Dati e Algoritmi (Pietracaprina)

Esercizi sulle Mappe

Problema 1 (Esercizio R-10.9 [GTG14]) *Stimare asintoticamente il numero di operazioni richiesto al caso migliore* (*best-case*) *e al caso pessimo* (*worst-case*) *per l'inserimento di* n *entry con chiavi distinte in una Mappa implementata tramite Tabella Hash, inizialmente vuota, di capacità* N *con separate chaining. Si assuma* $N > n$.

Soluzione. Nel caso migliore, la funzione hash mappa ogni entry in un bucket diverso, e quindi ciascun inserimento richiede $\Theta(1)$ operazioni. La complessità di tutti gli n inserimenti è quindi $\Theta(n)$. Nel caso peggiore, la funzione hash le mappa tutte nello stesso bucket. Dato che le n entry hanno chiavi distinte, in questo caso, l'i-esimo inserimento richiede $\Theta(i)$ operazioni in quanto vi sono già $i-1$ entry presenti nel bucket associato alla nuova entry da inserire, e la `put` deve guardarle tutte per accertarsi che non vi sia tra esse una entry con la stessa chiave di quella da inserire. La complessità di tutti gli n inserimenti è quindi $\Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$.

□

Problema 2 *Scrivere una versione iterativa di* `TreeSearch`.

Soluzione. L'algoritmo è il seguente:

Algoritmo `TreeSearch(`k`,`T`)`

Input: chiave k, Albero binario di ricerca T
Output: nodo $w \in T$ con chiave k, se esiste, o foglia nella posizione "giusta" per k

$w \longleftarrow T$.root();
while (T`.isInternal(`w`))` **do**
  $x \longleftarrow w$`.getElement().getKey();`
  if $(x = k)$ **then return** w;
  if $(k < x)$ **then** $w \longleftarrow T$`.left(`w`)`;
  else $w \longleftarrow T$`.right(`w`)`;
return w

La complessità di `TreeSearch(`k`)` è $\Theta(h)$, con h l'altezza dell'albero, dato che in ciascuna iterazione del while si esegue un numero costante di operazioni e w scende di un livello. □

Problema 3 *Progettare un algoritmo* (*iterativo o ricorsivo*) *che, dato un albero binario di ricerca* T *contenete entry con chiavi reali e un intervallo* $[A, B]$*, restituisca un nodo* $w \in T$ *contenente una entry con chiave* $k \in [A, B]$*, se esiste, altrimenti una foglia nella posizione "giusta" per una chiave in* $[A, B]$*, e analizzarne la complessità.*

Soluzione. Sviluppiamo un algoritmo iterativo modificando la versione iterativa di `TreeSearch` (lo sviluppo di un algoritmo ricorsivo è lasciato come esercizio).

Algoritmo `TreeSearchAB(`A`,`B`,`T`)`

Input: intervallo $[A, B]$, Albero binario di ricerca T
Output: nodo $w \in T$ con chiave $k \in [A, B]$, o w foglia nella posizione "giusta" per k

$w \longleftarrow T$.root();
while (T`.isInternal(`w`))` **do**
  $x \longleftarrow w$.getElement().getKey();
  if ($x \in [A, B]$) **then return** w;
  if ($B < x$) **then** $w \longleftarrow T$.`left`(w);
  else $w \longleftarrow T$.`right`(w);
return w

La complessità di `TreeSearchAB(`A`,`B`)` è $\Theta(h)$, con h l'altezza dell'albero, dato che in ciascuna iterazione del while si eseguono $\Theta(1)$ operazioni e w scende di un livello. □

Problema 4 *Progettare un algoritmo iterativo che, dato un albero binario di ricerca T contenete entry con chiavi reali* distinte*, determina la più piccola chiave positiva presente in T, e analizzarne la complessità. Se in T non esistono chiavi positive, l'algoritmo deve restituire* `'no positive key'`*.*

Soluzione. L'algoritmo esegue una discesa nell'albero: quando trova una chiave positiva va a sinistra (in questo caso il sottoalbero destro non può contenere la minima chiave positiva), mentre quando trova una chiave negativa o nulla va a destra (in questo caso il sottoalbero sinistro contiene solo chiavi negative o nulle e quindi non può contenere la minima chiave positiva). Durante la discesa si memorizza la minima chiave positiva incontrata che, alla fine, restituisce in output. Lo pseudocodice è il seguente.

Algoritmo `MinPositive(`T`)`

Input: Albero binario di ricerca T con chiavi reali distinte
Output: min chiave positiva, se esiste, altrimenti 'no positive key'

$v \longleftarrow T$.`root`(); minPosKey $\longleftarrow +\infty$;
while (T`.isInternal(`v`))` **do**
  $k \longleftarrow v$`.getElement().getKey();`
  if ($k > 0$) **then**
    `minPosKey` $\longleftarrow k$;
    $v \longleftarrow T$.`left`(v)
  else $v \longleftarrow T$.`right`(v);
if (`minPosKey`$< +\infty$) **then return** `minPosKey;`
else return `'no positive key';`

Per quanto semplice, la correttezza dell'algoritmo richiede la seguente osservazione. I valori positivi assegnati alla variabile `minPosKey` (se ne esistono) formano una sequenza decrescente in quanto ogni volta che alla variabile viene assegnato, come valore, la chiave k in un nodo v, l'algoritmo prosegue nel sottoalbero sinistro di v dove si incontreranno solo chiavi minori di k. Questo giustifica l'assegnamento a `minPosKey` fatto dentro al **while** senza prima verficare che k sia effettivamente minore del valore corrente di `minPosKey`. Riguardo alla complessità, dato che il nodo v scende di un livello ad ogni iterazione del **while**, il numero di iterazioni è proporzionale al più all'altezza di T, e in ciascuna iterazione si esegue un numero costante di

operazioni. Al di fuori del ciclo, si eseguono $O(1)$ operazioni. La complessità è quindi $O(h)$, con h altezza di T. □

Problema 5 *Si definisca* Interval Tree *un albero binario di ricerca le cui entry sono intervalli $[a, b]$, con $a \leq b$, sulla retta reale. La chiave di una entry $[a, b]$ è rappresentata dall'estremo sinistro a dell'intervallo. Nell'albero non possono esistere due intervalli $[a, b]$ e $[c, d]$ in cui il primo sia interamente contenuto nel secondo, cioè con $c \leq a \leq b \leq d$.*

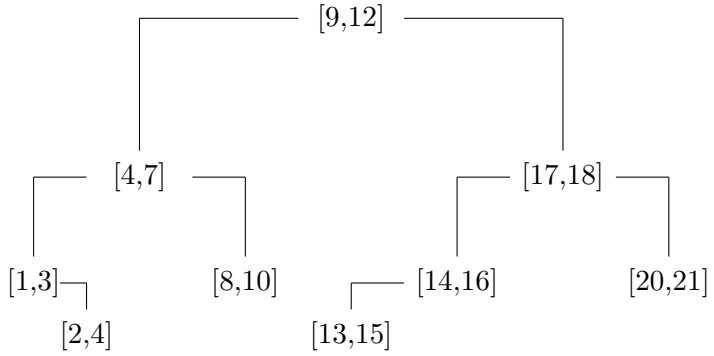
a. *Disegnare l'Interval Tree costruito inserendo nell'ordine i seguenti intervalli, a partire dall'albero vuoto:* $[9, 12]$, $[4, 7]$, $[17, 18]$, $[1, 3]$, $[8, 10]$, $[14, 16]$, $[20, 21]$, $[2, 4]$, $[13, 15]$.

b. *Progettare un algoritmo iterativo* `InsertCheck` *che, dato un intervallo $[a, b]$ e un Interval Tree T, determina se $[a, b]$ può essere inserito in T, ovvero se non esiste in T un itervallo che contiene interamente $[a, b]$ o che è contenuto interamente in $[a, b]$.*

c. *Analizzare la complessità di* `InsertCheck`.

Soluzione.

a. L'Interval Tree è il seguente (le foglie non sono disegnate):



b. Un'osservazione chiave per lo sviluppo dell'algoritmo è la seguente. Confrontiamo $[a, b]$ con l'intervallo $[c, d]$ memorizzato nella radice. Si possono verificare quattro casi:

- $[a, b]$ è interamente contenuto in $[c, d]$. In questo caso, l'algoritmo deve terminare rispondendo NO.
- $[a, b]$ contiene interamente $[c, d]$. In questo caso, l'algoritmo deve terminare rispondendo NO.
- $a < c$ e $b < d$. In questo caso un eventuale intervallo che contiene iteramente $[a, b]$ può stare solo nel sottoalbero sinistro. Analogamente un eventuale intervallo contenuto iteramente in $[a, b]$ non può stare nel sottoalbero destro perchè altrimenti tale intervallo sarebbe anche contenuto interamente in $[c, d]$, che contraddice la definizione di Interval Tree. La ricerca prosegue quindi nel sottoalbero sinistro.

- $c < a$ e $d < b$. In questo caso un eventuale intervallo contenuto iteramente in $[a, b]$ può stare solo nel sottoalbero destro. Analogamente un eventuale intervallo che contiene iteramente $[a, b]$ non può stare nel sottoalbero sinistro perchè altrimenti tale intervallo conterrebbe interamente anche $[c, d]$, che contraddice la definizione di Interval Tree. La ricerca prosegue quindi nel sottoalbero destro.

In base a tale osservazione, l'algoritmo è il seguente:

Algoritmo `InsertCheck(`a`,`b`,`T`)`

Input: Intervallo $[a, b]$, Interval Tree T
Output: YES/NO se $[a, b]$ può/non può essere inserito in T

$v \longleftarrow T$.`root`();
while (`!`T.`isInternal`(v)) **do**
  $[c, d] \longleftarrow v$.`getEelement().getValue()`;
  if (($c \leq a \leq b \leq d$) `OR` ($a \leq c \leq d \leq b$)) **then return** NO;
  if ($a < c$) **then** $v \longleftarrow T$.`left`(v);
  else $v \longleftarrow T$.`right`(v);
return YES

L'algoritmo verifica l'inseribilità dell'intervallo passato come input lungo un percorso radice-foglia dell'albero.

c. Al caso pessimo, il numero di iterazioni del ciclo **while** è pari all'altezza dell'albero e in ciascuna iterazione si esegue un numero costante di operazioni. La complessità dell'algoritmo è quindi $\Theta(h)$, dove h è l'altezza di T.

□

Problema 6 *Scrivere una versione ricorsiva dell'algoritmo* `InsertCheck` *sviluppato per il Problema 5.b, che mantenga la stessa complessità.*

Soluzione. La versione ricorsiva dell'algoritmo è la seguente (si assume che all'inizio venga invocato con $v = T$.root():

Algoritmo `InsertCheck(`a`,`b`,`v`,`T`)`

Input: Intervallo $[a, b]$, Interval Tree T, nodo $v \in T$
Output: YES/NO se $[a, b]$ può/non può essere inserito in T_v

if T.`isExternal`(v) **then return** YES;
$[c, d] \longleftarrow v$.`getEelement().getValue()`;
if (($c \leq a \leq b \leq d$) `OR` ($a \leq c \leq d \leq b$)) **then return** NO;
if ($a < c$) **then return** `InsertCheck`$(a, b, T, T$.`left`$(v))$;
else return `InsertCheck`$(a, b, T, T$.`right`$(v))$;

□

Problema 7 *Sia T un albero binario di ricerca di altezza h contenente n entry con chiavi distinte. Si supponga di avere per ogni nodo v una variabile v*`.size` *che memorizza il numero di entry presenti nel sottoalbero T_v. Sviluppare e analizzare un algoritmo ricorsivo*

`FindEntry(v,R)` *che dato un nodo* $v \in T$ *e un intero* R*, con* $1 \leq R \leq v$.size*, restituisca la entry di rango* R *tra le entry in* T_v*, ovvero quella che occuperebbe la posizione* R *nella sequenza delle entry di* T_v *ordinate per chiave. (La complessità va analizzata supponendo di invocare l'algoritmo con* $v =$ `T.root()`.)

Soluzione. L'algoritmo è il seguente:

Algoritmo `FindEntry`(v, R)

Input: nodo $v \in T$, intero $R \in [1, v.\texttt{size}]$
Output: entry di rango R in T_v

if (T`.isExternal(`v`)`) **then** segnala un errore;
$x \longleftarrow T.\texttt{left}(v).\texttt{size}$;
if ($R = x + 1$) **then return** v`.getElement()`;
if ($R < x + 1$) **then return** `FindEntry`$(T.\texttt{left}(v), R)$;
else return `FindEntry`$(T.\texttt{right}(v), R - (x + 1))$;

La correttezza dell'algoritmo discende dal fatto che il rango della entry nel nodo v è pari a $x + 1$, dove x è il numero di entry nel sottoalbero sinistro si v. Si noti in particolare che se $R > x + 1$ allora la entry cercata sarà nel sottoalbero destro di v e in tale sottoalbero dovrà avere rango $R - (x + 1)$. Per quanto riguarda la complessità, la struttura dell'algoritmo è la stessa di `TreeSearch`, e ogni invocazione ricorsiva esegue $\Theta(1)$ operazioni, escludendo le ulteriori invocazioni ricorsive al suo interno. In modo analogo all'analisi di `TreeSearch` si dimostra che la complessità di `FindEntry(T.root(),R)` è $\Theta(h)$, dove h è l'altezza di T. □

Problema 8 *Sia* T *un albero binario di ricerca dove ogni nodo* $v \in T$ *memorizza in una variabile* v`.size` *il numero di entry presenti nel sottoalbero* T_v *(inclusa quella in* v*). Progettare un algoritmo ricorsivo che conti quante entry in* T *hanno chiave* $\leq k$*, e analizzarne la complessità.*

Soluzione. L'algoritmo è il seguente e verrà invocato a partire dalla radice dell'albero.

Algoritmo `CountLE(`k`,`v`)`

Input: chiave k, nodo $v \in T$
Output: numero entry in T_v con chiave $\leq k$

if (T`.isExternal(`v`)`) **then return** 0;
if (v`.getElement().getKey()`$> k$) **then return** `CountLE(`k`,`T`.left(`v`))`;
else return $1 + T.\texttt{left}(v).\texttt{size}$+`CountLE(`$k$`,`$T$`.right(`$v$`))`;

La struttura dell'algoritmo e la sua analisi sono del tutto analoghe a quelle di `TreeSearch`. Se ne deduce quindi che, quando invocato dalla radice di un albero di altezza h, `CountLE` ha complessità $\Theta(h)$. □

Problema 9 *Risolvere il Problema 8 modificato per contare quante entry hanno chiave* $\geq k$*.*

Soluzione. È sufficiente una modifica molto semplice della soluzione del Problema 8, e la si lascia come esercizio. □

Problema 10 *Risolvere il Problema 8 usando un algoritmo iterativo.*

Soluzione. L'algoritmo è il seguente.

Algoritmo `CountLE(`k`,`T`)`

Input: chiave k, albero binario di ricerca T
Output: numero entry in T con chiave $\leq k$

$v \longleftarrow T$.`root`(); count $\longleftarrow 0$;
while (T.`isInternal`(v)) **do**
  if (v`.getElement().getKey()`$> k$) **then** $v \longleftarrow T$.`left`(v);
  else `count` $\longleftarrow$ `count` $+ 1 +$ T.`left`(v).`size`;
  $v \longleftarrow T$.`right`(v);
return count

La complessità è ovviamente determinata dal ciclo **while** che, al caso pessimo, esegue un numero di iterazioni pari all'altezza h dell'albero. Dato che in ciascuna iterazione si esegue un numero costante di operazioni, la complessità è $\Theta(h)$. □

Problema 11 *Risolvere il Problema 8 assumendo che al posto della variabile* v.size *si abbia a disposizione un metodo* T.size(v) *che restituisce il numero di entry in* T_v*, con complessità lineare nel valore restituito.* (**Suggerimento:** *esprimere la complessità come somma di due termini, uno dei quali è il costo aggregato di tutte le invocazioni del metodo* size.)

Soluzione. L'algoritmo è identico a quello riportato nella soluzione del Problema 8, con l'unica differenza che si sostituisce T.`left`(v).`size` con l'invocazione del metodo size (T.`size`(T.`left`(v))). Per quanto riguarda la complessità dell'algoritmo, quando invocato a partire dalla radice dell'albero, facciamo le seguenti osservazioni:

- Escludendo le invocazioni del metodo T.`size`($\cdot$), l'algoritmo ha la stessa complessità di quello della soluzione del Problema 8, ovvero $\Theta(h)$ dove h è l'altezza dell'albero.
- Il costo aggregato di tutte le invocazioni del metodo T.`size`($\cdot$) è proporzionale alla somma del numero di entry nei sottoalberi su cui viene invocato. Dato che il metodo viene invocato in sottoalberi diversi, e in questi sottoalberi tutte le entry hanno chiave $\leq k$, il costo aggregato di tutte le invocazioni del metodo T.`size`($\cdot$) è limitato superiormente dal numero totale m di entry con chiave $\leq k$ in T.

Dalle osservazioni, segue immediatamente che la complessità dell'algoritmo è $O(h + m)$. □

Problema 12 *Come cambia l'algoritmo sviluppato per il Problema 8 se non si hanno a disposizione né le variabili né il metodo* size*?*

Soluzione. L'algoritmo è identico a quello riportato nella soluzione del Problema 8, con l'unica differenza che si sostituisce T.`left`(v).`size` con l'invocazione ricorsiva `CountLE(`k`,`T`.left(`v`))`. Per completezza si riporta lo pseudocodice.

Algoritmo `CountLE(`k`,`v`)`

Input: chiave k, nodo $v \in T$
Output: numero entry in T_v con chiave $\leq k$

if (`T.isExternal(`v`)`) **then return** 0;
if (`v.getElement().getKey()`$> k$) **then return** `CountLE(`k`,`T`.left(`v`))`;
else return 1+`CountLE(`k`,`T`.left(`v`))`+`CountLE(`k`,`T`.right(`v`))`;

Analisi di complessità (non richiesta). Operiamo analogamente a quanto fatto nella soluzione del Problema 11. Consideriamo anzitutto la complessità dell'algoritmo (quando invocato a partire dalla radice dell'albero) escludendo le invocazioni ricorsive `CountLE(`k`,`T`.left(`v`))` inserite in sostituzione di T.`left`(v).`size` nella soluzione del Problema 8. Ragionando come nella soluzione del Problema 11, si conclude che tale complessità è $\Theta(h)$, dove h è l'altezza dell'albero. Analizziamo ora il costo aggregato di tutte le invocazioni ricorsive `CountLE(`k`,`T`.left(`v`))` inserite in sostituzione di T.`left`(v).`size` nella soluzione del Problema 8. Si osservi che ciascuna di queste invocazioni è fatta su un sottoalbero che contiene solo entry con chiave $\leq k$. In questo caso specifico, `CountLE` visita tutto il sottoalbero e la sua complessità sarà proporzionale al numero di nodi del sottoalbero che, essendo l'albero binario proprio, è a sua volta proporzionale al numero di nodi interni, ovvero al numero di entri del sottoalbero. Quindi, il costo aggregato di tutte le invocazioni ricorsive `CountLE(`k`,`T`.left(`v`))` inserite in sostituzione di T.`left`(v).`size` è asintoticamente lo stesso di tutte le invocazioni del metodo T.`size`($\cdot$) nella soluzione del Problema 11, ovvero $O(m)$, dove m è il numero di entry con chiave $\leq k$ in T. Concludendo, la complessità dell'algoritmo risulta essere $O(h+m)$, come nella soluzione del Problema 11. □

Problema 13 *Sia T un albero binario di ricerca le cui entry rappresentano studenti di un'università. Ogni studente è associato a una entry (k, x), dove k è la matricola, e x indica se lo studente è straniero ($x = 1$) o italiano ($x = 0$). Per ogni nodo $v \in T$ esiste un intero v.`numStr` che riporta il numero di studenti stranieri in T_v (sottoalbero con radice v). Si vuole progettare un algoritmo ricorsivo* `MinMatStraniero` *per determinare* la più piccola matricola di uno studente straniero in T. *Se non ci sono studenti stranieri in T_v l'algoritmo restituisce* `null`.

a. *Descrivere tramite pseudocodice la generica invocazione di* `MinMatStraniero`*, specificandone con attenzione l'input e l'output.*

b. *Analizzare la complessità di* `MinMatStraniero`

Soluzione.

a. L'algoritmo è basato sulle seguenti osservazioni.

- Se v è foglia o v.`numStr`=0, non ci sono studenti (stranieri) in T_v
- Se v è interno si hanno i seguenti casi. Se nel sottoalbero sinistro di v c'è almeno uno studente straniero, allora lo studente straniero con la matricola più piccola è in tale sottoalbero, altrimenti è quello associato a v (se straniero), oppure va cercato, se esiste, nel sottoalbero destro.

Lo pseudocodice è il seguente:

Algoritmo `MinMatStraniero`(T, v)

Input: Albero Binario di Ricerca T con le ipotesi date, nodo $v \in T$
Output: Chiave minima k di una entry $(k, x) \in T_v$ con $x = 1$, se esiste, o null

if ((`T.isExternal(v)`) `OR` (`v.numStr=0`)) **then return** `null`;
$m \longleftarrow$ `T.left(v).numStr`;
$x \longleftarrow$ `v.getElement().getValue()`;
if ($m > 0$) **then return** `MinMatStraniero(T,T.left(v))`;
else
  if ($x = 1$) **then return** `v.getElement().getKey()`;
  else return `MinMatStraniero(T,T.right(v))`;

b. La struttura dell'algoritmo è la stessa di `TreeSearch`, e ogni invocazione ricorsiva esegue $\Theta(1)$ operazioni, escludendo le ulteriori invocazioni ricorsive al suo interno. In modo analogo all'analisi di `TreeSearch` si dimostra che la complessità è $\Theta(h_v)$, dove h_v è l'altezza di T_v. (Se v è la radice di T la complessità è $\Theta(h)$, dove h è l'altezza dell'albero.)

Si lascia come esercizio la scrittura di una versione iterativa di `MinMatStraniero`. □

Problema 14 *Progettare e analizzare un algoritmo iterativo efficiente che determini l'altezza di un* $(2, 4)$*-Tree.*

Soluzione. Dall'osservazione che tutte le foglie di un (2,4)-Tree sono alla stessa profondità, che è pari all'altezza dell'albero, deriva il seguente algoritmo:

Algoritmo `24Height`(T)

Input: (2,4)-Tree T
Output: altezza di T

$h \longleftarrow 0$; $v \longleftarrow$ `T.root()`;
while (`T.isInternal(v)`) **do**
  $v \longleftarrow$ figlio di v piu' a sx;
  $h \longleftarrow h + 1$
return h

Dato che al di fuori del ciclo **while** e in ciascuna sua iterazione l'algoritmo esegue $\Theta(1)$ operazioni, la complessità è proporzionale al numero di iterazioni del **while**. Si osservi che il livello del nodo v nell'albero scende di 1 a ogni iterazione del **while**, e che il ciclo termina quando v raggiunge le foglie. Deduciamo quindi che la complessità è proporzionale all'altezza di T, ed è quindi $\Theta(\log n)$. □

Problema 15 *Sia* T *un* $(2, 4)$*-Tree le cui entry hanno chiavi distinte e dove ogni nodo* $v \in T$ *memorizza in una variabile* `v.size` *il numero di entry presenti nel sottoalbero* T_v *(incluse quelle in* v*). Progettare un algoritmo ricorsivo* `24CountLE` *che conti quante entry in* T *hanno chiave* $\leq k$*, e analizzarne la complessità.*

Soluzione. L'idea è di ispirarsi all'analogo algoritmo progettato per l'albero binario di ricerca. Sviluppiamo quindi un algoritmo `24CountLE(`v`,`k`)` che invocato su un nodo $v \in T$ restituisce il numero di entry in T_v con chiave $\leq k$. Basterà poi invocarlo con $v =$ `T.root()`. L'algoritmo è il seguente:

Algoritmo `24CountLE(`k`,`v`)`

Input: nodo v di un (2,4)-Tree T, chiave k
Output: numero di entry in T_v con chiave $\leq k$

if (`T.isExternal(`v`)`) **then return** 0;
Siano (k_1,x_1),..., (k_{d-1},x_{d-1}) le entry in v, con $k_1 < \ldots < k_{d-1}$;
Siano v_1, v_2, $\cdots$, v_d i figli di v;
$j \longleftarrow$ massimo indice tale che $k_j \leq k$; `/* `$j = 0$ `se` $k_1 > k$ `*/`
$C \longleftarrow 0$;
for $i \longleftarrow 1$ to j **do** $C \longleftarrow C + v_i$`.size` $+ 1$;
if $(k_j < k)$ **then** $C \longleftarrow C$+24CountLE(k, v_{j+1});
return C

La struttura dell'algoritmo e la sua analisi sono del tutto analoghe a quelle di `MWTreeSearch`. Se ne deduce quindi che `24CountLE` ha complessità $O(\log n)$ quando invocato dalla radice di un (sotto)albero con n entry. □

Problema 16 *Risolvere il Problema 15 tramite un algoritmo iterativo.*

Soluzione. L'algoritmo è il seguente

Algoritmo `24CountLE-IT` (k, T)

Input: (2,4)-Tree T, chiave k
Output: numero di entry in T con chiave $\leq k$

$v \longleftarrow T$`.root()`;
$C \longleftarrow 0$;
while (`T.isInternal(`v`)`) **do**
  Siano (k_1,x_1),..., (k_{d-1},x_{d-1}) le entry in v, con $k_1 < \ldots < k_{d-1}$;
  Siano v_1, v_2, $\cdots$, v_d i figli di v;
  $j \longleftarrow$ massimo indice tale che $k_j \leq k$; `/* `$j = 0$ `se` $k_1 > k$ `*/`
  $C \longleftarrow 0$;
  for $i \longleftarrow 1$ to j **do** $C \longleftarrow C + v_i$`.size` $+ 1$;
  if $(k_j = k)$ **then return** C;
  else $v \longleftarrow v_{j+1}$;
return C

La complessità dell'algoritmo è proporzionale al numero di iterazioni del ciclo **while** che sono al più pari all'altezza dell'albero, dato che ad ogni iterazione il nodo v scende di un livello. La complessità è quindi $O(\log n)$. □

Problema 17 *Risolvere il Problema 15, assumendo che al posto della variabile* v`.size` *si abbia a disposizione un metodo* T`.size(`v`)` *che restituisce il numero di entry in* T_v*, con complessità lineare nel valore restituito.*

Soluzione. Riscriviamo l'algoritmo `24CountLE` sostituendo a ogni uso della variabile v`.size` una chiamata al metodo T`.size(`v`)`.

Algoritmo `24CountLE(`k`,`v`)`

Input: (2,4)-Tree T, chiave k
Output: numero di entry in T con chiave $\leq k$
if (`T.isExternal(`v`)`) **then return** 0;
Siano (k_1,x_1),..., (k_{d-1},x_{d-1}) le entry in v, con $k_1 < \ldots < k_{d-1}$;
Siano v_1, v_2, $\cdots$, v_d i figli di v;
$j \longleftarrow$ massimo indice tale che $k_j \leq k$; `/* ` $j = 0$ `se` $k_1 > k$ `*/`
$C \longleftarrow 0$;
for $i \longleftarrow 1$ to j **do** $C \longleftarrow C + T$.`size`$(v_i) + 1$;
if $(k_j < k)$ **then** $C \longleftarrow C$+24CountLE(k, v_{j+1});
return C

Supponiamo di invocare l'algoritmo a partire dalla radice di un albero con n entry, di cui m hanno chiave $\leq k$. Facciamo le seguenti osservazioni

- Escludendo le invocazioni del metodo T.`size`$(\cdot)$, l'algoritmo ha la stessa complessità di quello della soluzione del Problema 15, ovvero $\Theta(\log n)$.
- Il costo aggregato di tutte le invocazioni del metodo T.`size`$(\cdot)$ è proporzionale alla somma del numero di entry nei sottoalberi su cui viene invocato. Dato che il metodo viene invocato in sottoalberi diversi, e in questi sottoalberi tutte le entry hanno chiave $\leq k$, il costo aggregato di tutte le invocazioni del metodo T.`size`$(\cdot)$ è limitato superiormente dal numero totale m di entry con chiave $\leq k$ in T.

Si deriva immediatamente che la complessità dell'algoritmo è $O(m + \log n)$. □

Problema 18 *Siano T e U due $(2,4)$-Tree di altezza h e tali che la massima chiave in T è* minore *della minima chiave in U.*

a. Progettare un algoritmo di complessità $O(h)$ per fondere T e U in un unico $(2,4)$-Tree TU.

b. Dire quali valori può assumere l'altezza di TU, e se la radice contiene una o più entry.

Soluzione.

a. È sufficiente creare una nuova radice contenente la entry con chiave massima ($e_{\max}$) di T, rendere T e U figli sinistro e destro, rispettivamente, della nuova radice, e rimuovere $e_{\max}$ dal nodo in cui si trovava in origine. Lo pseudocodice è il seguente.

Algoritmo `24TreeMergeEq`(T, U)

Input: (2,4)-Tree T, U di altezza h, max chiave in $T <$ min chiave in U
Output: (2,4)-Tree TU fusione di T e U

$v \longleftarrow T$.root();
$z \longleftarrow$ figlio più a destra di v;
while T`.isInternal(`z`)` **do**
  $v \longleftarrow z$;
  $z \longleftarrow$ figlio più a destra di v
$e_{\max} \longleftarrow$ entry con chiave max in v;
Crea la radice r di TU contenente $e_{\max}$;
Rendi T e U figli di r, a sx e dx di $e_{\max}$, rispettivamente;
TU.`Delete`$(e_{\max}, v)$;
return TU

(Il metodo `Delete(`$e_{\max}$`, v)` è stato descritto a lezione nella implementazione di `remove(k)`.)
L'algoritmo ha complessità $\Theta(h)$ in quanto:

- il ciclo **while** esegue $\Theta(h)$ iterazioni, ciascuna delle quali richiede $\Theta(1)$ operazioni;
- al di fuori del **while** si eseguono $\Theta(1)$ operazioni;
- l'invocazione del metodo `Delete` richiede $\Theta(h)$ operazioni al caso pessimo dato che l'altezza di TU (prima di invocare il metodo) è $h+1$, e, come visto a lezione, la complessità di `Delete` è proporzionale all'altezza del (2,4)-Tree.

b. Sia h l'altezza di T e U. Il (2,4)-Tree risultante può avere altezza h, se l'underflow a seguito della rimozione di $e_{\max}$ si propaga sino alla radice, oppure $h+1$. Nel secondo caso, la radice è sicuramente un 2-node, mentre ne primo caso può contenere un qualsiasi numero di entry (tra 1 e 3).

□

Problema 19 (Esercizio C-10.14 [GTG14]) *Siano T e U due $(2,4)$-Tree contenenti rispettivamente n ed m entry e tali che la massima chiave in T è* minore *della minima chiave in U. Progettare un algoritmo di complessità $O(\log n + \log m)$ per fondere T e U in un unico $(2,4)$-Tree TU.* **Suggerimento:** *Se i due alberi hanno altezze diverse, fondere l'albero di altezza minore con un opportuno sottoalbero dell'altro di uguale altezza (utilizzando l'algoritmo sviluppato per il Problema 18), e sostituirlo a tale sottoalbero.*

Soluzione. Sia `24height(X)` un algoritmo che calcola l'altezza h di un (2,4)-Tree X in tempo $\Theta(h)$. (Un tale algoritmo è stato sviluppato per il Problema 14.) Supponiamo di aver calcolato le altezze di T e U che denotiamo rispettivamente con h_T e h_U, e supponiamo anche che $h_T > h_U$. (L'altro caso è lasciato come esercizio.) Si osservi che dato che $h_T \in \Theta(\log n)$ e $h_U \in \Theta(\log m)$, utilizzando il metodo `24height(X)` menzionato prima, la complessità per il calcolo delle altezze rientra nel limite chiesto dall'esercizio. Sia `24TreeMergeEq(X,Y)` l'algoritmo di fusione di due (2,4)-Tree di uguale altezza sviluppato per il Problema 18. Si ricordi che il (2,4)-Tree risultato può avere la stessa altezza h di X e Y o altezza $h+1$ e che, nel secondo caso, la radice è un 2-node. La fusione di T e U viene eseguita come segue.

$v \longleftarrow T$.`root`();
for $i \longleftarrow 1$ **to** $h_T - h_U$ **do** $v \longleftarrow$ figlio più a dx di v;
$T' \longleftarrow$ `24TreeMergeEq`(T_v, U) ; `/* ` T_v `e` U `hanno altezza` h_U `*/`
$h \longleftarrow$ `24height`(T');
if $(h = h_U)$ **then** sostituisci T' al posto di T_v in T;
else
  `/* ` T' `ha altezza` $h_U + 1$ `e la sua radice è un 2-node */`
  $w \longleftarrow T$.`parent`(v);
  sia e la entry nella radice di T';
  siano A e B i due figli della radice di T';
  aggiungi e come entry più a dx in w;
  assegna A e B come figli di w a sx e dx di e eliminando v;
  if (w è un 5-node) **then** `Split`(w)`;`
return T

(Il metodo `Split(w)`, visto a lezione, serve per ripristinare le proprietà di (2,4)-Tree di T, dato che w è andato in overflow.) Per quanto riguarda la complessità, si noti che il ciclo **for** richiede tempo $O(h_T - h_U)$, le invocazioni di `24MergeEq(`T_v`,U)` e `24height(T')` richiedono tempo $O(h_U)$, e l'invocazione `Split(w)` richiede tempo $O(h_T - h_U)$. Dato che abbiamo assunto $h_T > h_U$, la complessità finale risulta

$$O(h_T) = O(h_T + h_U) = O(\log n + \log m).$$

□

Problema 20 (Variante dell'Esercizio C-11.35 del testo [GTG14]) *Sia T un $(2,4)$-Tree contenente n entry con chiavi distinte, dove ogni nodo $v \in T$ memorizza in una variabile v.*`size` *il numero di entry presenti nel sottoalbero T_v (incluse quelle in v). Si supponga che la radice di T contenga due entry con chiavi A e B $(A < B)$. Progettare un algoritmo iterativo che, dato $k < A$, conti quante entry in T hanno chiave nell'intervallo $I = [k, B]$, e analizzarne la complessità.*

Soluzione. Siano v_1, v_2 e v_3 i figli della radice e si osservi che le entry con chiave nell'intervallo I sono: le due entry nella radice, tutte le entry nel sottoalbero T_{v_2}, e tutte le entry con chiave $\geq k$ nel sottoalbero T_{v_1}. Sfruttando tale osservazione, l'algoritmo seguente prima inizializza un contatore C con il numero di entry nella radice e nel sottoalbero T_{v_2}, e poi va a contare le entry con chiave $\geq k$ nel sottoalbero T_{v_1}, aggiornando il contatore C di conseguenza.

Algoritmo `24CountInt`(T, A, B, k)

Input: (2,4)-Tree T la cui radice contiene 2 entry con chiavi $A < B$, e chiave $k < A$
Output: numero di entry in T con chiave in $[k, B]$

Siano v_1, v_2 e v_3 i figli della radice;
$C \longleftarrow 2 + v_2$.`size` ;
$u \longleftarrow v_1$;
while (`T.isInternal(`u`)`) **do**
  Siano $k_1 < k_2 < \cdots < k_{d-1}$ le chiavi in u ;
  Siano $u_1, u_2, \cdots, u_d$ i figli di u ;
  $j \longleftarrow$ minimo indice tale che $k_j \geq k$; `/*` $j = d$ `se` $k_{d-1} < k$ `*/`
  for $i \longleftarrow j + 1$ **to** d **do** $C \longleftarrow C + 1 + u_i$.`size`;
  $u \longleftarrow u_j$;
return C

La complessità dell'algoritmo è dominata da quella del ciclo while. In ciascuna iterazione del ciclo si esegue un numero costante di operazioni e il nodo u scende di un livello nell'albero. Quindi il numero totale di iterazioni, e di conseguenza la complessità dell'algoritmo, è proporzionale all'altezza $O(\log n)$ dell'albero. □

Problema 21 *Sia D un documento di N parole, rappresentato da un array $D[0 \div N-1]$.*

a. Progettare un algoritmo che, usando una Mappa d'appoggio, restituisca la parola con il massimo numero di occorrenze in D. Se ne esistono più di una, l'algoritmo ne restituisce una arbitraria tra esse.

b. Analizzare la complessità dell'algoritmo scegliendo una opportuna implementazione per la Mappa.

Soluzione. L'algoritmo è il seguente:

Algoritmo `MostFrequentWord`(D)

Input: Documento D di N parole: $D[i]$, $0 \leq i < N$
Output: Una della parole con il max numero di occorrenze in D

$M \leftarrow$ Mappa vuota di entry (parola,#occorrenze); maxCount $\leftarrow 0$;
for $i \leftarrow 0$ `to` $N - 1$ **do**
  count $\leftarrow M$.get($D[i]$);
  if (count = null) **then** count $\leftarrow 0$;
  M.put($D[i]$,++count);
  if (count > maxCount) **then**
    maxWord $\leftarrow D[i]$;
    maxCount $\leftarrow$ count
return maxWord

Si noti che il costo di ciascuna iterazione del **for** è dominato da quello della `get` e della `put` eseguite sulla mappa M. Analizziamo la complessità dell'algoritmo considerando due implementazioni alternative per M. Sia $m \leq N$ il numero di parole distinte contenute nel documento D.

- **(2,4)-Tree o Red-Black Tree**: dato che in ogni iterazione del ciclo for M contiene al più m entry, il costo di ciascuna invocazione dei metodi `get` e `put` è $O(\log m)$, e quindi la complessità dell'algoritmo risulta $O(N \log m)$, al caso pessimo.
- **Tabella Hash**: sotto l'ipotesi di uniform hashing e assumendo che la capacità del bucket array sia scelta garantendo un load factor $\lambda \in O(1)$ durante tutta l'esecuzione, il costo di ciascuna invocazione dei metodi `get` e `put` è $O(m)$, al caso pessimo, e $O(1)$, al caso medio. Quindi la complessità dell'algoritmo è $O(Nm)$ al caso pessimo, e $O(N)$ al caso medio.

Questo è uno dei casi in cui, in pratica, la Tabella Hash può essere la scelta migliore dal punto di vista delle prestazioni, nonostante che l'analisi al caso pessimo faccia preferire le implementazioni basate su (2,4)-Tree o Red-Black Tree. □