Dati e Algoritmi I (Pietracaprina)

Esercizi sulle Mappe

Problema 1 *Scrivere una versione iterativa di* `TreeSearch` *con la stessa specifica.*

Soluzione. L'algoritmo è il seguente:

Algoritmo `TreeSearch(`k`,` v`)`

Input: chiave k, nodo $v \in T$
Output: nodo $w \in T_v$ con chiave k, se esiste, o foglia nella posizione "giusta" per k

$w \leftarrow v$;
while `(true)` **do**
  if `(`T`.isExternal(`w`) OR (`w`.getElement().getKey()=`k`))` **then return** w;
  else
    if `(`$k <$ w`.getElement().getKey())` **then** $w \leftarrow T$.`left`(w) ;
    else $w \leftarrow T$.`right`(w) ;

La complessità 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 2 *Progettare un algoritmo iterativo che dato un nodo v di un albero binario di ricerca T e un intervallo $[A, B]$ di valori per le chiavi, restituisca un nodo $w \in T_v$ 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. L'algoritmo è una semplice modifica della versione iterativa di `TreeSearch`.

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

Input: intervallo $[A, B]$ di valori per le chiavi, nodo v di un ABR T
Output: nodo $w \in T_v$ con chiave $k \in [A, B]$, o w foglia nella posizione "giusta" per k

$w \leftarrow v$;
while `(true)` **do**
  if `(`T`.isExternal(`w`)OR(`w`.getElement().getKey()`$\in [A, B]$`))` **then return** w;
  else
    if `(`w`.getElement().getKey()`$< A$`)` **then** $w \leftarrow T$.`right`(w);
    else $w \leftarrow T$.`left`(w);

La complessità è la stessa di `TreeSearch`, ovvero $\Theta(h)$, con h l'altezza dell'albero. Come ulteriore esercizio, scrivere una versione ricorsiva di `TreeSearchAB`. □

Problema 3 *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 $c \leq a \leq b \leq d$.*

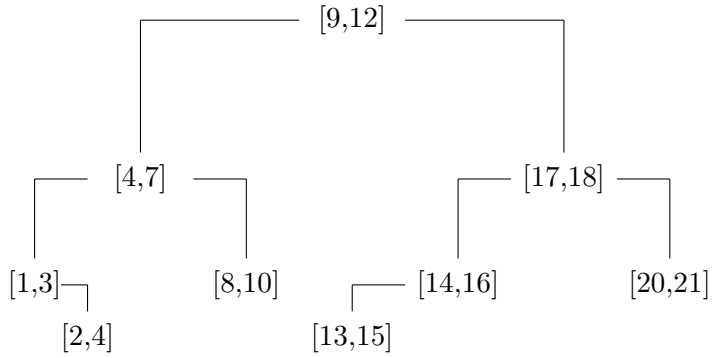
a. Disegnare l'Interval Tree costruito inserendo nell'ordine i seguenti intervalli, a partire dall'albero vuoto: $[9, 12]$, $[4, 7]$, $[17, 18]$, $[1, 3]$, $[8, 10]$, $[14, 16]$, $[20, 21]$, $[2, 4]$, $[13, 15]$.

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* $[a, b]$*. Determinare anche la complessità dell'algoritmo.*

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. 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 4 *Scrivere una versione ricorsiva dell'algoritmo* `InsertCheck` *sviluppato per il Problema 3 (Punto 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,T,v)

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 5 *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 $v_{\texttt{size}}$ 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 $1 \leq R \leq v_{\texttt{size}}$, 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`$(T, T.\texttt{left}(v), R)$;
else return `FindEntry`$(T, T.\texttt{right}(v), R - (x + 1))$;

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, si consideri la chiamata `FindEntry(T.root(),R)`, per un qualche R. Sia w il nodo contenete la entry cercata. Nell'esecuzione ci sarà una chiamata ricorsiva per ogni nodo nel percorso dalla radice a w, e in ciascuna chiamata si eseguiranno $O(1)$ operazioni, esclusa l'eventuale chiamata ricorsiva fatta al suo interno. La complessità sarà quindi $O(h)$, con h l'altezza dell'albero. □

Problema 6 *Sia T un albero binario di ricerca 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 (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 `CountLE` ha complessità $\Theta(h)$ quando invocato dalla radice di un (sotto)albero di altezza h. □

Problema 7 *Risolvere lo stesso problema dell'esercizio precedente usando un algoritmo non ricorsivo.*

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 ⟵ T.root();
count ⟵ 0;
while (T.isInternal(v)) do
    if (v.getElement().getKey()> k) then
        v ⟵ T.left(v)
    else count ⟵ count + 1 + T.left(v).size;
    v ⟵ 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 8 *Sia T un albero binario di ricerca le cui entry hanno chiavi distinte. Progettare un algoritmo ricorsivo che conti quante entry in T hanno chiave $\leq k$, e analizzarne la complessità. (N.B. Non si ha a disposizione la variabile* `size` *come nell'esercizio precedente.)*

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+ CountLE(k,T.left(v))+CountLE(k,T.right(v));
```

Analizziamo ora la complessità dell'algoritmo. Dato che il costo di ciascuna invocazione ricorsiva (escluse le invocazioni fatte al suo interno) è $O(1)$, la complessità dell'algoritmo è proporzionale al numero di nodi dell'albero della ricorsione, ovvero al numero di invocazioni ricorsive. Stimiamo tale numero riferendoci alla esecuzione di `CountLE(`k`,T.root())`, per una generica chiave k. A tale fine, consideriamo una generico nodo dell'albero della ricorsione associato alla invocazione `CountLE(`k`,`v`)`, per un qualche $v \in T$. Si osservi che:

- Se tutte le entry in T_v hanno chiave $\leq k$ allora l'algoritmo verrà alla fine invocato su ciascun $u \in T_v$.
- Se la chiave nel nodo v è $> k$, l'algoritmo richiama se stesso sul figlio sinistro di v.
- Se la chiave nel nodo v è $\leq k$, l'algoritmo richiama se stesso su entrambi i figli. Il figlio sinistro è radice di un sottoalbero che contiene solo entry con chiave $\leq k$, e, per il primo punto, l'algoritmo verrà alla fine invocato su tutti i nodi di tale sottoalbero.

Sia h l'altezza di T e s il numero di entry con chiave $\leq k$ in T. Possiamo definire una *dorsale* di nodi sui quali viene invocato l'algoritmo, che parte dalla radice e arriva a una foglia: se

un nodo v della dorsale ha chiave $> k$, il successore nella dorsale è il suo figlio sinistro, mentre se ha chiave $\leq k$ il successore nella dorsale è il suo figlio destro (ma in questo caso l'algoritmo sarà invocato anche sul figlio sinistro). È facile allora vedere l'algoritmo viene invocato ricorsivamente sui nodi della dorsale (che sono $\leq h$ dato che formano un percorso dalla radice-foglia) e su tutti i nodi a sinistra della dorsale che sono l'unione di sottoalberi T_u che contengono solo nodi con chiave $\leq k$, dove u non appartiene alla dorsale ma è figlio sinistro di un nodo della dorsale. Tali nodi a sinistra della dorsale sono al più s. Di conseguenza la complessità è $O(h + s)$. □

Problema 9 *Sia T un albero binario di ricerca contenente n entry con chiavi reali distinte. Descrivere un algoritmo* non ricorsivo *che 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 ⟵ T.root();
minPosKey ⟵ 0;
while (T.isInternal(v)) do
    if (v.getElement().getKey()> 0) then
        minPosKey ⟵ v.getElement().getKey();
        v ⟵ T.left(v)
    else v ⟵ T.right(v);
if (minPosKey> 0) 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. 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 10 *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). Progettare un algoritmo* ricorsivo `MinMatStraniero`(T, v) *che dato un nodo $v \in T$ restituisce* la più piccola matricola di uno studente straniero in T_v, *e analizzarne la complessità. Se non ci sono studenti stranieri in T_v l'algoritmo restituisce* `null`.

Soluzione. L'algoritmo è basato sulle seguenti osservazioni.

- Se v è foglia, 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, 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`)`) **then return** `null`;
$m \leftarrow T$`.left(`v`).numStr`;
$x \leftarrow v$`.getElement().getValue()`;
if ($m \geq 1$) **then return** `MinMatStraniero(`T`,`T`.left(`v`))`;
else
  if ($x = 1$) **then return** v;
  else return `MinMatStraniero(`T`,`T`.right(`v`))`;

La complessità è la stessa di `TreeSearch`, ovvero $\Theta(h)$, con h l'altezza di T_v, □

Problema 11 *Progettare e analizzare un algoritmo non ricorsivo efficiente che determini l'altezza di un (2,4)-Tree.*

Soluzione. L'algoritmo è il seguente:

Algoritmo `24height`(T)

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

$h \longleftarrow 0$; $v \longleftarrow T$`.root()`;
while (T`.isInternal(`v`)`) **do**
  $v \longleftarrow$ un qualsiasi figlio di v;
  $h \longleftarrow h + 1$
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 12 *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.*

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 un nuovo nodo r contenente $e_{\max}$;
Crea un (2,4)-tree TU con radice r e collega T e U come sottoalberi sx e dx di r ;
TU.`Delete`$(e_{\max}, v, R)$;
return TU

(Il metodo `Delete(`$e_{\max}$`, v, R)` 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, e al di fuori del ciclo si eseguono $\Theta(1)$ operazioni e una invocazione del metodo `Delete` che, come visto a lezione, ha complessità $\Theta(h)$

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 un 2-node.

□

Problema 13 (Esercizio C-10.14 del Goodrich-Tamassia, 5th Edition, 2010.) *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.*

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 11.) Supponiamo di aver calcolato le altezze di T e U che denotiamo rispettivamente con h_T e h_U, e supponiamo anche che $h_T > h_U$. Gli altri casi sono lasciati 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 bound chiesto dall'esercizio. Sia `24TreeMergeEq(X,Y)` l'algoritmo di fusione di due (2,4)-Tree di uguale altezza sviluppato per il Problema 12. 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 14 *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 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()`.

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

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(v_{j+1}, k);
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 15 *Risolvere il problema precedente tramite un algoritmo non ricorsivo.*

Soluzione. L'algoritmo è il seguente

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

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

È facile vedere che 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 16 *Analizzare l'algoritmo sviluppato per il Problema 14, assumendo che al posto della variabile* v.`size` *sia disponibile un metodo* T.`size`(v) *che restituisce sempre il numero di entry in* T_v *ma abbia una complessità proporzionale al valore restituito.*

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

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

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(v_{j+1}, k);
return C

Supponiamo di invocare l'algoritmo a partire dalla radice di un albero con n entry, di cui s hanno chiave $\leq k$. Si osservi che per ogni chiamata T.`size`(v_i) fatta dall'algoritmo il sottoalbero di T con radice v_i contiene solo entry con chiave $\leq k$ e quindi la complessità aggregata di tutte queste chiamate sarà $O(s)$. Il contributo alla complessità delle altre operazione è la stessa di quella dell'algoritmo `24CountLE` sviluppato nel problema precedente con l'uso della variabile v.`size`. Quindi, la complessità totale è $\Theta(s + \log n)$. □

Problema 17 (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 conti quante entry in T hanno chiave nell'intervallo $I = [k, B]$, con $k < A$, 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 `24CountI`(k, A, B)

Input: (2,4)-Tree T con radice 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. □