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ 2T(\frac{n}{2}) + \Theta(1) & \text{altrimenti} \end{cases}$$

La soluzione dell'equazione è $T(n) = \Theta(n)$ come si può dimostrare facilmente con il Master Theorem, o con alberi di ricorsione e metodo di sostituzione.

Usando il Master Theorem è pressoché immediato risolvere l'equazione: $1 = O(n^{1-\epsilon})$ per qualunque $\epsilon \leq 1$ per cui, per il primo caso del Master Theorem: $T(n) = \Theta(n)$.

1.5 Quinto esercizio

Si descriva un algorimo che dato un intero k ed un albero generale T, con attributi $key[x]$, $child[x]$, $sibling[x]$ e $parent[x]$, modifica il campo chiave di tutti i nodi di T ponendo $key[x]$ uguale al numero dei discendenti di x la cui chiave è minore di k.

Si può risolvere con una visita *depth-first-search*, ricorrendo sia sui figli che sui fratelli:

TREE-COUNT-LESSER-R(x, k)

```
1  if x = NIL
2     then return 0
3  if key[x] < k
4     then key[x] ← 1
5     else key[x] ← 0
6  key[x] ← key[x] + TREE-COUNT-LESSER-R(child[x], k)
7  return key[x] + TREE-COUNT-LESSER-R(sibling[x], k)
```

TREE-COUNT-LESSER(T, k)

```
1  TREE-COUNT-LESSER-R(root[T], k)
```

Algoritmi e Strutture Dati

a.a. 2012/13

Prima prova intermedia del 28/01/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

1. Dare la definizione di albero binario **completo**.
   Scrivere in C un programma **efficiente** per stabilire se un albero binario è **completo** e calcolarne la complessità al caso pessimo indicando, e risolvendo, la corrispondente relazione di ricorrenza.

2. Dato l'insieme delle chiavi {1,4,5,10,16,17,21}, quale è l'altezza minima *hmin* di un albero binario di ricerca che contenga esattamente queste chiavi? E l'altezza massima *hmax*?
   Disegnare 3 alberi binari di ricerca con le chiavi dell'insieme specificato rispettivamente di altezza *hmin*, *hmax* e di un'altezza *h* tale che $hmin < h < hmax$.

   Infine scrivere una versione **ricorsiva** della procedura `Tree-Insert` per gli alberi binari di ricerca. La procedura ha come input un albero binario di ricerca *T* e un nodo *z* da inserire in tale albero. Discutere la complessità al caso pessimo di tale procedura.

3. Si enunci e si dimostri il teorema fondamentale delle ricorrenze e lo si utilizzi per risolvere le seguenti ricorrenze (spiegando in quali casi del teorema ricade ciascuna di esse):

   - $T(n) = 4T(n/2) + n$
   - $T(n) = 4T(n/2) + n^2$
   - $T(n) = 4T(n/2) + n^3$

Algoritmi e Strutture Dati

a.a. 2012/13

Seconda prova intermedia del 30/05/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

1. In una tabella Hash di **m = 17** posizioni, inizialmente vuota, devono essere inserite le seguenti chiavi numeriche nell'ordine indicato:

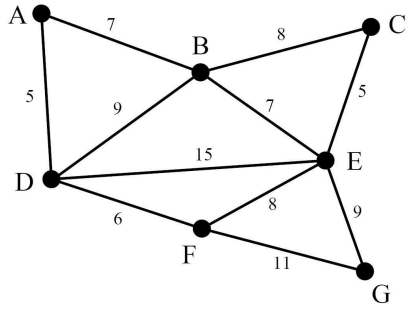
   101, 50, 20, 72, 3, 14, 25, 121, 115, 22, 73

   La tabella è a indirizzamento aperto e la scansione è eseguita per doppio Hashing:

   $$h(k, i) = (k \bmod m + i * 2^{k \bmod 5}) \bmod m$$

   Indicare per ogni chiave le posizioni scandite nella tabella e la posizione finale dove viene allocata.

2. Si scriva l'algoritmo di Kruskal per determinare gli alberi di copertura minimi, si discuta la sua complessità computazionale, se ne dimostri la correttezza e si simuli accuratamente la sua esecuzione sul seguente grafo:



3. L'*arbitraggio* è un'operazione finanziaria per lucrare dalla differenza di prezzi tra le varie piazze e mercati. Sia $V = \{ v_1, v_2, \ldots, v_n \}$ un insieme di n valute e si indichi con C_{ij} il tasso di cambio tra le valute v_i e v_j (cioè, vendendo 1 unità di valuta v_i si ottengono C_{ij} unità di valuta v_j). Un arbitraggio è possibile se esiste una sequenza di azioni elementari di cambio (*transazioni*) che inizi con 1 unità di una certa valuta e termini con più di 1 unità della stessa valuta. Per esempio, se i tassi di cambio sono: 1.53 franchi svizzeri per 1 euro, 0.94 dollari americani per 1 franco svizzero, e 0.77 euro per 1 dollaro americano, possiamo convertire 1 euro in 1.1 euro, realizzando un guadagno del 10%. Si formuli il problema dell'arbitraggio come un problema (noto) di ricerca su grafi e si sviluppi un algoritmo efficiente per la sua risoluzione, discutendone correttezza e complessità.

Algoritmi e Strutture Dati

Sessione estiva A.A. 2003/2004

Appello 09.07.04

1. *Per un certo problema sono stati trovati due possibili algoritmi risolutivi. Il tempo di esecuzione del primo è rappresentato dalla funzione* T_1 *riportata nel seguito al punto* (a) *mentre per il secondo è soddisfatta la relazione di ricorrenza riportate al punto* (b)*. Si dica, giustificando la risposta, quale dei due algoritmi è da preferire nel caso si debbano risolvere problemi di grandi dimensioni.*

(a) $T_1(n) = 2n^2 + nlgn$

(b) $T_2(n) = 4T_2(\frac{n}{2}) + 5n^2 + 2log\ n^2$

Utilizziamo il Master Theorem per valutare la complessità asintotica del secondo algoritmo.

- *a* Si ha: $a = 4, b = 2$, quindi $\log_2 4 = 2$. Poiché $5n^2 + 2log\ n^2 = \Theta(n^2)$ siamo nel caso 2 del Master Method. Allora $T_2(n) = \Theta(n^2 \text{lg } n)$.

Dobbiamo quindi confrontare $2n^2 + n\text{lg}n$ con $n^2\text{lg } n$. Poiché $2n^2 + n\text{lg}n = O(n^2\text{lg } n)$ ma non vale il viceversa (vedi lezione di tutorato del 14.10.04), l'algoritmo da scegliere è il primo.

2. *Considerare le seguenti procedure A e B e determinare la complessità asintotica della procedura B(n) su input* $n \in N$*.*

A(n)

```
1  s ← 0
2  for i ← 1 to n
3        do s ← s + i
4  return s
```

B(n)

```
1  m ← A(n)
2  s ← 1
3  for i ← 1 to m
4        do s ← s * i
5  return s
```

Soluzione: La procedura $A(n)$ è lineare rispetto all'input n e calcola la somma dei primi n numeri interi. La chiamata di A(n) all'interno di B ha quindi costo di ordine $\Theta(n)$ ed assegna ad m un valore in $\Theta(n^2)$ (la somma dei primi n numeri interi). Pertanto il ciclo for della procedura $B(n)$ ha costo $\Theta(m) = \Theta(n^2)$ e la complessità asintotica globale della procedura B(n) è $\Theta(n^2)$.

3. *Sia T un albero binario di ricerca bilanciato con chiavi intere. Descrivere un algoritmo efficiente per verificare se tutti i nodi di T hanno chiave strettamente compresa tra due valori dati $k1$ e $k2$.*
Discutere la complessità dell'algoritmo in funzione del numero n di nodi dell'albero.

Soluzione: E' sufficiente verificare che il minimo elemento di T sia maggiore di $k1$ ed il massimo minore di $k2$. Essendo l'albero bilanciato questo può essere fatto in tempo $O(lgn)$, dove n è il numero di nodi.

4. *Definiamo una operazione* concatenate *il cui input è costituito da due insiemi di chiavi $S1$ ed $S2$ tali che le chiavi in $S1$ sono tutte minori o uguali delle chiavi in $S2$ e il cui output è la fusione dei due insiemi. Supponendo che gli insiemi $S1$ ed $S2$ siano rappresentati con alberi binari di ricerca, progettare un algoritmo per realizzare l'operazione* concatenate. *L'algoritmo deve avere complessità $O(h)$ nel caso peggiore, dove h è l'altezza massima dei due alberi,.*

Soluzione: Poichè non si considerano alberi bilanciati è sufficiente porre $S1$ come figlio sinistro del nodo $BSTmin(S2)$ (o $S2$ come figlio destro di $BSTmax(S1)$). L'esercizio non specifica se gli insiemi sono rappresentati con o senza duplicazioni; nel secondo caso è sufficiente controllare se la chiave del massimo di $S1$ è uguale alla chiave del minimo di $S2$ e in caso affermativo chiamare la *BSTdelete* prima della fusione. Tutte queste operazioni hanno complessità $O(h)$.

5. *Definire la struttura dati* heap *e descriverne almeno una applicazione.*
Soluzione: Vedi testo.

6. *Sia L una lista concatenata il cui campo chiave contiene valori interi. Progettare un algoritmo che modifica L eliminando tutti gli elementi con chiave pari.*
Dimostrare la correttezza dell'algoritmo tramite l'uso di invarianti.

Soluzione: Possiamo scorrere la lista tenendo un puntatore *predy* all'ultimo elemento con chiave dispari considerato; scorrendo la lista aggiorneremo via via il campo next di *predy* fino a che questo non punterà correttamente al prossimo elemento con chiave dispari. All'inizio *predy* viene posto uguale alla costante NIL e il suo aggiornamento inizierà solo dopo che verrà incontato un elemento con chiave dispari; la testa della lista assumerà lo stesso valore di *predy* la prima volta che verrà trovato un elemento con chiave dispari. L'invariante è:
Gli elementi concatenati compresi tra HEAD$[L]$ e *predy* sono tutti e soli gli elementi che nella lista originale sono compresi tra HEAD$[L]$ e y e hanno chiave dispari.

```
ELIMINAPARI(L)
 1  y ← HEAD[L]
 2  predy ← NIL
 3  while y ≠ NIL
 4        do if (mod(KEY[y], 2) ≠ 0)
 5              then if (predy = NIL)
 6                      then HEAD[L] ← y
 7                   predy ← y
 8              else  if (predy ≠ NIL)
 9                      then NEXT[predy] ← NEXT[y]
10           y ← NEXT[y]
```

7. *Definire l'operazione di rotazione a sinistra di un sottoalbero di un albero binario. Mostrare un esempio in cui è richiesta una tale operazione dopo l'inserimento o la cancellazione di una chiave in un albero rosso/nero.*

Soluzione: Si veda il testo. Un semplice esempio si trova nel caso 1 di RB-delete.

Algoritmi e Strutture Dati

Sessione estiva A.A. 2003/2004

Appello 10.06.04

1. *Data la ricorrenza*

$$T(n) = 2 \cdot T(\frac{2n}{3}) + n^2$$

utilizzando il metodo di sostituzione dimostrare che $T(n) = \Omega(n^2)$.

Soluzione: Dobbiamo dimostrare che esistono due costanti positive c e n_0 tale che $T(n) \geq cn^2$, per ogni $n \geq n_0$. Poiché $T(n) = 2 \cdot T(\frac{2n}{3}) + n^2$, assumendo $T(\frac{2n}{3}) \geq c \cdot (\frac{2n}{3})^2$ si ottiene: $T(n) = 2 \cdot T(\frac{2n}{3}) + n^2 \geq 2c \cdot (\frac{2n}{3})^2 + n^2 = (\frac{8}{9} \cdot c + 1)n^2$
La diseguaglianza $(\frac{8}{9} \cdot c + 1)n^2 \geq cn^2$ è soddisfatta per ogni $c \leq 9$ e per ogni $n_0 > 0$. Quindi esistono c ed n_0 tali che $T(n) \geq cn^2$ per ogni $n \geq n_0$.

2. *Date le seguenti procedure* A *e* B, *si determini la complessità asintotica della procedura* A(n) *su input* $n \in N$

A(n)

```
1  s ← 0
2  for i ← 1 to n
3        do s ← s + B(n)
4  return s
```

B(m)

```
1  if m = 1
2     then return 0
3     else  return B(m/2) + m
```

Soluzione: La complessità di B può essere espressa tramite la ricorrenza $T_B(n) = T_B(n/2) + \Theta(1)$ che si risolve facilmente con il master method ottenendo $T_B(n) = \Theta(logn)$. Per la complessità di A abbiamo

A(n)

1	$s \leftarrow 0$	1
2	**for** $i \leftarrow 1$ **to** n	$n + 1$
3	**do** $s \leftarrow s + \mathrm{B}(n)$	$\sum_{i=1}^{n} T_B(n) = nT_B(n)$
4	**return** s	1

e quindi:

$$T_A(n) = \sum_{i=1}^{n} T_B(n) = \sum_{i=1}^{n} \Theta(logn) = \Theta(nlogn)$$

3. *Si consideri la struttura dati albero (posizionale e generale) e si assuma che ad ogni nodo x, oltre agli usuali attributi $key[x]$, $child[x]$, $sibling[x]$, sia associato un attributo $color[x]$ che può assumere i valori* bianco *o* nero. *Si descriva un algoritmo che dato un albero T calcola il numero dei nodi di T che hanno tutti i figli bianchi.*

Soluzione: Si può risolvere con una visita in ampiezza *breadth-first-search* verificando la condizione richiesta durante il ciclo while di visita di ciascun gruppo di fratelli. La chiamata esterna è COUNT($root[T]$).

```
COUNT(x)
 1  tot ← 0
 2  if x = NIL
 3     then return tot
 4  ENQUEUE(x,Q)
 5  while not QUEUE-EMPTY[Q]
 6        do y ← HEAD[Q]
 7           DEQUEUE[Q]
 8           equal ← TRUE
 9           while y ≠ NIL
10                 do if COLOR[Y] = NERO
11                       then equal ← FALSE
12                    if CHILD[y] ≠ NIL
13                       then ENQUEUE[CHILD[y],Q]
14                    y ← SIBLING[y]
15           if equal = TRUE
16              then tot ← tot + 1
17  return tot
```

4. *Considerare le seguenti procedure e per ciascuna dire se può essere utilizzata per verificare una struttura dati nota. (Giustificare bene la risposta)*

VERIFICA-1(A, k)

```
1  test ← TRUE
2  for i ← k downto 2
3        do if (A[i] < A[i/2])
4              then test ← FALSE
5  return test
```

VERIFICA-2(T)

```
1  if T = NIL
2     then return TRUE
3     else  test ← VERIFICA-2(left[T])
4           test ← test  &  VERIFICA-2(right[T])
5           test ← test  &  KEY[left[T]] ≤ KEY[T]
6           test ← test  &  KEY[right[T]] ≥ KEY[T]
7  return test
```

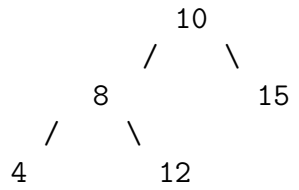
Soluzione: La procedura Verifica-1 può essere utilizzata per verificare la proprietà di min-heap.
Per quanto riguarda la procedura Verifica-2, anche sostituendo le istruzioni

```
1  test ← test  &  KEY[left[T]] ≤ KEY[T]
2  test ← test  &  KEY[right[T]] ≥ KEY[T]
```

con

```
1  if (left[T] ≠ NIL)
2     then test ← test  &  KEY[left[T]] ≤ KEY[T]
3  if (right[T] ≠ NIL)
4     then test ← test  &  KEY[right[T]] ≥ KEY[T]
```

la procedura NON può essere utilizzata per verificare la proprietà BST perchè non confronta la chiave di un nodo con TUTTE le chiavi nei sottoalberi sinistro e destro. Si consideri ad esempio il seguente albero (che NON è un BST dato che la chiave 12 è alla sinistra della chiave 10):



5. *(a) Scrivere un algoritmo che trovi il massimo elemento di un array utilizzando un approccio divide-et-impera.*
(b) Determinare la complessità dell' algoritmo sviluppato.
(c) Dimostrare la correttezza dell' algoritmo sviluppato.

Soluzione: Vedi lezione di tutorato del 7 Novembre 2003

6. *Disegnare, se posibile, un albero Rosso/Nero con altezza uguale a 3 ed altezza nera uguale a 2.*

Soluzione: Ci sono molti alberi Rosso/Neri che possono essere presentati, un esempio può essere l'albero completo di altezza 3 che ha tutti i nodi neri a parte quelli sul primo livello, figli della radice.

7. *Dire in cosa consiste il problema delle collisioni in una tabella hash e spiegare le tecniche usate per affrontarlo.*

Soluzione: Vedi testo.

Algoritmi e Strutture Dati

Sessione invernale A.A. 2003/2004

Appello 20.02.04

Prima parte

1. *Data la ricorrenza*

$$T(n) = 2 \cdot T(\frac{2n}{3}) + n^2$$

(a) *Utilizzando il metodo di sostituzione dimostrare che* $T(n) = O(n^2)$.

Dobbiamo dimostrare che esistono due costanti c e n_0 tale che $T(n) \leq cn^2$, per ogni $n \geq n_0$. Procediamo per induzione e supponiamo $T(\frac{2n}{3}) \leq c \cdot (\frac{2n}{3})^2$. Otteniamo: $T(n) = 2 \cdot T(\frac{2n}{3}) + n^2 \leq 2c \cdot (\frac{2n}{3})^2 + n^2 = (\frac{8}{9} \cdot c + 1)n^2$
È ora facile vedere che $(\frac{8}{9} \cdot c + 1) \leq c$ per un qualsiasi costante $c \geq 9$. Quindi $T(n) \leq cn^2$, per ogni $c \geq 9$ e per ogni $n_0 > 0$.

(b) *Utilizzando poi il Teorema Principale, dare i limiti inferiore e superiore stretti per* $T(n)$. (Nota: $\log_{\frac{3}{2}} 2 \approx 1,710$ e $\log_2 \frac{3}{2} \approx 0,585$).

Abbiamo: $a = 2$, $b = \frac{3}{2}$ e $1 < \log_{\frac{3}{2}} 2 < 2$. Poiché $f(n) = n^2 = \Omega(n^{\log_{\frac{3}{2}} 2+\epsilon})$ se vale la condizione di regolarità ($\exists c < 1$ tale che $af(\frac{n}{b}) < cf(n)$) siamo nel caso 3 del Master Method. La condizione vale in quanto scelto $\frac{8}{9} < c < 1$ è facile vedere che $2 \cdot (\frac{2n}{3})^2 = \frac{8}{9} \cdot n^2 < c \cdot n^2$. Allora $T(n) = \Theta(f(n)) = \Theta(n^2)$.

2. *Nell'ipotesi che* $\textsc{Proc}(m) = \Theta(\sqrt{m})$*, determinare la complessità asintotica (caso peggiore) della seguente procedura* $\textsc{Fun}(A, n)$ *al crescere di* $n \in N$.

```
FUN(A, n)
1  if n < 1 return 1
2  t ← FUN(A, n/2)                          T_FUN(n/2)
3  if t > n^2
4     then t ← t − 1/2 · FUN(A, n/2)        T_FUN(n/2)
5  for j ← 1 to n
6        do t ← t + A[j] + PROC(n)          n√n
7  return t
```

Nel caso peggiore vengono effettuate entrambe le chiamate ricorsive e quindi $T_{\textsc{Fun}}(n)$ soddisfa la ricorrenza $T_{\textsc{Fun}}(n) = 2 \cdot T_{\textsc{Fun}}(n/2) + \Theta(n\sqrt{n})$ e la complessità asintotica della procedura FUN si trova risolvendo tale ricorrenza. Questo si ottiene facilmente utilizzando il Master Method. Si ha $a = 2, b = 2$, $\log_2 2 = 1$, $n\sqrt{n} = \Omega(n^{1+\epsilon})$, e $n\sqrt{n}$ soddisfa la proprietà di regolarità. Quindi siamo nel caso 3 ed abbiamo $T_{\textsc{Fun}}(n) = \Theta(n\sqrt{n}) = \Theta(n^{\frac{3}{2}})$.

3. *Si scriva lo pseudocodice di un algoritmo che dato un albero binario* T *calcola il numero dei nodi di* T *che hanno due figli con lo stesso numero di discendenti.*

Si risolve facilmente con una visita in post-ordine di T che calcola il numero dei discendenti di ogni nodo (includendo il nodo stesso) ed incrementa una variabile globale

Equals inizializzata a 0. Il test ($nl \neq 0$) serve ad escludere i nodi che non hanno due figli (perchè entrambi uguali a NIL).

```
DISCENDENTI(x)
1  if x = NIL
2      then return 0
3  nl ← DISCENDENTI(LEFT[x])
4  nr ← DISCENDENTI(RIGHT[x])
5  if (nl = nr)  &  (nl ≠ 0)
6      then Equals ← Equals + 1
7  return nl + nr + 1
```

Seconda parte

4. *Si supponga di voler modificare gli elementi di un array* A *incrementando di una costante* k1 *tutti gli elementi minori di un'altra costante* k2. *Supposto che* A *soddisfi le proprietà di max-heap, scrivere lo pseudocodice di un algoritmo efficiente che trasforma* A *come richiesto producendo ancora un max-heap.*

Una semplice soluzione si ottiene modificando la procedura BUILD-MAX-HEAP.

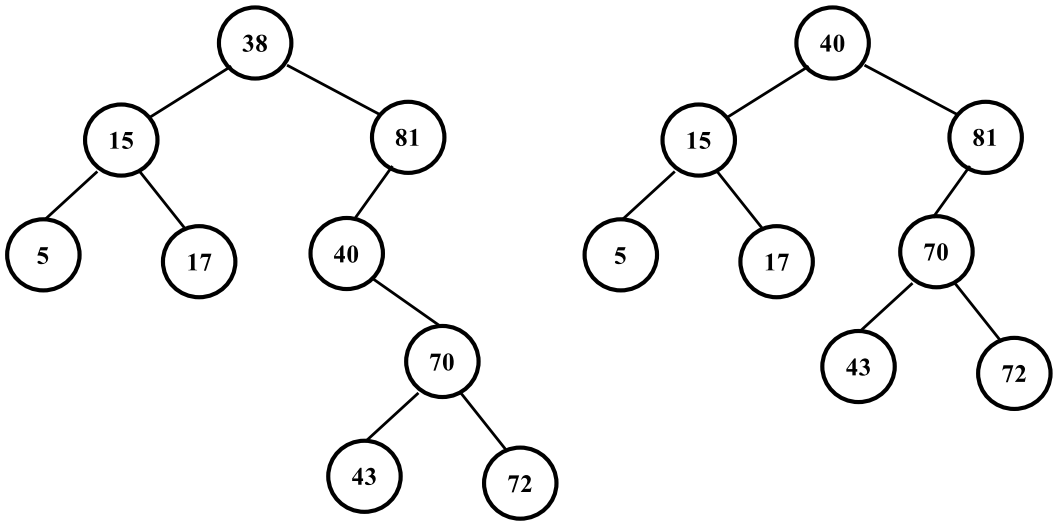
```
INCREASE-HEAP(A, i, n, k1, k2)
1  for i ← n downto 1
2      if A[i] < k2
3      then A[i] ← A[i] + k1
4      else  HEAPIFY(A, i, n)
```

Si noti che quando si esegue l'incremento dell'elemento $A[i]$ non è necessario applicare la procedura HEAPIFY perchè l'array di partenza rappresentava un max-heap e quindi se $A[i] < k2$ anche tutti i discendenti di $A[i]$ lo erano, e sono quindi già stati incrementati senza alterare la proprietà di max-heap. Al contrario quando $A[i]$ non viene incrementato potrebbe essere diventato minore di uno dei figli (precedentemente incrementati) e in questo caso si rende necessaria una chiamata alla procedura HEAPIFY.

5. (a) *Dare la definizione di* albero binario di ricerca
(b) *Data la struttura ad albero indicata in figura, inserire gli elementi 38, 72, 70, 15, 81, 17, 40, 43, 5 in modo da ottenere un albero binario di ricerca*
(c) *Successivamente si mostri l'albero di ricerca ottenuto dopo la rimozione del nodo corrispondente alla chiave 38.*

Proprietà BST. Per ogni coppia di nodi x e y in T
- se y è nel sottoalbero sinistro di x allora $key[y] \leq key[x]$
- se y è nel sottoalbero destro di x allora $key[x] \leq key[y]$

ATTENZIONE! molti hanno dato la seguente risposta **errata**: *per ogni nodo x deve essere $key[left[x]] \leq key[x] \leq key[right[x]]$.*



6. *Descrivere le principali caratteristiche di una tabella hash realizzata con la tecnica ad indirizzamento aperto.*

Vedi testo.

7. *Sia* T *un* albero binario di ricerca *ai cui nodi sono associati gli attributi* key, left, right, color, bh. *I primi sono gli usuali attributi di un albero* Rosso/Nero *mentre* bh[x] *è progettato per indicare l'altezza nera dell'albero radicato in* x.
Supponendo che T soddisfi le prime 4 proprietà caratteristiche degli alberi Rosso/Neri e che i campi key, left, right, color *siano già stati correttamente assegnati, scrivere lo pseudocodice di un algoritmo efficiente che verifica se T soddisfa anche la proprietà sui cammini.*

Dobbiamo controllare solo la proprietà relativa alle altezze nere. Si noti che il testo dell'esercizio non garantisce il contenuto del campo emphbh[x]. Questo può venir utilizzato durante la visita di T come nel seguente algoritmo.

```
RN-CHECK(x)
 1  if x = FOGLIA-NIL
 2     then BH[x] ← 1
 3          return TRUE
 4  if color[x] = BLACK
 5     then s ← 1
 6     else  s ← 0
 7  ck ← RN-CHECK(LEFT[x]) & RN-CHECK(RIGHT[x])
 8  if ck & (BH[LEFT[x]] = BH[RIGHT[x]])
 9     then BH[x] ← BH[LEFT[x]] + s
10          return TRUE
11     else  return FALSE
```

Algoritmi e Strutture Dati

Sessione invernale A.A. 2003/2004

Appello 26.01.04

Prima parte

1. *Per un certo problema sono stati trovati due possibili algoritmi risolutivi. I loro tempi di esecuzione soddisfano alle due relazioni di ricorrenza riportate nei seguenti punti* (a) *e* (b). *Si dica, giustificando la risposta, quale dei due algoritmi è da preferire nel caso si debbano risolvere problemi di grandi dimensioni.*

(a) $T_1(n) = 3T_1(\frac{n}{2}) + 3n^2 lg^2 n$

(b) $T_2(n) = 4T_2(\frac{n}{2}) + 2n^2 + n + 2lg^2 n$

Utilizziamo il Master Theorem per valutare la complessità asintotica dei due algoritmi.

(a) Si ha: $a = 3, b = 2$, quindi $1 < \log_2 3 < 2$. Poiché $3n^2 \text{lg}^{\,2} n = \Omega(n^2) = \Omega(n^{\log_2 3+\epsilon})$ se vale la condizione di regolarità ($\exists c < 1$ tale che $af(\frac{n}{b}) < cf(n)$) siamo nel caso 3 del Master Method. La condizione vale in quanto scelto $c = 3/4$ è facile vedere che $3(3\frac{n^2}{4}\ \text{lg}^{\,2}(\frac{n}{2})) < \frac{3}{4}(3n^2 \text{lg}^2 n)$. Allora $T_1(n) = \Theta(3n^2 \text{lg}^2 n)$.

(b) Si ha: $a = 4, b = 2$, quindi $\log_2 4 = 2$. Poiché $2n^2 + n + 2\text{lg}^{\,2} n = \Theta(n^2)$ siamo nel caso 2 del Master Method. Allora $T_2(n) = \Theta(n^2 \text{lg}\ n)$.

Dobbiamo quindi confrontare $3n^2 \text{lg}^2 n$ con $n^2 \text{lg}\ n$. Poiché $n^2 \text{lg}\ n = O(3n^2 \text{lg}^2 n)$ ma non vale il viceversa, l'algoritmo da scegliere è il secondo.

2. *Data la seguente procedura* FUN *se ne determini la complessità asintotica al crescere di* $n \in N$

```
FUN(A, n)
1  if n < 1 return 1
2  s ← 0
3  for j ← 1 to n
4        do s ← s + A[j]
5  return s + 2 FUN(A, n/2)
```

```
FUN(A, n)
1  if n < 1 return 1
2  s ← 0                                   1
3  for j ← 1 to n                          n + 1
4        do s ← s + A[j]                   ∑_{j=1}^{n} 1
5  return s + 2 FUN(A, n/2)                T(n/2)
```

Poiché $T_{\text{FUN}}(n)$ soddisfa la ricorrenza $T_{\text{FUN}}(n) = T_{\text{FUN}}(n/2) + kn$ la complessità asintotica della procedura FUN si trova risolvendo tale ricorrenza. Questo si ottiene facilmente utilizzando il Master Method. Si ha $a = 1, b = 2$, $\log_b a = 0$, $kn = \Omega(n^{0+\epsilon})$, e kn soddisfa la proprietà di regolarità. Quindi siamo nel caso 3 ed abbiamo $T_{\text{FUN}}(n) = \Theta(n)$.

3. *Si consideri la struttura dati albero (posizionale e generale) con gli attributi key[x], child[x], sibling[x] asociati ad ogni nodo x. Si descriva un algorimo che dato un intero k ed un albero T, calcola il numero dei nodi di T che hanno esattamente k figli.*

Si può risolvere con una visita in ampiezza *breadth-first-search* contando le iterazioni del ciclo while che visita ciascun gruppo di fratelli. Usiamo una variabile globale *tot* posta uguale a 0 prima della chiamata esterna CHILDREN-COUNT($root[T], k$).

```
CHILDREN-COUNT(x, k)
 1  if x = NIL return
 2  ENQUEUE(x,Q)
 3  while not QUEUE-EMPTY[Q]
 4        do y ← HEAD[Q]
 5           DEQUEUE[Q]
 6           s ← 1
 7           while y ≠ NIL
 8                 do s ← s + 1
 9                    if CHILD[y] ≠ NIL
10                       then ENQUEUE[CHILD[y],Q]
11                    y ← SIBLING[y]
12           if s = k
13              then tot ← tot + 1
```

Seconda parte

4. *Si consideri l'array A[1..7] contenente gli elementi 3,25,10,50,2,5,7.*

   (a) *Dire se A soddisfa la proprietà di max-heap o di quasi-max-heap o nessuna delle due. Giustificare la risposta.*

   Non è né un max-heap né un quasi-max-heap. Infatti $A[1] < A[\text{LEFT}[1]]$ e quindi non è un max-heap, inoltre $A[\text{LEFT}[1]] < A[\text{LEFT}[\text{LEFT}[1]]]$ e quindi non è neppure un quasi-max-heap.

   (b) *Nel caso in cui A non sia uno heap descrivere il risultato dell'applicazione della procedura* BUILD-MAX-HEAP(A).

   L'array che si ottiene dopo l'applicazione di BUILD-MAX-HEAP(A) è $A = [50, 25, 10, 3, 2, 5, 7]$.

   (c) *Descrivere infine quale è il risultato dell'applicazione della procedura* HEAP-EXTRACT-MAX(A) *al max-heap risultante dai punti precedenti.*

   Dopo l'estrazione del massimo l'array diviene è $A = [25, 7, 10, 3, 2, 5]$.

5. *Scrivere un algoritmo che dato un* albero binario di ricerca *T ed una chiave k restituisce il numero di chiavi di T il cui valore è minore di k. Valutare la complessità dell'algoritmo proposto.*

Proseguiamo come per una ricerca della chiave k contando tutti i nodi che vengono lasciati alla sinistra del cammino percorso durante la ricerca. Si noti che la stessa procedura può essere utilizzata anche per *contare* i nodi che vengono lasciati alla sinistra dato che alla sinistra di una chiave minore di k ci sono solo nodi minori di k. La chiamata esterna saraà LESS(ROOT[T],K).

```
LESS(x, k)
1  if x = NIL
2      then return 0
3  if KEY[x] ≥ k
4      then return LESS(LEFT[x])
5      else  return 1 + LESS(LEFT[x]) + LESS(RIGHT[x]).
```

6. *Descrivere le proprietà della procedura di partizione di una array che viene utilizzata dall'algoritmo* QUICKSORT(A, p, q).

L'algoritmo di quicksort si basa su di una procedura di partizione che deve soddisfare la seguente proprietà.

PARTIZIONA$[A, p, q]$ riorganizza la porzione $p \cdots q$ dell'array A e restituisce un indice r compreso tra p e q in modo che nell'array riorganizzata tutti gli elementi in $A[p \cdots r]$ siano minori o uguali a tutti gli elementi in $A[r+1 \cdots q]$. Formalmente

precondizione: $A[1..n]$ *è una array di lunghezza* n, $1 \leq p \leq q \leq n$.

$r \leftarrow$ PARTIZIONA$[A, p, q]$

postcondizione: $p \leq r \leq q$ *e* $A[p \cdots r] \leq A[r+1 \cdots q]$.

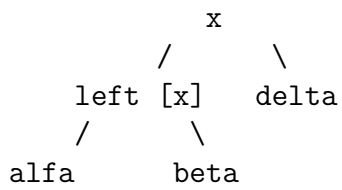
Si osservi che l'uso del pivot nelle realizzazioni studiate *serve* a garantire questa proprietà.

7. *Sia x un nodo in un albero Rosso/Nero T tale che* colore[x]=nero, colore[left[x]]=rosso, colore[right[x]]=nero. *Dire, giustificando formalmente la risposta, se l'applicazione di una rotazione a destra al nodo* x *distrugge la proprietà di albero Rosso/Nero o no.*

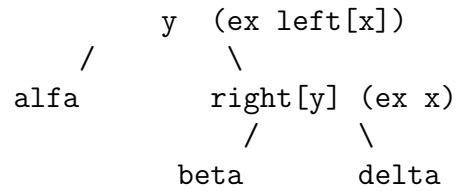
La rotazione non mantiene la proprietà (N.B. in nessun caso).

Chiamiamo altezza nera estesa di T l'altezza nera di T incrementata di 1 se la radice di T è nera e la denotiamo con $bh^*(T)$.

Siano α e β i figli di left[x] e chiamiamo δ il nodo right[x]. Poiché T è un albero Rosso/Nero, e colore[left[x]]=rosso, deve essere $bh^*(\alpha) = bh^*(\beta) = bh^*(\delta)$.



Chiamiamo y la nuova radice del sottoalbero dopo la rotazione.



Il figlio sinistro di y è α e quindi la sua altezza nera estesa è proprio $bh^*(\alpha)$. I figli di right[y] sono β e δ che hanno la stessa altezza nera estesa, cioè $bh^*(\beta)$. Pertanto right[y], che è nero ha altezza nera estesa $bh^*(\beta) + 1$.

Poiché $bh^*(\beta) + 1 = bh^*(\alpha) + 1 \neq bh^*(\alpha)$ l'albero dopo la rotazione non gode più della proprietà sulle altezza nere.

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 5/2/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Completare la seguente tabella indicando la complessità delle operazioni che si riferiscono a un dizionario di *n* elementi. Si noti che tutte le operazioni, tranne la **Ricerca**, assumono di aver già raggiunto l'elemento *x* a cui si applica l'operazione.

	Ricerca	**Cancellazione**	**Successore**	**Costruzione**
Lista doppia non ordinata				
Array ordinato				
Alberi binari di ricerca				

2. Risolvere le seguenti relazioni di ricorrenza (giustificando la risposta):

   (a) $T(n) = 4T(n/2) + n$ *(b)* $T(n) = 4T(n/2) + n^2$ *(c)* $T(n) = 4T(n/2) + n^3$

3. Si definisca la relazione di "riducibilità polinomiale" tra problemi ($\leq_P$) e si stabilisca se valgono le seguenti proprietà: *a*) riflessiva, *b*) simmetrica, *c*) transitiva (giustificando tecnicamente le risposte).

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 5/2/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

1. Progettare un algoritmo che, ricevuto in input un intero k e un array a, **non ordinato**, di n elementi **distinti**, restituisca il k-esimo elemento più piccolo di a.

- **a.** Progettare una soluzione di costo in tempo $\Theta(n \log n)$.
- **b.** Progettare una soluzione di costo in tempo $O(n + k \log n)$.

2. Un nodo di un albero binario è detto **centrale** se il numero di foglie del sottoalbero di cui è radice è pari alla somma delle chiavi dei nodi appartementi al percorso dalla radice al nodo stesso.
   a. Scrivere una funzione **efficiente** in C che restituisca il numero di nodi centrali.
   b. Discutere la complessità della soluzione trovata.
   c. Se vogliamo modificare la funzione in modo che restituisca l'**insieme** dei nodi centrali che tipo di struttura dati si può utilizzare per rappresentare l'insieme?
   La complessità dell'algoritmo deve rimanere la stessa che nel caso (a).

   Si deve utilizzare il seguente tipo per la rappresentazione di un albero binario:

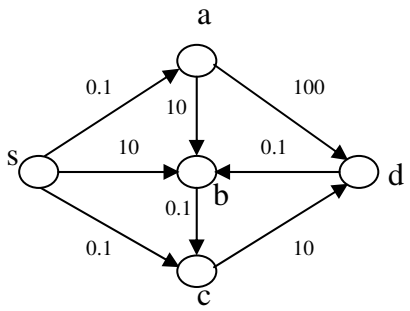
```
typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;
```

3. Si enunci e si dimostri la proprietà fondamentale degli alberi di copertura minimi e la si utilizzi per dimostrare la correttezza degli algoritmi di Kruskal e Prim.

4. Dato un grafo orientato e pesato $G=(V,E)$ con pesi strettamente positivi, cioè $w(u,v)>0$ per ogni $(u,v)\in E$, si vuole determinare se esiste in G un ciclo $c\equiv\langle x_0,x_1,\ldots,x_q\rangle$ raggiungibile da un dato vertice "sorgente" s, in cui il prodotto dei pesi sugli archi sia minore di 1, cioè:

$$\prod_{i=1}^{q} w(x_{i-1}, x_i) < 1.$$

   Si sviluppi un algoritmo per risolvere questo problema, se ne discuta la correttezza e si determini la sua complessità computazionale. Inoltre, si simuli la sua esecuzione sul seguente grafo:



Algoritmi e Strutture Dati

a.a. 2012/13

Compito del 06/09/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. In una tabella Hash di **m = 17** posizioni, inizialmente vuota, devono essere inserite le seguenti chiavi numeriche nell'ordine indicato:

   52, 1, 92, 37

   La tabella è a indirizzamento aperto e la scansione è eseguita per doppio Hashing:

   $$h(k, i) = (k \bmod m + i * 2^{k \bmod 5}) \bmod m$$

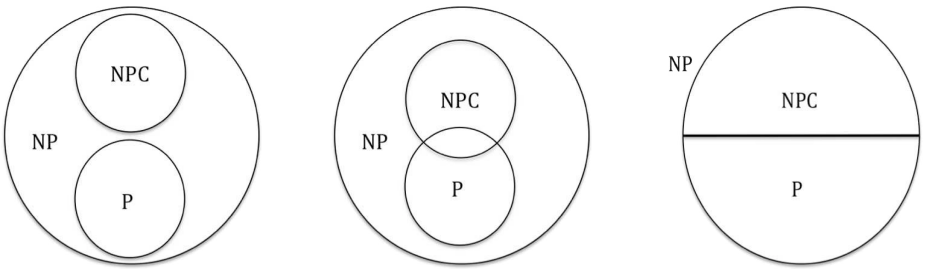
   Indicare per ogni chiave le posizioni scandite nella tabella e la posizione finale dove viene allocata.

2. Si definiscano le relazioni O, Ω, Θ e, utilizzando le definizioni date e nient'altro, si mostri che valgono le seguenti proprietà:

   a) $f(n) = O(g(n))$ se e solo se $g(n) = \Omega(f(n))$

   b) $f(n) = \Theta(g(n))$ se e solo se $g(n) = \Theta(f(n))$

3. Si definiscano le classi P, NP, NPC e si stabilisca, giustificando formalmente la risposta, quale delle seguenti relazioni è ritenuta vera (o verosimile):



(a) (b) (c)

Algoritmi e Strutture Dati

a.a. 2012/13

Compito del 06/09/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

1. Sia v un array di n interi distinti, tale che esiste una posizione j, $0 \leq j < n$, per cui:
   a. gli elementi nel sottoarray $v[0, j]$ sono ordinati in ordine decrescente;
   b. gli elementi in v[j+1, n-1] sono in ordine crescente;
   c. $v[j] < v[j+1]$, se $j < n - 1$

   Scrivere un algoritmo di tipo *divide-et-impera* **efficiente** che restituisca la posizione j. Calcolare la complessità al caso pessimo dell'algoritmo indicando, e risolvendo, la corrispondente relazione di ricorrenza.

   Per l'esame da **12 CFU**, deve essere fornita **una funzione C**.
   Per l'esame da **9 CFU**, è sufficiente specificare lo pseudocodice.

2. Dato un albero di ricerca T, progettare un algoritmo **efficiente** che restituisca il numero di elementi che occorrono una sola volta e analizzarne la complessità.

   a. Non si possono usare strutture ausiliarie di dimensione **O(n)** dove n è il numero dei nodi dell'albero.

   b. Devono essere definite esplicitamente eventuali funzioni/procedure ausiliarie. Si consideri la rappresentazione dell'albero binario di ricerca che utilizza i campi **left**, **right, p** e **key**.

3. Sia $G = (V, E)$ un grafo orientato e pesato e sia $s \in V$ un vertice "sorgente". Supponendo che G sia stato inizializzato con INIT-SINGLE-SOURCE(G, s), si dimostri che se una qualsiasi sequenza di passi di rilassamento assegna a $\pi[s]$ un valore diverso da NIL, allora G contiene un ciclo di peso negativo raggiungibile dalla sorgente s. (Suggerimento: si utilizzino le definizioni delle funzioni citate e le proprietà elementari della funzione RELAX).

4. Si vuole costruire una rete stradale che colleghi cinque città (A-E), minimizzando i costi complessivi di realizzazione. I costi per la costruzione di una strada tra due città sono sintetizzati nella seguente tabella (dove $+\infty$ significa che la strada è irrealizzabile):

	A	B	C	D	E
A	0	3	5	11	9
B	3	0	3	9	8
C	5	3	0	$+\infty$	10
D	11	9	$+\infty$	0	7
E	9	8	10	7	0

   Si formuli il problema dato in termini di un problema di ottimizzazione su grafi, e si descriva un algoritmo per la sua soluzione discutendone correttezza e complessità. Infine, si simuli accuratamente l'algoritmo presentato per determinare una soluzione del problema.

Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 10/09/2012

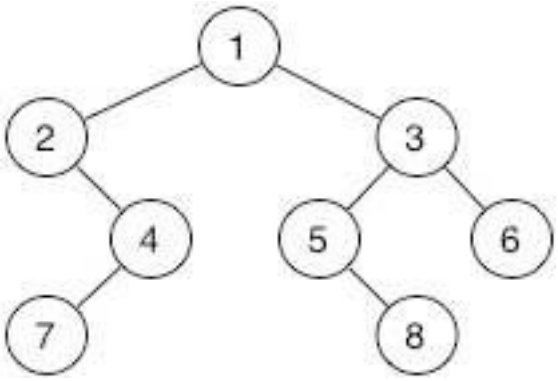
Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Dato il seguente albero



   Eseguire una visita in preordine, una visita in ordine simmetrico e una visita in postordine elencando nei tre casi la sequenza dei nodi incontrati.

2. Sia G un grafo orientato sparso con pesi sugli archi positivi. Si scriva un algoritmo per determinare le distanze tra tutte le coppie di vertici in G che sia asintoticamente più efficiente dell'algoritmo di Floyd-Warshall. (Si giustifichi tecnicamente la risposta.)

3. Si stabilisca se il seguente *ragionamento* è errato o meno, giustificando la risposta:
   a) Il problema ISOMORFISMO-DI-GRAFI, che consiste nel determinare se due grafi sono identici (isomorfi), può essere ricondotto al problema CLIQUE su un grafo ausiliario detto "grafo di associazione";
   b) Il problema CLIQUE è NP-completo;
   c) Quindi, il problema ISOMORFISMO-DI-GRAFI è NP-completo.

Algoritmi e Strutture Dati

a.a. 2012/13

Compito del 13/06/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

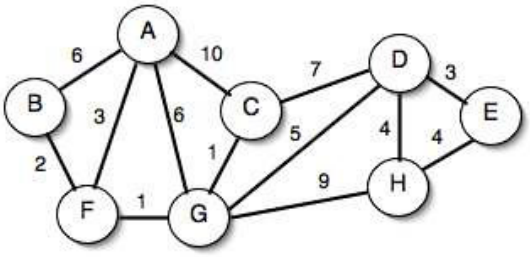
(30 minuti; ogni esercizio vale 2 punti)

1. Si vuole ordinare in senso crescente un array di lunghezza *n*. Indicare il tempo di esecuzione dei seguenti algoritmi di ordinamento nei casi specificati nelle due colonne:

	Array già ordinato in senso crescente	**Array già ordinato in senso decrescente**
Insertion sort		
Quicksort		
Heapsort		

2. Si mostri, utilizzando la definizione, che la relazione O soddisfa la proprietà transitiva, ovvero:
"Se $f(n) = O(g(n))$ e $g(n) = O(h(n))$, allora $f(n) = O(h(n))$"

3. Si determini un albero di copertura minimo nel seguente grafo:



Algoritmi e Strutture Dati

a.a. 2012/13

Compito del 13/06/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

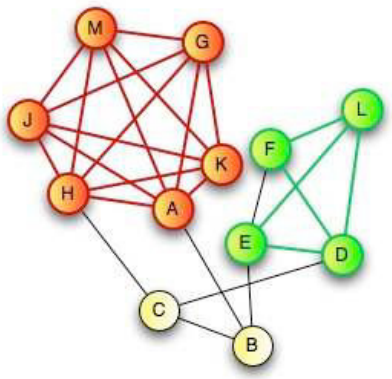
1. Dato un albero binario, i cui nodi contengono chiavi intere, progettare un algoritmo **efficiente** che stabilisca se le chiavi nei nodi soddisfano la proprietà del **max-heap**, e analizzarne la complessità.

   Per l'esame da **12 CFU**, deve essere fornita **una funzione C** e si deve dichiarare il tipo **Node** utilizzato per rappresentare l'albero binario.
   Per l'esame da 9 CFU, è sufficiente specificare lo pseudocodice.

2. Dato un array A di n elementi, progettare un algoritmo **efficiente** che costruisca **ricorsivamente** un albero binario **bilanciato** tale che $A[i]$ sia l'$(i+1)$-*esimo* campo *u.key* in ordine di visita posticipata (postordine).

   Discutere la complessità al caso pessimo indicando, e risolvendo, la corrispondente relazione di ricorrenza.

3. Si scriva un algoritmo di complessità $O(n^2)$ per determinare una clique massimale all'interno di un grafo non orientato con n vertici. Si discuta la sua complessità e la sua correttezza e si simuli accuratamente la sua esecuzione sul seguente grafo:



   L'algoritmo è in grado di determinare anche clique massime? In caso negativo si fornisca un controesempio.

4. Si definisca formalmente la relazione di riducibilità polinomiale tra problemi decisionali ($\leq_P$) e si stabilisca se le seguenti affermazioni sono vere o false:

   1) La relazione $\leq_P$ è transitiva
   2) La relazione $\leq_P$ è riflessiva
   3) Se $\leq_P$ è simmetrica, allora P = NP
   4) Se $\mathcal{P} \leq_P \mathcal{Q}$ e $\mathcal{Q} \in$ P, allora $\mathcal{P} \in$ P
   5) Se $\mathcal{P}, \mathcal{Q} \in$ NPC, allora $\mathcal{P} \leq_P \mathcal{Q}$ se e solo se $\mathcal{Q} \leq_P \mathcal{P}$

   Nel primo caso si fornisca una dimostrazione rigorosa, nel secondo un controesempio.
   (Nota: in caso di discussioni poco formali l'esercizio non verrà valutato pienamente.)

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 14/9/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Determinare il numero di **nodi interni** in un albero **d-ario completo** in funzione dell'altezza h e dimostrare per induzione la correttezza della risposta.

2. Si diano le definizioni di O, Ω e Θ e si stabilisca, utilizzando le definizioni date, se $3n^2 + 7n = \Theta(n^2)$.

3. Si definisca la relazione di "riducibilità polinomiale" tra problemi ($\leq_P$) e si stabilisca se valgono le seguenti proprietà: *a*) riflessiva, *b*) simmetrica, *c*) transitiva (giustificando tecnicamente le risposte).

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 14/9/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

1. Dato un albero binario *T,* definire una funzione **efficiente** in C che restituisca una copia *T'* di *T*, che contenga anche, in ogni nodo, il numero di nodi del sottoalbero di cui è radice (radice inclusa).

   Discutere la complessità della soluzione trovata.

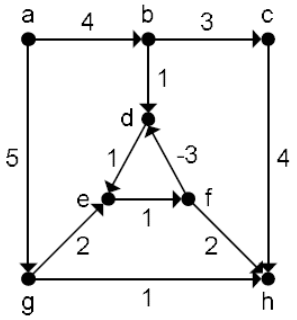
   Il tipo dell'albero *T* è:

   ```
   typedef struct node{
       int key;
       struct node * left;
       struct node * right;
   } * Node;
   ```

   Modificare in modo adeguato il tipo dell'albero *T* per ottenere il tipo dell'albero *T'.*

2. Progettare un algoritmo che, ricevuto in input un intero *k* e un array *a,* **non ordinato**, di *n* elementi **distinti**, restituisca il *k*-esimo elemento più piccolo di *a.* La soluzione **deve** essere di costo in tempo *O(n log k)* e utilizza uno heap di *k* elementi.

3. Si scriva l'algoritmo di Bellman-Ford, si dimostri la sua correttezza, si fornisca la sua complessità computazionale e si simuli accuratamente la sua esecuzione sul seguente grafo (uitlizzando il vertice *a* come sorgente):



4. Si scriva l'algoritmo di Dijkstra, si dimostri la sua correttezza e si fornisca la sua complessità computazionale. Si supponga inoltre di cambiare l'istruzione **while Q ≠ Ø** dell'algoritmo, con la seguente: **while |Q| > 1**. Questa variazione fa eseguire il ciclo |V|– 1 volte invece di |V|. L'algoritmo proposto è corretto? (Giustificare "tecnicamente" la risposta).

Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 15/05/2012

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Dato un albero binario completamente bilanciato, descrivere a parole un algoritmo qualsiasi che abbia complessità:
   a. lineare rispetto all'altezza;
   b. lineare rispetto al numero dei nodi;
   c. esponenziale rispetto all'altezza.

2. Si inseriscano le chiavi del seguente insieme

   $$K = \{ 68, 39, 57, 15, 70, 74, 99, 24 \}$$

   in una tabella hash di dimensione $m = 11$, inizialmente vuota, utilizzando il metodo della divisione e le liste di collisione.

3. Si scriva un algoritmo di complessità $O(nm\log n)$ per determinare le distanze tra tutte le coppie di vertici in un grafo orientato G avente pesi sugli archi positivi, dove n e m sono, rispettivamente, il numero di vertici e il numero di archi in G.

Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 15/05/2012

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

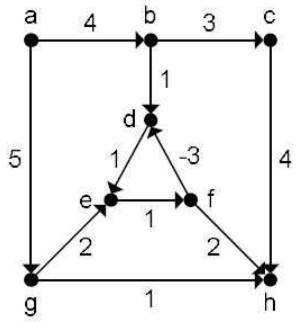
1. Un nodo di un albero binario è detto **pari** se il numero di foglie del sottoalbero di cui è radice è pari.
   a. Progettare un algoritmo **efficiente** che dato un albero binario restituisca il numero di nodi pari.
   b. Discutere brevemente la complessità della soluzione trovata.

   La rappresentazione dell'albero binario utilizza esclusivamente i campi **left, right** e **key.**

2. Dato un array non ordinato di **n** interi, eventualmente ripetuti, progettare un algoritmo **efficiente** che restituisca il numero di elementi che occorrono una sola volta e analizzarne la complessità in tempo.

   L'algoritmo deve utilizzare spazio aggiuntivo costante e devono essere definite esplicitamente eventuali funzioni/procedure ausiliarie.

3. Si scriva l'algoritmo di Bellman-Ford, si dimostri la sua correttezza, si fornisca la sua complessità computazionale e si simuli accuratamente la sua esecuzione sul seguente grafo:



4. Si enunci e si dimostri il teorema fondamentale delle ricorrenze e lo si utilizzi per risolvere le seguenti ricorrenze (spiegando in quali casi del teorema ricade la soluzione di ciascuna equazione):

   a. $T(n) = 3T(n/2) + n^2$
   b. $T(n) = 4T(n/2) + n^2$
   c. $T(n) = T(n/2) + 2^n$
   d. $T(n) = 16T(n/4) + n$

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 16/7/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Determinare il numero di foglie e il numero totale di nodi in un albero binario completo in funzione dell'altezza *h* e dimostrare per induzione la correttezza della risposta.

2. Ordinare le seguenti funzioni in una lista in modo che se *f* viene prima di *g*, allora $f(n) = O(g(n))$. Se due o più funzioni hanno lo stesso ordine asintotico, lo si indichi esplicitamente.

$$\begin{array}{cccc} n & 2^n & n \lg n & n^3 \\ n^2 & \lg n & n - n^3 + 7n^5 & n^2 + \lg n \end{array}$$

3. Si enunci e si dimostri il teorema fondamentale della NP-completezza.

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 16/7/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

1. Dato un vettore v di n interi non necessariamente distinti, si dice che un elemento è di **maggioranza** se appare in v almeno $\lceil n/2 \rceil$ volte. Scrivere due funzioni in C di complessità in spazio aggiuntivo costante e in tempo
   a. $\Theta(n^2)$
   b. $\Theta(n \log n)$

   per stabilire se v contiene un elemento di maggioranza, e in caso affermativo lo restituisca.

2. Dato un albero binario, progettare un algoritmo **ricorsivo** che costruisce un array bidimensionale m tale che, per ogni coppia di nodi u e v, l'elemento $m[u][v]$ sia il minimo antenato comune di u e v. L'algoritmo deve richiedere tempo $O(n^2)$.
   Osservate che durante la ricorsione sul nodo corrente u, potete individuare quali coppie di nodi hanno u come minimo antenato comune.

   [Il **minimo antenato comune** di due nodi u e v è l'antenato comune di u e v che si trova più lontano dalla radice dell'albero.]

3. L'*arbitraggio* è un'operazione finanziaria per lucrare dalla differenza di prezzi tra le varie piazze e mercati. Sia $V = \{ v_1, v_2, \ldots, v_n \}$ un insieme di n valute e si indichi con C_{ij} il tasso di cambio tra le valute v_i e v_j (cioè, vendendo 1 unità di valuta v_i si ottengono C_{ij} unità di valuta v_j). Un arbitraggio è possibile se esiste una sequenza di azioni elementari di cambio (*transazioni*) che inizi con 1 unità di una certa valuta e termini con più di 1 unità della stessa valuta. Per esempio, se i tassi di cambio sono: 1.53 franchi svizzeri per 1 euro, 0.94 dollari americani per 1 franco svizzero, e 0.77 euro per 1 dollaro americano, possiamo convertire 1 euro in 1.1 euro, realizzando un guadagno del 10%. Si formuli il problema dell'arbitraggio come un problema (noto) di ricerca su grafi e si sviluppi un algoritmo efficiente per la sua risoluzione, discutendone correttezza e complessità.

4. Si stabilisca se le seguente affermazioni sono vere o false, fornendo una dimostrazione nel primo caso e un controesempio nel secondo:
   a. « Sia $G = (V, E, w)$ un grafo orientato e pesato, e sia $G' = (V, E, w')$ il grafo pesato ottenuto da G aggiungendo una costante k ai pesi (in altri termini, G' ha gli stessi vertici e gli stessi archi di G, e $w'(u,v) = k + w(u,v)$, per ogni arco (u,v) di $E'=E$). Allora, $p=\langle x_0,\ldots,x_q\rangle$ è un cammino minimo in G se e solo se p è un cammino minimo in G'. »
   b. « Sia $G = (V, E, w)$ un grafo connesso non orientato e pesato, e sia $G' = (V, E, w')$ il grafo pesato ottenuto da G aggiungendo una costante k ai pesi (in altri termini, G' ha gli stessi vertici e gli stessi archi di G, e $w'(u,v) = k + w(u,v)$, per ogni arco (u,v) di $E'=E$). Allora, T è un albero di copertura minimo di G se e solo se T è un albero di copertura minimo di G'. »

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 18/6/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Completare la seguente tabella indicando la complessità delle operazioni che si riferiscono a un dizionario di *n* elementi. Si noti che l'operazione **Predecessore** assume di aver già raggiunto l'elemento *x* a cui si applica l'operazione.

	Ricerca	**Predecessore**	**Costruzione**
Tabelle Hash con liste di collisione (caso medio)*			
Tabelle Hash a indirizzamento aperto (caso pessimo)*			
Tabelle Hash a indirizzamento aperto (caso medio)*			
Alberi binari di ricerca bilanciati			

*La Tabella Hash ha dimensione *m* e il fattore di carico è α

2. Risolvere le seguenti relazioni di ricorrenza (giustificando le risposte):

   (a) $T(n) = 3 \cdot T(n/2) + n^2$ *(b)* $T(n) = 16 \cdot T(n/4) + n$ *(c)* $T(n) = 4 \cdot T(n/2) + n^2$

3. Si definiscano le classi di complessità P, NP, NPC, e si dimostri che $P \cap NPC \neq \varnothing \Rightarrow P = NP$.

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 18/6/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

1. Dato un albero binario T scrivere una funzione **efficiente** in C che restituisca 1 se **per ogni** nodo *u* di T vale la seguente proprietà: il sottoalbero sinistro di *u* ha una dimensione **almeno doppia** di quella del sottoalbero destro di *u* (la dimensione è il numero di nodi in esso contenuti), 0 altrimenti.
   Inoltre:
   a. discutere la complessità della soluzione trovata.
   b. dimostrare la correttezza della soluzione proposta (facoltativo).

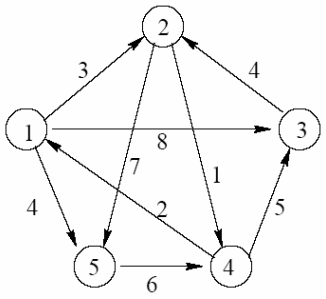
   Si deve utilizzare il seguente tipo per la rappresentazione di un albero binario:

```
typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;
```

2. Sono dati due alberi binari completamente bilanciati, di radice *r* e *s* rispettivamente, aventi la stessa altezza *h* e dimensione totale (somma dei nodi dei due alberi) *n***.** Le chiavi memorizzate nei nodi di entrambi gli alberi soddisfano la **proprietà di max-heap.**
   Si vogliono fondere i due alberi, ottenendo un unico albero completo a sinistra, di altezza *h+1* e dimensione *n*, che soddisfi la proprietà di max-heap.
   a. Progettare una soluzione di costo in tempo $\Theta(n)$ e in spazio aggiuntivo $\Theta(n)$.
   b. Progettare una soluzione di costo in tempo $O(\log n)$ e spazio aggiuntivo costante.

   La rappresentazione dell'albero binario utilizza esclusivamente i campi **left, right** e **key.**

3. Si scriva l'algoritmo di Dijkstra, si dimostri la sua correttezza, si fornisca la sua complessità computazionale e si simuli accuratamente la sua esecuzione sul seguente grafo (utilizzando il vertice 1 come sorgente):



4. Sia $G=(V,E)$ un grafo non orientato, connesso e pesato avente tutti i pesi distinti. Sia *C* un ciclo di *G* e sia *(u,v)* l'arco in *C* avente peso massimo. Si stabilisca se esiste o meno in *G* un albero di copertura minimo che contiene l'arco *(u,v)*. Potremmo dire lo stesso se i pesi sugli archi non fossero distinti? Giustificare formalmente le risposte.

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 19/2/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Completare la seguente tabella indicando la complessità delle operazioni che si riferiscono a un dizionario di *n* elementi. Si noti che tutte le operazioni, tranne la **Ricerca**, assumono di aver già raggiunto l'elemento *x* a cui si applica l'operazione.

	Minimo	**Ricerca**	**Cancellazione**	**Successore**	**Costruzione**
Lista semplice ordinata					
minHeap					

2. Risolvere le seguenti relazioni di ricorrenza (giustificando le risposte):

   (a) $T(n) = 2 \cdot T(n/2) + \sqrt{n}$ *(b)* $T(n) = 27 \cdot T(n/3) + n^3$ *(c)* $T(n) = 5 \cdot T(n/4) + n^2$

3. Si definiscano le classi di complessità P, NP, NPC, e si dica quale delle seguenti affermazioni è vera, quale è falsa e quale è improbabile (giustificando le risposte):

   (a) $NP \subseteq P$ (b) $P \subseteq NP$ (c) P u NPC = NP (d) $P \cap NPC \neq \emptyset \Rightarrow P = NP$

Algoritmi e Strutture Dati

a.a. 2009/10

Compito del 19/2/2010

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

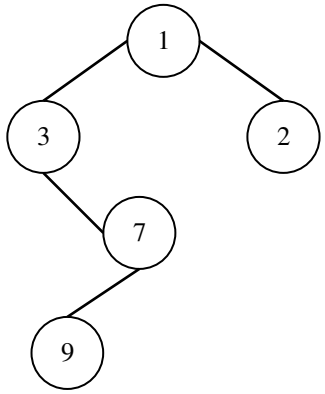
(2.5 ore; ogni esercizio vale 6 punti)

1. Un albero binario si dice **t-bilanciato** se per ogni suo nodo vale la proprietà: le altezze dei sottoalberi radicati nei suoi due figli differiscono per al più **t** unità.
   a. Dato un albero binario, scrivere una funzione **efficiente** in C che restituisca il minimo valore **t** per cui l'albero risulti **t-bilanciato**.
   b. Discutere la complessità della soluzione trovata.

   Si deve utilizzare il seguente tipo per la rappresentazione di un albero binario:

```
typedef struct node{
    int key;
    struct node * left;
    struct node * right;
} * Node;
```

2. Dato il seguente albero:



   a. Eseguire una visita in ordine anticipato e una visita in ordine simmetrico elencando nei due casi la sequenza dei nodi incontrati.
   b. Progettare un algoritmo che dati due vettori contenenti rispettivamente i valori dei nodi tutti **distinti** ottenuti da una visita in ordine anticipato e da una visita in ordine simmetrico di un albero binario, ricostruisca l'albero binario.
      Un possibile prototipo della funzione è:

      ricostruisci(array vant, int infant, int supant, array vsim, int infsim, int supsim) →Node

      Suggerimento : il vettore vant permette di determinare la radice dell'albero mentre il vettore vsim la suddivisione dei nodi fra sottoalbero sinistro e destro.
   c. Analizzare la complessità dell'algoritmo nel caso pessimo.

La rappresentazione dell'albero binario utilizza esclusivamente i campi **left, right** e **key.**

3. La seguente tabella fornisce le distanze (in unità di 100 miglia) tra gli aeroporti delle città di Londra, Città del Messico, New York, Parigi, Pechino e Tokyo:

	L	*CM*	*NY*	*Pa*	*Pe*	*T*
L	–	56	35	2	51	60
CM	56	–	21	57	78	70
NY	35	21	–	36	68	68
Pa	2	57	36	–	51	61
Pe	51	78	68	51	–	13
T	60	70	68	61	13	–

Utilizzando sia l'algoritmo di Kruskal che quello di Prim, si determini un albero di copertura minimo per il grafo corrispondente. Le esecuzioni di entrambi gli algoritmi dovranno essere simulate accuratamente.

4. Si scriva l'algoritmo di Floyd-Warshall per il problema dei cammini minimi tra tutte le coppie, si fornisca la sua complessità computazionale, si dimostri la sua correttezza e si simuli accuratamente la sua esecuzione sulla seguente matrice:

$$W = \begin{bmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 8 & \infty & 1 \\ 6 & 2 & 0 & 4 & 3 \\ 1 & \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 1 & 0 \end{bmatrix}.$$

Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 26/1/2012

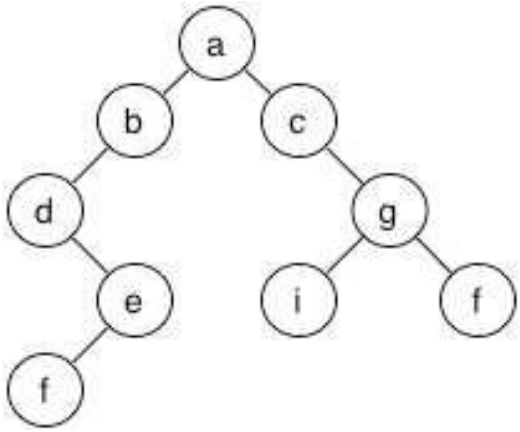
Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Dato il seguente albero:



   Eseguire una visita in preordine , una visita in ordine simmetrico e una visita in postordine elencando nei tre casi la sequenza dei nodi incontrati.

2. Si diano le definizioni di O, Ω, Θ, o, ω e si stabilisca, utilizzando le definizioni date, se $7n^3 + 2n = \Theta(n^3)$.

3. Si definiscano le classi di complessità P, NP, NPC, e si mostri che se esiste un problema NP-completo risolvibile polinomialmente allora P = NP.

Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 26/1/2012

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

1. Dato un albero binario con chiavi intere positive e negative, progettare un algoritmo **efficiente** che restituisca la somma totale di tutte le chiavi dell'albero.
   Definire un secondo algoritmo, anch'esso **efficiente**, che restituisca il massimo peso di tutti i sottoalberi, dove il peso di un sottoalbero è la somma di tutte le chiavi dei suoi nodi.

   Valutare la complessità dei due algoritmi.

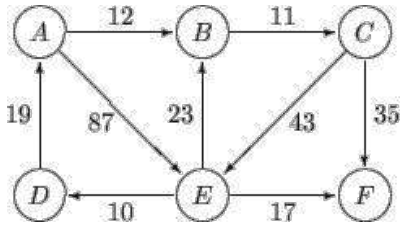
   La rappresentazione dell'albero binario utilizza esclusivamente i campi **left, right** e **key.**

2. Modificare la funzione PARTITION del quicksort per ottenere una funzione PARTITION'(A, p, r) che **permuta** gli elementi di A[p..r] e restituisce due indici q e t, con $p \le q \le t \le r$, tali che
   a. tutti gli elementi di A[q..t] siano uguali
   b. ogni elemento di A[p..q-1] sia minore di A[q]
   c. ogni elemento di A[t+1..r] sia maggiore di A[q]

   Come PARTITION, anche PARTITION' dovrà richiedere un tempo $\Theta(r-p)$.

   Dimostrare la correttezza di PARTITION' definendo un'invariante di ciclo.

3. Si scriva l'algoritmo di Dijkstra, si dimostri la sua correttezza, si fornisca la sua complessità computazionale e si simuli accuratamente la sua esecuzione sul seguente grafo (utilizzando il vertice A come sorgente):



4. Sia $G=(V,E)$ un grafo non orientato, connesso e pesato avente tutti i pesi distinti. Sia C un ciclo di G e sia (u,v) l'arco in C avente peso massimo. Si stabilisca se esiste o meno in G un albero di copertura minimo che contiene l'arco (u,v). Potremmo dire lo stesso se i pesi sugli archi non fossero distinti? Giustificare formalmente le risposte.

Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 28/01/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Dato l'insieme delle chiavi {1,4,5,10,16,17,21}, quale è l'altezza minima *hmin* di un albero binario di ricerca che contenga esattamente queste chiavi? E l'altezza massima *hmax*?
   Disegnare 3 alberi binari di ricerca con le chiavi dell'insieme specificato rispettivamente di altezza *hmin*, *hmax* e di un'altezza *h* tale che $hmin < h < hmax$.

2. Si enunci la proprietà fondamentale degli alberi di copertura minimi (definendo accuratamente tutti i termini impiegati) e la si utilizzi per mostrare che se (u,v) è un arco di peso minimo in un grafo non orientato G, allora (u,v) appartiene a un albero di copertura minimo di G.

3. Si stabilisca se la seguente affermazione è vera o falsa, fornendo nel primo caso una dimostrazione, nel secondo un controesempio:

   "Se $P \neq NP$, allora $P \cap NPC = \varnothing$".

Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 28/01/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

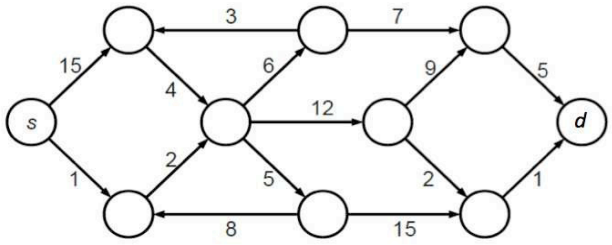
1. Dare la definizione di albero binario **completo**. Progettare un algoritmo **efficiente** per stabilire se un albero binario è **completo** e calcolarne la complessità al caso pessimo indicando, e risolvendo, la corrispondente relazione di ricorrenza.

2. Insertion sort può essere espresso come una procedura ricorsiva nel modo seguente: per ordinare A[1..n], si ordina in modo ricorsivo A[1..n-1] e poi si inserisce A[n] nell'array ordinato A[1..n-1]. Scrivere la versione **ricorsiva** dell'insertion sort. Infine scrivere una ricorrenza per il tempo di esecuzione di questa versione ricorsiva e risolverla.

3. Determinare il costo computazionale T(n) del seguente algoritmo, in funzione del parametro $n \geq 0$:

```
MyAlgorithm( int n ) → int
int
  a, i, j;
if ( n > 1 ) then
   a := 0;
   for i := 1 to n-1
      for j := 1 to n-1
         a := a + (i+1)*(j+1);
      endfor
   endfor
   for i := 1 to 16
      a := a + MyAlgorithm(n/4);
   endfor
   return a;
else
   return n-1;
endif
```

4. La rete ferroviaria italiana può essere descritta mediante un grafo orientato pesato $G=(V, E, w)$, dove i vertici rappresentano le stazioni, la presenza di un arco tra due vertici indica l'esistenza di una tratta ferroviaria diretta tra le corrispondenti stazioni e, per ogni arco $(u,v)\in E$, il peso $w(u,v)$ rappresenta la quantità di carburante necessaria per raggiungere la stazione v partendo da u. Si scriva un algoritmo che, dati in ingresso il grafo G, la quantità C di carburante inizialmente presente nel serbatoio della locomotiva di un treno, e due nodi s e d, restituisca TRUE se esiste un cammino che consente al treno di raggiungere la stazione d partendo dalla stazione s, e FALSE in caso contrario. Si discuta della correttezza e della complessità computazionale dell'algoritmo proposto e si simuli accuratamente la sua esecuzione sul seguente grafo, con $C = 15$.



Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 29/05/2012

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Discutere qual è il caso ottimo di Quicksort e determinarne la complessità. Scrivere esplicitamente la ricorrenza.

2. Per un certo problema sono stati trovati due algoritmi risolutivi (A_1 e A_2) con i seguenti tempi di esecuzione:

   A_1: $T(n) = 5\ T(n/6) + 2n^2$

   A_2: $T(n) = 7\ T(n/6) + n$

   Si dica, giustificando tecnicamente la risposta, quale dei due algoritmi è preferibile per input di dimensione sufficientemente grande.

3. Si definisca formalmente la relazione di riducibilità polinomiale tra problemi decisionali ($\leq_P$) e si spieghi (tecnicamente) perché se fosse simmetrica si avrebbe $P = NP$.

Algoritmi e Strutture Dati

a.a. 2011/12

Compito del 29/05/2012

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

1. Sia T un albero generico i cui nodi hanno chiavi intere e campi: **key**, **left-child**, **right-sib**. Scrivere un algoritmo che calcoli l'altezza di tale albero.

   Discutere brevemente la complessità della soluzione trovata.

2. Dato un array di **n** interi, progettare un algoritmo **efficiente** che costruisca un albero binario di ricerca di altezza **Θ(log n)** che contenga gli interi dell'array come chiavi, e analizzarne la complessità.

   Devono essere definite esplicitamente eventuali funzioni/procedure ausiliarie. Si consideri la rappresentazione dell'albero binario che utilizza i campi **left**, **right** e **key**.

3. Si stabilisca se le seguenti affermazioni sono vere o false, fornendo una dimostrazione nel primo caso e un controesempio nel secondo:

   a) « Sia $G = (V, E, w)$ un grafo orientato e pesato, e sia $G' = (V, E, w')$ il grafo pesato ottenuto da G aggiungendo una costante k ai pesi (in altri termini, G' ha gli stessi vertici e gli stessi archi di G, e $w'(u,v) = k + w(u,v)$, per ogni arco (u,v) di $E'=E$). Allora, $p = \langle x_0,\ldots,x_q \rangle$ è un cammino minimo in G se e solo se p è un cammino minimo in G'. »

   b) « Sia $G = (V, E, w)$ un grafo connesso non orientato e pesato, e sia $G' = (V, E, w')$ il grafo pesato ottenuto da G aggiungendo una costante k ai pesi (in altri termini, G' ha gli stessi vertici e gli stessi archi di G, e $w'(u,v) = k + w(u,v)$, per ogni arco (u,v) di $E'=E$). Allora, T è un albero di copertura minimo di G se e solo se T è un albero di copertura minimo di G'. »

4. Si scriva l'algoritmo di Floyd-Warshall, si dimostri la sua correttezza, si fornisca la sua complessità computazionale e si simuli accuratamente la sua esecuzione sulla seguente matrice:

$$W = \begin{bmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 8 & \infty & 1 \\ 6 & 2 & 0 & 4 & 3 \\ 1 & \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 1 & 0 \end{bmatrix}$$

Algoritmi e Strutture Dati

a.a. 2012/13

Compito del 30/05/2013

Cognome: _____ Nome: _____

Matricola: _____ E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

1. Quali sono il numero minimo e il numero massimo di elementi in un heap di altezza h? Giustificare brevemente la risposta.

2. Utilizzando la definizione di O, e nient'altro, si stabilisca se le seguente affermazioni sono vere o false:

   a) $f(n) = O(n)$

   b) $f(n) = O(n^2)$

   dove $f(n) = n(n+1)/2$.

3. Si definiscano le classi di complessità P, NP, NPC, e si dica quali delle seguenti affermazioni è vera (giustificando le risposte):

   (a) NP = P *(b)* NPC ∩ NP = NPC *(c)* P ∪ NPC = NP *(d)* P ∩ NPC ≠ ∅

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

– Appello dell' 8 Febbraio 2005 –

Esercizio 1 (ASD)

1. Dire quale delle seguenti affermazioni è vera giustificando la risposta.
   (a) $\lg n = \Omega(n)$
   (b) $n = \Omega(\lg n)$
   (c) $n = O(\lg n)$
   (d) Nessuna delle precedenti è vera
2. Un algoritmo di tipo divide et impera per risolvere un problema di dimensione n lo decompone in 4 sottoproblemi di dimensione $(n/2)$ ciascuno, e ricombina le loro soluzioni con un procedimento la cui complessità asintotica è caratterizzata dalla funzione $f(n) = 3n^2 + \lg n$. Qual è la complessità asintotica dell' algoritmo? Giustificare la risposta.

Soluzione

1. Risposta esatta: (b) - Si può applicare la definizione della classe $\Omega(\lg n)$: $\lg n \leq n$ per ogni $n \geq 2$.
   Si può anche calcolare il limite: $\lim_{n\to\infty} \frac{n}{\lg n} = \infty$ per concludere che $n = \omega(\lg n)$ e quindi $n = \Omega(\lg n)$.
2. Si tratta di risolvere la ricorrenza: $T(n) = 4T(\frac{n}{2}) + 3n^2 + \lg n$.
   E' facile vedere che è applicabile il Master Theorem. Si ha: $a = 4$, $b = 2$ e $\log_b a = \log_2 4 = 2$ ed $f(n) = 3n^2 + \lg n$. Poichè $f(n) = \Theta(n^2)$ ($n^2 \leq 3n^2 + \lg n \leq 4n^2$ per ogni $n \geq 2$), $f(n) = \Theta(n^2) = \Theta(n^{\log_b a})$ e quindi (caso 2 del Master Theorem) la soluzione è: $T(n) = \Theta(n^2 \lg n)$.

Esercizio 2 (ASD)

Dire quale delle seguenti affermazioni è esatta relativamente a ciascuna delle seguenti domande.
1) Quanti scambi vengono effettuati dall'algoritmo HeapSort?
2) Quanti confronti vengono effettuati dall'algoritmo QuickSort?

(a) $O(n)$ nel caso medio

(b) $O(n^2)$ nel caso peggiore

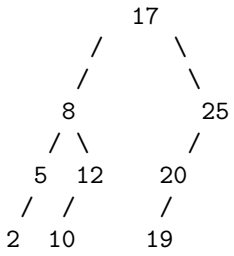
(c) $O(n \lg n)$ nel caso peggiore

Soluzione

1) Risposta esatta: (c)
2) Risposta esatta: (b)

Esercizio 3 (ASD)

Dire quali delle seguenti affermazioni sono applicabili al seguente albero, giustificando la risposta.



(a) È un albero binario di ricerca che puo' essere colorato in modo da divenire R/N
(b) È un albero binario di ricerca non bilanciato
(c) È un max-heap
(d) È un min-heap

Soluzione

La sola affermazione applicabile è la (b) in quanto è facile vedere che è soddisfatta la condizione BST ma che l'abero non può essere colorato poichè sul cammino 17,25,20,19,NIL non può esserci lo stesso numero di nodi neri del cammino 17,25,NIL senza violare altri punti della proprietà RB. Inoltre è facile verificare che l'albero non può rappresentare né un max-heap (es. $17 < 25$) né un min-heap (es. $17 > 8$).

Esercizio 4 (ASD e Laboratorio)

Si consideri la classe *BTNode* per memorizzare i nodi di un albero binario (riportata alla fine del testo d'esame). Si consideri inoltre la seguente classe *AlmostComplete* che costruisce alberi quasi completi (ovvero alberi completi in cui mancano zero o più foglie "più a destra" dell'ultimo livello). Gli alberi vengono costruiti per livelli e in modo che siano sempre quasi completi. Quindi ogni nuovo elemento viene inserito dopo la foglia più a destra dell'ultimo livello.

```
public class AlmostComplete  {
    BTNode root;      // radice dell'albero
    int count;           // numero di elementi dell'albero

    // post: crea un albero vuoto
    public AlmostComplete() { root = null; count = 0; }
    ...
    ...
    // pre: albero non vuoto e quasi completo
    // post: ritorna il riferimento all'ultima foglia inserita nell'albero
    private BTNode findLastLeaf() {...}
}
```

Si richiede di implementare il metodo *findLastLeaf* che ritorna il riferimento all'ultima foglia inserita nell'albero. In particolare, si richiede di:

1. (**ASD e Laboratorio**) scrivere lo pseudocodice di un algoritmo iterativo che risolva il problema
2. (**ASD e Laboratorio**)provare la correttezza dell'algoritmo
3. (**Laboratorio**) scrivere l'implementazione Java del metodo
4. (**EXTRA**) Scrivere lo pseudocodice di un algoritmo che risolva il problema in O(log(n)) (dove n è il numero di nodi dell'albero), sfruttando l'informazione relativa al numero di elementi presenti nell'albero quasi completo e le relazioni padre-figlio tra i nodi.

Soluzione

1. Una soluzione consiste nell'effettuare la visita in ampiezza dell'albero salvando il riferimento di ciascun nodo. L'ultimo nodo visitato sarà proprio la foglia cercata. Un'altra soluzione è la seguente:

   findLastLeaf()
   pos ← root
   while left[pos] ≠ nil do
   if height(right[pos]) = height(left[pos])
   then pos ← right[pos]
   else pos ← left[pos]
   return pos

2. Dobbiamo dimostrare che l'algoritmo ritorna correttamente il riferimento all'ultima foglia inserita nell'albero. L'invariante del ciclo è il seguente:

   INV = l'ultima foglia inserita nel sottoalbero radicato in pos è l'ultima foglia dell'albero radicato in root.

   Infatti:

Inizializzazione; all'inizio pos punta all'albero radicato in root e quindi l'invariante è vero.

Mantenimento; supponiamo che l'invariante sia vero per pos fissato ovvero che l'ultima foglia del sottoalbero radicato in pos sia anche l'ultima foglia dell'albero radicato in root. Se il sottoalbero sinistro e destro di pos hanno la stessa altezza allora la foglia cercata è l'ultima foglia del sottoalbero destro di pos; altrimenti è l'ultima foglia del sottoalbero sinistro di pos. L'assegnamento a pos ristabilisce l'invariante per il ciclo successivo.

Terminazione: il ciclo termina quando pos punta ad una foglia. L'invariante assicura che essa è l'ultima foglia dell'albero radicato in root.

3.
```
// pre: albero non vuoto e quasi completo
// post: ritorna il riferimento all'ultima foglia inserita nell'albero
private BTNode findLastLeaf() {
    BTNode pos = root;
    while (pos.left != null) {
        if (BNode.height(pos.right) ==
            BNode.height(pos.left))
            pos = pos.right;
        else
            pos = pos.left;
    }
    return pos;
}
```

4. **findLastLeaf()**

pos ← root
k ← count
i ← 1
while k >1 do
    A[i] ← k mod 2
    k ← k div 2
    i ← i+1
for j ← i downto 1
    if A[j] = 0
        pos ← pos.left
    else
        pos ← pos.right
return pos

Soluzione Java:

```
// pre: albero non vuoto e quasi completo
// post: ritorna il riferimento all'ultima foglia inserita nell'albero
private BTNode findLastLeaf() {
    BTNode pos = root;

    // la codifica binaria del numero di elementi nell'albero quasi completo
    // serve da guida per trovare il percorso che arriva all'ultima foglia inserita
    String s = Integer.toBinaryString(count);
    int k = s.indexOf("1") + 1;
    while (k < s.length()) {

        if (s.charAt(k) == '0')
            pos = pos.left;
        else
            pos = pos.right;
        k++;
    }
    return pos;
}
```

Esercizio 5 (ASD e Laboratorio)

1. (**ASD**) Relativamente alle visite di alberi generali, scegliere tra le seguenti una affermazione corretta e giustificare la risposta riportando lo pseudocodice dell'algoritmo corrispondente.

(a) La visita in profondità utilizza una struttura dati coda
(b) La visita in ampiezza utilizza una struttura dati coda
(c) La visita in profondità utilizza una struttura dati pila
(d) La visita in ampiezza utilizza una struttura dati pila

2. (**Laboratorio**) Scrivere l'implementazione Java della struttura dati scelta (solo le operazioni di inserimento e rimozione).

Soluzione

1. Ci sono diverse soluzioni possibili dato che la visita in profondità iterativa utilizza una struttura dati pila mentre la visita in ampiezza utilizza una struttura dati coda. Supponendo di avere inizializzato le strutture S (pila) e Q (coda) e di effettuare una chiamata esterna con x uguale alla radice dell'albero, i due algoritmi sono:

```
visita-iter-DFS(x)
    push(x,S)
    while (not empty-stack(S))
       do x <- top(S)
          pop(S)
          if not(empty-tree(x))
             then "visita x"
                  push(sibling[x],S)
                  push(child[x],S)

visita-BFS(x)
    enqueue(x,Q)
    while (not empty-queue(Q))
       do x <- head(Q)
          dequeue(Q)
          while not(empty-tree(x))
             do "visita x"
                 if not(empty-tree(child[x]))
                     then enqueue(child[x],Q)
                 x <- sibling[x]
```

2. Si vedano ad esempio le classi StackArray.java e QueueArray.java realizzate durante il corso.

Esercizio 6 (Laboratorio)

Sia s una stringa contenente solamente parentesi tonde aperte e chiuse. Diciamo che s è ben formata se rispetta le regole sulle parentesi, ovvero che 1) ogni parentesi aperta deve essere seguita (anche non immediatamente) da una parentesi chiusa e che 2) ogni parentesi chiusa deve corrispondere ad una parentesi precedentemente aperta.
Ad esempio le stringhe (()(()())) e ()(()) sono ben formate, mentre le stringhe)(e (()())) non lo sono.
Data una lista semplice L di tipo *SLList* in cui ciascun record contiene un carattere (precisamente un oggetto di tipo *Character*) con valore '(' oppure ')', si richiede di implementare il metodo *BenFormata* della seguente classe:

```
import BasicLists.*;
import Utility.*;
public class Esercizio6 {

   // pre: L diverso da null
   // post: ritorna true se la stringa di parentesi memorizzate in L e' ben formata;
   //       ritorna false altrimenti
   public boolean BenFormata(SLList L) {...}
}
```

Soluzione

```
import BasicLists.*;
import Utility.*;
public class prova {

    public boolean BenFormata(SLList L) {
        Iterator iter = L.iterator();
        int k = 0;
        Character parentesi = new Character('(');
        while (iter.hasNext() && k >= 0) {
            if (((Character)iter.next()).equals(parentesi))
                k++;
            else
                k--;
        }
        return (k==0);
    }
}
```

```
********************** CLASSE BTNode ************************************************************
class BTNode {
    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) { key = ob; parent = left = right = null; }

     // post: ritorna l'altezza del nodo n rispetto all'albero
    //       in cui si trova
    static int height(BTNode n) {...}

    ...
}
********************** CLASSE SLRecord ************************************************************
package BasicLists;
class SLRecord {
    Object key;             // valore memorizzato nell'elemento
    SLRecord next;         // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con valore v, e prossimo elemento nextel
    SLRecord(Object ob, SLRecord nextel) { key = ob; next= nextel; }

    // post: costruisce un nuovo elemento con valore v, e niente next
    SLRecord(Object ob) { this(ob,null); }
}
********************** CLASSE SLList ************************************************************
package BasicLists;
import Utility.Iterator;
public class SLList {
    SLRecord head;         // primo elemento
    int count;                  // num. elementi nella lista

    // post: crea una lista vuota
    public SLList() { head = null; count = 0; }

    // post: ritorna il numero di elementi della lista
    public int size() {...}

    // post: ritorna true sse la lista non ha elementi
    public boolean isEmpty() {...}

    // post: svuota la lista
    public void clear() {...}

    // pre:  ob non nullo
    // post: aggiunge l'oggetto ob in testa alla lista
    //       Ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean insert(Object ob) {...}

    // pre: l'oggetto passato non e' nullo
    // post: ritorna true sse nella lista c'e' un elemento uguale a value
    public boolean contains(Object value) {...}

    // pre: l'oggetto passato non e' nullo
    // post: rimuove l'elemento uguale a value
    //       ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean remove(Object value) {...}

    // post: ritorna un oggetto che scorre gli elementi della lista
    public Iterator iterator() { return new SLListIterator(head);}
 }
********************** CLASSE SLListIterator ************************************************************v**
package BasicLists;
import Utility.Iterator;
public class SLListIterator implements Iterator {
    SLRecord first;              // riferimento al primo elemento
    SLRecord current;            // riferimento all'elemento corrente

    // post: inizializza first e current secondo il parametro passato
    public SLListIterator(SLRecord el) {first = current = el;}

    // post: reset dell'iteratore per ricominciare la visita
    public void reset() {...}

    // post: ritorna true se la lista non e' terminata
    public boolean hasNext() {...}

    // pre: lista non terminata
    // post: ritorna il valore dell'elemento corrente e sposta l'iteratore all'elemento successivo
    public Object next() {...}

    // post: ritorna il valore dell'elemento corrente
    public Object val() {...}
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 9 Febbraio 2006 —

Esercizio 1 (ASD)

Considerata la ricorrenza:

$$T(n) = 8T(\frac{n}{3}) + 2n^2 \lg n$$

si richiede di:

- risolverla utilizzando il teorema principale;
- dire se $T(n) = O(n^3)$, giustificando la risposta.

Esercizio 2 (ASD)

Dire quali tra le seguenti affermazioni sono vere, giustificando la risposta:

1. L'altezza di un albero binario di ricerca con n nodi è nella classe $O(\lg n)$.
2. L'altezza di un albero binario di ricerca con n nodi è nella classe $\Omega(\lg n)$.
3. L'altezza di un albero binario di ricerca con n nodi è nella classe $O(n)$.
4. L'altezza di un albero binario di ricerca con n nodi è nella classe $\Omega(n)$.

Esercizio 3 (ASD)

Si consideri la struttura dati coda di max-priorità realizzata con un array che rappresenta un max-heap. Si aggiunga l'operazione terzomax(Q) che è definita solo se la coda contiene almeno tre elementi e restituisce il valore dell'elemento che ha la terza priorità massima. Scrivere lo pseudocodice di terzomax(Q).

Esercizio 4 (ASD e Laboratorio)

Si vuole realizzare il tipo di dato *insieme ordinato* mediante una lista semplice (singly-linked list). Si consideri quindi la seguente classe *SortedSet* appartenente al package *Sets*:

```
package Sets;
public class SortedSet {
    SetRecord head;        // riferimento alla testa della lista che rappresenta l'insieme
    int count;             // totale elementi nell'insieme

    // post: inserisce un nuovo record con chiave ob nella lista
    //       che rappresenta l'insieme, rispettando l'ordinamento
    //       crescente delle chiavi. Nel caso ob sia gia' presente
    //       nell'insieme non effettua l'inserimento e ritorna false.
    //       Altrimenti effettua l'inserimento, aggiorna count e
    //       ritorna true.
    public boolean insert(Comparable ob) { // implementare!}

    // pre: insieme non vuoto
    // post: cancella il record con chiave ob dalla lista che rappresenta
    //       l'insieme e aggiorna count. Ritorna true se l'operazione e'
```

```
    //       andata a buon fine; ritorna false se ob non e' presente nella
    //       lista
    public boolean delete(Comparable ob) {// non implementare!}
}
```

Si richiede di:

1. scrivere la classe *SetRecord.java* del package *Sets* che memorizza un singolo elemento dell'insieme;
2. implementare in modo efficiente il metodo *insert* della classe *SortedSet.java*, rispettando la post-condizione assegnata;
3. dimostrare la correttezza del metodo *insert*.

Esercizio 5 (Laboratorio)

Implementare nella classe *GenTree.java* del package *Trees* il seguente metodo usando la ricorsione:

```
// pre: k >=0
// post: ritorna true sse tutti i nodi dell'albero (tranne le foglie) hanno ALMENO k figli;
public boolean ktree(int k) {...}
```

L'algoritmo proposto è lineare rispetto al numero di nodi dell'albero? Giustificare la risposta.

Esercizio 6 (ASD)

1. Scrivere un algoritmo ricorsivo che stampa tutte le chiavi presenti in un BST (albero binario di ricerca) senza ripetizioni.
2. **(difficile)** Discutere la correttezza e la complessità dell'algoritmo proposto.

```
***************************** classe TreeNode.java ***************************
package Trees;
class TreeNode {

    Object key;       // valore associato al nodo
    TreeNode parent;    // padre del nodo
    TreeNode child;      // figlio sinistro del nodo
    TreeNode sibling;     // fratello destro del nodo


    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }


    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob,
        TreeNode parent,
        TreeNode child,
        TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }

}
***************************** classe GenTree.java ***************************
package Trees;
import java.util.Iterator;
public class GenTree implements Tree{
    private TreeNode root;      // radice dell'albero
    private int count;           // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
    ....
    ....
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 10 Gennaio 2005 —

Esercizio 1 (ASD)

1. Sia $T(n) = T(n/4) + T(3/4n) + O(n)$. Supponendo $T(1) = 1$, dire, quale delle seguenti risposte è quella esatta. Giustificare la risposta.

(a) $T(n) = O(\lg n)$

(b) $T(n) = O(n)$

(c) $T(n) = O(n \lg n)$

(d) Nessuna delle precedenti risposte è esatta.

2. Qual è la complessità dell'algoritmo di ricerca binaria, in funzione del numero di elementi n? Dire quale delle seguenti risposte è quella esatta. Giustificare la risposta.

(a) O(n) nel caso peggiore

(b) O(log n) nel caso peggiore

(c) O(log log n) nel caso medio

(d) O(log n) nel caso medio ed O(n) nel caso peggiore

Traccia di Soluzione

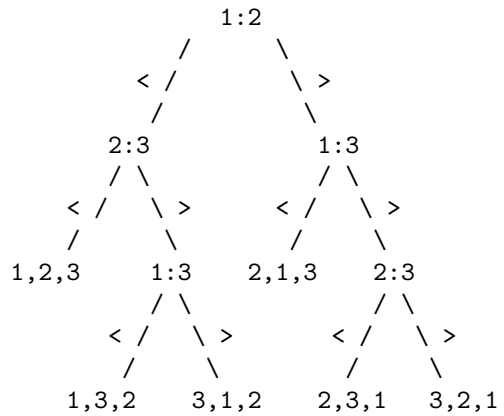
1. La risposta corretta è la (c). Si può dimostrare sia tramite l'albero di ricorsione sia con il metodo di sostituzione. E' forse piu' semplice sviluppare l'albero di ricorsione che ha somma c n su tutti i livelli pieni ed altezza $O(\lg n)$. Con il metodo di sostituzione si puo' procedere come segue. Assumiamo $T(n) \leq dn \lg n$

$$
\begin{aligned}
T(n) &= T(\tfrac{1}{4}n) + T(\tfrac{3}{4}n) + O(n) \\
&\leq T(\tfrac{1}{4}n) + T(\tfrac{3}{4}n) + cn && \text{per definizione di } O(n), \text{ con } c > 0 \\
&\leq \tfrac{1}{4}dn \lg \tfrac{1}{4}n + \tfrac{3}{4}dn \lg \tfrac{3}{4}n + cn && \text{per ipotesi induttiva} \\
&\leq \tfrac{1}{4}dn \lg \tfrac{1}{4}n + \tfrac{3}{4}dn \lg n + cn && \text{infatti } \lg \tfrac{3}{4}n \leq \lg n \\
&= \tfrac{1}{4}dn(\lg n - 2) + \tfrac{3}{4}dn \lg n + cn \\
&= dn \lg n - (\tfrac{1}{2}d - c)n \\
&\leq dn \lg n && \text{se } d \geq 2c
\end{aligned}
$$

2. La risposta corretta è la (b).

Esercizio 2 (ASD)

Si consideri il seguente albero di decisione. Quale algoritmo di ordinamento rappresenta?



(a) Selection Sort

(b) Mergesort

(c) Insertion Sort

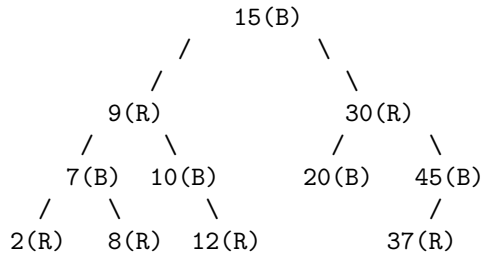
(d) Non rappresenta alcun algoritmo di ordinamento

Traccia di Soluzione

La risposta corretta è la (c).

Esercizio 3 (ASD)

1. In quanto tempo è possibile trovare il minimo in un albero binario di ricerca bilanciato di n elementi? Giustificare la risposta.
2. Dato il seguente albero R/B

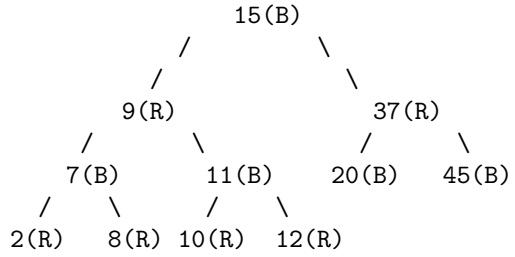


si consideri l'inserimento della chiave 11 seguito dalla cancellazione della chiave 30 e si disegni l'albero risultante.

Traccia di Soluzione

1. La risposta corretta è $O(\lg n)$.

2. La parte piu' complessa è certamente l'inserimento della chiave 11 che richiede due rotazioni. L'albero finale è:



Esercizio 4 (ASD)

Scrivete lo pseudocodice di un algoritmo che rimuove l'elemento massimo in un max-heap memorizzato in un array A. Valutare la complessità dell'algoritmo descritto.

Traccia di Soluzione

Si tratta di realizzare l'algoritmo che elimina il massimo da un max-heap. Vedi testo.

Esercizio 5 (ASD e Laboratorio)

Si consideri il seguente metodo della classe *SLList*, che ritorna l'i-esimo elemento della lista:

```
// pre: indice i valido (cioe' 1 <= i <= size())
// post: ritorna il valore (campo key) dell'elemento i-esimo della lista;
public Object getAtIndex(int i) {...}
```

Si richiede di:

1. scrivere lo pseudocodice dell'algoritmo (ASD e Laboratorio)
2. provare la correttezza dell'algoritmo (ASD e Laboratorio)
3. determinare la complessità dell'algoritmo nel caso pessimo, giustificando la risposta (ASD e Laboratorio)
4. scrivere l'implementazione Java del metodo (Laboratorio)

Traccia di Soluzione

1. **getAtIndex(i)**
   index ← head
   for j ← 1 to i-1
   do index ← next[index]
   return key[index]

2. Dobbiamo dimostrare che l'algoritmo ritorna il valore (cioè il contenuto del campo key) dell'i-esimo elemento della lista. L'invariante del ciclo for è il seguente:

   INV = index dista (i-j) elementi dall'i-esimo della lista

   Inizializzazione: all'inizio j=1 e index è già posizionato sul primo elemento della lista. Quindi mancano i-1 elementi per arrivare all'i-esimo: l'invariante è vero.

   Mantenimento: supponiamo che l'invariante sia vero per j fissato. Allora index dista (i-j) elementi dall'i-esimo della lista. Il corpo del ciclo sposta index al successivo elemento. Quindi, index dista (i-(j+1)) elementi dall'i-esimo della lista, cioè l'invariante viene mantenuto per il ciclo successivo.

   Terminazione: all'uscita dal ciclo j=i e quindi index dista 0 elementi dall'i-esimo della lista.

   L'algoritmo ritorna correttamente key[index], cioè il valore dell'i-esimo elemento della lista.

3. Sia n il numero degli elementi contenuti nella lista. Nel caso pessimo i=n e index attraversa tutti gli elementi della lista. In tal caso il ciclo viene ripetuto n-1 volte e quindi la complessità nel caso pessimo è $\Theta(n)$.

4.
```
// pre: indice i valido (cioe' 1 <= i <= size())
// post: ritorna il valore (campo key) dell'elemento i-esimo della lista;
public Object getAtIndex(int i) {
    SLRecord index = head;
    for (int j = 1; j<i; j++)
        index = index.next;
    return index.key;
}
```

Esercizio 6 (Laboratorio)

Si vogliono implementare due stack utilizzando un solo array. Implementare i metodi **isEmpty1**, **isFull2**, **push1**, **pop2** e il costruttore della classe *TwoStacksOneArray*. I metodi devono avere complessità costante.

```
public class TwoStacksOneArray {
    private static final int MAXSIZE = 100;  // dimensione massima dell'array
    private Object[] A;  // array che rappresenta i due stack
    /* ... dichiarare qui eventuali altri campi della classe ... */

    // post: inizializza i due stack vuoti
    public TwoStacksOneArray() {...}

    // post: ritorna true sse il primo stack e' vuoto
    public boolean isEmpty1() {...}

    // post: ritorna true sse il secondo stack e' vuoto
    public boolean isEmpty2() {...}

    // post: ritorna true sse il primo stack e' pieno
    public boolean isFull1() {...}

    // post: ritorna true sse il secondo stack e' pieno
    public boolean isFull2() {...}

    // post: se il primo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push1(Object ob) {...}

    // post: se il secondo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push2(Object ob) {...}

    // pre: primo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al primo stack
    public Object pop1() {...}

    // pre: secondo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al secondo stack
    public Object pop2() {...}
}
```

Traccia di Soluzione

```
public class TwoStacksOneArray {
    private static final int MAXSIZE = 100;  // dimensione massima dell'array
    private Object[] A;  // array che rappresenta i due stack
    int top1;    // top del primo stack
    int top2;    // top del secondo stack

    // post: inizializza i due stack vuoti
    public TwoStacksOneArray() {
        A = new Object[MAXSIZE];
        top1 = -1;
        top2 = MAXSIZE;
    }

    // post: ritorna true sse il primo stack e' vuoto
    public boolean isEmpty1() {
        return (top1 == -1);
    }

    // post: ritorna true sse il secondo stack e' vuoto
    public boolean isEmpty2() {
        return (top2 == MAXSIZE);
    }
```

```
    // post: ritorna true sse il primo stack e' pieno
    public boolean isFull1() {
        return (top1 + 1 == top2);
    }

    // post: ritorna true sse il secondo stack e' pieno
    public boolean isFull2() {
         return (top2 - 1 == top1);
    }

    // post: se il primo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push1(Object ob) {
         if (!isFull1())
            A[++top1] = ob;
    }

    // post: se il secondo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push2(Object ob) {
        if (!isFull2())
           A[--top2] = ob;
    }

    // pre: primo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al primo stack
    public Object pop1() {
        return A[top1--];
    }

    // pre: secondo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al secondo stack
    public Object pop2() {
       return A[top2++];
    }
}
```

Esercizio 7 (Laboratorio)

Si consideri il package *Trees* visto durante il corso e relativo agli alberi generali. Si vuole aggiungere alla classe *GenTree* il seguente metodo, che ritorna una stringa contenente tutti i nodi dell'albero di livello k:

```
    // post: ritorna una stringa contenente tutti i nodi dell'albero di livello k
    public String levelKnodes(int k) {
        StringBuffer sb = new StringBuffer();
        sb.append("elenco nodi di livello " + k + ": ");
        if (root != null)
            getlevelKnodes(root,sb,k);

        return sb.toString();
    }
```

Si richiede di completare il metodo aggiungendo l'implementazione del metodo ricorsivo:

```
    // pre: parametri diversi da null
    // post: memorizza in sb i nodi del livello richiesto
    private void getlevelKnodes(TreeNode n, StringBuffer sb, int k) {...}
```

Si osservi che non ci sono precondizioni relative al livello k ricevuto in input. Quindi il metodo deve gestire anche i casi di k non valido (es. k minore di zero o maggiore dell'altezza dell'albero).

Traccia di Soluzione

```
    // pre: parametri diversi da null
```

```
// post: memorizza in sb i nodi del livello richiesto
private void getlevelKnodes(TreeNode n, StringBuffer sb, int k) {

    if (k == 0)
       sb.append(n.key.toString() + " ");

    if (k > 0 && n.child != null)
       getlevelKnodes(n.child, sb, k-1);

    if (k >= 0 && n.sibling != null)
       getlevelKnodes(n.sibling, sb, k);
}
```

```
********************** CLASSE SLRecord ****************************************************************
package BasicLists;
class SLRecord {
    Object key;             // valore memorizzato nell'elemento
    SLRecord next;         // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con valore v, e prossimo elemento nextel
    SLRecord(Object ob, SLRecord nextel) { key = ob; next= nextel; }

    // post: costruisce un nuovo elemento con valore v, e niente next
    SLRecord(Object ob) { this(ob,null); }
}
********************** CLASSE SLList ******************************************************************
package BasicLists;
import Utility.Iterator;
public class SLList {
    SLRecord head;         // primo elemento
    int count;                  // num. elementi nella lista

    // post: crea una lista vuota
    public SLList() { head = null; count = 0; }
 ...
 }
********************** CLASSE TreeNode ****************************************************************
package Trees;
class TreeNode {

    Object key;       // valore associato al nodo
    TreeNode parent;    // padre del nodo
    TreeNode child;      // figlio sinistro del nodo
    TreeNode sibling;     // fratello destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi sinistro e destro vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }

    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob, TreeNode parent, TreeNode child, TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }
}
********************** CLASSE GenTree *****************************************************************
package Trees;
import Utility.*;
public class GenTree implements Tree{
    private TreeNode root;      // radice dell'albero
    private int count;           // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
...
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 16 Gennaio 2006 —

Esercizio 1 (ASD)

Si consideri la ricorrenza:

$$T(n) = T(\frac{n}{3}) + T(\frac{n}{6}) + 2n$$

e si dimostri la verità o falsità delle seguenti affermazioni, utilizzando il metodo di sostituzione o le proprietà delle classi di complessità.

1. $T(n) = \Theta(n)$
2. $T(n) = O(n \lg n)$
3. $T(n) = \Theta(n \lg n)$

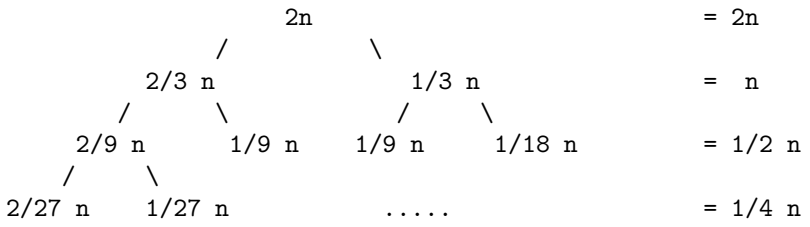
Si disegni inoltre l'albero di ricorsione relativo alla ricorrenza data.

Soluzione

E' facile dimostrare che $T(n) = \Theta(n)$. Dobbiamo dimostrare che esistono c_1, c_2, n_0 positive tali che $c_1 n \leq T(n) \leq c_2 n$, per ogni $n \geq n_0$. Procedendo con il metodo di sotituzione, verifichiamo che esiste $n_0 \geq 0$ e

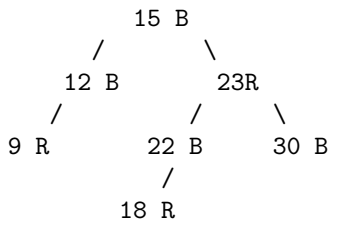
- (i) esiste $c_1 > 0$ tale che $c_1 n \leq c_1 \frac{n}{3} + c_1 \frac{n}{6} + 2n$, per ogni $n \geq n_0$.
  E' sufficiente scegliere $c_1 = 1$ per ottenere: $n \leq \frac{n}{2} + 2n$, per ogni $n \geq 0$.
- (ii) esiste $c_2 > 0$ tale che $c_2 \frac{n}{3} + c_2 \frac{n}{6} + 2n \leq c_2 n$, per ogni $n \geq n_0$. E' sufficiente scegliere $c_2 = 6$ per ottenere: $6\frac{n}{3} + 6\frac{n}{6} + 2n = 5n \leq 6n$, per ogni $n \geq 0$.

La verità della seconda affermazione segue dalle seguenti considerazioni: (a) $T(n) = \Theta(n)$ implica $T(n) = O(n)$ e (b) poiché $n \leq n \lg n$, per ogni $n > 1$ si ha $n = O(n \lg n)$. Quindi $T(n) = O(n \lg n)$ segue da (a) e (b) per la proprietà transitiva relativa alla classe O.
La terza affermazione è invece falsa perchè $n = o(n \lg n)$, come si dimostra facilmente calcolando il limite.



Esercizio 2 (ASD)

Dire se il seguente albero binario gode della proprietaà R/B. Giustificare la risposta e, in caso di risposta negativa, dire se è possibile ottenere un albero R/B tramite ricolorazione dei nodi.



Soluzione

Si, è un albero R/B. (Chiaramente si suppone che tutte le foglie-NIL lasciate implicite siano nere).

Esercizio 3 (ASD)

Si consideri l'algoritmo di ordinamento Quicksort realizzato con la seguente procedura di partizione di Lomuto (come nel testo):

```
Partition(A,p,r)
  x <- A[r]
  i <- p-1
  for j <- p to r-1
     do if A[j] <= x
           then i <- i+1
                scambia (A[i], A[j])
  scambia (A[i+1], A[r])
  return i+1
```

Si dica quali tra le seguenti affermazioni sono corrette:

(a) L'algoritmo ha un comportamento pessimo quando l'array di partenza è già ordinato;

(b) L'algoritmo ha un comportamento pessimo quando l'array di partenza è ordinato in ordine inverso;

(c) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene solo valori inclusi in un intervallo prefissato;

(d) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene valori positivi e negativi alternati;

(e) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene tutti valori uguali;

Giustificare la risposta.

Soluzione

Sono corrette le affermazioni a,b,e perchè in questi casi l'algoritmo di partizione costruisce (sempre) una partizione che contiene un solo elemento in una delle due parti. Con una tale partizione la ricorrenza che caratterizza il comportamento di Quicksort è $T(n) = T(n-1) + \Theta(n)$ la cui soluzione è $T(n) = \Theta(n^2)$, che sappiamo caratterizzare il comportamento pessimo di Quicksort.

Esercizio 4 (ASD e Laboratorio)

Scrivere un algoritmo ITERATIVO *isMinHeap* che, dato un array A contenente numeri interi, verifica in tempo lineare se A è un min-heap. Si richiede di:

1. scrivere l'algoritmo in pseudo-codice
2. dimostrare la correttezza dell'algoritmo
3. completare l'implementazione della seguente classe *Esercizio4.java*, il cui unico metodo *isMinHeapRic* deve risolvere in modo RICORSIVO lo stesso problema.

```
public class Esercizio4.java {

    // pre: A non nullo
    // post: ritorna true se l'array A e' un min-heap; ritorna false altrimenti
    public static boolean isMinHeapRic(int[] A) {...}
}
```

Se necessario, si possono aggiungere alla classe eventuali metodi privati di supporto.

Soluzione

1. ```
   isMinHeap(A)
         i <- 2
         while (i <= lenght[A]) and (A[i div 2] <= A[i])
             do i <- i+1
         return (i > length(A))
   ```

   Algoritmo alternativo:

   ```
   isMinHeap(A)
      flag <- true
      i <- 1
      while (flag and i <= (length(A) div 2))
         do  if (2*i+1 > length(A))
                then flag <- A[i] <= A[2*i]
                else flag <- A[i] <= A[2*i] and A[i] <= A[2*i +1]
             i <- i+1
      return flag
   ```

2. *Invariante*: minheap($A[1..i-1]$)] ovvero A[1..i-1] è un minheap.
   Inizializzazione: i=2 e A[1] contiene un solo elemento e quindi è un minheap.
   Mantenimento: L'incremento di i avviene solo se la condizione ($A[\text{padre}[i]] \leq A[i]$) è soddisfatta e

   $$[(A[\text{padre}[i]] \leq A[i]) \text{ and } \text{minheap}(A[1..i-1])] \implies \text{minheap}(A[1..i]).$$

   Terminazione: Il ciclo puo' terminare perché

   - $i = \text{lenght}[A] + 1$ e in questo caso l'invariante implica minheap(A[1..lenght[A]])
   - oppure perchè isheap è false ma questo può avvenire solo se $A[\text{padre}[i]] > A[i]$) con $i \leq \text{lenght}[A]$, e in questo caso A non è un minheap.

3. ```
   public class Esercizio4 {

       // pre: A non nullo
       // post: ritorna true se l'array A e' un min-heap; ritorna false altrimenti
       public static boolean isMinHeapRic(int[] A) {
           return verifyMinHeap(A,0);
       }

       private static boolean verifyMinHeap(int[] A, int i) {

           // i punta ad una foglia
           if (i >= (A.length / 2))
               return true;

           // manca il figlio destro
           if (2*(i+1) >= A.length)
               return (A[i] <= A[2*i+1]);
           else // ci sono entrambi i figli
               return ((A[i] <= A[2*i+1]) && (A[i] <= A[2*(i+1)]) &&
                      verifyMinHeap(A, 2*i+1) && verifyMinHeap(A, 2*(i+1)));

       }
   }
   ```

Esercizio 5 (Laboratorio)

Estendere la classe *BinaryTree.java* del package *BinTrees* relativo agli alberi binari, aggiungendo il metodo

```
// post: ritorna una stringa contenente la chiave di tutti i nodi dell'albero che hanno un solo cugino.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConUnCugino() {...}
```

Si richiede di implementare il metodo mediante un algoritmo ricorsivo. Se necessario, si possono aggiungere alla classe eventuali metodi privati di supporto.

Soluzione

Proponiamo due soluzioni: la prima è più lineare e permette di provare semplicemente la correttezza.

PRIMA SOLUZIONE

```
// post: ritorna una stringa contenente la chiave di tutti i nodi
//       dell'albero che hanno un solo cugino.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConUnCugino() {

    StringBuffer sb = new StringBuffer("Nodi con un solo cugino: ");
    visita(root, sb);
    return sb.toString();
}

// post: esegue una visita in preordine
private void visita(BTNode n, StringBuffer sb) {
    if (n != null) {
        cugini(n,sb);
        visita(n.left, sb);
        visita(n.right, sb);
    }
}

// pre: n diverso da null
// post: aggiunge n ad sb sse n ha un solo cugino
private void cugini(BTNode n, StringBuffer sb) {
    BTNode fp; // fratello del padre di n

    if (n.parent != null && n.parent.parent != null) {
       if (n.parent.parent.left == n.parent)
           fp = n.parent.parent.right;
       else
           fp = n.parent.parent.left;

       if ((fp != null) && ((fp.left == null) != (fp.right == null)))
           sb.append(n.key.toString() + " ");
    }
}
```

SECONDA SOLUZIONE

```
// post: ritorna una stringa contenente la chiave di tutti i nodi
//       dell'albero che hanno un solo cugino.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConUnCugino() {
    StringBuffer sb = new StringBuffer("Nodi con un solo cugino: ");
    seleziona(root, sb);
    return sb.toString();
}
```

```
private void seleziona(BTNode n, StringBuffer sb) {

    if (n == null)
        return;
    BTNode l = n.left;
    BTNode r = n.right;

    if (l == null && r == null)
        return;

    if (l != null && r == null) {
        seleziona(l, sb);
        return;
    }

    if (l == null && r != null) {
        seleziona(r, sb);
        return;
    }

    if ((l.left != null) && ((r.left != null) != (r.right != null)))
        sb.append(l.left.key.toString() + " ");

    if ((l.right != null) && ((r.left != null) != (r.right != null)))
        sb.append(l.right.key.toString() + " ");

    if ((r.left != null) && ((l.left != null) != (l.right != null)))
        sb.append(r.left.key.toString() + " ");

    if ((r.right != null) && ((l.left != null) != (l.right != null)))
        sb.append(r.right.key.toString() + " ");

    seleziona(l, sb);
    seleziona(r, sb);
}
```

Esercizio 6 (ASD)

Scrivere un algoritmo che dato un albero binario di ricerca T con chiavi tutte distinte ed un nodo x di T stampa tutte le chiavi di T strettamente minori di key[x].

Soluzione

Utilizzando le funzioni min(x) e succ(x), possiamo scrivere:

```
y <- min(root[T])
while key[y] < key[x]
      do write key[y]
         y <- succ(y)
```

In alternativa possiamo richiamare la seguente procedura ricorsiva con y inizializzato a root[T].

```
minore(y,x)
     if (y == nil) then return
     if (y == x) then stampa-tree(left[x])
                      return
     if (key[x] < key[y])
        then minore(left[y],x)
        else write(key[y])
             stampa-tree(left[y])
             minore(right[y],x)
```

dove stampa-tree(z) stampa tutte le chiavi dell'albero radicato nel nodo z e può essere sviluppata con una semplice visita in profondità tipo:

```
stampa-tree(z)
    if z =/= NIl
       then write(key[z])
            stampa-tree(left[z])
            stampa-tree(right[z])
```

Nota: La soluzione di percorrere l'albero senza tener conto della proprietà BST, non è valutabile positivamente.

```
***************************** classe BTNode.java ***************************
package BinTrees;
class BTNode {

    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo


    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) {
        key = ob;
        parent = left = right = null;
    }


    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTNode(Object ob,
           BTNode left,
           BTNode right,
           BTNode parent) {
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
    ....
    ....
}
***************************** classe BinaryTree.java ***************************
package BinTrees;
import java.util.Iterator;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;      // puntatore al nodo corrente
    private int count;          // numero nodi dell'albero


    // post: crea un albero binario vuoto
    public BinaryTree() {
        root = null;
        cursor = null;
        count = 0;
    }
    ....
    ....
}
```

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

– Appello del 14 Giugno 2005 –

Esercizio 1 (ASD)

1. Dire quale delle seguenti affermazioni è vera giustificando la risposta.
   (a) $\lg n = O(n^2)$
   (b) $n^2 = O(\lg n)$
   (c) $n^2 = O(n \lg n)$
   (d) Nessuna delle precedenti risposte è esatta.
2. Un algoritmo di tipo divide et impera per risolvere un problema di dimensione n lo decompone in 3 sottoproblemi di dimensione $(n/3)$ ciascuno, e ricombina le loro soluzioni con un procedimento la cui complessità asintotica è lineare. Qual è la complessità asintotica dell' algoritmo? Giustificare la risposta.

Esercizio 2 (ASD)

Qual è la complessità dell'algoritmo di cancellazione di una chiave in una coda con priorità di n elementi realizzata con un max-heap binario? Dire quale delle seguenti risposte è esatta. Giustificare la risposta.

(a) O(n) nel caso peggiore

(b) O(log n) nel caso peggiore

(c) O(n log n) nel caso peggiore

(d) O(log n) nel caso medio ed O(n) nel caso peggiore

Esercizio 3 (ASD)

Disegnare un albero R/B che contiene le chiavi: 28,23,15,7,24,2,9,18,37,3,10

Esercizio 4 (ASD)

Si consideri il seguente algoritmo scritto nello pseudo codice:

```
test
    i ← 2
    while (i ≤ n) and (T[i] ≤ T[parent(i)])
        do i ← i+1
    return (i=n+1)
```

e si dimostri, utilizzando la tecnica dell'invariante, che l'algoritmo restituisce **true** se e solo se l'array T contiene un max-heap.

Esercizio 5 (ASD e Laboratorio)

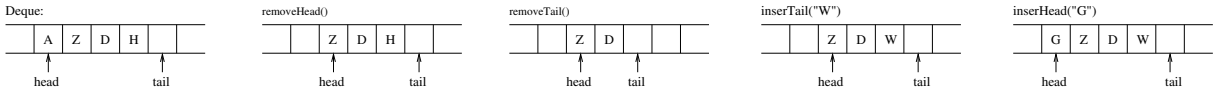
1. **(ASD)** Scrivere lo pseudo codice di un algoritmo che calcola il numero di nodi che hanno piú di un figlio in un albero generale rappresentato utilizzando gli attributi: key, child, sibling.
2. **(LABORATORIO)** Si consideri il package *Trees* visto durante il corso e relativo agli alberi generali. Si vuole aggiungere alla classe *GenTree* il seguente metodo, che ritorna il numero di nodi dell'albero che sono figli unici (cioé non hanno fratelli).

```
// post: ritorna il numero di nodi dell'albero che sono figli unici
//        NB: si assume che la radice, se diversa da null, sia figlio unico
public int contaFigliUnici() {...}
```

Si richiede di completare l'implementazione del metodo usando la ricorsione.

Esercizio 6 (Laboratorio)

La struttura dati *Deque* è simile ad una coda in cui però è possibile inserire e rimuovere dati sia dalla testa (*head*) che dalla coda (*tail*). Nella figura che segue sono riportati alcuni esempi di rimozione e inserimento in una *Deque*.



Si vuole realizzare la struttura dati *Deque* utilizzando un array circolare. Si richiede quindi di contribuire all'implementazione della seguente classe scrivendo i metodi *Deque*, *isEmpty*, *Tail*, *insertHead* e *removeTail*, gestendo l'array D in modo circolare.

```
public class Deque {
    private static final int MAX=100;   // dimensione massima della deque
    private Object[] D;                 // la deque
    private int head;                   // puntatore alla testa
    private int tail;                   // puntatore alla coda
    private int numel;                  // totale elementi nella deque

    // post: costruisce una deque vuota
    public Deque() {...}

    // post: ritorna true sse la deque e' vuota
    public boolean isEmpty() {...}

    // post: ritorna true sse la deque e' piena
    public boolean isFull() {...}

    // pre: deque non vuota
    // post: ritorna il valore dell'elemento puntato da head
    public Object Head() {...}

    // pre: deque non vuota
    // post: ritorna il valore dell'elemento puntato da tail
    public Object Tail() {...}

    // pre: value non nullo
    // post: inserisce in deque (parte tail)
    public void insertTail(Object ob) {...}

    // pre: deque non vuota
    // post: ritorna e rimuove l'elemento puntato da head
    public Object removeHead() {...}

    // pre: value non nullo
    // post: inserisce in deque (parte head)
    public void insertHead(Object ob) {...}

    // pre: deque non vuota
    // post: ritorna e rimuove l'elemento puntato da tail
    public Object removeTail() {...}
}
```

```
********************** CLASSE TreeNode  ********************************************************
package Trees;
class TreeNode {
    Object key;       // valore associato al nodo
    TreeNode parent;    // padre del nodo
    TreeNode child;      // figlio sinistro del nodo
    TreeNode sibling;     // fratello destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi sinistro e destro vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }

    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob, TreeNode parent, TreeNode child, TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }
}
********************** CLASSE GenTree  *********************************************************
package Trees;
import Utility.*;
public class GenTree implements Tree{
    private TreeNode root;      // radice dell'albero
    private int count;           // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
    ...
    ...
 }
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 5 Giugno 2006 —

Esercizio 1 (ASD)

Considerata la ricorrenza:

$$T(n) = 10T(\frac{n}{3}) + 3n^2 + n$$

si richiede di:

- risolverla utilizzando il teorema principale;
- dire se $T(n) = O(n^2)$, giustificando la risposta.

Esercizio 2 (ASD)

Si consideri l'operazione `successore` che dato un nodo x in un albero binario di ricerca T restituisce il nodo di T che segue x in una visita in in-ordine di T. Nell'ipotesi che `successore`$[x]$ sia diverso da `NIL`, dire quali delle seguenti affermazioni sono vere e quali false, giustificando la risposta.

1. `key`$[x] >$ `key`[successore$[x]$].
2. Il nodo `successore`$[x]$ si trova nel sottoalbero destro di x.
3. Il nodo `successore`$[x]$ si trova nel sottoalbero sinistro di x.
4. Esiste un cammino dalla radice di T ad una foglia che passa per entrambi i nodi x e `successore`$[x]$.

Esercizio 3 (ASD)

Si consideri la struttura dati albero generale i cui nodi hanno gli attributi: `key`, `fratello`, `figlio` e `padre` e soddisfano la seguente proprietà speciale:

`key`[`padre`[x]] > `key`[x], per ogni nodo x diverso dalla radice

Si sviluppi un algoritmo in pseudo-codice che dato un nodo x di T lo elimina e restituisce un albero generale che soddisfa ancora la proprietà speciale.

Esercizio 4 (ASD + Laboratorio)

Si vuole realizzare un'implementazione del tipo di dato Dizionario mediante una tabella hash ad indirizzamento aperto, in cui le collisioni sono risolte mediante scansione lineare.
Data la classe:

```
package Esercizio4;
class Coppia {
    Comparable key;          // chiave
    Object elem;             // elemento

    // post: costruisce un nuovo elemento con chiave key e valore ob
    Coppia(Comparable k, Object ob) {
        key = k;
        elem = ob;
    }
}
```

che rappresenta una coppia (chiave, elemento) da memorizzare nel dizionario, si richiede di:

1. completare l'implementazione della seguente classe *DizHashAperto*, ipotizzando che il dizionario non ammetta chiavi duplicate:

```
package Esercizio4;
public class DizHashAperto {
    private static final int DEFSIZE = 113;     // capacita' array
    private Coppia[] diz;                       // tabella hash
    private int count;                          // totale coppie nel dizionario

    //post: costruisce un dizionario vuoto
    public DizHashAperto() { diz = new Coppia[DEFSIZE]; count = 0; }

    // pre: key diverso da null
    // post: ritorna l'indice dell'array associato a key
    private int hash(Comparable key) { return (key.hashCode() % DEFSIZE); }

   // pre: key, ob diversi da null
   // post: aggiunge la coppia (key,ob) nel dizionario. Ritorna true
   //       se l'operazione e' andata a buon fine; false altrimenti
    public boolean insert(Comparable key, Object ob) {...} // COMPLETARE!
}
```

   Si osservi che la funzione hash è implementata dal metodo privato *hash*.

   N.B: è possibile definire eventuali metodi privati di supporto.

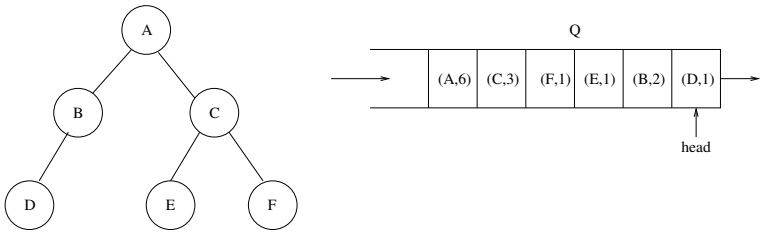
2. discutere gli eventuali problemi nella gestione delle collisioni che il metodo di scansione lineare può comportare.

Esercizio 5 (Laboratorio)

Si richiede di implementare *usando la ricorsione* il seguente metodo per la classe *BinaryTree* del package *BinTrees*, relativo agli alberi binari:

```
// post: ritorna una coda contenente, per ciascun nodo n dell'albero, una stringa che rappresenta
//       la coppia (key(n), num-nodi(n)) dove key(n) e' il valore della chiave di n e num-nodi(n)
//       e' il numero di nodi del sottoalbero radicato in n, compreso n stesso.
//       Se l'albero e' vuoto ritorna null.
public QueueCollegata sottoalberi() {...}
```

Si richiede che la coda restituita dal metodo *sottoalberi*, rappresenti una visita in post-ordine delle chiavi dell'albero. Esempio:



Nel foglio allegato sono descritte tutte le classi utili allo svolgimento dell'esercizio. In particolare, la classe *QueueCollegata* fa parte del package *Queues* visto a lezione.
N.B: è possibile utilizzare eventuali metodi privati di supporto.

Esercizio 6 (ASD)

1. Sia L una lista semplice (con attributi `key` e `next`) con sentinella (`sent`[L]) di numeri interi. Scrivere un algoritmo che trasforma L eliminando tutti gli elementi che contengono una chiave maggiore di quella dell'elemento immediatamente successivo nella lista di partenza. Esempi: la lista 7,5,6,3,2 si trasforma in 5,2 e la lista 3,4,6,1,5 si trasforma in 3,4,1,5.
2. Dimostrare la correttezza dell'algoritmo proposto.

```
***************************** classe BTNode.java ***************************
package BinTrees;
class BTNode {

    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo


    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) {
        key = ob;
        parent = left = right = null;
    }


    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTNode(Object ob,
           BTNode left,
           BTNode right,
           BTNode parent) {
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
    ....
    ....
}
***************************** classe BinaryTree.java ***************************
package BinTrees;
import java.util.Iterator;
import Queues.*;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;      // puntatore al nodo corrente
    private int count;          // numero nodi dell'albero


    // post: crea un albero binario vuoto
    public BinaryTree() {
        root = null;
        cursor = null;
        count = 0;
    }
    ....
    ....
}
***************************** classe QueueCollegata.java ***************************
package Queues;
public class QueueCollegata implements Queue {
    private QueueRecord head;    // puntatore al primo elemento in coda
    private QueueRecord tail;    // puntatore all'ultimo elemento della coda
    private int count;           // numero di elementi in coda

    // post: costruisce una coda vuota
    public QueueCollegata() {
        head = null;
        tail = null;
        count = 0;
    }

    // post: ritorna il numero di elementi nella coda
    public int size() {...}

    // post: ritorna true sse la coda e' vuota
    public boolean isEmpty() {...}

    // post: svuota la coda
    public void clear() {...}

    // pre: coda non vuota
    // post: ritorna il valore del primo elemento della coda
    public Object front() {...}

    // pre: value non nullo
    // post: inserisce value in coda
    public void enqueue(Object ob) {...}

    // pre: coda non vuota
    // post: ritorna e rimuove l'elemento il primo elemento in coda
    public Object dequeue() {...}
}
```

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

– Appello del 12 Luglio 2005 –

Esercizio 1 (ASD)

1. Qual è il tempo di esecuzione di una operazione *insert* in un array ordinato di n elementi nel caso peggiore? Giustificare la risposta.
   (a) $O(\sqrt{n})$
   (b) $O(\log n)$
   (c) $O(n^2)$
   (d) $O(n)$.
2. Scrivere una ricorrenza che può essere risolta con il Master Theorem (caso 1) e trovarne la soluzione.

Esercizio 2 (ASD)

Qual è la complessità dell'algoritmo di cancellazione di una chiave in un albero R/N? Giustificare la risposta.

(a) O(n) nel caso peggiore

(b) O(log n) nel caso peggiore

(c) O(n log n) nel caso peggiore

(d) O(log n) nel caso medio ed O(n) nel caso peggiore

Esercizio 3 (ASD)

1. Disegnare un min-heap binario che contiene le chiavi: 28,23,15,7,24,2,9,18,37,3,10
2. Scrivere poi la sua rappresentazione lineare (array).

Esercizio 4 (ASD)

1. Scrivere un algoritmo (pseudo-codice) per trovare il *secondo minimo* in un array di $n \geq 2$ numeri interi non ordinati. (Il secondo minimo è l'elemento che precede il minimo nell'ordinamento descrescente).
2. Dimostrare la correttezza dell'algoritmo proposto utilizzando la tecnica dell'invariante.

Esercizio 5 (ASD e Laboratorio)

1. **(ASD)** Scrivere lo pseudo codice di un algoritmo che ritorna true se due alberi generali sono *strutturalmente uguali* (sovrapponibili, tranne al più il contenuto dei nodi).

   Per gli alberi generali si consideri la rappresentazione key, child, sibling per ciascun nodo.
2. **(LABORATORIO)** Si consideri il package *BinTrees* visto durante il corso e relativo agli alberi binari. Si vuole aggiungere alla classe *BinaryTree* il seguente metodo, che ritorna true se e solo se i sottoalberi sinistro e destro dell'albero sono uguali (sovrapponibili), sia strutturalmente che nel contenuto dei nodi.

```
// post: ritorna true se i sottoalberi sinistro e destro sono uguali
// sia strutturalmente che nel contenuto dei nodi; ritorna false altrimenti
public boolean uguali() {...}
```

   Si richiede di completare l'implementazione del metodo usando la ricorsione. Se necessario si utilizzi un metodo privato di supporto.

Esercizio 6 (Laboratorio)

Si vuole utilizzare un albero binario di ricerca per memorizzare le parole distinte di un testo. Ciascun nodo dell'albero è quindi relativo ad una parola (stringa) distinta e deve conteggiare il numero delle sue occorrenze nel testo in esame. Si richiede di:

1. Scrivere la classe *Nodo* che rappresenta un nodo dell'albero binario di ricerca;
2. Completare l'implementazione della seguente classe *ContaOccorrenze* scrivendo i metodi *insert* e *stampaOccorrenze*.

```
public class ContaOccorrenze {
    Nodo root;           // radice dell'albero
    int paroledistinte;  // conta le parole distinte del testo
    int paroletesto;     // conta tutte le parole del testo

    // NOTA: si assume che il testo in input sia memorizzato in un array di stringhe
    public ContaOccorrenze(String[] testo) {
        root = null;
        paroledistinte = 0;
        paroletesto = 0;
        if (testo != null)
           memorizzaTesto(testo);
    }

    // post: inserisce il testo nell'albero binario di ricerca
    private void memorizzaTesto(String[] testo) {
        for (int i=0; i < testo.length; i++) {
           insert(testo[i].toUpperCase());
           paroletesto++;
        }
    }

    // post: se parola non esiste nell'albero la inserisce come nuovo nodo;
    //       altrimenti aggiorna il numero di occorrenze del nodo corrispondente.
    //       Aggiorna il numero di parole distinte del testo.
    public void insert(String parola) {...}

    // post: stampa le parole dell'albero e le loro rispettive occorrenze
    //       in ordine crescente di parola
    public void stampaOccorrenze( ) {...}
}
```

Se necessario, definire eventuali metodi privati di supporto.

```
********************** CLASSE BTNode      ********************************************************
package BinTrees;
class BTNode {

    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo


    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) {
        key = ob;
        parent = left = right = null;
    }


    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTNode(Object ob,
           BTNode left,
           BTNode right,
           BTNode parent) {
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
}
********************** CLASSE BinaryTree  ********************************************************
package BinTrees;
import Utility.*;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;      // puntatore al nodo corrente
    private int count;          // numero nodi dell'albero

    // post: crea un albero binario vuoto
    public BinaryTree() {
        root = null;
        cursor = null;
        count = 0;
    }
    ...
}
```

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

Appello dell' 8 Febbraio 2005

Esercizio 1 (ASD)

1. Dire quale delle seguenti affermazioni è vera giustificando la risposta.
   (a) $n \lg n = O(\lg n)$
   (b) $\lg n = \Omega(n)$
   (c) $n \lg n = \Omega(n)$
   (d) Nessuna delle precedenti è vera
2. Un algoritmo di tipo divide et impera per risolvere un problema di dimensione n lo decompone in 4 sottoproblemi di dimensione $(n/2)$ ciascuno, e ricombina le loro soluzioni con un procedimento la cui complessità asintotica è caratterizzata dalla funzione $f(n) = 2n^3 + 3n$. Qual è la complessità asintotica dell' algoritmo? Giustificare la risposta.

Soluzione

1. Risposta esatta: (c) - Si può applicare la definizione della classe $\Omega(n)$: $n \leq n \lg n$ per ogni $n \geq 2$.
   Si può anche calcolare il limite: $\lim_{n\to\infty} \frac{n \lg n}{n} = \infty$ per concludere che $n \lg n = \omega(n)$ e quindi $n \lg n = \Omega(n)$.
2. Si tratta di risolvere la ricorrenza:
   $T(n) = 4T(\frac{n}{2}) + 2n^3 + 3n$. E' facile vedere che la ricorrenza può essere trattata con il Master Theorem. Si ha: $a = 4$, $b = 2$ e $\log_b a = \log_2 4 = 2$ E' quindi facile verificare che $f(n) = \Omega(n^{\log_b a+1}) = \Omega(n^3)$, ad esempio mostrando che $n^3 \leq 3n^3 + n$ per ogni $n \geq 1$. Poichè vale anche (condizione di regolarità): $af(\frac{n}{b}) = 4[2(\frac{n}{2})^3 + 3(\frac{n}{2})] = n^3 + 6n \leq \frac{2}{3}(2n^3 + 3n) = \frac{4}{3}n^3 + 2n$, per ogni $n \geq 4$, si ricade nel caso 3 del Master Theorem. La soluzione è quindi: $T(n) = \Theta(n^3)$.

Esercizio 2 (ASD)

Dire quale delle seguenti affermazioni è esatta relativamente a ciascuna delle seguenti domande.
1) Quanti scambi vengono effettuati dall'algoritmo HeapSort?
2) Quanti confronti vengono effettuati dall'algoritmo QuickSort?

(a) $O(\lg n)$ nel caso medio

(b) $O(n^3)$ nel caso peggiore

(c) $O(n^2)$ nel caso peggiore

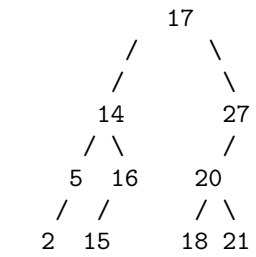
(d) $O(n \lg n)$ nel caso peggiore

Soluzione

1) Risposta esatta: (d)
2) Risposta esatta: (c)

Esercizio 3 (ASD)

Dire quali delle seguenti affermazioni sono applicabili al seguente albero, giustificando la risposta.



(a) L'albero è un max-heap
(b) L'albero è un min-heap
(c) L'albero è un albero binario di ricerca che puo' essere colorato in modo da divenire R/N
(d) L'albero è un albero binario di ricerca non bilanciato

Soluzione

La sola affermazione applicabile è la (d) in quanto è facile vedere che è soddisfatta la condizione BST ma che l'abero non può essere colorato poichè sul cammino 17,27,20,18,NIL non può esserci lo stesso numero di nodi neri del cammino 17,27,NIL senza violare altri punti della proprietà RB. Inoltre è facile verificare che l'albero non può rappresentare né un max-heap (es. $17 < 27$) né un min-heap (es. $17 > 14$).

Esercizio 4 (ASD e Laboratorio)

Si consideri la classe *BTNode* per memorizzare i nodi di un albero binario (riportata alla fine del testo d'esame). Si consideri inoltre la seguente classe *QuasiCompleto* che costruisce alberi quasi completi (ovvero alberi completi in cui mancano zero o più foglie "più a destra" dell'ultimo livello). Gli alberi vengono costruiti per livelli e in modo che siano sempre quasi completi. Quindi ogni nuovo elemento viene inserito dopo la foglia più a destra dell'ultimo livello.

```
public class QuasiCompleto  {
    BTNode root;      // radice dell'albero
    int count;           // numero di elementi dell'albero

    // post: crea un albero vuoto
    public QuasiCompleto() { root = null; count = 0; }
    ...
    ...
    // pre: albero non vuoto e quasi completo
    // post: ritorna il riferimento all'ultima foglia inserita nell'albero
    private BTNode ultimaFoglia() {...}
}
```

Si richiede di implementare il metodo *ultimaFoglia* che ritorna il riferimento all'ultima foglia inserita nell'albero. In particolare, si richiede di:

1. (**ASD e Laboratorio**) scrivere lo pseudocodice di un algoritmo iterativo che risolva il problema
2. (**ASD e Laboratorio**)provare la correttezza dell'algoritmo
3. (**Laboratorio**) scrivere l'implementazione Java del metodo
4. (**EXTRA**) Scrivere lo pseudocodice di un algoritmo che risolva il problema in O(log(n)) (dove n è il numero di nodi dell'albero), sfruttando l'informazione relativa al numero di elementi presenti nell'albero quasi completo e le relazioni padre-figlio tra i nodi.

Soluzione

1. Una soluzione consiste nell'effettuare la visita in ampiezza dell'albero salvando il riferimento di ciascun nodo. L'ultimo nodo visitato sarà proprio la foglia cercata. Un'altra soluzione è la seguente:

ultimaFoglia()

pos ← root
while left[pos] ≠ nil do
    if height(right[pos]) = height(left[pos])
        then pos ← right[pos]
        else pos ← left[pos]
return pos

2. Dobbiamo dimostrare che l'algoritmo ritorna correttamente il riferimento all'ultima foglia inserita nell'albero. L'invariante del ciclo è il seguente:

   INV = l'ultima foglia inserita nel sottoalbero radicato in pos è l'ultima foglia dell'albero radicato in root.

   Infatti:

   Inizializzazione; all'inizio pos punta all'albero radicato in root e quindi l'invariante è vero.

   Mantenimento; supponiamo che l'invariante sia vero per pos fissato ovvero che l'ultima foglia del sottoalbero radicato in pos sia anche l'ultima foglia dell'albero radicato in root. Se il sottoalbero sinistro e destro di pos hanno la stessa altezza allora la foglia cercata è l'ultima foglia del sottoalbero destro di pos; altrimenti è l'ultima foglia del sottoalbero sinistro di pos. L'assegnamento a pos ristabilisce l'invariante per il ciclo successivo.

   Terminazione: il ciclo termina quando pos punta ad una foglia. L'invariante assicura che essa è l'ultima foglia dell'albero radicato in root.

3.
```
// pre: albero non vuoto e quasi completo
// post: ritorna il riferimento all'ultima foglia inserita nell'albero
private BTNode ultimaFoglia() {
    BTNode pos = root;
    while (pos.left != null) {
        if (BNode.height(pos.right) ==
            BNode.height(pos.left))
            pos = pos.right;
        else
            pos = pos.left;
    }
    return pos;
}
```

4. **ultimaFoglia()**

   pos ← root
   k ← count
   i ← 1
   while k >1 do
       A[i] ← k mod 2
       k ← k div 2
       i ← i+1
   for j ← i downto 1
       if A[j] = 0
           pos ← pos.left
       else
           pos ← pos.right
   return pos

   Soluzione Java:

```
// pre: albero non vuoto e quasi completo
// post: ritorna il riferimento all'ultima foglia inserita nell'albero
private BTNode ultimaFoglia() {
    BTNode pos = root;

    // la codifica binaria del numero di elementi nell'albero quasi completo
    // serve da guida per trovare il percorso che arriva all'ultima foglia inserita
    String s = Integer.toBinaryString(count);
    int k = s.indexOf("1") + 1;
```

```
        while (k < s.length()) {

            if (s.charAt(k) == '0')
                pos = pos.left;
            else
                pos = pos.right;
            k++;
        }
        return pos;
    }
```

Esercizio 5 (ASD e Laboratorio)

1. (**ASD**) Relativamente alle visite di alberi generali rappresentati come un insieme di nodi a cui sono associati gli attributi `child` e `sibling`, scegliere tra le seguenti una affermazione corretta e giustificare la risposta riportando lo pseudocodice dell'algoritmo corrispondente.

   (a) La visita in ampiezza utilizza una struttura dati coda
   (b) La visita in profondità utilizza una struttura dati pila
   (c) La visita in profondità utilizza una struttura dati coda
   (d) La visita in ampiezza utilizza una struttura dati pila

2. (**Laboratorio**) Scrivere l'implementazione Java della struttura dati scelta (solo le operazioni di inserimento e rimozione).

Soluzione

1. Ci sono diverse soluzioni possibili dato che la visita in profondità iterativa utilizza una struttura dati pila mentre la visita in ampiezza utilizza una struttura dati coda. Supponendo di avere inizializzato le strutture S (pila) e Q (coda) e di effettuare una chiamata esterna con x uguale alla radice dell'albero, i due algoritmi sono:

```
visita-iter-DFS(x)
    push(x,S)
    while (not empty-stack(S))
       do x <- top(S)
          pop(S)
          if not(empty-tree(x))
             then "visita x"
                  push(sibling[x],S)
                  push(child[x],S)

visita-BFS(x)
    enqueue(x,Q)
    while (not empty-queue(Q))
       do x <- head(Q)
          dequeue(Q)
          while not(empty-tree(x))
             do "visita x"
                 if not(empty-tree(child[x]))
                     then enqueue(child[x],Q)
                 x <- sibling[x]
```

2. Si vedano ad esempio le classi StackArray.java e QueueArray.java realizzate durante il corso.

Esercizio 6 (Laboratorio)

Sia s una stringa contenente solamente parentesi tonde aperte e chiuse. Diciamo che s è ben formata se rispetta le regole sulle parentesi, ovvero che 1) ogni parentesi aperta deve essere seguita (anche non immediatamente) da una

parentesi chiusa e che 2) ogni parentesi chiusa deve corrispondere ad una parentesi precedentemente aperta. Diciamo invece che s è mal formata se non rispetta queste regole.
Ad esempio le stringhe (()(()())) e ()(()) sono ben formate, mentre le stringhe)(e (()())) sono mal formate.
Data una lista semplice L di tipo *SLList* in cui ciascun record contiene un carattere (precisamente un oggetto di tipo *Character*) con valore '(' oppure ')', si richiede di implementare il metodo *MalFormata* della seguente classe:

```
import BasicLists.*;
import Utility.*;
public class Esercizio6 {

   // pre: L diverso da null
   // post: ritorna true se la stringa di parentesi memorizzate in L e' mal formata;
   //       ritorna false altrimenti
   public boolean MalFormata(SLList L) {...}
}
```

Soluzione

```
import BasicLists.*;
import Utility.*;
public class prova {

    public boolean MalFormata(SLList L) {
        Iterator iter = L.iterator();
        int k = 0;
        Character parentesi = new Character('(');
        while (iter.hasNext() && k >= 0) {
            if (((Character)iter.next()).equals(parentesi))
                k++;
            else
                k--;
        }
        return (k!=0);
    }
}
```

```
********************** CLASSE BTNode ************************************************************
class BTNode {
    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) { key = ob; parent = left = right = null; }

     // post: ritorna l'altezza del nodo n rispetto all'albero
    //       in cui si trova
    static int height(BTNode n) {...}

    ...
}
********************** CLASSE SLRecord ************************************************************
package BasicLists;
class SLRecord {
    Object key;             // valore memorizzato nell'elemento
    SLRecord next;         // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con valore v, e prossimo elemento nextel
    SLRecord(Object ob, SLRecord nextel) { key = ob; next= nextel; }

    // post: costruisce un nuovo elemento con valore v, e niente next
    SLRecord(Object ob) { this(ob,null); }
}
********************** CLASSE SLList ************************************************************
package BasicLists;
import Utility.Iterator;
public class SLList {
    SLRecord head;         // primo elemento
    int count;                  // num. elementi nella lista

    // post: crea una lista vuota
    public SLList() { head = null; count = 0; }

    // post: ritorna il numero di elementi della lista
    public int size() {...}

    // post: ritorna true sse la lista non ha elementi
    public boolean isEmpty() {...}

    // post: svuota la lista
    public void clear() {...}

    // pre:  ob non nullo
    // post: aggiunge l'oggetto ob in testa alla lista
    //       Ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean insert(Object ob) {...}

    // pre: l'oggetto passato non e' nullo
    // post: ritorna true sse nella lista c'e' un elemento uguale a value
    public boolean contains(Object value) {...}

    // pre: l'oggetto passato non e' nullo
    // post: rimuove l'elemento uguale a value
    //       ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean remove(Object value) {...}

    // post: ritorna un oggetto che scorre gli elementi della lista
    public Iterator iterator() { return new SLListIterator(head);}
 }
********************** CLASSE SLListIterator ************************************************************v**
package BasicLists;
import Utility.Iterator;
public class SLListIterator implements Iterator {
    SLRecord first;              // riferimento al primo elemento
    SLRecord current;            // riferimento all'elemento corrente

    // post: inizializza first e current secondo il parametro passato
    public SLListIterator(SLRecord el) {first = current = el;}

    // post: reset dell'iteratore per ricominciare la visita
    public void reset() {...}

    // post: ritorna true se la lista non e' terminata
    public boolean hasNext() {...}

    // pre: lista non terminata
    // post: ritorna il valore dell'elemento corrente e sposta l'iteratore all'elemento successivo
    public Object next() {...}

    // post: ritorna il valore dell'elemento corrente
    public Object val() {...}
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

Appello del 9 Febbraio 2006

Esercizio 1 (ASD)

Considerata la ricorrenza:

$$T(n) = 3T(\frac{n}{2}) + 4n^2\sqrt{n}$$

si richiede di:

- risolverla utilizzando il teorema principale;
- dire se $T(n) = O(n^3)$, giustificando la risposta.

Esercizio 2 (ASD)

Dire quali tra le seguenti affermazioni sono vere, giustificando la risposta:

1. L'altezza di un albero binario di ricerca con n nodi è nella classe $\Omega(\lg n)$.
2. L'altezza di un albero binario di ricerca con n nodi è nella classe $O(n)$.
3. L'altezza di un albero binario di ricerca con n nodi è nella classe $\Omega(n)$.
4. L'altezza di un albero binario di ricerca con n nodi è nella classe $O(\lg n)$.

Esercizio 3 (ASD)

Si consideri la struttura dati coda di min-priorità realizzata con un array che rappresenta un min-heap. Si aggiunga l'operazione terzomin(Q) che è definita solo se la coda contiene almeno tre elementi e restituisce il valore dell'elemento che ha la terza priorità minima. Scrivere lo pseudocodice di terzomin(Q).

Esercizio 4 (Laboratorio)

Si vuole realizzare il tipo di dato *insieme ordinato* mediante una lista semplice (singly-linked list). Si consideri quindi la seguente classe *SortedSet* appartenente al package *Sets*:

```
package Sets;
public class SortedSet {
    SetRecord head;        // riferimento alla testa della lista che rappresenta l'insieme
    int count;             // totale elementi nell'insieme

    // post: inserisce un nuovo record con chiave ob nella lista
    //       che rappresenta l'insieme, rispettando l'ordinamento
    //       crescente delle chiavi. Nel caso ob sia gia' presente
    //       nell'insieme non effettua l'inserimento e ritorna false.
    //       Altrimenti effettua l'inserimento, aggiorna count e
    //       ritorna true.
    public boolean insert(Comparable ob) {// non implementare!}

    // pre: insieme non vuoto
    // post: cancella il record con chiave ob dalla lista che rappresenta
    //       l'insieme e aggiorna count. Ritorna true se l'operazione e'
```

```
    //       andata a buon fine; ritorna false se ob non e' presente nella
    //       lista
    public boolean delete(Comparable ob) {// implementare!}
}
```

Si richiede di:

1. scrivere la classe *SetRecord.java* del package *Sets* che memorizza un singolo elemento dell'insieme;
2. implementare in modo efficiente il metodo *delete* della classe *SortedSet.java*, rispettando le pre- e post-condizioni assegnate;
3. dimostrare la correttezza del metodo *delete*

Esercizio 5 (Laboratorio)

Implementare nella classe *GenTree.java* del package *Trees* il seguente metodo usando la ricorsione:

```
// pre: k >= 0
// post: ritorna true sse tutti i nodi dell'albero  hanno AL PIU' k figli;
public boolean ktree(int k) {...}
```

L'algoritmo proposto è lineare rispetto al numero di nodi dell'albero? Giustificare la risposta.

Esercizio 6 (ASD)

1. Scrivere un algoritmo ricorsivo che stampa tutte le chiavi presenti in un BST (albero binario di ricerca) senza ripetizioni.
2. **(difficile)** Discutere la correttezza e la complessità dell'algoritmo proposto.

```
***************************** classe TreeNode.java ***************************
package Trees;
class TreeNode {

    Object key;       // valore associato al nodo
    TreeNode parent;    // padre del nodo
    TreeNode child;      // figlio sinistro del nodo
    TreeNode sibling;     // fratello destro del nodo


    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }


    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob,
        TreeNode parent,
        TreeNode child,
        TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }

}
***************************** classe GenTree.java ***************************
package Trees;
import java.util.Iterator;
public class GenTree implements Tree{
    private TreeNode root;      // radice dell'albero
    private int count;           // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
    ....
    ....
}
```

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

Appello del 10 Gennaio 2005

Esercizio 1 (ASD)

1. Sia $T(n) = T(n/4) + T(n/2) + O(n)$. Supponendo $T(1) = 1$, dire, quale delle seguenti risposte è quella esatta. Giustificare la risposta.

(a) $T(n) = O(\lg n)$

(b) $T(n) = O(n)$

(c) $T(n) = O(n \lg n)$

(d) Nessuna delle precedenti risposte è esatta.

2. Qual è la complessità dell'algoritmo di ricerca sequenziale, in funzione del numero di elementi n? Dire quale delle seguenti risposte è quella esatta. Giustificare la risposta.

(a) O(n) nel caso peggiore

(b) O(log n) nel caso peggiore

(c) O(log n) nel caso medio

(d) O(log n) nel caso medio ed O(n) nel caso peggiore

Soluzione

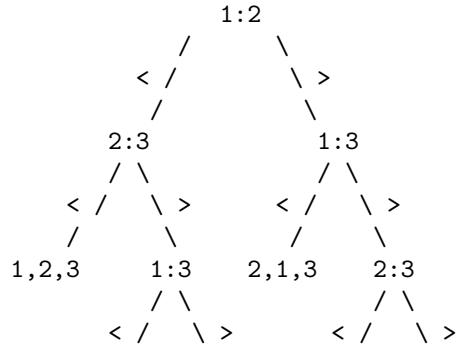
1. La risposta corretta è la (b) (e di conseguenza anche la (c)). Si può dimostrare sia tramite l'albero di ricorsione sia con il metodo di sostituzione come segue. Assumiamo $T(n) \leq dn$

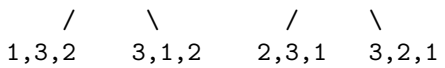
$$
\begin{aligned}
T(n) &= T(\tfrac{1}{4}n) + T(\tfrac{1}{2}n) + O(n) \\
&\leq T(\tfrac{1}{4}n) + T(\tfrac{1}{2}n) + cn && \text{per definizione di } O(n), \text{ con } c > 0 \\
&\leq \tfrac{1}{4}dn + \tfrac{1}{2}dn + cn && \text{per ipotesi induttiva} \\
&= (\tfrac{3}{4}d + c)n \\
&\leq dn && \text{se } d \geq 4c
\end{aligned}
$$

2. La risposta corretta è la (a).

Esercizio 2 (ASD)

Si consideri il seguente albero di decisione. Quale algoritmo di ordinamento rappresenta?





(a) Selection Sort

(b) Mergesort

(c) Insertion Sort

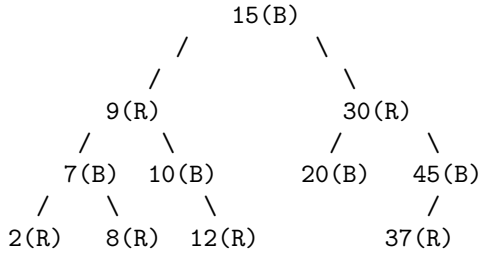
(d) Non rappresenta alcun algoritmo di ordinamento

Soluzione

La risposta corretta è la (c).

Esercizio 3 (ASD)

1. In quanto tempo è possibile trovare una chiave in un albero binario di ricerca bilanciato di n elementi?
2. Dato il seguente albero R/B

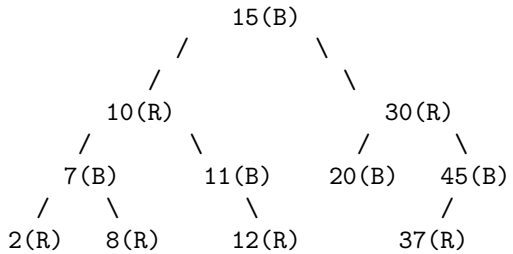


si consideri l'inserimento della chiave 11 seguito dalla cancellazione della chiave 9 e si disegni l'albero risultante.

Soluzione

1. La risposta corretta è $O(\lg n)$.

2.



Esercizio 4 (ASD)

Scrivete lo pseudocodice di un algoritmo che rimuove il secondo elemento più grande (cioe' solo il massimo lo precede nell'ordine decrescente) in un max-heap memorizzato in un array A. Valutare la complessità dell'algoritmo descritto.

Soluzione

Vedi testo.

Esercizio 5 (ASD e Laboratorio)

Si consideri il seguente metodo della classe SLList.java, che modifica il valore dell'i-esimo elemento della lista, secondo il parametro passato

```
// pre: ob diverso da null e indice i valido (cioe' 1 <= i <= size())
// post: imposta l'elemento i-esimo della lista ad "ob"
public void setAtIndex(Object ob, int i)  {...}
```

Si richiede di:

1. scrivere lo pseudocodice dell'algoritmo
2. provare la correttezza dell'algoritmo
3. determinare la complessità dell'algoritmo nel caso pessimo, giustificando la risposta.
4. scrivere l'implementazione Java del metodo

Soluzione

1. **setAtIndex(ob, i)**
   index ← head
   for j ← 1 to i-1
   do index ← next[index]
   key[index] ← ob

2. Dobbiamo dimostrare che l'algoritmo modifica il valore (cioè il campo key) dell'i-esimo elemento della lista. L'invariante del ciclo for è il seguente:

   INV = index dista (i-j) elementi dall'i-esimo della lista

   Inizializzazione: all'inizio j=1 e index è già posizionato sul primo elemento della lista. Quindi mancano i-1 elementi per arrivare all'i-esimo: l'invariante è vero.

   Mantenimento: supponiamo che l'invariante sia vero per j fissato. Allora index dista (i-j) elementi dall'i-esimo della lista. Il corpo del ciclo sposta index al successivo elemento. Quindi, index dista (i-(j+1)) elementi dall'i-esimo della lista, cioè l'invariante viene mantenuto per il ciclo successivo.

   Terminazione: all'uscita dal ciclo j=i e quindi index dista 0 elementi dall'i-esimo della lista.

   L'algoritmo memorizza correttamente ob in key[index], cioè nell'i-esimo elemento della lista.

3. Sia n il numero degli elementi contenuti nella lista. Nel caso pessimo i=n e index attraversa tutti gli elementi della lista. In tal caso il ciclo viene ripetuto n-1 volte e quindi la complessità nel caso pessimo è $\Theta(n)$.

4.
```
// pre: ob diverso da null e indice i valido (cioe' 1 <= i <= size())
// post: imposta l'elemento i-esimo della lista ad "ob"
public void setAtIndex(Object ob, int i) {
    SLRecord index = head;
    for (int j = 1; j<i; j++)
        index = index.next;
    index.key = ob;
}
```

Esercizio 6 (Laboratorio)

Si vogliono implementare due stack utilizzando un solo array. Implementare i metodi **isEmpty2**, **isFull1**, **push2**, **pop1**, e il costruttore della classe TwoStacksOneArray.java. I metodi devono avere complessità costante.

```
public class TwoStacksOneArray {
    private static final int MAXSIZE = 100;  // dimensione massima dell'array
    private Object[] A;  // array che rappresenta i due stack
    /* ... dichiarare qui eventuali altri campi della classe ... */

    // post: inizializza i due stack vuoti
    public TwoStacksOneArray() {...}
```

```
    // post: ritorna true sse il primo stack e' vuoto
    public boolean isEmpty1() {...}

    // post: ritorna true sse il secondo stack e' vuoto
    public boolean isEmpty2() {...}

    // post: ritorna true sse il primo stack e' pieno
    public boolean isFull1() {...}

    // post: ritorna true sse il secondo stack e' pieno
    public boolean isFull2() {...}

    // post: se il primo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push1(Object ob) {...}

    // post: se il secondo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push2(Object ob) {...}

    // pre: primo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al primo stack
    public Object pop1() {...}

    // pre: secondo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al secondo stack
    public Object pop2() {...}
}
```

Soluzione

```
public class TwoStacksOneArray {
    private static final int MAXSIZE = 100;  // dimensione massima dell'array
    private Object[] A;  // array che rappresenta i due stack
    int top1;    // top del primo stack
    int top2;    // top del secondo stack

    // post: inizializza i due stack vuoti
    public TwoStacksOneArray() {
        A = new Object[MAXSIZE];
        top1 = -1;
        top2 = MAXSIZE;
    }

    // post: ritorna true sse il primo stack e' vuoto
    public boolean isEmpty1() {
        return (top1 != -1);
    }

    // post: ritorna true sse il secondo stack e' vuoto
    public boolean isEmpty2() {
        return (top2 != MAXSIZE);
    }

    // post: ritorna true sse il primo stack e' pieno
    public boolean isFull1() {
        return (top1 + 1 == top2);
    }

    // post: ritorna true sse il secondo stack e' pieno
    public boolean isFull2() {
         return (top2 - 1 == top1);
    }
```

```
    // post: se il primo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push1(Object ob) {
         if (!isFull1())
            A[++top1] = ob;
    }

    // post: se il secondo stack non e' pieno, inserisce ob; altrimenti non fa nulla
    public void push2(Object ob) {
        if (!isFull2())
           A[--top2] = ob;
    }

    // pre: primo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al primo stack
    public Object pop1() {
        return A[top1--];
    }

    // pre: secondo stack non vuoto
    // post: rimuove e ritorna l'elemento in cima al secondo stack
    public Object pop2() {
       return A[top2++];
    }
}
```

Esercizio 7 (Laboratorio)

Si consideri il package *Trees* visto durante il corso e relativo agli alberi generali. Si vuole aggiungere alla classe *GenTree* il seguente metodo, che ritorna una stringa contenente tutte le foglie dell'albero di livello k:

```
    // post: ritorna una stringa contenente tutte le foglie dell'albero di livello k
    public String leafK(int k) {
        StringBuffer sb = new StringBuffer();
        sb.append("elenco foglie di livello " + k + ": ");
        if (root != null)
            getleafK(root,sb,k);

        return sb.toString();
    }
```

Si richiede di completare il metodo aggiungendo l'implementazione del metodo ricorsivo:

```
    // pre: parametri diversi da null
    // post: memorizza in sb le foglie del livello richiesto
    private void getleafK(TreeNode n, StringBuffer sb, int k) {...}
```

Si osservi che non ci sono precondizioni relative al livello k ricevuto in input. Quindi il metodo deve gestire anche i casi di k non valido (es. k minore di zero o maggiore dell'altezza dell'albero).

Soluzione

```
    // pre: parametri diversi da null
    // post: memorizza in sb le foglie del livello richiesto
    private void getleafK(TreeNode n, StringBuffer sb, int k) {

        if (k == 0 && n.child == null)
           sb.append(n.key.toString() + " ");

        if (k > 0 && n.child != null)
           getleafK(n.child, sb, k-1);

        if (k >= 0 && n.sibling != null)
           getleafK(n.sibling, sb, k);
    }
```

```
********************** CLASSE SLRecord ************************************************************
package BasicLists;
class SLRecord {
    Object key;             // valore memorizzato nell'elemento
    SLRecord next;         // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con valore v, e prossimo elemento nextel
    SLRecord(Object ob, SLRecord nextel) { key = ob; next= nextel; }

    // post: costruisce un nuovo elemento con valore v, e niente next
    SLRecord(Object ob) { this(ob,null); }
}
********************** CLASSE SLList **************************************************************
package BasicLists;
import Utility.Iterator;
public class SLList {
    SLRecord head;         // primo elemento
    int count;                  // num. elementi nella lista

    // post: crea una lista vuota
    public SLList() { head = null; count = 0; }
 ...
 }
********************** CLASSE TreeNode ************************************************************
package Trees;
class TreeNode {

    Object key;       // valore associato al nodo
    TreeNode parent;    // padre del nodo
    TreeNode child;      // figlio sinistro del nodo
    TreeNode sibling;     // fratello destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi sinistro e destro vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }

    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob, TreeNode parent, TreeNode child, TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }
}
********************** CLASSE GenTree *************************************************************
package Trees;
import Utility.*;
public class GenTree implements Tree{
    private TreeNode root;      // radice dell'albero
    private int count;           // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
...
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

Appello del 16 Gennaio 2006

Esercizio 1 (ASD)

Si consideri la ricorrenza:

$$T(n) = T(\frac{n}{4}) + T(\frac{n}{8}) + 4n$$

e si dimostri la verità o falsità delle seguenti affermazioni, utilizzando il metodo di sostituzione o le proprietà delle classi di complessità.

1. $T(n) = \Theta(n)$
2. $T(n) = O(n \lg n)$
3. $T(n) = \Theta(n \lg n)$

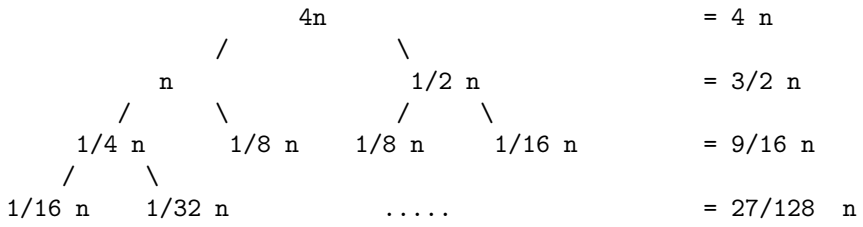
Si disegni inoltre l'albero di ricorsione relativo alla ricorrenza data.

Soluzione

E' facile dimostrare che $T(n) = \Theta(n)$. Dobbiamo dimostrare che esistono c_1, c_2, n_0 positive tali che $c_1 n \leq T(n) \leq c_2 n$, per ogni $n \geq n_0$. Procedendo con il metodo di sotituzione, verifichiamo che esiste $n_0 \geq 0$ e

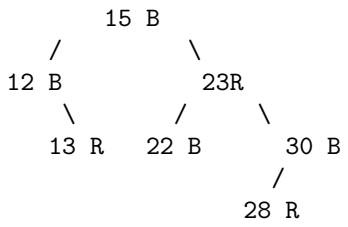
- (i) esiste $c_1 > 0$ tale che $c_1 n \leq c_1 \frac{n}{4} + c_1 \frac{n}{8} + 4n$, per ogni $n \geq n_0$.
  E' sufficiente scegliere $c_1 = 1$ per ottenere: $n \leq \frac{3}{8}n + 2n$, per ogni $n \geq 0$.
- (ii) esiste $c_2 > 0$ tale che $c_2 \frac{n}{4} + c_2 \frac{n}{8} + 4n \leq c_2 n$, per ogni $n \geq n_0$. E' sufficiente scegliere $c_2 = 8$ per ottenere: $8\frac{n}{4} + 8\frac{n}{8} + 4n = 7n \leq 8n$, per ogni $n \geq 0$.

La verità della seconda affermazione segue dalle seguenti considerazioni: (a) $T(n) = \Theta(n)$ implica $T(n) = O(n)$ e (b) poiché $n \leq n \lg n$, per ogni $n > 1$ si ha $n = O(n \lg n)$. Quindi $T(n) = O(n \lg n)$ segue da (a) e (b) per la proprietà transitiva relativa alla classe O.
La terza affermazione è invece falsa perchè $n = o(n \lg n)$, come si dimostra facilmente calcolando il limite.



Esercizio 2 (ASD)

Dire se il seguente albero binario gode della proprietà R/B. Giustificare la risposta e, in caso di risposta negativa, dire se è possibile ottenere un albero R/B tramite ricolorazione dei nodi.



Soluzione

Si, è un albero R/B. (Chiaramente si suppone che tutte le foglie-NIL lasciate implicite siano nere).

Esercizio 3 (ASD)

Si consideri l'algoritmo di ordinamento Quicksort realizzato con la seguente procedura di partizione di Lomuto (come nel testo):

```
Partition(A,p,r)
  x <- A[r]
  i <- p-1
  for j <- p to r-1
     do if A[j] <= x
           then i <- i+1
                scambia (A[i], A[j])
  scambia (A[i+1], A[r])
  return i+1
```

Si dica quali tra le seguenti affermazioni sono corrette:

(a) L'algoritmo ha un comportamento pessimo quando l'array di partenza è già ordinato;

(b) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene valori positivi e negativi alternati;

(c) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene solo valori inclusi in un intervallo prefissato;

(d) L'algoritmo ha un comportamento pessimo quando l'array di partenza contiene tutti valori uguali;

(e) L'algoritmo ha un comportamento pessimo quando l'array di partenza è ordinato in ordine inverso;

Giustificare la risposta.

Soluzione

Sono corrette le affermazioni a,d,e perchè in questi casi l'algoritmo di partizione costruisce (sempre) una partizione che contiene un solo elemento in una delle due parti. Con una tale partizione la ricorrenza che caratterizza il comportamento di Quicksort è $T(n) = T(n-1) + \Theta(n)$ la cui soluzione è $T(n) = \Theta(n^2)$, che sappiamo caratterizzare il comportamento pessimo di Quicksort.

Esercizio 4 (ASD e Laboratorio)

Scrivere un algoritmo ITERATIVO *isMaxHeap* che, dato un array A contenente numeri interi, verifica in tempo lineare se A è un max-heap. Si richiede di:

1. scrivere l'algoritmo in pseudo-codice
2. dimostrare la correttezza dell'algoritmo
3. completare l'implementazione della seguente classe *Esercizio4.java*, il cui unico metodo *isMaxHeapRic* deve risolvere in modo RICORSIVO lo stesso problema.

```
public class Esercizio4.java {

    // pre: A non nullo
    // post: ritorna true se l'array A e' un max-heap; ritorna false altrimenti
    public static boolean isMaxHeapRic(int[] A) {...}
}
```

Se necessario, aggiungere alla classe eventuali metodi privati di supporto.

Soluzione

1. ```
   isMaxHeap(A)
         i <- 2
         while (i <= lenght[A]) and (A[i div 2] >= A[i])
             do i <- i+1
         return (i > length(A))
   ```

   Algoritmo alternativo:

   ```
   isMaxHeap(A)
      flag <- true
      i <- 1
      while (flag and i <= (length(A) div 2))
         do  if (2*i+1 > length(A))
                then flag <- A[i] >= A[2*i]
                else flag <- A[i] >= A[2*i] and A[i] >= A[2*i +1]
             i <- i+1
      return flag
   ```

2. *Invariante*: maxheap($A[1..i-1]$)] ovvero A[1..i-1] è un maxheap.
   Inizializzazione: i=2 e A[1] contiene un solo elemento e quindi è un maxheap.
   Mantenimento: L'incremento di i avviene solo se la condizione (A[padre[i]] $\geq A[i]$) è soddisfatta e

   $$[(A[\text{padre}[i]] \geq A[i]) \text{ and } \text{maxheap}(A[1..i-1])] \implies \text{maxheap}(A[1..i]).$$

   Terminazione: Il ciclo può terminare perché

   - $i = \text{lenght}[A] + 1$ e in questo caso l'invariante implica maxheap(A[1..lenght[A]])
   - oppure perchè isheap è false ma questo può avvenire solo se $A[\text{padre}[i]] < A[i]$) con $i \leq \text{lenght}[A]$, e in questo caso A non è un maxheap.

3. ```
   public class Esercizio4 {

       // pre: A non nullo
       // post: ritorna true se l'array A e' un max-heap; ritorna false altrimenti
       public static boolean isMaxHeapRic(int[] A) {
           return verifyMaxHeap(A,0);
       }

       private static boolean verifyMaxHeap(int[] A, int i) {

           // i punta ad una foglia
           if (i >= (A.length / 2))
               return true;

           // manca il figlio destro
           if (2*(i+1) >= A.length)
               return (A[i] >= A[2*i+1]);
           else // ci sono entrambi i figli
               return ((A[i] >= A[2*i+1]) && (A[i] >= A[2*(i+1)]) &&
                      verifyMaxHeap(A, 2*i+1) && verifyMaxHeap(A, 2*(i+1)));

       }
   }
   ```

Esercizio 5 (Laboratorio)

Estendere la classe *BinaryTree.java* del package *BinTrees* relativo agli alberi binari, aggiungendo il metodo:

```
// post: ritorna una stringa contenente la chiave dei nodi dell'albero che hanno esattamente due cugini.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConDueCugini() {...}
```

Si richiede di implementare il metodo mediante un algoritmo ricorsivo. Se necessario, è possibile aggiungere alla classe eventuali metodi privati di supporto.

Soluzione

Proponiamo due soluzioni: la prima è più lineare e permette di provare semplicemente la correttezza.

PRIMA SOLUZIONE

```
// post: ritorna una stringa contenente la chiave di tutti i nodi
//       dell'albero che hanno esattamente due cugini.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConDueCugini() {
    StringBuffer sb = new StringBuffer("Nodi con due cugini: ");
    visita(root, sb);
    return sb.toString();
}

// post: esegue una visita in preordine
private void visita(BTNode n, StringBuffer sb) {
    if (n != null) {
        cugini(n,sb);
        visita(n.left, sb);
        visita(n.right, sb);
    }
}

// pre: n diverso da null
// post: aggiunge n ad sb sse n ha due cugini
private void cugini(BTNode n, StringBuffer sb) {
    BTNode fp; // fratello del padre di n

    if (n.parent != null && n.parent.parent != null) {
       if (n.parent.parent.left == n.parent)
           fp = n.parent.parent.right;
       else
           fp = n.parent.parent.left;

       if ((fp != null) && (fp.left != null) && (fp.right != null))
           sb.append(n.key.toString() + " ");
    }
}
```

SECONDA SOLUZIONE

```
// post: ritorna una stringa contenente la chiave di tutti i nodi
//       dell'albero che hanno esattamente due cugini.
// NOTA: x e' cugino di y sse i padri di x e y sono fratelli
public String NodiConDueCugini() {
    StringBuffer sb = new StringBuffer("Nodi con due cugini: ");
    seleziona(root, sb);
    return sb.toString();
}

private void seleziona(BTNode n, StringBuffer sb) {
```

```
    if (n == null)
        return;
     BTNode l = n.left;
     BTNode r = n.right;

     if (l == null && r == null)
         return;

     if (l != null && r == null) {
         seleziona(l, sb);
         return;
     }

     if (l == null && r != null) {
         seleziona(r, sb);
         return;
     }

     if ((l.left != null) && (r.left != null) && (r.right != null))
         sb.append(l.left.key.toString() + " ");

     if ((l.right != null) && (r.left != null) && (r.right != null))
         sb.append(l.right.key.toString() + " ");

     if ((r.left != null) && (l.left != null) && (l.right != null))
         sb.append(r.left.key.toString() + " ");

     if ((r.right != null) && (l.left != null) && (l.right != null))
         sb.append(r.right.key.toString() + " ");

     seleziona(l, sb);
     seleziona(r, sb);
}
```

Esercizio 6 (ASD)

Scrivere un algoritmo che dato un albero binario di ricerca T con chiavi tutte distinte ed un nodo x di T stampa tutte le chiavi di T strettamente maggiori di key[x].

Soluzione

Utilizzando le funzioni max(x) e prev(x), possiamo scrivere:

```
y <- max(root[T])
while key[y] > key[x]
      do write key[y]
         y <- prev(y)
```

In alternativa possiamo richiamare la seguente procedura ricorsiva con y inizializzato a root[T].

```
maggiore(y,k)
     if (y == nil) then return
     if (y == x) then stampa-tree(right[x])
                      return
     if (key[x] > key[y])
         then maggiore(right[y],x)
         else write(key[y])
              stampa-tree(right[y])
              maggiore(left[y],x)
```

dove stampa-tree(z) stampa tutte le chiavi dell'albero radicato nel nodo z e può essere sviluppata con una semplice visita in profondità tipo:

```
stampa-tree(z)
    if z =/= NIl
       then write(key[z])
            stampa-tree(left[z])
            stampa-tree(right[z])
```

Nota: La soluzione di percorrere l'albero senza tener conto della proprietà BST, non è valutabile positivamente.

```
***************************** classe BTNode.java ***************************
package BinTrees;
class BTNode {

    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo


    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) {
        key = ob;
        parent = left = right = null;
    }


    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTNode(Object ob,
           BTNode left,
           BTNode right,
           BTNode parent) {
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
    ....
    ....
}
***************************** classe BinaryTree.java ***************************
package BinTrees;
import java.util.Iterator;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;      // puntatore al nodo corrente
    private int count;          // numero nodi dell'albero


    // post: crea un albero binario vuoto
    public BinaryTree() {
        root = null;
        cursor = null;
        count = 0;
    }
    ....
    ....
}
```

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

Appello del 14 Giugno 2005

Esercizio 1 (ASD)

1. Dire quale delle seguenti affermazioni è vera giustificando la risposta.
   (a) $\lg n = \Omega(n^2)$
   (b) $n^2 = \Omega(\lg n)$
   (c) $n^2 = \Omega(n \lg n)$
   (d) Nessuna delle precedenti risposte è esatta.
2. Un algoritmo di tipo divide et impera per risolvere un problema di dimensione n lo decompone in 2 sottoproblemi di dimensione $(n/2)$ ciascuno, e ricombina le loro soluzioni con un procedimento la cui complessità asintotica è quadratica. Qual è la complessità asintotica dell' algoritmo? Giustificare la risposta.

Esercizio 2 (ASD)

Qual è la complessità dell'algoritmo di inserimento di una chiave in una coda con priorità di n elementi realizzata con un max-heap binario? Dire quale delle seguenti risposte è esatta. Giustificare la risposta.

(a) O(n) nel caso peggiore

(b) O(log n) nel caso peggiore

(c) O(n log n) nel caso peggiore

(d) O(log n) nel caso medio ed O(n) nel caso peggiore

Esercizio 3 (ASD)

Disegnare un albero R/B che contiene le chiavi: 6,23,13,11,16,2,4,17,35,3,7

Esercizio 4 (ASD)

Si consideri il seguente algoritmo scritto nello pseudo codice:

test
i $\leftarrow$ 1
while (i < n) and (T[i+1] $\leq$ T[parent(i+1)])
do i $\leftarrow$ i+1
return (i=n)

e si dimostri, utilizzando la tecnica dell'invariante, che l'algoritmo restituisce **true** se e solo se l'array T contiene un max-heap.

Esercizio 5 (ASD e Laboratorio)

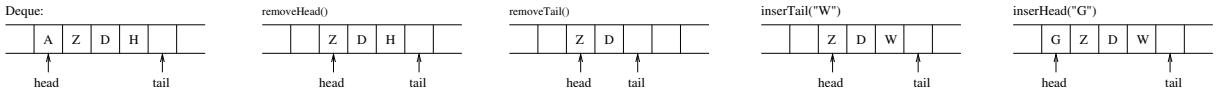
1. **(ASD)** Scrivere lo pseudo codice di un algoritmo che calcola il numero di figli unici in un albero generale rappresentato utilizzando gli attributi: key, child, sibling.
2. **(LABORATORIO)** Si consideri il package *Trees* visto durante il corso e relativo agli alberi generali. Si vuole aggiungere alla classe *GenTree* il seguente metodo, che ritorna il numero di nodi dell'albero che hanno piú di un figlio.

```
// post: ritorna il numero di nodi dell'albero che hanno piu' di un figlio
public int contaNodiConPiuDiUnFiglio() {...}
```

   Si richiede di completare l'implementazione del metodo usando la ricorsione.

Esercizio 6 (Laboratorio)

La struttura dati *Deque* è simile ad una coda in cui però è possibile inserire e rimuovere dati sia dalla testa (*head*) che dalla coda (*tail*). Nella figura che segue sono riportati alcuni esempi di rimozione e inserimento in una *Deque*.



Si vuole realizzare la struttura dati *Deque* utilizzando un array circolare. Si richiede quindi di contribuire all'implementazione della seguente classe scrivendo i metodi *Deque*, *isFull*, *Head*, *insertTail* e *removeHead*, gestendo l'array *D* in modo circolare.

```
public class Deque {
    private static final int MAX=100;   // dimensione massima della deque
    private Object[] D;                 // la deque
    private int head;                   // puntatore alla testa
    private int tail;                   // puntatore alla coda
    private int numel;                  // totale elementi nella deque

    // post: costruisce una deque vuota
    public Deque() {...}

    // post: ritorna true sse la deque e' vuota
    public boolean isEmpty() {...}

    // post: ritorna true sse la deque e' piena
    public boolean isFull() {...}

    // pre: deque non vuota
    // post: ritorna il valore dell'elemento puntato da head
    public Object Head() {...}

    // pre: deque non vuota
    // post: ritorna il valore dell'elemento puntato da tail
    public Object Tail() {...}

    // pre: value non nullo
    // post: inserisce in deque (parte tail)
    public void insertTail(Object ob) {...}

    // pre: deque non vuota
    // post: ritorna e rimuove l'elemento puntato da head
    public Object removeHead() {...}

    // pre: value non nullo
    // post: inserisce in deque (parte head)
    public void insertHead(Object ob) {...}

    // pre: deque non vuota
    // post: ritorna e rimuove l'elemento puntato da tail
    public Object removeTail() {...}
}
```

```
********************** CLASSE TreeNode  ********************************************************
package Trees;
class TreeNode {
    Object key;       // valore associato al nodo
    TreeNode parent;    // padre del nodo
    TreeNode child;      // figlio sinistro del nodo
    TreeNode sibling;     // fratello destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi sinistro e destro vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }

    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob, TreeNode parent, TreeNode child, TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }
}
********************** CLASSE GenTree  *********************************************************
package Trees;
import Utility.*;
public class GenTree implements Tree{
    private TreeNode root;      // radice dell'albero
    private int count;           // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
    ...
    ...
 }
```

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

Appello del 12 Luglio 2005

Esercizio 1 (ASD)

1. Qual è il tempo di esecuzione di una operazione *delete* in un array ordinato di n elementi nel caso peggiore? Giustificare la risposta.

   (a) $O(\sqrt{n})$
   (b) $O(\log n)$
   (c) $O(n^2)$
   (d) $O(n)$.

2. Scrivere una ricorrenza che può essere risolta con il Master Theorem (caso 3) e trovarne la soluzione.

Esercizio 2 (ASD)

Qual è la complessità dell'algoritmo di inserimento di una chiave in un albero R/N? Giustificare la risposta.

(a) O(n) nel caso peggiore

(b) O(log n) nel caso peggiore

(c) O(n log n) nel caso peggiore

(d) O(log n) nel caso medio ed O(n) nel caso peggiore

Esercizio 3 (ASD)

1. Disegnare un min-heap binario che contiene le chiavi: 6,23,13,11,16,2,4,17,35,3,7

2. Scrivere poi la sua rappresentazione lineare (array).

Esercizio 4 (ASD)

1. Scrivere un algoritmo (pseudo-codice) per trovare il *secondo massimo* in un array di $n \geq 2$ numeri interi non ordinati. (Il secondo massimo è l'elemento che segue il massimo nell'ordinamento descrescente).

2. Dimostrare la correttezza dell'algoritmo proposto utilizzando la tecnica dell'invariante.

Esercizio 5 (ASD e Laboratorio)

1. **(ASD)** Scrivere lo pseudo codice di un algoritmo che ritorna true se due alberi generali sono uguali (sovrapponibili), sia strutturalmente che nel contenuto dei nodi.

   Per gli alberi generali si consideri la rappresentazione key, child, sibling per ciascun nodo.

2. **(LABORATORIO)** Si consideri il package *BinTrees* visto durante il corso e relativo agli alberi binari. Si vuole aggiungere alla classe *BinaryTree* il seguente metodo, che ritorna true se e solo se i sottoalberi sinistro e destro dell'albero sono *strutturalmente uguali*, (sovrapponibili, tranne al più il contenuto dei nodi).

```
// post: ritorna true se i sottoalberi sinistro e destro sono strutturalmente uguali,
// (cioe' sovrapponibili, tranne al piu' il contenuto dei nodi)
public boolean structEquals() {...}
```

   Si richiede di completare l'implementazione del metodo usando la ricorsione. Se necessario si utilizzi un metodo privato di supporto.

Esercizio 6 (Laboratorio)

Si vuole utilizzare un albero binario di ricerca per memorizzare i voti distinti di un appello d'esame per successive analisi statistiche sulla distribuzione dei voti. Ciascun nodo dell'albero è quindi relativo ad un voto distinto e deve conteggiare il numero delle sue occorrenze nell'appello d'esame che si sta considerando.
Si richiede di:

1. Scrivere la classe *Voto* che rappresenta un nodo dell'albero binario di ricerca;
2. Completare l'implementazione della seguente classe *ContaVoti* scrivendo i metodi *insert* e *stampaVoti*.

```
public class ContaVoti {
    Voto root;           // radice dell'albero
    int votidistinti;    // conta i voti distinti dell'appello
    int votiappello;     // conta tutti i voti dell'appello

    // NOTA si assume che l'appello d'esame in input sia
    // memorizzato in un array di interi
    public ContaVoti(int[] appello) {
        root = null;
        votidistinti = 0;
        votiappello = 0;
        if (appello != null)
           memorizzaAppello(appello);
    }

    // post: inserisce l'appello nell'albero binario di ricerca
    private void memorizzaAppello(int[] appello) {
        for (int i=0; i < appello.length; i++) {
           insert(appello[i]);
           votiappello++;
        }
    }

    // post: se il voto non esiste nell'albero lo inserisce come nuovo nodo;
    //       altrimenti aggiorna il numero di occorrenze del nodo corrispondente
    public void insert(int voto) {...}

    // post: stampa i voti dell'albero e le loro rispettive occorrenze
    //       in ordine crescente di voto
    public void stampaVoti( ) {...}
}
```

Se necessario, definire eventuali metodi privati di supporto.

```
********************** CLASSE BTNode      ********************************************************
package BinTrees;
class BTNode {

    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo


    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) {
        key = ob;
        parent = left = right = null;
    }


    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTNode(Object ob,
           BTNode left,
           BTNode right,
           BTNode parent) {
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
}
********************** CLASSE BinaryTree  ********************************************************
package BinTrees;
import Utility.*;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;      // puntatore al nodo corrente
    private int count;          // numero nodi dell'albero

    // post: crea un albero binario vuoto
    public BinaryTree() {
        root = null;
        cursor = null;
        count = 0;
    }
    ...
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 21 Febbraio 2007 —

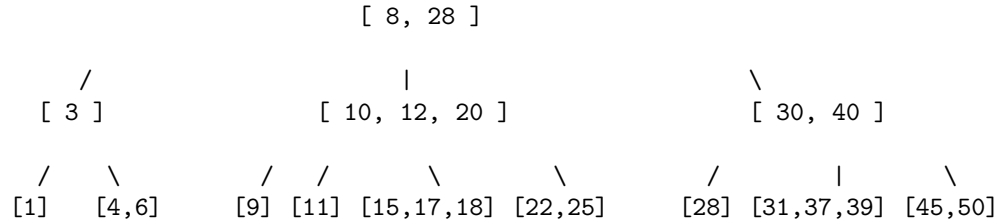
Esercizio 1 (ASD)

Si risolvano le seguenti ricorrenze, giustificando la risposta.

- $T(n) = 5T(\frac{n}{2}) + n^2 + 3n$
- $T(n) = 2T(\frac{n}{3}) + T(\frac{n}{4}) + n$

Esercizio 2 (ASD)

- Quale è la complessità di una operazione di inserimento di una chiave in un B-albero con n chiavi, in funzione di n? Giustificare la risposta.
- Si consideri il seguente albero 2-3-4 e lo si trasformi in un albero R/B.



Esercizio 3 (ASD)

Si consideri la struttura dati albero generale i cui nodi hanno gli attributi: `key`, `fratello`, `figlio` e `padre` e si sviluppi un algoritmo per verificare se è soddisfatta la seguente proprietà speciale:

`key`[`padre`[x]] > `key`[x], per ogni nodo x diverso dalla radice

Esercizio 4 (ASD)

Sia L una lista semplice (con attributi `key` e `next` e sentinella `sent`[L]) di numeri interi e $k > 0$ un numero intero. Sia inoltre new_node(c,x) una funzione che restituisce un nuovo nodo con campo key uguale a c e campo next uguale ad x. Scrivere un algoritmo che trasforma L inserendo dopo ogni k nodi un nuovo nodo che ha come chiave la somma dei k nodi precedenti.
Esempio. Assumiamo k=5. L= (3,2,1,1,5,3,4,1,2,6,1,8) viene trasformata in L'= (3,2,1,1,5,12,3,4,1,2,6,16,1,8).
Dimostrare la correttezza dell'algoritmo.

Esercizio 5 (LAB)

Sia data la seguente interfaccia, dove per tutti i metodi con parametro di tipo `Node` assumiamo la precondizione p != null, e per tutti i metodi che restituiscono un `Node`, utilizziamo il valore null quando il metodo sarebbe indefinito (ad esempio, per `parent()` applicato alla radice, o `root()` applicato ad un albero vuoto).

```
interface BinTree  {
      Node root();               // la radice dell'albero
      Node parent(Node p);       // padre di p nell'albero
      Node left(Node p);         // figlio sinistro di p nell'albero
      Node right(Node p);        // figlio destro di p nell'albero
      boolean internal(Node p);  // true sse p e' un nodo interno
      boolean leaf(Node p);      // true sse p e' una foglia
  int key(Node p);           // la chiave contenuta nel nodo p
}
```

Definite, in Java, l'implementazione del metodo `prec()` descritto dalla seguente specifica.

```
     /**
      * PRE:  T != null e' un BinTree quasi completo, p un Nodo di T
      *
      * POST: restituisce un riferimento al Node che precede p in una visita in
      *       ampiezza di T che attraversi i livelli da sinistra a  destra.
      *       Eccezione se p non ha precedenti in T
      */
      public static Node prec(BinTree T, Node p) throws NoSuchElementException
```

L'implementazione deve garantire una **complessità** $O(log\ n)$, dove n è il numero di nodi dell'albero.

Esercizio 6 (LAB)

Sia data la seguente definizione parziale della classe `List` che realizza una lista semplice.

```
class List {
      /**
       *  NOTAZIONE:
       *   - next^i[head] = il ListItem raggiunto seguendo "i" riferimenti
   *     next a partire da head.
       *   - next^0[head] = head
   *
   *  INVARIANTE DI RAPPRESENTAZIONE
       *  (a) size >= 0
       *  (b) size > 0  => next^i[head] != null (0 <= i <= size-1)
       *                   tail = next^(size-1)[head], next[tail] = null
       *  (c) size = 0 <=> head = tail = null
   *
       *  FUNZIONE DI ASTRAZIONE
       *  this  = []                    se size = 0
       *        = [x_0.... x_{size-1}] dove xi = item[next^i[head]]
   */

      private class ListItem { int item;  ListItem next; }
      private ListItem head, tail;
      private int size;

      public void reverse() {
         // POST: trasforma this invertendone l'ordine degli elemeni, ovvero
     //       se this = [x1,x2,...,xk], this_post = [xk,...,x2,x1]
      }
      // costruttori, altri metodi, ... etc
}
```

Fornite l'implementazione del metodo `reverse()`. L'implementazione deve garantire una **complessità** $\Theta(n)$, dove n è la dimensione della lista, ed utilizzare spazio costante, ovvero modificare la lista sul posto, senza creare una nuova lista.

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

Appello del 24 Gennaio 2007

Esercizio 1 (ASD)

1. Sia $T(n) = T(n/6) + T(n/3) + \Theta(n)$. Considerare ciascuna delle seguenti affermazioni e dire se è corretta o no. Giustificare la risposta.

(a) $T(n) = \Omega(n)$

(b) $T(n) = O(\lg n)$

(c) $T(n) = O(n \lg n)$

Soluzione

(a) $T(n) = \Omega(n)$ è vera. Usando il metodo di sostituzione, possiamo dimostrare che esistono due costanti positive n_0, d tali che $T(n) \geq dn$, per ogni $n > n_0$.

$$\begin{aligned} T(n) &= T(\tfrac{1}{6}n) + T(\tfrac{1}{3}n) + \Theta(n) \\ &\geq T(\tfrac{1}{6}n) + T(\tfrac{1}{3}n) + cn && \text{per definizione di } \Theta(n), \text{ con } c > 0 \\ &\geq \tfrac{1}{6}dn + \tfrac{1}{3}dn + cn && \text{per ipotesi induttiva} \\ &\geq \tfrac{1}{2}dn + cn \\ &\geq dn && \text{se } d \leq 2c \end{aligned}$$

(b) $T(n) = O(\lg n)$ è falsa. Poiché $n = \omega(\lg n)$, per il punto precedente e le proprieta' sulle classi si ottiene $T(n) = \omega(\lg n)$. Poichè per ogni $g(n)$, $\omega(g(n)) \cap O(g(n)) = \emptyset$, si deduce $T(n) \neq O(\lg n)$.

(c) $T(n) = O(n \lg n)$. E' vera. Usando il metodo di sostituzione, è facile dimostrare che esistono due costanti positive n_0, d tali che $T(n) \leq dn \lg n$, per ogni $n > n_0$.

$$\begin{aligned} T(n) &= T(\tfrac{1}{6}n) + T(\tfrac{1}{3}n) + \Theta(n) \\ &\leq T(\tfrac{1}{6}n) + T(\tfrac{1}{3}n) + cn && \text{per definizione di } \Theta(n), \text{ con } c > 0 \\ &\leq d(\tfrac{1}{6}n \lg \tfrac{1}{6}n) + d(\tfrac{1}{3}n \lg \tfrac{1}{3}n) + cn && \text{per ipotesi induttiva} \\ &\leq \tfrac{1}{2}dn \lg n - \lg 6n - \lg 3n + cn && \text{per ipotesi induttiva} \\ &\leq \tfrac{1}{2}dn \lg n + cn \lg n \\ &\leq dn \lg n && \text{se } d \geq 2c \end{aligned}$$

Esercizio 2 (ASD)

Si assuma di disporre dei seguenti costruttori di alberi binari:
- tree_NIL() che restituisce una foglia_NIL,
- tree(x, T1,T2) che data una chiave x e due alberi binari T1 e T2 restituisce un nuovo albero binario con radice x, sottoalbero sinistro T1 e sottoalbero destro T2.

Dato un max-heap memorizzato in un array A[1..n], si sviluppi un algoritmo efficiente per costruire un BST T che contiene tutte le chiavi di A. Si dica quale è la complessità dell'algoritmo e si dimostri la sua correttezza.

Nota: Si osservi che non si richiede che T sia bilanciato.

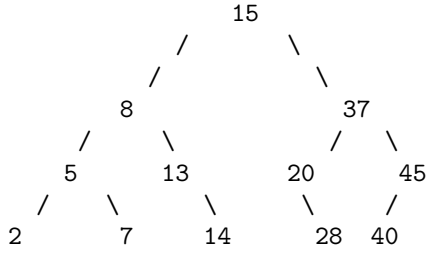
Soluzione

```
trasforma(A,n)
  if n>1
    then
        root <- A[1]
        A[1] <- A[n]
        heapify(A,1,n-1)
        T1 <- trasforma(A,n-1)
        T2 <- tree_NIL()
        return tree(root,T1,T2)
    else return tree_NIL()
```

Si tratta di una diversa formulazione dell'algoritmo heapsort; la complessità si valuta in modo analogo ed è $\Theta(n \lg n)$. Per la correttezza possiamo dimostrare che per ogni $m \geq 0$ se A[1..m] è un max-heap allora `trasforma`(A,m) costruisce un albero BST che è in realta' una catena a sinistra ordinata in ordine decrescente dalla radice alla foglia più a sinistra. La correttezza è ovvia se m=0. Per $m = n > 0$, osserviamo che prima della chiamata ricorsiva A[1..(n-1)] contiene un max-heap e pertanto, per ipotesi induttiva, `trasforma`(A,n-1) costruisce un BST T1 che è una catena a sinistra e contiene tutte le chiavi di A[1..n-1]. Osservando poi che root è l'elemento più grande di A[1..n] otteniamo il risultato.

Esercizio 3 (ASD)

1. In quanto tempo è possibile trovare una chiave in un albero R/B di n elementi? Giustificare la risposta.
2. Dato il seguente albero BST

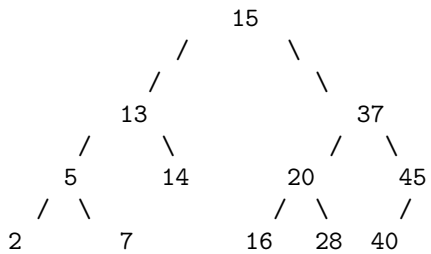


si consideri l'inserimento della chiave 16 seguito dalla cancellazione della chiave 8 e si disegni l'albero risultante.

Soluzione

1. La risposta corretta è $O(\lg n)$, dato che un albero R/B è un BST bilanciato.

2. L'albero finale è:



Esercizio 4 (ASD)

Sia T un albero generale i cui nodi hanno chiavi intere e gli attributi: chiave, figlio, fratello. Scrivere un algoritmo che trasforma T raddoppiando i valori di tutte le chiavi sui livelli dispari dell'albero.

Soluzione

Si tratta di realizzare una visita dell'albero che tiene conto del livello del nodo considerato. La chiamata esterna sarà `trasforma(root[T],false)`, assumendo che la radice si trovi sul livello 0 (pari).

```
trasforma(x,raddoppia)
    if x =/= NIL
       then
        if raddoppia then chiave[x] <- 2*chiave[x]
        trasforma(fratello[x],raddoppia)
        trasforma(figlio[x], not(raddoppia))
```

Per gli esercizi seguenti siano date le seguenti interfacce.

```
public interface BinTree  {

      Node root();               //  la radice del BinTree

      Node parent(Node p);       // padre di p nel BinTree

      Node left(Node p);         // figlio sinistro di p nel BinTree

      Node right(Node p);        // figlio destro di p nel BinTree

      boolean internal(Node p);  // true sse p e‘ un nodo interno nel BinTree

      boolean leaf(Node p);      // true sse p e‘ una foglia nel BinTree

}

public interface Node {

      Comparable key()           // la chiave memorizzata nel nodo

}
```

Esercizio 5 (LAB)

Definite, in Java, l'implementazione del metodo `enumerate()` descritto dalla seguente specifica. L'implementazione deve garantire una complessità asintotica pari a $\Theta(m + h)$, dove m è il numero di chiavi enumerate e h è l'altezza dell'albero.

```
/**
 * POST: restituisce una enumerazione di tutte le chiavi k tali che a <= k <= b nel
 *       sottoalbero di T radicato nel nodo p, dove <= e‘ la relazione di ordine tale
 *       che a <= b sse a.compareTo(b) <= 0
 *
 * PRE:  T != null e‘ un BinTree con nodi di tipo Node, in cui ogni nodo ha zero o
 *       due figli. Le chiavi sono memorizzate nei soli nodi interni di T, e sono
 *       organizzate in modo da soddisfare la BST property.
 *       Il nodo p e‘ un nodo di T, e a <= b.
 */
public static Iterator enumerate(BinTree T, Node p, Comparable a, Comparable b)
```

Soluzione

```
public static Iterator enumerate(BinTree T, Node p, Comparable a, Comparable b) {
if (T.leaf(p))  return new Vector().iterator();

if (((Comparable)p.key()).compareTo(a) < 0)
    return enumerate(T, T.right(p),a,b);
else if (((Comparable)p.key()).compareTo(b) <= 0) {
        Vector els = new Vector();
    Iterator itleft = enumerate(T, T.left(p), a, b);
    Iterator itright = enumerate(T, T.right(p), a, b);
    els.add(p.element());
    while (itleft.hasNext()) els.add(itleft.next());
    while (itright.hasNext()) els.add(itright.next());
    return els.iterator();
}
else
    return enumerate(T, T.left(p),a,b);
}
```

Esercizio 6 (LAB)

Definite, in Java, l'implementazione del metodo `find()` descritto dalla seguente specifica. L'implementazione deve garantire una complessità asintotica pari a $O(\mathtt{n})$.

```
/**
 * POST: restituisce un riferimento al nodo raggiunto all'n-esimo passo di una
 *       visita per livelli di T, in cui i livelli sono attraversati da sinistra
 *       a destra. La visita attraversa i soli nodi interni di T e causa l'eccezione
 *       se T ha meno di n nodi interni.
 *
 *
 * PRE: T != null e' un BinTree con nodi di tipo Node, in cui ogni nodo ha zero o
 *      due figli; n >= 0
 */
 public static Node find(BinTree T, int n) throws NoSuchElementException
```

Soluzione

```
public static Node find (BinTree T, int n) {
Vector Q = new Vector();
Node p = null; int m = 0;

if (T.internal(T.root())) {
      Q.add(T.root());  m = n;
}
while (!Q.isEmpty() && m > 0) {
    p = (Node)Q.elementAt(0);
    Q.removeElementAt(0); m--;

    if (T.internal(T.left(p))) Q.add(T.left(p));
    if (T.internal(T.right(p))) Q.add(T.right(p));
}
if (m > 0) throw new NoSuchElementException();
else return p;
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 19 Giugno 2007 —

Esercizio 1 (ASD)

Si risolvano le seguenti ricorrenze, giustificando la risposta.

- $T(n) = 3T(\frac{n}{4}) + 3n$
- $T(n) = 2T(\frac{n}{4}) + T(\frac{n}{3}) + n$

Esercizio 2 (ASD)

Si disegni un albero R/B che contiene le chiavi: $\{8, 28, 3, 10, 12, 20, 30, 40, 1, 4, 6, 9, 11, 15, 45, 50\}$.

Esercizio 3 (ASD)

Si sviluppi un algoritmo per calcolare il grado massimo dei nodi di un albero generale rappresentato tramite gli attributi: `fratello`, `figlio` e `padre`. (Ricordiamo che il grado di un nodo di un albero è pari al numero dei suoi figli.)

Esercizio 4 (ASD)

Si sviluppi un algoritmo che dato un intero x ed una lista di interi L costruisce una nuova lista L' che contiene tutti gli elementi di L che sono multipli di x. Si definisca la struttura dati utilizzata per rappresentare le liste, si dimostri la correttezza dell'algoritmo sviluppato e se ne valuti la complessità.

[continua sul retro]

Esercizio 5 (LAB)

Sia data la seguente interfaccia per alberi binari con struttura a nodi: ogni nodo hai una chiave e puntatori ai figli destro e sinistro (*senza puntatore al padre*).

```
interface BinTree  {
      Node root();               // la radice dell'albero
      Node left(Node p);         // figlio sinistro di p nell'albero
      Node right(Node p);        // figlio destro di p nell'albero
      int key(Node p);           // la chiave contenuta nel nodo p
}
```

Per tutti i metodi con parametro di tipo `Node` assumiamo la la precondizione `p != null`, mentre tutti i metodi con risultato di tipo `Node`, restituiscono `null` quando il loro valore è indefinito (ad esempio, `root()` su un albero vuoto). Definite, in Java, l'implementazione per il metodo `succ()` descritto dalla seguente specifica.

```
public static Node succ(BinTree T, int k) throws NoSuchElementException
/**
 * PRE:  T != null e' un albero binario di ricerca con chiavi distinte, k e' una chiave contenuta in T.
 *
 * POST: restituisce un riferimento al nodo che contiene la minima tra le chiavi
 * maggiori (il successore) di k in T. Se tale nodo non esiste lancia l'eccezione.
 */
```

L'implementazione deve garantire una complessità $O(h)$, dove h è l'altezza dell'albero.

Esercizio 6 (LAB)

Fornite l'implementazione del metodo `checkSum()` specificato qui di seguito.

```
 public static bool checkSum(int[] A, int[] B, int x)
/**
 * PRE:  A e B sono due array ordinati
 *
 * POST: restituisce true se esistono a in A e b in B tali che a+b = x
 */
```

L' implementazione deve garantire una complessità $O(n \log n)$ dove n è il numero complessivo degli elementi in `A` e `B`.

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 17 Luglio 2007 —

Esercizio 1 (ASD)

1. Si discuta la verità o falsità di ciascuna delle seguenti affermazioni.
   (a) $n \lg n = O(n^2)$
   (b) $\sqrt{n} = O(n \lg n)$
   (c) $n^2 = \Omega(n \lg n)$
   (d) $\sqrt{n} + n^2 = \Theta(n^2 + \lg n)$
2. Si scriva la ricorrenza che descrive la conplessità asintotica di un algoritmo di tipo divide et impera che per risolvere un problema di dimensione n lo decompone in 3 sottoproblemi di dimensione $(n/3)$ ciascuno, e ricombina le loro soluzioni con un procedimento la cui complessità asintotica è quadratica.

Esercizio 2 (ASD)

Si consideri la complessità asintotica dell'algoritmo di cancellazione di una chiave in un albero R/N e si discuta la verità o falsità di ciascuna delle seguenti risposte.

(a) O(n).

(b) Ω(log n).

(c) O(n log n).

(d) Θ(n log n).

Esercizio 3 (ASD)

Si consideri la struttura dati *albero binario di ricerca* (`BST`) con gli attributi `key`, `left`, `right` associati a ciascun nodo. Si scriva lo pseudo codice di un algoritmo che dato un albero binario T memorizza le chiavi di T in un array A[1..n] che rappresenta un *max-heap*. Si discuta la complessità e la correttezza dell'algoritmo proposto.

Esercizio 4 (ASD)

Si consideri la struttura dati *albero generale* con gli attributi `key`, `child`, `sibling` associati a ciascun nodo. Si scriva lo pseudo codice di un algoritmo che applicato ad un albero generale T ritorna `true` se T contiene almeno un figlio unico il cui padre è pure figlio unico. (Nota: la radice viene considerata figlio unico)

[continua sul retro]

Esercizio 5 (LAB)

Considerate la seguente specifica della struttura dati *coda*.

```
class Queue {
    // POST: crea un coda vuota
    public Queue();
     // POST: true sse la coda non ha elementi
    public boolean empty();
    // POST: inserisce l'oggetto in coda
    public void enqueue(Object e);
    // POST: estrae l'oggetto che e' sulla testa della coda
    public Object dequeue() throws EmptyQueueException;
}
```

Implementate il metodo seguente.

```
 public static Queue replica(int n, Queue Q)
  /**
   * PRE:  Q != null, n >= 0
   *
   * POST: restituisce una nuova coda ottenuta replicando ciclicamente gli elementi
   *       di Q fino ad ottenere una coda di lunghezza n. Se la coda non ha
   *       elementi, restituisce la coda vuota. Esempi:
   *        replica(7, [a,b,c]) restituisce la coda [a,b,c,a,b,c,a]
   *        replica(2, [a,b,c]) restituisce la coda [a,b]
   *        replica(5, []) restituisce la coda []
   */
```

Esercizio 6 (LAB)

Considerate la seguente specifica della struttura dati *coda a priorità*.

```
class PQueue {
    // POST: restituisce una coda a priorita' vuota
    public PQueue();
    // POST: true sse la coda non ha elementi
    public boolean empty();
    // POST: inserisce l'oggetto e con chiave k nella coda
    public void insert(int k, Object e);
    // POST: restituisce l'elemento associato alla massima chiave della coda;
    public Object max() throws EmptyQueueException;
    // POST: estrae l'oggetto associato alla chiave massima
    public Object extractMax() throws EmptyQueueException;
}
```

Utilizzate `PQueue` per realizzare la struttura dati *pila* specificata come segue:

```
class Stack {
    // POST: crea uno stack vuoto
    public Stack();
    // POST: true sse lo stack  non ha elementi
    public boolean empty();
    // POST: inserisce l'oggetto e sullo stack
    public void push(object e);
    // POST: restituisce l'ultimo elemento inserito sullo stack
    public Object top() throws EmptyStackException;
    // POST: estrae l'ultimo elemento inserito sullo stack
    public Object pop() throws EmptyStackException;
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 26 Giugno 2006 —

Esercizio 1 (ASD)

Considerata la ricorrenza:

$$T(n) = 2T(\frac{n}{3}) + n \lg n$$

si richiede di:

- risolverla utilizzando il teorema principale;
- dire se $T(n) = O(n^2)$, giustificando la risposta.

Esercizio 2 (ASD)

Sia T un albero R/B. Per ciascuna delle seguenti affermazioni dire se essa è vera o falsa. Giustificare la risposta.

- Per ogni nodo x di T, su ogni cammino da x ad una foglia esiste almeno un nodo nero.
- Per ogni nodo x di T, su ogni cammino da x alla radice esiste almeno un nodo rosso.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia esiste lo stesso numero di nodi neri.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia il numero dei nodi neri è almeno il doppio di quelli rossi.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia il numero dei nodi neri è al più il doppio di quelli rossi.

Esercizio 3 (ASD)

Si consideri la struttura dati max-heap e si sviluppi un algoritmo efficiente (scrivere lo pseudo codice) che dato un max-heap memorizzato in un array A ed un indice i, $1 \leq i \leq heap\_size[A]$, incrementa A[i] di una quantità positiva k e restituisce un nuovo max-heap, ancora memorizzato in A.
Discutere la complessità dell'algoritmo proposto.

Esercizio 4 (ASD + Laboratorio)

Si consideri il package *Dizionario* visto a lezione e, in particolare, la classe *DizBST* che implementa il tipo di dato Dizionario mediante un albero binario di ricerca.

1. **(Laboratorio)** Si vuole aggiungere alla classe *DizBST* il seguente metodo, che verifica se l'albero che rappresenta il dizionario è un albero AVL:

```
// post: ritorna true sse l'albero che rappresenta il dizionario e' un albero AVL
public boolean isAVL() {...}
```

   Si richiede di implementare il metodo *isAVL* usando la ricorsione. La complessità del metodo deve essere $O(n)$, dove n è il numero degli elementi presenti nel dizionario.

   Se necessario, è possibile definire un eventuale metodo privato di supporto.

   NOTA: si ricorda che un albero AVL gode della seguente proprietà di bilanciamento: *Per ogni nodo x, le altezze dei sottoalberi sinistro e destro di x differiscono di al più una unità.*

2. **(ASD)** Scrivere l'algoritmo al punto precedente in pseudo-codice. Dimostrare la correttezza dell'algoritmo ricorsivo proposto.

Esercizio 5 (Laboratorio)

Si vuole realizzare un'implementazione del tipo di dato Dizionario mediante una tabella hash, in cui le collisioni sono risolte mediante liste (semplici) di collisioni. Data la classe:

```
package Esercizio5;
class Nodo {
    Comparable key;          // chiave
    Object elem;             // elemento
    Nodo next;               // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con chiave key e valore ob
    Nodo(Comparable k, Object ob, Nodo nextel) {key = k; elem = ob; next = nextel;}

    // post: costruisce un nuovo elemento con chiave key e valore ob
    Nodo(Comparable k, Object ob) {
        this(k,ob,null);
    }
}
```

che rappresenta una coppia (chiave, elemento) da memorizzare nel dizionario e il riferimento al prossimo elemento della lista di collisione, si richiede di:

1. Completare l'implementazione della seguente classe *DizHashCollisioni*, ipotizzando che il dizionario non ammetta chiavi duplicate e che le liste di collisione siano ordinate rispetto alla chiave:

   ```
   package Esercizio5;
   public class DizHashCollisioni {
       private static final int DEFSIZE = 113;     // capacita' array
       private Nodo[] diz;                         // tabella hash
       private int count;                          // totale coppie nel dizionario

       //post: costruisce un dizionario vuoto
       public DizHashCollisioni() { diz = new Nodo[DEFSIZE]; count = 0; }

       // pre: key diverso da null
       // post: ritorna l'indice dell'array associato a key
       private int hash(Comparable key) { return (key.hashCode() % DEFSIZE); }

      // pre: key, ob diversi da null
      // post: aggiunge la coppia (key,ob) nel dizionario. Ritorna true
      //       se l'operazione e' andata a buon fine; false altrimenti
       public boolean insert(Comparable key, Object ob) {...} // COMPLETARE!
   }
   ```

2. Definire la nozione di *fattore di carico* di una tabella hash. Nel caso in cui le collisioni siano gestite mediante liste di collisioni, cosa succede al fattore di carico della tabella hash?

Esercizio 6 (ASD)

1. Si elenchino tutte le operazioni definite sulla struttura dati albero binario di ricerca (BST), descrivendone le funzionalità.

2. Siano A un albero binario di ricerca contenente n interi e B un array ordinato ($\leq$) contenente m interi.

   (a) Scrivere lo pseudo codice di una procedura efficiente per stampare in ordine non decrescente ($\leq$) l'unione (compresi evenutali duplicati) di tutti gli elementi di A e di B.

   (b) Indicare la complessità della procedura in funzione di n ed m.

```
***************************** classe DizBSTNode.java ***************************
package Dizionario;
class DizBSTNode {

    Comparable key;       // chiave associato al nodo
    Object elem;          // elemento associato alla chiave
    DizBSTNode parent;    // padre del nodo
    DizBSTNode left;      // figlio sinistro del nodo
    DizBSTNode right;     // figlio destro del nodo


    // post: ritorna un albero di un solo nodo, con chiave key, elemento ob
    //       e sottoalberi sinistro e destro vuoti
    DizBSTNode(Comparable key, Object ob) {
        this.key = key;
        elem = ob;
        parent = left = right = null;
    }
}
***************************** classe DizBST.java ***************************
package Dizionario;
public class DizBST implements Dizionario {
    private DizBSTNode root;            // radice dell'albero che rappresenta il dizionario
    private int count;              // numero di nodi dell'albero

    // post: costruisce un albero di ricerca vuoto
    public DizBST() {
        root = null;
        count = 0;
    }
    ....
    ....
}
***************************** interfaccia Dizionario.java **********************
package Dizionario;
public interface Dizionario {

   // post: ritorna il numero di elementi nel dizionario
   public int size();

   // post: ritorna true sse il dizionario e' vuoto
   public boolean isEmpty();

   // post: svuota il dizionario
   public void clear();

   // pre:  key, ob diversi da null
   // post: aggiunge la coppia (key,ob) nel dizionario. Ritorna true
   //       se l'operazione e' andata a buon fine; false altrimenti
   public boolean insert(Comparable key, Object ob);

   // pre:  key diverso da null
   // post: cancella dal dizionario la coppia con chiave key. Ritorna
   //       true se l'operazione e' andata a buon fine; false altrimenti
   public boolean delete(Comparable key);

   // pre:  key diverso da null
   // post: se key e' presente nel dizionario ritorna l'elemento ad essa
   //       associato. Ritorna null altrimenti.
   public Object search(Comparable key);
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 26 Giugno 2006 —

Esercizio 1 (ASD)

Considerata la ricorrenza:

$$T(n) = 2T(\frac{n}{3}) + n \lg n$$

si richiede di:

- risolverla utilizzando il teorema principale;
- dire se $T(n) = O(n^2)$, giustificando la risposta.

Soluzione

- Poichè $n^{\lg_3 2} < 1$, esiste una costante positiva ϵ tale che $n = n^{\lg_3 2+\epsilon}$. Quindi $n \lg n = \Omega(n^{\lg_3 2+\epsilon})$. Siamo nel caso 3 del metodo principale. Verificato che $2n/3 \lg n/3 \leq 2/3n \lg n$, otteniamo $T(n) = \Theta(n \lg n)$.
- Si'. E' facile dimostrare che $n \lg n = O(n^2)$.

Esercizio 2 (ASD)

Sia T un albero R/B. Per ciascuna delle seguenti affermazioni dire se essa è vera o falsa. Giustificare la risposta.

- Per ogni nodo x di T, su ogni cammino da x ad una foglia esiste almeno un nodo nero.
- Per ogni nodo x di T, su ogni cammino da x alla radice esiste almeno un nodo rosso.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia esiste lo stesso numero di nodi neri.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia il numero dei nodi neri è almeno il doppio di quelli rossi.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia il numero dei nodi neri è al più il doppio di quelli rossi.

Soluzione

- Per ogni nodo x di T, su ogni cammino da x ad una foglia esiste almeno un nodo nero.
  VERA, la foglia è sempre nera
- Per ogni nodo x di T, su ogni cammino da x alla radice esiste almeno un nodo rosso.
  FALSA, potrebbero essere tutti neri.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia esiste lo stesso numero di nodi neri.
  VERA, per la proprietà R/B.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia il numero dei nodi neri è almeno il doppio di quelli rossi.
  FALSA, si pensi ad un albero R/B con il cammino N-R-N-R-N. La proprietà vale invece per l'intero albero.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia il numero dei nodi neri è al più il doppio di quelli rossi.
  FALSA, si pensi ad un albero con solo nodi neri.

Esercizio 3 (ASD)

Si consideri la struttura dati max-heap e si sviluppi un algoritmo efficiente (scrivere lo pseudo codice) che dato un max-heap memorizzato in un array A ed un indice i, $1 \leq i \leq heap\_size[A]$, incrementa A[i] di una quantità positiva k e restituisce un nuovo max-heap, ancora memorizzato in A.
Discutere la complessità dell'algoritmo proposto.

Soluzione

Si tratta di realizzare l'operazione increase_key (vedi testo).

Esercizio 4 (ASD + Laboratorio)

Si consideri il package *Dizionario* visto a lezione e, in particolare, la classe *DizBST* che implementa il tipo di dato Dizionario mediante un albero binario di ricerca.

1. **(Laboratorio)** Si vuole aggiungere alla classe *DizBST* il seguente metodo, che verifica se l'albero che rappresenta il dizionario è un albero AVL:

   ```
   // post: ritorna true sse l'albero che rappresenta il dizionario e' un albero AVL
   public boolean isAVL() {...}
   ```

   Si richiede di implementare il metodo *isAVL* usando la ricorsione. La complessità del metodo deve essere $O(n)$, dove n è il numero degli elementi presenti nel dizionario.

   Se necessario, è possibile definire un eventuale metodo privato di supporto.

   NOTA: si ricorda che un albero AVL gode della seguente proprietà di bilanciamento: *Per ogni nodo x, le altezze dei sottoalberi sinistro e destro di x differiscono di al più una unità.*

2. **(ASD)** Scrivere l'algoritmo al punto precedente in pseudo-codice. Dimostrare la correttezza dell'algoritmo ricorsivo proposto.

Soluzione

1.
   ```
   // post: ritorna true sse l'albero che rappresenta il dizionario e' un albero AVL
   public boolean isAVL() {
       return isAVL(root) != -2;
   }

   // post: ritorna un valore diverso da -2 sse l'albero radicato in n e' un albero AVL
   private int isAVL(DizBSTNode n) {
       if (n == null)
           return -1;

       int left = isAVL(n.left);
       int right = isAVL(n.right);

       if (left == -2 || right == -2 || Math.abs(left - right) > 1)
           return -2;
       else
           return 1 + Math.max(left, right);
   }
   ```

2.

Esercizio 5 (Laboratorio)

Si vuole realizzare un'implementazione del tipo di dato Dizionario mediante una tabella hash, in cui le collisioni sono risolte mediante liste (semplici) di collisioni. Data la classe:

```
package Esercizio5;
class Nodo {
    Comparable key;          // chiave
    Object elem;             // elemento
    Nodo next;               // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con chiave key e valore ob
    Nodo(Comparable k, Object ob, Nodo nextel) {key = k; elem = ob; next = nextel;}

    // post: costruisce un nuovo elemento con chiave key e valore ob
    Nodo(Comparable k, Object ob) {
        this(k,ob,null);
    }
}
```

che rappresenta una coppia (chiave, elemento) da memorizzare nel dizionario e il riferimento al prossimo elemento della lista di collisione, si richiede di:

1. Completare l'implementazione della seguente classe *DizHashCollisioni*, ipotizzando che il dizionario non ammetta chiavi duplicate e che le liste di collisione siano ordinate rispetto alla chiave:

```
package Esercizio5;
public class DizHashCollisioni {
    private static final int DEFSIZE = 113;     // capacita' array
    private Nodo[] diz;                         // tabella hash
    private int count;                          // totale coppie nel dizionario

    //post: costruisce un dizionario vuoto
    public DizHashCollisioni() { diz = new Nodo[DEFSIZE]; count = 0; }

    // pre: key diverso da null
    // post: ritorna l'indice dell'array associato a key
    private int hash(Comparable key) { return (key.hashCode() % DEFSIZE); }

   // pre: key, ob diversi da null
   // post: aggiunge la coppia (key,ob) nel dizionario. Ritorna true
   //       se l'operazione e' andata a buon fine; false altrimenti
    public boolean insert(Comparable key, Object ob) {...} // COMPLETARE!
}
```

2. Definire la nozione di *fattore di carico* di una tabella hash. Nel caso in cui le collisioni siano gestite mediante liste di collisioni, cosa succede al fattore di carico della tabella hash?

Soluzione

1.
```
package Esercizio5;
public class DizHashCollisioni {
    private static final int DEFSIZE = 113;     // capacita' array
    private Nodo[] diz;                         // tabella hash
    private int count;                          // totale coppie nel dizionario

    //post: costruisce un dizionario vuoto
    public DizHashCollisioni() { diz = new Nodo[DEFSIZE]; count = 0; }

    // pre: key diverso da null
    // post: ritorna l'indice dell'array associato a key
    private int hash(Comparable key) { return (key.hashCode() % DEFSIZE); }

    // pre: key, ob diversi da null
    // post: aggiunge la coppia (key,ob) nel dizionario. Ritorna true
    //       se l'operazione e' andata a buon fine; false altrimenti
    public boolean insert(Comparable key, Object ob) {
        Nodo temp;
        int pos = hash(key);
```

```
        if (diz[pos] == null)  // inserimento in testa
            diz[pos] = new Nodo(key,ob);
        else { // verifico se la chiave e' gia' presente
            Nodo previous = null;
            Nodo index = diz[pos];
            while (index != null && index.key.compareTo(key) < 0) {
                previous = index;
                index = index.next;
            }
            if (index != null && index.key.equals(key))
                return false; // chiave gia' presente
            else  // inserimento in mezzo o in coda
                previous.next = new Nodo(key, ob, index);
        }
        count++;
        return true;
    }
}
```

2. Il fattore di carico di una tabella hash è definito come il rapporto $\alpha = n/m$ tra il numero n di elementi presenti nella tabella e la dimensione totale m della tabella stessa. Nel caso di gestione delle collisioni tramite liste di collisioni, il fattore di carico della tabella può essere maggiore di uno. Infatti gli elementi che causano collisioni vengono memorizzati in liste concatenate che, per definizione, possono avere dimensione arbitraria. Inoltre, assumendo una distribuzione uniforme delle chiavi, il fattore di carico identifica la lunghezza media delle liste di collisioni.

Esercizio 6 (ASD)

1. Si elenchino tutte le operazioni definite sulla struttura dati albero binario di ricerca (BST), descrivendone le funzionalità.

2. Siano A un albero binario di ricerca contenente n interi e B un array ordinato ($\leq$) contenente m interi.

   (a) Scrivere lo pseudo codice di una procedura efficiente per stampare in ordine non decrescente ($\leq$) l'unione (compresi evenutali duplicati) di tutti gli elementi di A e di B.

   (b) Indicare la complessità della procedura in funzione di n ed m.

Soluzione

2.(a) Si tratta di realizzare un algoritmo di *merge* degli elementi delle due strutture.

```
x <- minimum(A);
i <- 1;
while (x =/= NIL) and (i <= m)
   do if (key[x] <= B[i]
         then stampa key[x];
              x <- tree-successor(x)
         else stampa B[i];
              i <- i+1
while (x =/= NIL)
    do stampa key[x]; x <- tree-successor[x]
while (i <= m)
    do stampa B[i]; i <- i+1
```

2.(b) Si ricorda che la visita di un albero BST tramite la funzione tree-successor richiede un tempo lineare nel numero dei nodi dell'albero. Poichè anche l'array B viene percorso linearmente, la complessità totale è $\Theta(n + m)$.

```
***************************** classe DizBSTNode.java ***************************
package Dizionario;
class DizBSTNode {

    Comparable key;       // chiave associato al nodo
    Object elem;          // elemento associato alla chiave
    DizBSTNode parent;    // padre del nodo
    DizBSTNode left;      // figlio sinistro del nodo
    DizBSTNode right;     // figlio destro del nodo


    // post: ritorna un albero di un solo nodo, con chiave key, elemento ob
    //       e sottoalberi sinistro e destro vuoti
    DizBSTNode(Comparable key, Object ob) {
        this.key = key;
        elem = ob;
        parent = left = right = null;
    }
}
***************************** classe DizBST.java ***************************
package Dizionario;
public class DizBST implements Dizionario {
    private DizBSTNode root;            // radice dell'albero che rappresenta il dizionario
    private int count;              // numero di nodi dell'albero

    // post: costruisce un albero di ricerca vuoto
    public DizBST() {
        root = null;
        count = 0;
    }
    ....
    ....
}
***************************** interfaccia Dizionario.java **********************
package Dizionario;
public interface Dizionario {

   // post: ritorna il numero di elementi nel dizionario
   public int size();

   // post: ritorna true sse il dizionario e' vuoto
   public boolean isEmpty();

   // post: svuota il dizionario
   public void clear();

   // pre:  key, ob diversi da null
   // post: aggiunge la coppia (key,ob) nel dizionario. Ritorna true
   //       se l'operazione e' andata a buon fine; false altrimenti
   public boolean insert(Comparable key, Object ob);

   // pre:  key diverso da null
   // post: cancella dal dizionario la coppia con chiave key. Ritorna
   //       true se l'operazione e' andata a buon fine; false altrimenti
   public boolean delete(Comparable key);

   // pre:  key diverso da null
   // post: se key e' presente nel dizionario ritorna l'elemento ad essa
   //       associato. Ritorna null altrimenti.
   public Object search(Comparable key);
}
```

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

– Appello del 1 Settembre 2005 –

Esercizio 1 (ASD)

Sia $T(n) = 4T(\frac{n}{2}) + O(n^3)$. Dire, quale delle seguenti risposte è quella esatta. Giustificare la risposta.

(a) $T(n) = \Theta(n^2 \lg n)$

(b) $T(n) = \Theta(n^3)$

(c) $T(n) = \Theta(n^2)$

(d) Nessuna delle precedenti risposte è esatta.

Esercizio 2 (ASD)

Qual è la complessità dell'algoritmo di heapsort? Giustificare la risposta.

(a) O(n)

(b) O(log n)

(c) O(n log n)

(d) $\Theta(n^2)$

Esercizio 3 (ASD)

Disegnare un albero R/B che contiene le chiavi: 1,2,3,4,5,6,7,8,9,10,11,12 ed ha altezza massimale rispetto agli altri alberi R/B che contengono le stesse chiavi.

Esercizio 4 (Laboratorio)

Si vuole realizzare una struttura dati a lista concatenata che consenta di gestire un multi-insieme di numeri interi ordinati in senso crescente. Si implementi quindi:

1. una classe *Elemento* che memorizza un singolo numero intero del multi-insieme e tiene conto del numero delle sue occorrenze;
2. una classe *MultiInsieme* che gestisce il multi-insieme. La classe deve prevedere
   - un metodo costruttore;
   - un metodo *insert* per l'inserimento ordinato di un nuovo elemento nel multi-insieme;
   - un metodo *remove* per la cancellazione di un elemento dal multi-insieme. Il metodo deve cancellare una singola occorrenza dell'elemento e deve ritornare un valore booleano che indichi se l'operazione è andata a buon fine (elemento trovato e cancellato) oppure no (elemento non presente nel multi-insieme).

Esercizio 5 (ASD)

Scrivere lo pseudo codice di un algoritmo che verifica se un albero binario T è un albero binario di ricerca. Si assuma che i nodi di T siano stati aumentati aggiungendo agli attributi *key*, *left* e *right* anche gli attributi *min* e *max* che individuano rispettivamente la chiave minima e la chiave massima tra tutte le chiavi memorizzate nel sottoalbero radicato in quel nodo.

Esercizio 6 (ASD e Laboratorio)

1. **(ASD)** Scrivere lo pseudo codice di un algoritmo che trasforma un albero binario rappresentato come un albero generale, ovvero tramite gli attributi *key*, *child*, *sibling*, in un un albero binario rappresentato utilizzando gli attributi: *key*, *left*, *right*.

   Si utilizzi la funzione get_bnode per creare un nuovo nodo con gli attributi *key*, *left*, *right*.

2. **(LABORATORIO)** Si consideri il package *Trees* sviluppato durante il corso e relativo agli alberi generali. Si vuole aggiungere alla classe *GenTree* il seguente metodo che verifica se l'albero generale è un albero binario.

   ```
   // post: ritorna true se l'albero e' binario; ritorna false altrimenti
   public boolean isBinary() {...}
   ```

   Si richiede di implementare il metodo *isBinary* usando la ricorsione. Il metodo ritorna true se l'istanza corrente dell'albero è un albero binario, cioé se ogni suo nodo ha al più due figli. In caso contrario il metodo ritorna false. Se necessario utilizzare un metodo privato di supporto.

```
********************** CLASSE TreeNode  ********************************************************
package Trees;
class TreeNode {
    Object key;       // valore associato al nodo
    TreeNode parent;    // padre del nodo
    TreeNode child;      // figlio sinistro del nodo
    TreeNode sibling;     // fratello destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi sinistro e destro vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }

    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob, TreeNode parent, TreeNode child, TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }
}
********************** CLASSE GenTree  *********************************************************
package Trees;
import Utility.*;
public class GenTree implements Tree{
    private TreeNode root;      // radice dell'albero
    private int count;           // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
    ...
    ...
 }
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 4 Settembre 2007 —

Esercizio 1 (ASD)

Descrivete a parole le proprietà espresse dalle seguenti equazioni e per ciascuna dite se è sempre vera oppure se vale solo sotto particolari condizioni.

- $f(n) + O(f(n)) = \Theta(f(n))$
- $f(n) + \Omega(f(n)) = \Theta(f(n))$

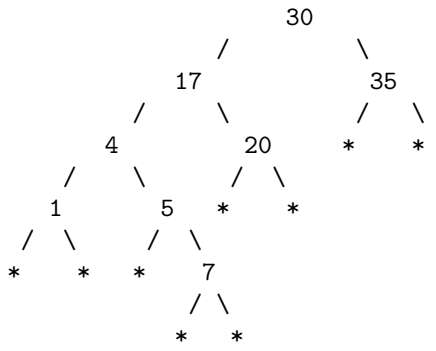
Esercizio 2 (ASD)

Risolvete le seguenti ricorrenze, giustificando la risposta.

- $T(n) = 2T(\frac{n}{3}) + 2n \log n + 3n$
- $T(n) = T(\frac{n}{2}) + T(\sqrt{n}) + n$

Esercizio 3 (ASD)

- Dite se il seguente albero binario di ricerca può essere colorato in modo da diventare un albero R/B. Giustificare la risposta. (Il simbolo * rappresenta la foglia-NIL.)



- Disegnate l'albero che si ottiene applicando una rotazione destra alla radice dell'albero precedente.

Esercizio 4 (ASD)

Sia $A[1..n]$, $n \geq 1$ un array di interi che soddisfa la seguente proiprietà:

$$\begin{aligned} \exists m: \quad & 1 \leq m \leq n \\ & \forall i: \ 1 \leq i < m \implies A[i] < A[i+1] \\ & \forall i: \ m \leq i < n \implies A[i] > A[i+1] \end{aligned}$$

Sviluppate un algoritmo che trovi l'indice m in tempo $O(\log n)$ e dimostrarne la correttezza.

[continua sul retro]

Esercizio 5 (LAB)

Sia data la seguente interfaccia per alberi binari generalizzati, con struttura a nodi in cui ogni nodo ha un colore ed un riferimento alla lista dei figli. L'interfaccia `ColorTree` ha un metodo `root()` che restituisce la radice dell'albero e due metodi che, dato un nodo, restituiscono rispettivamente il colore del nodo ed un iteratore che permette di scorrere la lista dei sui figli.

```
interface ColorTree  {
      Node root();               // la radice dell'albero
  Color color(Node p);       // il colore del nodo p
      Iterator figli(Node p);    // lista di figli di p
}
```

Diciamo che `T:ColorTree` è un BV-albero se `T` ha tutti i nodi di colore Blue (`Color.BLUE`) o Verde (`Color.GREEN`) ogni nodo Blue in `T` ha tutti i suoi figli Blue.

Fornite una implementazione del metodo `count()` specificato qui di seguito. La vostra implementazione deve garantire una complessità $O(k)$ dove k è il numero dei nodi verdi dell'albero.

```
public static void count(Tree T) {
  // PRE: T != null e' un BV-albero
  // POST: il numero dei nodi verdi in T.
}
```

Esercizio 6 (ASD+LAB)

Dato un albero binario T e due nodi u e v in T, definiamo il *minimo antenato comune* di u e v il nodo a di T di altezza minima che è antenato dei due nodi. Data l'interfaccia

```
interface BinTree  {
      Node root();               // la radice dell'albero
      Node left(Node p);         // figlio sinistro di p nell'albero
      Node right(Node p);        // figlio destro di p nell'albero
  int key(Node p);           // la chiave associata a p
}
```

fornite l'implementazione del metodo `mac` specificato qui di seguito e determinate la complessità della vostra implementazione.

```
public static Node mac(BinTree T, Node u, Node v) {
  // PRE: T != null e' un albero binario di ricerca (BST)
  con chiavi tutte distinte.
  // POST: restituisce il minimo antenato comune di u e v
}
```

Nota bene: l'albero `T` è un BST e **non** ha puntatore al padre.

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 1 Settembre 2006 —

Esercizio 1 (ASD)

Considerata la ricorrenza:

$$T(n) = 3T(\frac{n}{4}) + n^{\frac{3}{2}}$$

si richiede di:

- risolverla utilizzando il teorema principale;
- dire se $T(n) = \Omega(n)$, giustificando la risposta.

Esercizio 2 (ASD)

1. Sia A un array di $n > 10$ elementi che contiene un max-heap. Per ciascuna delle seguenti affermazioni dire se essa è necessariamente vera oppure no. Giustificare la risposta.
   - A[1] $\geq$ A[3].
   - A[2] $\geq$ A[6].
   - A[1] $\leq$ A[5].
   - A[5] $\leq$ A[2].
2. Sia T un BST (albero binario di ricerca) che contiene $n > 10$ chiavi. Si descriva un algoritmo efficiente (scrivere lo pseudo codice) per trasferire le chiavi memorizzate in T in un array A che rappresenta un max-heap.

Esercizio 3 (ASD)

Considerare la seguente procedura e determinare la sua complessità asintotica in funzione dell'input n.

```
proc(n) ::=
    m <- n*n
    while m > 0 do
        h <- 2*m + n
        for i = 1 to h do
            B[i] <- 0
        h <- 2*n
        for i = 1 to h do
            C[i] <- 0
        m <- m-1
```

Esercizio 4 (ASD)

Scrivere lo pseudocodice per una funzione che, dato un albero binario di ricerca T contenente chiavi distinte, determina se la radice di T contiene l'elemento *mediano superiore* dell'insieme delle chiavi contenute in T.
DEF: l'elemento mediano superiore di un insieme di n elementi è l'elemento che si trova in posizione $\lceil n/2 \rceil$ nella sequenza ordinata degli elementi dell'insieme. Esempio: il mediano dell'insieme $\{5, 1, 10, 2, 4\}$ è 4 mentre il mediano dell'insieme $\{9, -1, 5, 7, 8, 2\}$ è 5.

Esercizio 5 (Laboratorio)

Implementare il seguente metodo per la classe *BinaryTree* del package *BinTrees* visto a lezione:

```
    // post: ritorna una lista concatenata di tipo SLList contenente le chiavi di tutti nodi
    //       dell'albero che sono nonni di almeno un nipote.
    //       La lista e' vuota se non esiste alcun nodo nell'albero con questa proprieta'.
public SLList nonni() {...}
```

Se necessario, è possibile definire metodi privati di supporto.

Esercizio 6 (Laboratorio)

Si vuole realizzare l'implementazione di una *lista di liste* in cui i nodi della lista principale sono di tipo:

```
package Esercizio6;
class NodoLL {
    Object  key;       // chiave
    NodoL   headL;     // riferimento alla testa della lista associata al nodo
    NodoLL  next;      // riferimento al prossimo nodo

    NodoLL(Object ob) { key = ob; next = null; headL = null; }
}
```

mentre i nodi delle liste associate alla lista principale sono di tipo:

```
package Esercizio6;
class NodoL {
    Object key;    // chiave
    NodoL  next;   // riferimento al prossimo nodo

    NodoL(Object ob) { key = ob; next = null; }
}
```

Si richiede di:

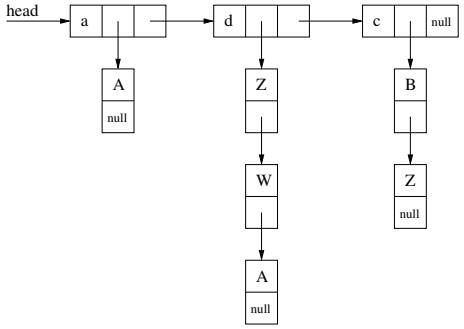
1. completare l'implementazione della seguente classe *ListaDiListe*:

   ```
   package Esercizio6;
   public class ListaDiListe {
       private NodoLL head = null;       // riferimento alla lista di liste

       // pre: ob1 e ob2 non nulli
       // post: ricerca l'oggetto ob1 nella lista principale e:
       //       - se ob1 e' presente inserisce ob2 in coda alla lista associata al nodo di ob1
       //       - se ob1 non e' presente inserisce un nuovo record con chiave ob1 in coda alla
       //         lista principale e un nuovo nodo con chiave ob2 nella lista associata al nodo di ob1
       public void insert(Object ob1, Object ob2) {...}
   }
   ```

2. [**Facoltativo**] scrivere gli invarianti del metodo *insert* relativi ai cicli di posizionamento nella lista principale e nella lista associata.

Un esempio di lista di liste costruita dalla classe *ListaDiListe* è il seguente:



```
***************************** interfaccia List.java ***************************
package BasicLists;
import java.util.Iterator;
public interface List {
    // post: ritorna il numero di elementi della lista
    public int size();

    // post: ritorna true sse la lista non ha elementi
    public boolean isEmpty();

    // post: svuota la lista
    public void clear();

    // pre:  ob non nullo
    // post: aggiunge l'oggetto ob in testa alla lista. Ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean insert(Object ob);

    // pre: l'oggetto passato non e' nullo
    // post: ritorna true sse nella lista c'e' un elemento uguale a value
    public boolean contains(Object value);

    // pre: l'oggetto passato non e' nullo
    // post: rimuove l'elemento uguale a value; ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean remove(Object value);

    // post: ritorna un oggetto che scorre gli elementi
    public Iterator iterator();

    // post: ritorna una lista che rappresenta tutti gli elementi della lista, in sequenza
    public String toString();
}
***************************** classe SLList.java ***************************
package BasicLists;
import java.util.Iterator;
public class SLList implements List {
    SLRecord head;              // primo elemento
    int count;                  // num. elementi nella lista

    // post: crea una lista vuota
    public SLList() { head = null; count = 0; }
    ....
}
***************************** classe SLRecord.java *********************
package BasicLists;
class SLRecord {
    Object key;             // valore memorizzato nell'elemento
    SLRecord next;         // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con valore v,  e prossimo elemento nextel
    SLRecord(Object ob, SLRecord nextel) { key = ob; next= nextel; }

    // post: costruisce un nuovo elemento con valore v, e niente next
    SLRecord(Object ob) { this(ob,null); }
}
***************************** classe BTNode.java ***************************
package BinTrees;
class BTNode {
    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi sinistro e destro vuoti
    BTNode(Object ob) { key = ob; parent = left = right = null; }

    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTNode(Object ob, BTNode left, BTNode right, BTNode parent) {
        key = ob; this.parent = parent; setLeft(left); setRight(right);
    }
    ....
}
***************************** classe BinaryTree.java ***************************
package BinTrees;
import java.util.Iterator;
import Queues.*;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;      // puntatore al nodo corrente
    private int count;          // numero nodi dell'albero

    // post: crea un albero binario vuoto
    public BinaryTree() { root = null;  cursor = null; count = 0; }
    ....
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 14 Gennaio 2009 —

Esercizio 1 - ASD

Provare la verità o la falsità di ciascuna delle seguenti affermazioni.

- La soluzione della ricorrenza $T(n) = 5T(\frac{n}{3}) + 3n^2 + 2\sqrt{n^3}$ è nella classe $\Theta(n^2)$
- $\Omega(n \lg n) + n^2 = O(n^2)$
- La soluzione della ricorrenza $T(n) = 2T(\sqrt{n}) + 3n^2$ è nella classe $\Theta(n^2)$

Esercizio 2 - ASD

Discutere la correttezza di ciascuna delle seguenti affermazioni. Dimostrare formalmente la validità delle risposte date.

1. La complessità asintotica dell'algoritmo di ricerca di una chiave in un albero R/B con n nodi è $\Theta(\lg n)$.
2. La complessità asintotica dell'algoritmo di ricerca di una chiave in un albero BST con n nodi è $\Omega(\lg n)$.
3. La complessità asintotica dell'algoritmo di ricerca di una chiave in un min-heap binario con n nodi è $\Theta(n)$.
4. Esistono alberi R/N che hanno tutti i nodi neri.
5. Possiamo trasformare un albero R/B con n nodi in un min-heap con complessità asintotica $\Theta(n)$.

Esercizio 3 - ASD

Diciamo che T è un *interval* BST se è un BST a chiavi intere che soddisfa la seguente proprietà: per ogni intero k, se le chiavi k e $k+2$ sono in T allora anche la chiave $k+1$ è in T.

Proporre un algoritmo che verifica se un dato BST a chiavi intere è un interval BST.

Dire qual è la complessità dell'algoritmo e spiegare perché è corretto.

Esercizio 4 - ASD

Si realizzi una operazione RIMUOVI(A, k, P) che soddisfa la seguente specifica:

Precondizione: A è un array che rappresenta un max-heap di dimensione `heapsize`$[A]$ a chiavi intere; $0 < k \leq 100$ e $P > 0$ sono due costanti.

Postcondizione: L'algoritmo restituisce un nuovo array A che rappresenta il max-heap ottenuto eliminando da quello iniziale i k più grandi elementi di A maggiori di P oppure tutti gli elementi di A maggiori di P nel caso vi siano in A meno di k elementi maggiori di P.

Dire qual è la complessità dell'algoritmo rispetto alla dimensione iniziale di A e spiegare perché è corretto.

Esercizio 1 (Laboratorio)

Dato il package *Liste* realizzato durante il corso, aggiungere il seguente metodo alla classe *ListaDoppia* che realizza le operazioni sulle liste mediante una lista doppia circolare con sentinella:

```
// post: ritorna il numero di elementi distinti presenti in lista
public int  distinct() {...}
```

Ad esempio, se la lista contiene, nell'ordine, gli elementi *x, g, h, g, v, x, y, g* allora il metodo deve ritornare il valore 5. Si richiede di completare l'implementazione del metodo e di scrivere gli invarianti di ciclo (senza dimostrarli!).
NOTA: per la soluzione di questo esercizio non possono essere utilizzate strutture dati d'appoggio e non possono essere richiamati gli altri metodi della classe *ListaDoppia*.

Esercizio 2 (Laboratorio)

Dato il package *Tree* realizzato durante il corso e relativo agli alberi generali, aggiungere il seguente metodo alla classe *GenTree*:

```
// pre: k diverso da null
// post: ritorna true sse tutte le foglie dell'albero hanno chiave uguale a k
public boolean foglieUguali(Object k) {...}
```

Si richiede di completare l'implementazione del metodo usando la ricorsione. L'algoritmo deve avere complessità lineare rispetto al numero di nodi dell'albero. È possibile utilizzare un metodo privati di appoggio.

```
********************** classe RecordLD ************************************
package Liste;
class RecordLD {
    Object key;             // valore memorizzato nell'elemento
    RecordLD next;          // riferimento all'elemento successivo
    RecordLD prev;          // riferimento all'elemento precedente

    // post: costruisce un nuovo elemento con valore v,
    //       elemento successivo nextel e precedente prevel
    RecordLD(Object ob, RecordLD nextel, RecordLD prevel) {
        key = ob;
        next= nextel;
        if (next != null)
            next.prev = this;
        prev = prevel;
        if (prev != null)
            prev.next = this;
    }

    // post: costruisce un nuovo elemento con valore v, e niente next e prev
    RecordLD(Object ob) {
        this(ob,null, null);
    }
}
********************** classe ListaDoppia **********************************
package Liste;
public class ListaDoppia implements Lista {
    private RecordLD sentinel;          // riferimento alla sentinella
    private int count;                  // num. elementi nella lista


    // metodo costruttore
    // post: crea una lista vuota
    public ListaDoppia() {
        // la sentinella ha chiave puntatori nulli
        sentinel = new RecordLD(null,null, null);
        sentinel.next = sentinel;
        sentinel.prev = sentinel;
        count = 0;
    }

   ...
}
********************** classe TreeNode *************************************
package Trees;
class TreeNode {

    Object key;                 // valore associato al nodo
    TreeNode parent;     // padre del nodo
    TreeNode child;        // primo figlio del nodo
    TreeNode sibling;     // fratello destro del nodo


    // post: ritorna un albero di un solo nodo, con chiave ob e sottoalberi vuoti
    TreeNode(Object ob) {
        key = ob;
        parent = child = sibling = null;
    }


    // post: ritorna un albero contenente la chiave ob e i sottoalberi specificati
    TreeNode(Object ob,
             TreeNode parent,
             TreeNode child,
             TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }

}
********************** classe GenTree **************************************
package Trees;
public class GenTree implements Tree{
    private TreeNode root;      // radice dell'albero
    private int count;           // numero di nodi dell'albero
    private TreeNode cursor;    // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
   ...
}
```

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 14 Gennaio 2009 —

Esercizio 1 - ASD

Si risolvano le seguenti ricorrenze, giustificando la risposta.

- $T(n) = 9T(\frac{n}{3}) + 3n^2 + 2\sqrt{n}$
- $T(n) = \frac{5}{2}T(\frac{n}{2}) + n$

Soluzione

- Poiché $\lg_3 9 = 2$ e $f(n) = 3n^2 + 2\sqrt{n} = \Theta(n^2)$ siamo nel caso 2 del MT e la soluzione è $T(n) = \Theta(n^2 \lg n)$.
- Poiché $\lg_2 \frac{5}{2} > 1$ esiste $\epsilon > 0$ tale che $1 = \lg_2 \frac{5}{2} - \epsilon$. Quindi $f(n) = n = O(n^{\lg_2 \frac{5}{2} - \epsilon})$. Siamo nel caso 1 del MT e la soluzione è $T(n) = \Theta(n^{\lg_2 \frac{5}{2}})$.

Esercizio 2 - ASD

Date le seguenti procedure A e B, si determini la complessità asintotica della procedura A(n) su input n.

A(n)
1 $s \leftarrow 0$
2 **for** $i \leftarrow 1$ **to** n
3 **do** $s \leftarrow s +$ B(i)
4 **return** s

B(m)
1 $s \leftarrow 0$
2 **for** $j \leftarrow 1$ **to** m
3 **do** $s \leftarrow s + 1$
4 **return** s

Soluzione

La complessità di B(i) è lineare in i pertanto ad ogni iterazione del ciclo la terza istruzione di A ha costo ki per una qualche costante k. Sommando su tutti i valori che assume la variabile i otteniamo $T_A(n) = \sum_{i=1}^{n}(ki) = \Theta(n^2)$

Esercizio 3 - ASD

Scrivere un algoritmo che dato un albero binario di ricerca T, bilanciato e contenente chiavi tutte distinte, e due chiavi $k_1 < k_2$ restituisce **true** se e solo se T soddisfa anche la seguente proprietà:
- per ogni nodo x di T se $key[x] = k_1$ allora non esiste alcun nodo $y \neq x$ in T tale che $k_1 < key[y] < k_2$.
Dire qual è la complessità dell'algoritmo rispetto al numero delle chiavi memorizzate nell'albero e spiegare perché è corretto.

Soluzione

Precondizione: x è la radice di un BST bilanciato.

Postcondizione: L'algoritmo risponde **true** se l'albero radicato in x gode della proprietà data, **false** altrimenti.

CHECK$(x, k1, k2)$

1 $z \leftarrow$ BSTSEARCH$(x, k1)$
2 **if** $((z = \text{NIL}) \vee (\text{SUCC}(z) = \text{NIL})$
3 **then return true**
4 **else return** $(key[\text{SUCC}(z)] \geq k2)$

Correttezza: Se la chiave k_1 non compare nell'albero (la BSTSEARCH ritorna NIL) allora la condizione è soddisfatta; idem se compare ma è la più grande (SUCC(z) ritorna NIL); altrimenti (compare e non è la più grande), è sufficiente verificare che la chiave del successore di z sia maggiore o uguale a k_2. Infatti la proprietà BST ci assicura che tutti i predecessori di un nodo hanno chiave minore o uguale a quella del nodo mentre i suoi successori hanno chiave maggiore o uguale.

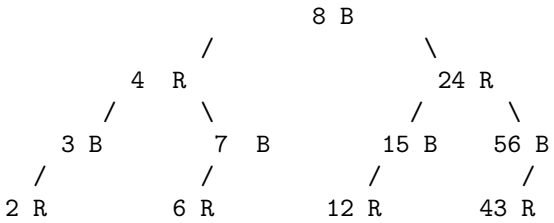
Complesità: La complessità è logaritmica nel numero delle chiavi. Infatti sia la ricerca di una chiave che l'individuazione del successore sono operazioni lineari nell'altezza dell'albero, e l'albero è bilanciato.

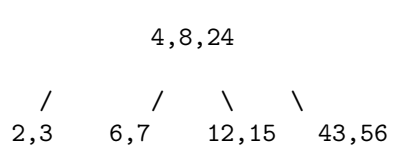
Esercizio 4 - ASD

Si disegni un albero R/B che contiene le seguenti chiavi: 6, 15, 8, 2, 24, 56, 3, 43, 12, 4, 7. Lo si trasformi poi in un albero 2-3-4.

Soluzione

Una possibile soluzione è la seguente.





Esercizio 5 - ASD

Si sviluppi un algoritmo che, dati un albero generale T e un intero positivo $k > 0$, conta il numero di nodi di grado k in T. Si supponga che l'albero generale sia rappresentato tramite gli attributi: `fratello` e `figlio`. (Ricordiamo che il grado di un nodo di un albero è pari al numero dei suoi figli.)

Soluzione

```
CONTAGRADO(x, k)
 1  if (x = NIL)
 2      then return 0
 3      else  z ← figlio[x]
 4            n ← 0
 5            while (z ≠ NIL)
 6                  do n ← n + 1
 7                     z ← fratello[z]
 8            if (n = k)
 9               then return 1 + CONTAGRADO(fratello[x], k) + CONTAGRADO(figlio[x], k)
10               else  return CONTAGRADO(fratello[x], k) + CONTAGRADO(figlio[x], k)
```

Esercizio 1 (Laboratorio)

Una sequenza ordinata è una collezione in cui gli elementi compaiono in modo ordinato e sono ammesse più copie dello stesso elemento. Si vuole realizzare una classe *SequenzaOrdinata* per rappresentare una sequenza ordinata di elementi di tipo stringa. La struttura dati scelta per memorizzare la sequenza ordinata è l'array.

```
public class SequenzaOrdinata {
    private static final int defaultSize = 100;
    private String[] S = new String[defaultSize];    // la sequenza e' un array
    private int count;     // tot. elementi della sequenza


    // pre:  s non nulla
    // post: aggiunge la stringa s alla sequenza ponendola nella
    //       posizione corretta rispetto all'ordine e aggiornando count.
    //       Ritorna true se l'operazione e' riuscita, false se non c'e' piu'
    //       spazio libero nell'array
    public boolean insert(String s) {...}

    ...
}
```

Si richiede di completare l'implementazione del metodo *insert* della classe *SequenzaOrdinata* riportata sopra e di dimostrarne la correttezza.

Soluzione

- Una possibile implementazione è la seguente:

```
public class SequenzaOrdinata {
    private static final int defaultSize = 100;
    private String[] S = new String[defaultSize];    // la sequenza e' un array
    private int count;     // tot. elementi della sequenza


    // pre:  s non nulla
    // post: aggiunge la stringa s alla sequenza ponendola nella
    //       posizione corretta rispetto all'ordine e aggiornando count.
    //       Ritorna true se l'operazione e' riuscita, false se non c'e' piu'
    //       spazio libero nell'array
    public boolean insert(String s) {

         if (count == defaultSize)
            return false;

         // INV: gli elementi originariamente in S[j+1,count-1] sono maggiori
         //      di s e sono stati spostati in S[j+2,count]
         int j;
         for (j = count-1; j >= 0 && S[j].compareTo(s) > 0 ; j--)
             S[j+1] = S[j];
         S[j+1] = s;
         count++;
         return true;
    }

   ...
}
```

- Verifichiamo l'invariante riportato nel codice:

  Inizializzazione: All'inizio j=count-1 e quindi la porzione di array S[cont,count-1] e' vuota. Lo stesso per S[count+1,count]. L'invariante è banalmente verificato.

Mantenimento: Sia INV vero per j fissato. Allora gli elementi originariamente in S[j+1, count-1] sono maggiori di s e sono stati spostati in S[j+2,count] . Si entra nel ciclo solo se S[j] > s e, in tal caso l'elemento S[j] viene copiato in S[j+1]. Allora, dopo l'esecuzione del corpo del ciclo e' vero che gli elementi originariamente in S[j...count-1] sono maggiori di s e sono stati spostati in S[j+1,count]. Quindi l'invariante viene mantenuto al decrementare di j.

Terminazione: il ciclo termina se j >= 0 e S[j] < s oppure se j=-1. In entrambi i casi è vero che gli elementi originariamente in S[j+1..count-1] sono maggiori di s e sono stati spostati in S[j+2, count]. L'invariante garantisce quindi la correttezza dell'inserimento di s in posizione j+1.

Esercizio 2 (Laboratorio)

1. Si consideri una tabella hash T[0,...,6] = [-, 1, C, -, 11, 19, 26] in cui le posizioni 0 e 3 sono libere e la posizione 2 risulta marcata in seguito ad un'operazione di cancellazione. La tabella è stata costruita utilizzando la funzione hash
$$h(k) = k \bmod 7$$
e tecnica di gestione delle collisioni ad indirizzamento aperto con scansione lineare. Si consideri il problema di inserire la chiave 67 nella tabella. Dire se la chiave viene inserita, eventualmente in quale posizione e quali posizioni della tabella vengono esaminate con insuccesso.

2. Si consideri una tabella hash T[0...22] in cui si vogliono memorizzare dati relativi agli studenti che sostengono questo appello d'esame. Gli studenti vengono identificati dal numero di matricola, che diventa la chiave della tabella hash. Si consideri la seguente funzione hash basata sul metodo del ripiegamento:
$$h(K) = h(k_1k_2k_3k_4k_5k_6) = (k_1k_2 \ - \ k_3k_4 \ + \ k_5k_6) \bmod 23$$
dove $k_1k_2k_3k_4k_5k_6$ sono le cifre decimali che compongono K. Ad esempio, h(816145) = (81 - 61 + 45) mod 23 = 65 mod 23 = 19. Partendo dalla tabella contenente solamente la chiave utilizzata come esempio (816145), si richiede di inserire in tabella le seguenti chiavi specificando per ciascuna la posizione di inserimento e il numero di accessi effettuati:
   (a) 837120
   (b) 829133
   (c) 818326
   (d) 792364

   Per la soluzione di eventuali collisioni utilizzare la scansione quadratica
$$c(k,i) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod 23 \quad 0 \le i \le 22$$
   con $c_1 = 1$ e $c_2 = 3$.

Soluzione

1. La chiave 67 viene inserita in posizione 0 dopo aver esaminato le posizioni 4, 5 e 6.

2. (a) h(837120) = (83 - 71 + 20) mod 23 = (103 - 71) mod 23 = 32 mod 23 = 9, libera. Un accesso.
   (b) h(829133) = (82 - 91 + 33) mod 23 = (115 - 91) mod 23 = 24 mod 23 = 1, libera. Un accesso
   (c) h(818326) = (81 - 83 + 26) mod 23 = 24 mod 23 = 1, occupata.
   Per i = 1 la nuova posizione è ((1 + 1 + 3) mod 23) = 5. Due accessi.
   (d) h(792364) = (79 - 23 + 64) mod 23 = 120 mod 23 = 5, occupata.
   Per i = 1 la nuova posizione è ((5 + 1 + 3) mod 23) = 9, occupata.
   Per i = 2 la nuova posizione è ((5 + 2 + 3*4) mod 23) = 19, occupata.
   Per i = 3 la nuova posizione è ((5 + 3 + 3*9) mod 23) = 12, libera. Quattro accessi.

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 8 Giugno 2009 —

Esercizio 1 - ASD

- Giustificando la risposta, si risolva la seguente ricorrenza:
  $T(n) = 2T(\frac{n}{3}) + 3\sqrt{n^3}$
- Giustificando la risposta, si dica se la seguente affermazione è corretta.
  Per ogni funzione $f(n)$ asintoticamente positiva si ha $\Theta(\sqrt{n^3} + f(n)) = \Omega(\sqrt{n^3})$

Esercizio 2 - ASD

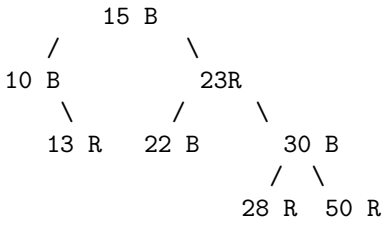
Chiamiamo *BST-foresta* una lista $F = T_1, T_2, ..., T_n$ di BST non vuoti a chiavi intere che soddisfa la seguente proprietà: per ogni i, $1 \leq i < n$, la chiave più grande memorizzata in T_i è minore o uguale alla chiave più piccola memorizzata in T_{i+1}.
Specificare la struttura dati BST-foresta (strutture dati usate, attributi dei nodi, operazioni, ...) e proporre un algoritmo per l'operazione di inserimento di una chiave k in una BST-foresta.

Esercizio 3 - ASD

Si descriva l'algoritmo per l'inserimento di un nuovo elemento in una coda di priorità Q, realizzata con un array che rappresenta un max-heap.

Esercizio 4 - ASD

1) Dire se il seguente albero binario gode della proprietà R/B. (Chiaramente si suppone che tutte le foglie-NIL lasciate implicite siano nere).



2) In caso di risposta positiva, disegnare l'albero che si ottiene dopo aver simulato l'inserimento della chiave 35.

Esercizio 1 (Laboratorio)

Una sequenza ordinata è una collezione in cui gli elementi compaiono in modo ordinato e sono ammesse più copie dello stesso elemento. Si vuole realizzare un package per rappresentare e gestire una sequenza ordinata di elementi di tipo integer. La struttura dati scelta per memorizzare la sequenza ordinata è la lista semplice circolare con sentinella. Ovviamente la lista viene mantenuta ordinata in senso crescente rispetto agli elementi della sequenza.
Data la classe:

```
package SO;
class NodoLista {
    int key;
    NodoLista next;

    NodoLista(int k) {
       key = k;
       next = null;
    }
}
```

si richiede di completare l'implementazione dei metodi *insert* e *oneOccurrence* della seguente classe. I due metodi devono avere complessità lineare rispetto al numero degli elementi in lista.

```
package SO;
public class SequenzaOrdinata {
    private NodoLista sentinel; // nodo sentinella
    private int count; // totale elementi in lista

    // post: costruisce una sequenza ordinata vuota
    public SequenzaOrdinata() {
        sentinel = new NodoLista(0); // valore convenzionale per la sentinella
        sentinel.next = sentinel;
        count = 0;
    }

    // post: inserisce l'elemento k nella sequenza ordinata e aggiorna count
    public void insert(int k) {...}

   // post: ritorna true se e solo se tutti gli elementi della sequenza ordinata
   //       occorrono una sola volta
   // NOTA: se la sequenza ordinata e' vuota ritorna true
   public boolean oneOccurrence() {...}

}
```

Esercizio 2 (Laboratorio)

Si consideri una tabella hash T[0...9] con gestione delle collisioni mediante liste di collisioni, ordinate in senso crescente rispetto alla chiave. La chiave della tabella hash è un numero reale $k \in (0, 1)$ e la funzione hash è $h(k) = \lfloor km \rfloor$, dove m è la dimensione della tabella hash. Ad esempio, per la tabella considerata, $h(0,57) = \lfloor 0,57 * 10 \rfloor = 5$. Partendo dalla tabella contenente solamente la chiave utilizzata come esempio (k = 0,57) si richiede di inserire in tabella le seguenti chiavi mostrando tutti i passaggi del procedimento:

- $0,123$
- $0,619$
- $0,24$
- $0,95$
- $0,5321$
- $0,67$

- $0,299$
- $0,91$
- $0,652$
- $0,74$

Qual è il fattore di carico della tabella?

Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

— Appello del 30 Giugno 2009 —

Esercizio 1 - ASD

1. Sia $T(n) = 4T(n/2)+n^2+3n$. Considerare ciascuna delle seguenti affermazioni e dire se è corretta o no. Giustificare la risposta.

(a) $T(n) = \Omega(n)$

(b) $T(n) = O(n^3)$

(c) $T(n) = O(n \lg n)$

2. Sia $T(n) = T(n/6) + T(n/3) + n$. Verificare usando il metodo di sostituzione la correttezza della seguente affermazione. $T(n) = O(n \lg n)$

Esercizio 2 - ASD

Si consideri il seguente array A=[14,5,12,9,7,8,24,6,10,5] e lo si trasformi applicando l'algoritmo BuildMaxheap.

Esercizio 3 - ASD

1. Si sviluppi un algoritmo che, dato un albero generale T rappresentato tramite gli attributi: `fratello`, `figlio` e `padre` e un intero $k > 0$, calcola il numero dei nodi di grado k presenti in$T$. (Ricordiamo che il grado di un nodo di un albero è pari al numero dei suoi figli.)

2. Si dimostri la correttezza dell'algoritmo proposto.

Esercizio 4 - ASD

Sia T un albero R/B. Per ciascuna delle seguenti affermazioni dire se essa è vera o falsa. Giustificare la risposta.

- L'operazione di inserimento di una nuova chiave in T ha complessità logaritmica nel numero dei nodi di T.
- L'operazione di ricerca di una chiave in T ha complessità logaritmica nel numero dei nodi di T.
- Per ogni nodo x di T, su ogni cammino da x alla radice esiste almeno un nodo rosso.
- Per ogni nodo x di T, su ogni cammino da x ad una foglia esiste lo stesso numero di nodi neri.

Esercizio 1 (Laboratorio)

Date le classi *StackArray* (che realizza uno stack con un array) e *QueueArray* (che realizza una coda con un array) sviluppate durante il corso, si richiede di completare l'implementazione dei due metodi della seguente classe:

```
import Queues.QueueArray;
import Stacks.StackArray;
public class Esercizio1 {

    // pre: S non nullo
    // post: scambia l'ordine del primo e dell'ultimo elemento
    //       presenti nello stack S, lasciando inalterati tutti
    //       gli altri elementi. Se S contiene meno di due elementi
    //       allora rimane inalterato
    public static void scambiaS(StackArray S) {...}


    // pre: Q non nulla
    // post: scambia l'ordine del primo e dell'ultimo elemento
    //       presenti nella coda Q, lasciando inalterati tutti
    //       gli altri elementi. Se Q contiene meno di due elementi
    //       allora rimane inalterata
    public static void scambiaQ(QueueArray Q) {...}
}
```

È possibile utilizzare strutture dati di appoggio solo di tipo *StackArray* e solo se strettamente necessarie.

Esercizio 2 (Laboratorio)

Data la definizione:

> Un nodo n di un albero binario si dice *nodo sinistro* se è figlio sinistro di suo padre. Si dice invece *nodo destro* se è figlio destro di suo padre. La radice è un nodo a parte, cioè non è né sinistro né destro.

Si consideri il package *BinTrees* sviluppato durante il corso e relativo agli alberi binari. Si richiede di aggiungere alla classe *BinaryTree* l'implementazione del seguente metodo:

```
    // post: ritorna la differenza tra il numero di nodi sinistri e il numero
    //       di nodi destri dell'albero
     public int nodiSxmenoDx() {...}
```

Il metodo deve essere lineare rispetto al numero di nodi dell'albero.
Se necessario, è possibile utilizzare metodi privati di supporto.

```
********************** classe StackArray ************************************
package Stacks;
public class StackArray implements Stack {
    private static final int MAX=100;   // dimensione massima dello stack
    private Object[] S;                 // lo stack
    private int head;                   // puntatore al top dello stack

    // post: costruisce uno stack vuoto
    public StackArray() {
        S = new Object[MAX];
        head = -1;
    }

    // post: ritorna il numero di elementi nello stack
    public int size() {
         return head +1;
    }

   // post: ritorna lo stack vuoto
    public void clear() {
         head = -1;
    }

    // post: ritorna true sse lo stack e' vuoto
    public boolean isEmpty() {
         return (head == -1);
    }

    // post: ritorna true sse la coda e' piena
    public boolean isFull() {
         return (head == MAX -1);
    }

    // pre: stack non vuoto!
    // post: ritorna l'oggetto in cima allo stack
    public Object top() {
         return S[head];
    }

   // post: inserisce ob in cima allo stack
    public void push(Object ob) {
        if (isFull())
           return;
        S[++head] = ob;
    }

    // pre: stack non vuoto!
    // post: ritorna e rimuove l'elemento in cima allo stack
    public Object pop() {
        return S[head--];
    }
}
********************** classe QueueArray ************************************
package Queues;
public class QueueArray implements Queue {
    private static final int MAX=100;   // dimensione massima della coda
    private Object[] Q;                 // la coda
    private int head;                   // puntatore al primo elemento in coda
    private int tail;                   // puntatore all'ultimo elemento della coda

    // post: costruisce una coda vuota
    public QueueArray() {
        Q = new Object[MAX];
        head = 0;
        tail = 0;
    }

    // post: ritorna il numero di elementi nella coda
    public int size() {
    return tail - head;
    }

    // post: ritorna true sse la coda e' vuota
    public boolean isEmpty() {
    return (head  == tail);
    }

    // post: ritorna true sse la coda e' piena
    public boolean isFull() {
    return (tail - head == MAX);
    }

    // post: svuota la coda
    public void clear() {
        head = 0;
        tail = 0;
    }
```

```
    // pre: coda non vuota
    // post: ritorna il valore del primo elemento della coda
    public Object front() {
         return Q[head % MAX];
    }

    // pre: value non nullo
    // post: inserisce value in coda
    public void enqueue(Object ob) {
        if (isFull())
           return;

        Q[tail % MAX] = ob;
        tail = tail + 1;
        if (head > MAX) {
            tail = tail - MAX;
            head = head - MAX;
        }
    }

    // pre: coda non vuota
    // post: ritorna e rimuove l'elemento il primo elemento in coda
    public Object dequeue() {
        Object temp = Q[head % MAX];
        head = (head + 1);
        return temp;
    }
}
********************** classe BTNode ****************************************
package BinTrees;
class BTNode {
    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    //       sinistro e destro vuoti
    BTNode(Object ob) {
        key = ob;
        parent = left = right = null;
    }

    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTNode(Object ob,
           BTNode left,
           BTNode right,
           BTNode parent) {
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
    ...
}
********************** classe BinaryTree ************************************
package BinTrees;
import Queues.*;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;      // puntatore al nodo corrente
    private int count;          // numero nodi dell'albero


    // post: crea un albero binario vuoto
    public BinaryTree() {
        root = null;
        cursor = null;
        count = 0;
    }

    ...
}
```