Alma Mater Studiorum - Università di Bologna

Campus di Cesena

Corso di Laurea in Ingegneria e Scienze Informatiche

Treap persistenti

Tesi di Laurea in Algoritmi e Strutture Dati

Relatore:

Prof. Vittorio Maniezzo

Presentata da:

Filippo Soldati

Anno Accademico 2019/2020

Indice

Elenco delle figure

Capitolo 1

Introduzione

Sommario

1.1 Introduzione

Nella vita del programmatore medio, non capita spessissimo di dover implementare un albero binario bilanciato. Tipicamente, capita solo una volta - durante il corso di "Algoritmi e Strutture Dati" all'università.

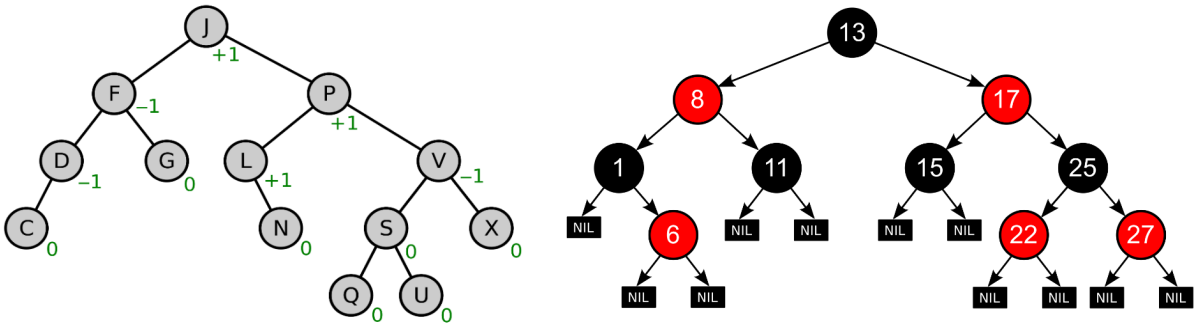

Figura 1.1: Esempi di alberi binari bilanciati.
[13][14]

Nonostante gli alberi binari di ricerca possano essere considerati una struttura dati piuttosto semplice, non si può sempre dire lo stesso della loro controparte bilanciata: basandosi spesso su insiemi di regole e proprietà non banali che raramente compaiono su altre strutture più semplici, non è facilissimo ricordarsene l'intera implementazione.

Esistono però una serie di contesti in cui sapere come implementare un albero binario bilanciato diventa una necessità - ad esempio, nell'implementare una libreria di runtime per un linguaggio puramente funzionale, o nel cercare di risolvere alcuni problemi di programmazione competitiva, in cui le funzionalità offerte dalle implementazioni di libreria non bastano.

1.1.1 Programmazione competitiva

La programmazione competitiva[10] è un gioco (o un "mind sport", che dir si voglia) in cui i partecipanti si sfidano nel programmare le soluzioni di una serie di problemi proposti.

Vengono organizzate competizioni di diverso tipo: dall'*International Collegiate Programming Contest* (ICPC) che viene organizzato su base annuale dal 1977 e a cui nel 2020 hanno partecipato squadre rappresentanti più di 3400 università di tutto il mondo; a gare online organizzate settimanalmente a cui partecipano anche più di 20 mila persone[1].

La programmazione competitiva ha subito una forte crescita negli ultimi anni[12] e di questo si può trovare un certo riscontro anche in Italia: se tipicamente solo aziende internazionali come Google o Facebook organizzavano competizioni in quest'ambito, dal 2019 anche alcune aziende italiane[2] organizzano eventi di questo tipo.

I problemi trattati possono riguardare quasi ogni aspetto dell'informatica, tuttavia la maggior parte di quelli proposti sono di natura logico/matematica e riguardano combina-

[1] Codeforces: Results of 2020

[2] Bending Spoons (Codeflows) e Reply (Reply Challnges)

toria, teoria dei grafi, strutture dati, geometria computazionale e analisi di stringhe.

Non in tutte le competizioni viene consentito l'utilizzo di codice scritto al di fuori del tempo di gara, né di consultare materiale - la maggior parte dei concorrenti si prepara quindi studiando gli algoritmi e le strutture dati che potrebbero essere più utili durante una competizione: essendo il tempo di gara limitato, non è importante solamente l'efficienza di un determinato algoritmo ma anche la difficoltà effettiva di implementazione.

Può capitare che per la soluzione di un problema sia necessario utilizzare un albero binario bilanciato ed in questi contesti utilizzare un *Treap* è lo standard de facto: una risposta breve potrebbe essere che questa struttura offre una implementazione particolarmente semplice e compatta, combinando le idee degli alberi binari di ricerca con quelle degli heap; pur garantendo performance comparabili con quelle degli altri alberi più complessi.

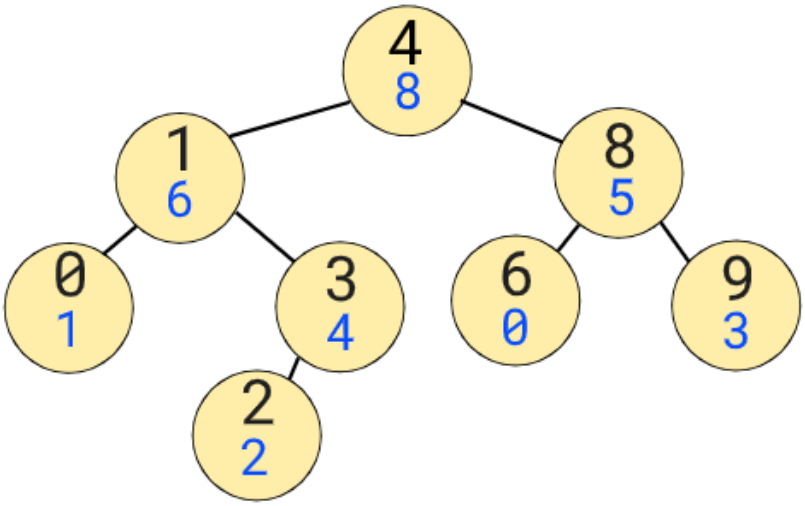

Figura 1.2: Rappresentazione di un treap.

Una risposta più lunga e completa è invece quello che cerco di fornire in questa tesi, che si pone l'obbiettivo di presentare e analizzare non solo i treap, ma anche le possibili varianti e le possibili estensioni che si possono utilizzare per riuscire a risolvere efficientemente una vasta gamma di problemi - il tutto mantenendo un buon compromesso tra difficoltà implementativa e l'effettiva performance ottenuta.

Capitolo 2

Treap

Sommario

In questo capitolo, viene illustrato il funzionamento dei *treap*, analizzandone alcune possibili implementazioni insieme alle possibili operazioni che possono essere supportate. Viene anche mostrata la variante dei *Randomized Binary Search Tree*, e degli alberi a chiave implicita.

2.1 Treap

Il ***Treap*** è un albero binario di ricerca bilanciato (*BBST*) introdotto nell'89 da Seidel e Aragon[1], il cui nome deriva dall'unione di *Tree* e *Heap*, in quanto il *Treap* mantiene le proprietà di entrambe le strutture: ciascun nodo possiede infatti oltre alla chiave anche

un secondo valore detto "priorità" - le chiavi vengono mantenute ordinate rispetto ad una visita *in-order* come in un regolare albero di ricerca binario (*BST*); mentre le priorità devono invece soddisfare la *heap property* - ciascun nodo deve avere sempre una priorità maggiore o uguale di quella dei suoi figli.

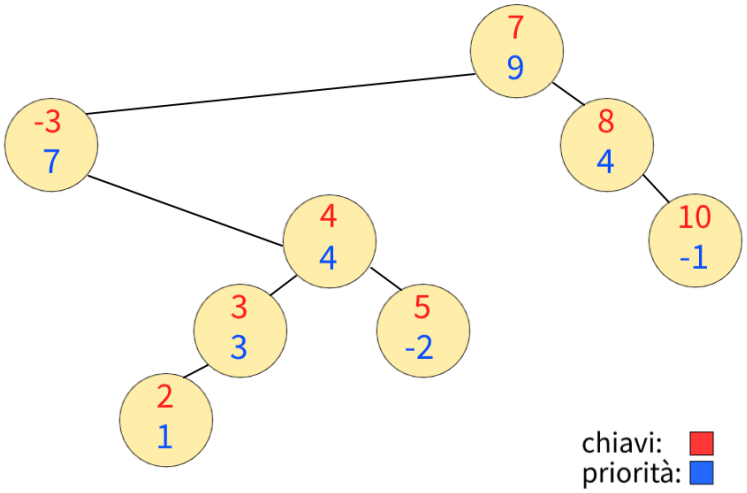

Figura 2.1: Rappresentazione grafica di un treap.

Abbiamo già detto che rispetto ad un BST tradizionale, sarà necessario aggiungere a ciascuna chiave inserita una relativa priorità - cerchiamo ora di capire come la scelta di questi valori possa influire sull'altezza dell'albero ottenuto.

Per farlo, iniziamo osservando questi esempi di sequenze di inserimenti, insieme ai treap così formati:

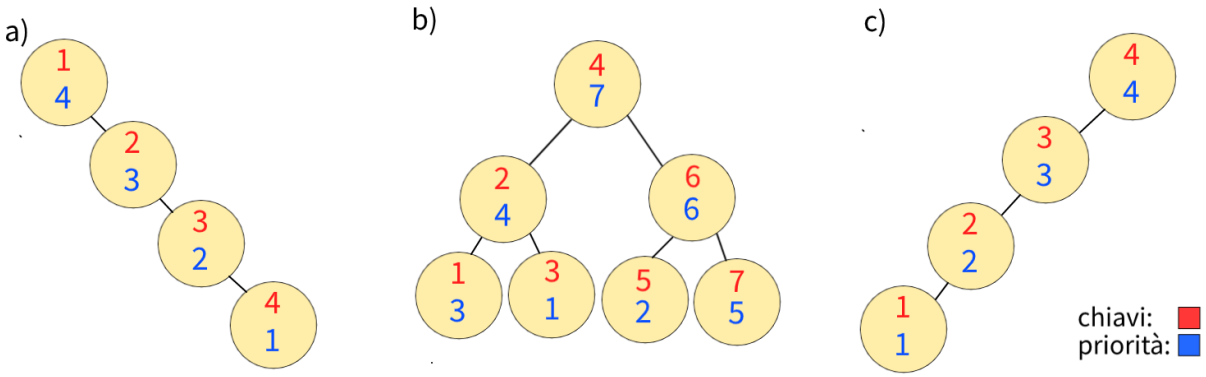

Figura 2.2: Rappresentazione grafica di tre possibili treap.

Possiamo subito notare alcune cose:

- Per ciascuno dei tre alberi, la disposizione proposta è l'unica a soddisfare sia "heap property" che "binary search property" per le coppie chiave/priorità date: questo avviene perché sono state utilizzate priorità tutte distinte tra loro, ma potrebbe non essere vero per priorità duplicate.

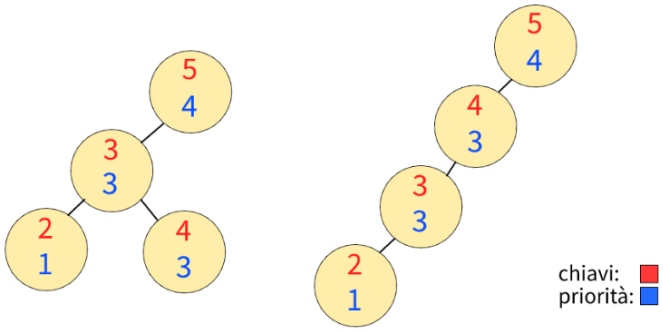

Figura 2.3: Esempio di un treap con priorità duplicate e due possibili configurazioni.

- Nonostante la heap property sia rispettata, possiamo comunque ottenre alberi degeneri, di altezza lineare.

- La radice dell'albero è sempre il nodo di priorità massima.

Estendendo la terza osservazione e assumendo di avere chiavi distinte, possiamo notare che l'albero ottenuto coincida con il BST che si ottiene inserendo i nodi in ordine decrescente di priorità: la radice di un BST è sempre il primo nodo inserito e ricorsivamente, divedendo i nodi in minori e maggiori dell'ultimo valore inserito, si hanno come figlio sinistro e destro rispettivamente il primo nodo inserito tra quelli minori ed il primo tra quelli maggiori.

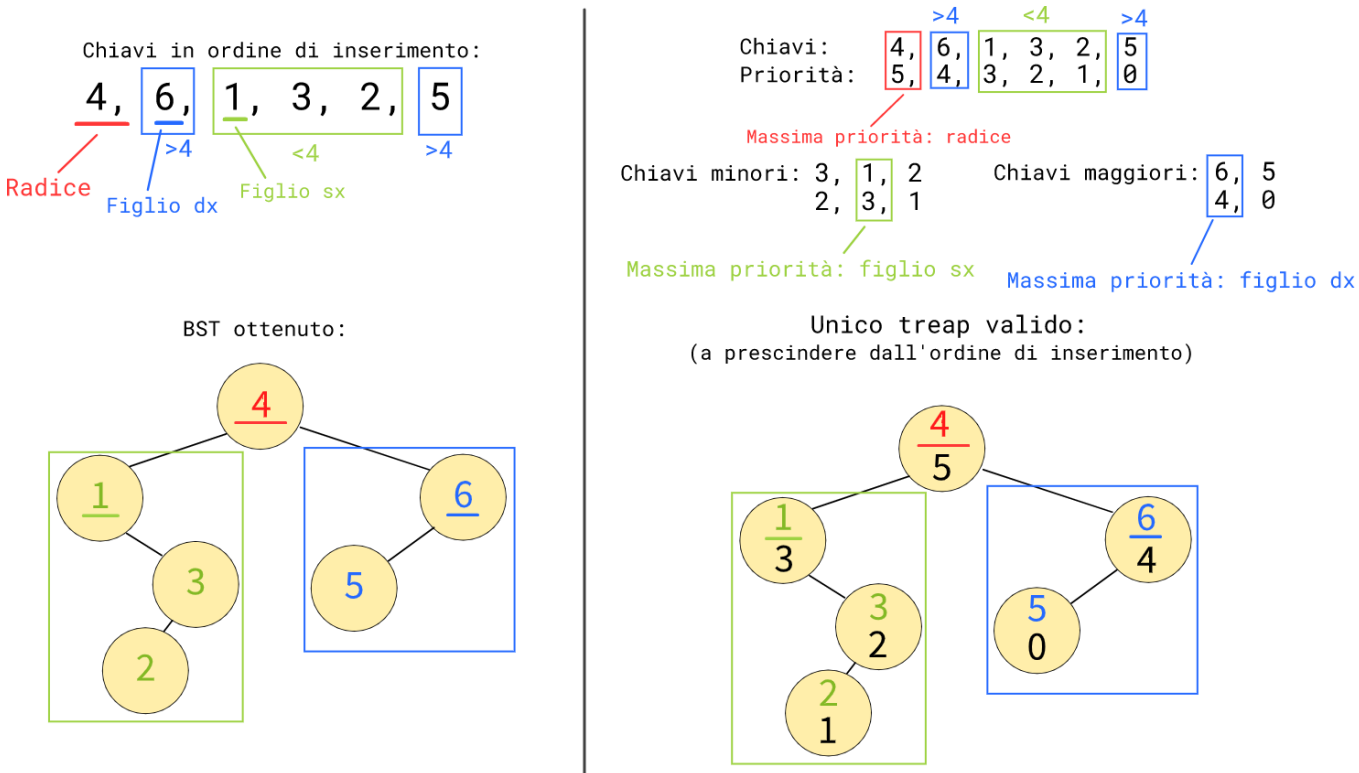

Figura 2.4: *BST* affiancato da un possibile *treap* corrispondente.

In modo analogo, in un treap avremo come radice il nodo di massima priorità e dividendo ricorsivamente le chiavi in minori e maggiori, avremo come figlio sinistro e destro i nodo di priorità massima rispettivamente tra i nodi con chiave minore e tra quelli con chiave maggiore.

A primo impatto, questa corrispondenza potrebbe non sembrare particolarmente utile, in quanto è noto che i BST possano avere altezza lineare. Tuttavia, i BST hanno un'altezza attesa logaritmica quando costruiti su chiavi casuali - ne consegue che se le priorità vengono assegnate in modo casuale (evitando il più possibilmente collisioni), allora l'altezza attesa sarà di $O(logN)$.

Assegnando quindi a ciascun nodo una priorità casuale, il treap garantisce un'altezza media di $O(logN)$ - nonostante rimanga possibile ottenere alberi di altezza $O(N)$, la probabilità che questo si verifichi tuttavia diminuisce esponenzialmente al crescere di N.

Condividendo le proprietà dei BST, la ricerca di un elemento all'interno di un treap avviene come in ogni altro albero binario di ricerca[7], ignorando completamente la priorità dei nodi:

Algorithm 1: Funzione di ricerca per BST e Treap.

```
SEARCH(root, k)
    node = root;
    while root ≠ null do
        if k = root.key then
            return [true]
        else if k < root.key then
            root = root.left
        else
            root = root.right
    return [false]
```

Tuttavia, la stessa cosa non si può dire per inserimento e cancellazione - vediamo ora due metodi diversi per poter implementare queste operazioni, continuando a garantire la "heap property".

2.1.1 Implementazione con rotazioni

Il primo metodo, più simile a quelli utilizzati nei BBST più comuni, si basa sull'utilizzo delle rotazioni:

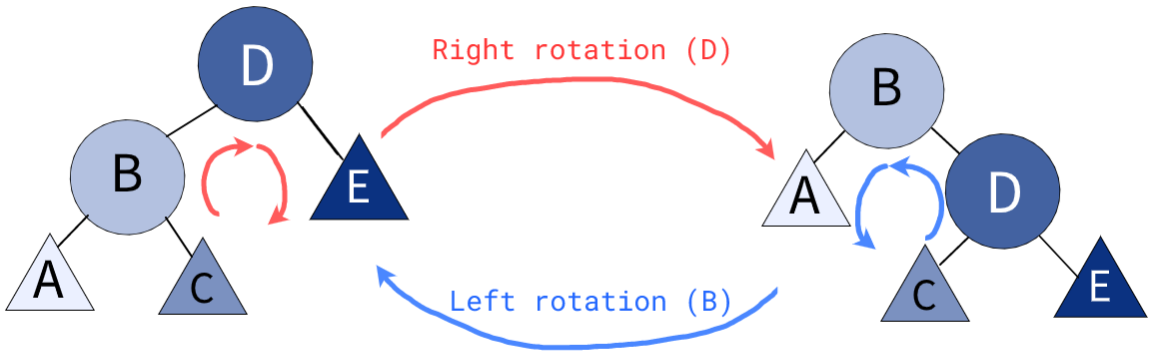

Figura 2.5: Operazioni di rotazione su albero binario.

Algorithm 2: Rotazioni per BST.

```
ROTATE-RIGHT(root)
    pivot = root.left;
    root.left = pivot.right;
    pivot.right = root;
    return pivot

ROTATE-LEFT(root)
    pivot = root.right;
    root.right = pivot.left;
    pivot.left = root;
    return pivot
```

Inserimento

In un BST, l'inserimento segue questo algoritmo:

- Partendo dalla radice, si confronta la chiave del nodo attuale - in maniera analoga alla ricerca, se la chiave è minore di quella che vogliamo inserire, il prossimo nodo da considerare sarà il figlio sinistro, altrimenti il destro
- Quando succede che il figlio su cui vorremmo spostarci per l'inserimento è inesistente, significa che abbiamo trovato la posizione in cui inserire il nostro nuovo nodo, che verrà quindi creato ed inserito. Un nodo appena inserito sarà quindi sempre una foglia, almeno inizialmente.

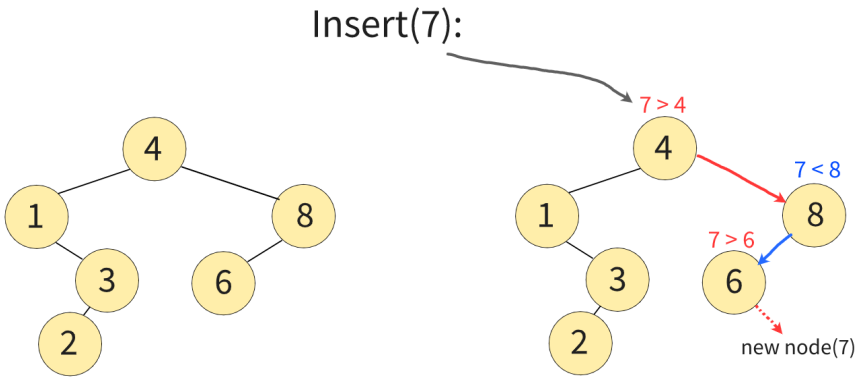

Figura 2.6: Esempio di inserimento in un BST.

Questo procedimento può essere implementato in modo sia ricorsivo che iterativo - per facilitare alcuni dei passaggi successivi, viene proposto uno pseudocodice ricorsivo:

Algorithm 3: Inserimento in un BST.

```
INSERT(root, k)
    if root = null then
        return new node(k)
    if k < root.key then
        root.left = INSERT(root.left, k);
        return root
    else
        root.right = INSERT(root.right, k);
        return root
```

Effettuando inserimenti in questo modo in un treap però, quello che si potrebbe verificare è che la priorità del nodo inserito sia superiore a quella del padre - in questo modo, l'"heap property" non verrebbe più mantenuta.

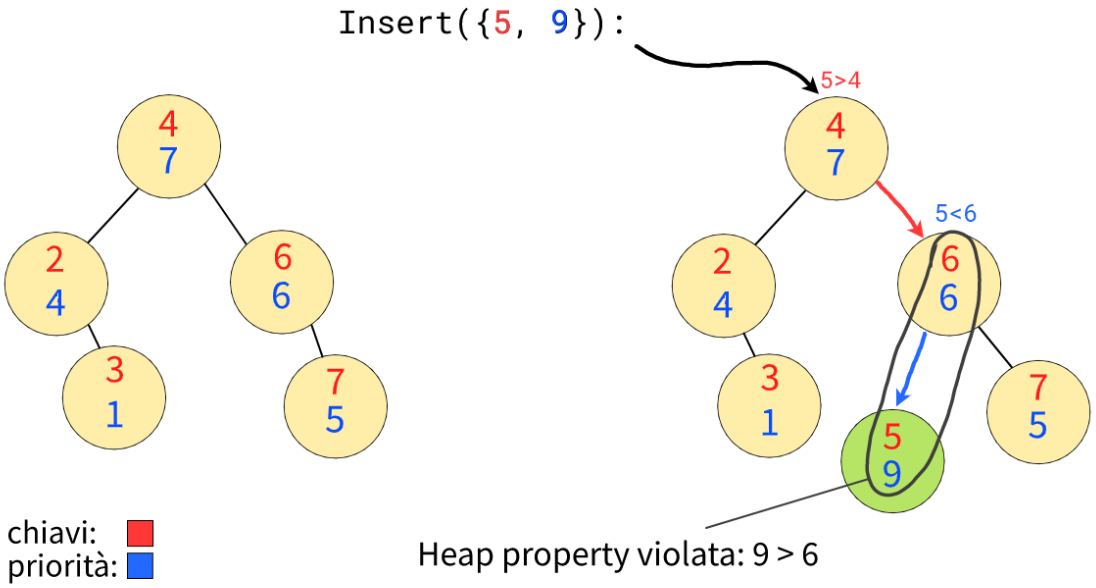

Figura 2.7: Esempio di inserimento non valido.

Applicando una rotazione sul nodo inserito in direzione del padre, il problema viene "localmente" risolto - ora, la heap property è garantita nel subtree che abbiamo appena ruotato, ma potrebbe non esserlo per il "padre attuale" del nodo inserito - ripetendo questo procedimento finché necessario, la "heap property" viene mantenuta per l'intero albero.

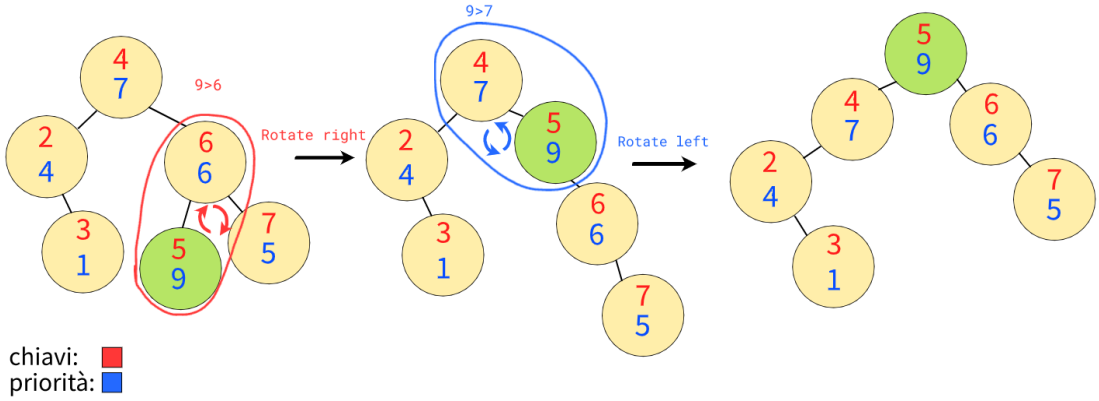

Figura 2.8: Rotazioni post inserimento.

La complessità dell'inserimento in un BST è di $O(h)$, a cui si vanno ad aggiungere le $O(h)$ rotazioni che possono essere necessarie per ripristinare la heap property - possiamo facilmente verificare che siano al più h poiché il nodo inserito sale di un "livello" per ciascuna rotazione effettuata. Siccome ci aspettiamo che l'altezza di un treap sia $O(logN)$, abbiamo una complessità attesa di $O(logN)$ per le operazioni di inserimento.

Algorithm 4: Inserimento in un Treap con rotazioni.

```
INSERT(root, x, y)
    if root = [null] then return new node(x, y);
    if x < root.key then
        root.left = INSERT(root.left, x, y);
        if root.left.priority > root.priority then return ROTATE-RIGHT(root);
        else return root;
    else
        root.right = INSERT(root.right, k);
        if root.right.priority > root.priority then return ROTATE-LEFT(root);
        else KwRetroot;
```

Cancellazione

In un BST, la cancellazione può essere implementata in più modi - un possibile metodo è il seguente:

- Viene identificato il nodo V da rimuovere, nello stesso modo con cui viene individuato nella ricerca

- Se V è una foglia, può essere eliminato direttamente in quanto la sua rimozione non ha effetto su altri nodi (escluso il padre, di cui verranno aggiornati i puntatori ai figli)

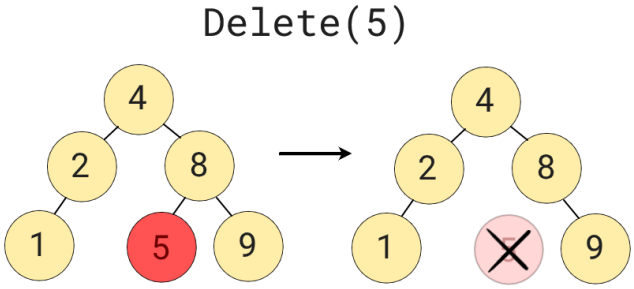

Figura 2.9: Rimozione di una foglia.

- Se V ha un solo figlio, V può essere rimosso solo dopo averlo rimpiazzato con suo figlio (aggiornando i puntatori dei figli del padre di V)

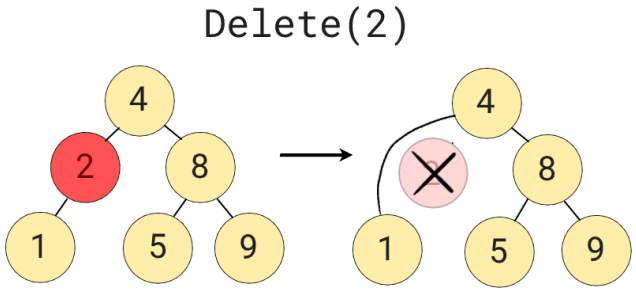

Figura 2.10: Rimozione di un nodo con un solo figlio.

- Se V ha due figli, scambiamo V con il suo predecessore (o successore) - siccome il suo predecessore ha al più un figlio, ora V si trova in una posizione dell'albero in cui può essere rimosso con una delle due opzioni sopra citate.

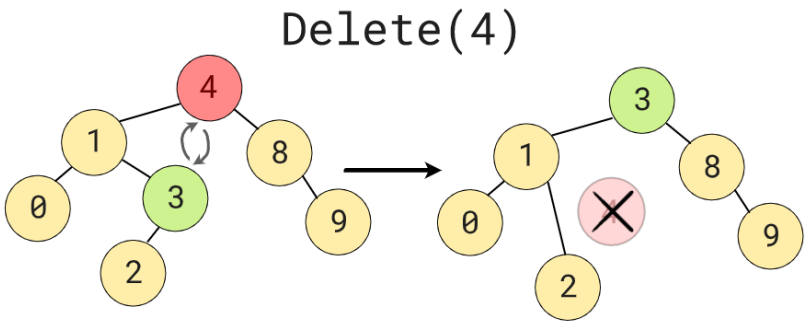

Figura 2.11: Rimozione di un nodo con entrambi i figli.

Nei primi due casi, non è necessario apportare modifiche all'algoritmo, in quanto nessuna delle due operazioni può invalidare la heap property. Tuttavia, nel terzo caso, è possibile che il predecessore che viene spostato al posto del nodo rimosso abbia una priorità minore di quella di almeno uno dei figli.

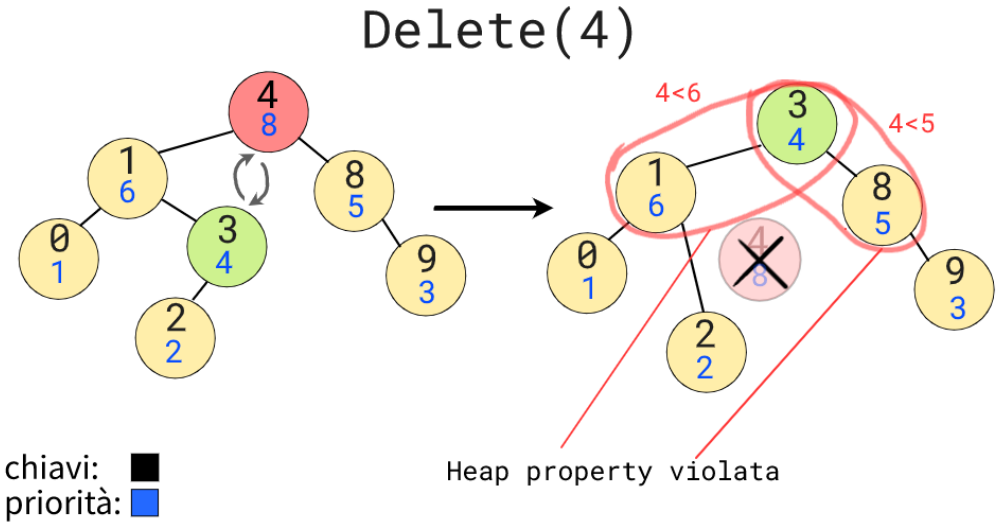

Figura 2.12: Esempio di rimozione che invalida la heap property.

Come per l'inserimento, la cosa può essere risolta applicando una serie di rotazioni. È importante prestare particolare attenzione al caso in cui un nodo abbia priorità inferiore di entrambi i suoi figli: in quel caso, sarà necessario ruotare in direzione del figlio di priorità minore, in modo da portare come radice quello di priorità maggiore.

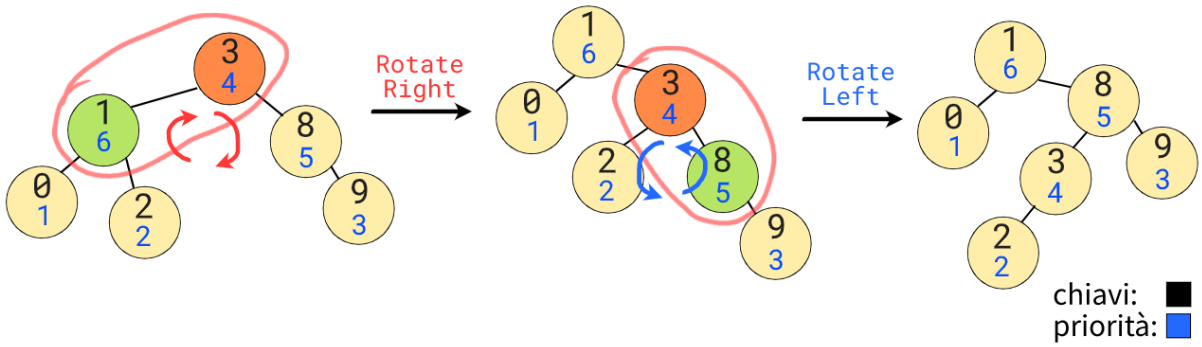

Figura 2.13: Rotazioni per ripristinare la heap property.

Lo stesso risultato si può però ottenere anche con un approccio leggermente differente: utilizzando le rotazioni, possiamo portare il nodo che dobbiamo eliminare verso il fondo all'albero. Rendendolo una foglia, o quando ha un singolo figlio, l'eliminazione potrà poi procedere senza problemi, e l'heap property non verrà invalidata (se non temporaneamente, durante la "discesa" del nodo da eliminare).

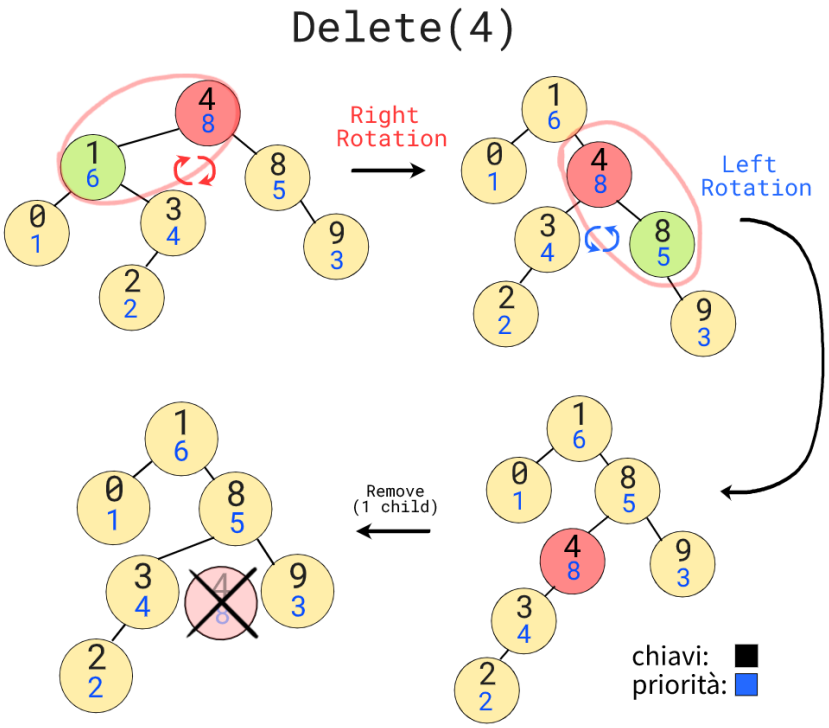

Figura 2.14: Rimozione basata su rotazioni.

Il codice della rimozione così implementata risulterà più semplice della versione precedentemente proposta:

Algorithm 5: Rimozione con rotazioni da un treap.

```
ERASE(root, k)
    if root = null then
        return null
    if root.key = k then
        if root.left = null then
            current = root.right;
            delete(root);
            return current;
        if root.right = null then
            current = root.left;
            delete(root);
            return current;
        if root.left.priority > root.right.priority then
            current =ROTATE-RIGHT(root);
            current.right =ERASE(current.right, k);
            return current;
        else
            current =ROTATE-LEFT(root);
            current.l =ERASE(current.l, k);
            return current;
```

La complessità della rimozione rimarrà di $O(h) = O(logN)$, in quanto la ricorsione scende di un livello ciascuna volta che viene chiamata.

2.1.2 Split - join

Nonostante la versione con le rotazioni non sia necessariamente troppo complessa, soprattutto se paragonata ad altre possibili implementazioni di *BBST*; uno dei vantaggi dei *treap* sta nella semplicità di un'altra implementazione, basata sulle operazioni di *split* e *join*.

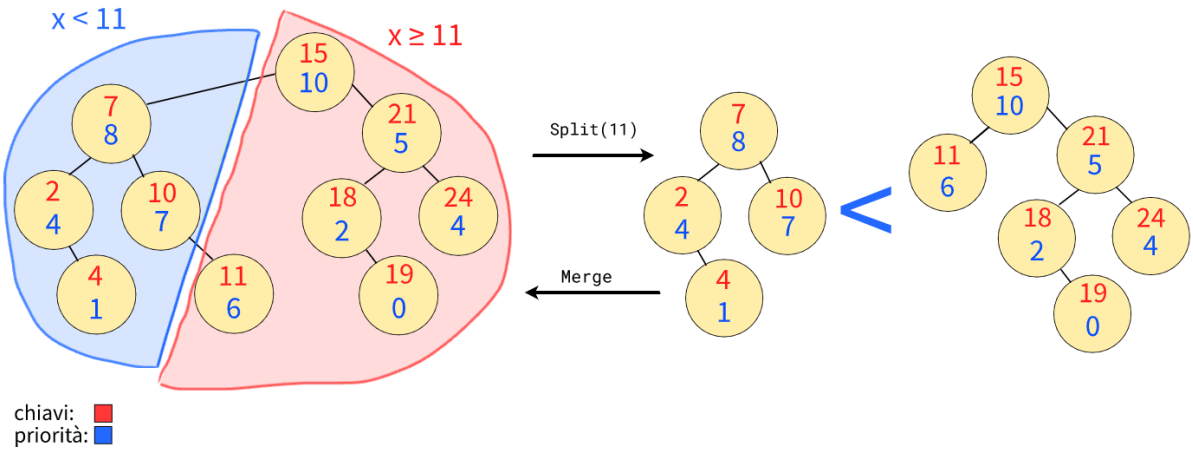

Figura 2.15: Split e join.

Split

L'operazione di *split* permette di spezzare un treap in due treap più piccoli T_1 e T_2, dividendo i nodi rispettivamente in minori e maggiori o uguali rispetto ad una chiave k; sempre con una complessità uguale a $O(logN)$.

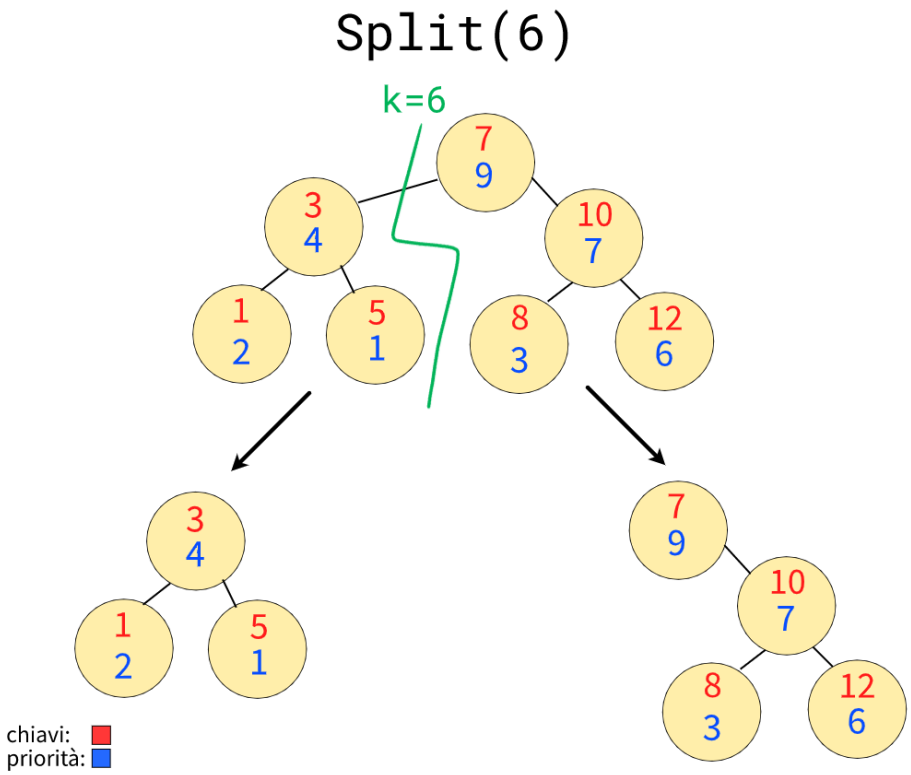

Figura 2.16: Rappresentazione dell'operazione di split.

Per effettuare uno split, si può procedere in questo modo ricorsivo:

- Per prima cosa, confrontiamo la radice con la chiave utilizzata per lo split, determinando quindi se collocarla in T_1 o in T_2.

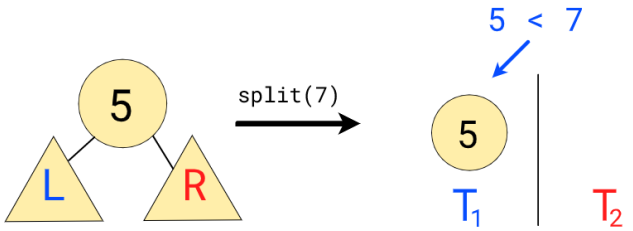

Figura 2.17: Esempio di collocazione della radice durante uno split.

- Dei due possibili figli della radice, uno di essi sarà già correttamente collocato: se la radice appartiene a T_1, allora il sottoalbero individuato dal figlio sinistro conterrà già solo nodi per cui $x < k$; viceversa se la radice è collocata in T_2, il figlio destro

delimiterà già un sottoalbero composto di soli nodi per cui $x > k$.

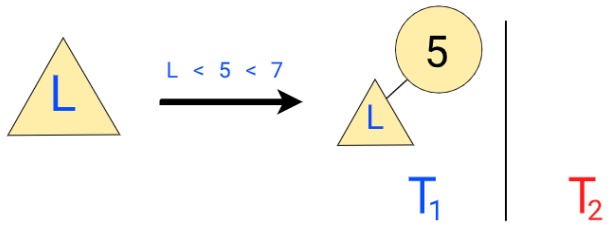

Figura 2.18: Collocazione del primo figlio.

- Dell'altro figlio invece, non sono note informazioni - ciascun nodo del suo sottoalbero potrebbe appartenere a T_1 come a T_2. Di conseguenza, ci servirà "splittarlo" interamente, utilizzando la ricorsione.

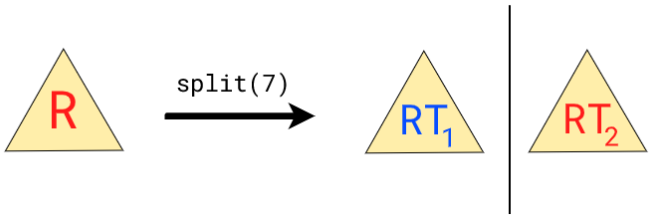

Figura 2.19: Split del secondo figlio.

- A questo punto, non ci rimane che piazzare correttamente i due sottoalberi ottenuti dal figlio che abbiamo appena splittato: la parte che si deve trovare nello stesso albero in cui abbiamo collocato la radice viene collegata direttamente alla radice come figlio, mentre invece l'altra contiene già tutti i nodi che effettivamente devono essere assegnati e può quindi essere utilizzata direttamente come risultato.

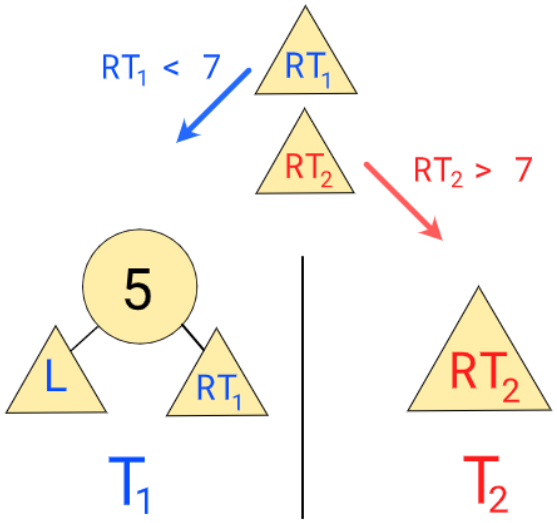

Figura 2.20: Ricollocamento del figlio splittato.

Possiamo osservare che durante un'operazione di split, non viene mai fatto alcun riferimento alle priorità dei nodi - l'algoritmo rimane quindi invariato rispetto a quello utilizzato per un BST normale.

Algorithm 6: Operazione di split.

```
SPLIT(root, k)
    if root = null then
        return null, null
    if root.key < k then
        T1, T2 = SPLIT(root.right, k);
        root.right = T1;
        return root, T2
    else
        T1, T2 = SPLIT(root.left, k);
        root.left = T2;
        return T1, root
```

Siccome ad ogni passaggio la distanza dalla radice dell'albero aumenta di 1, la complessità dell'algoritmo è di $O(h) = O(logN)$.

Join

L'operazione di *join*, talvolta riportata in altre fonti anche come *merge*, consiste nell'unire due treap A e B in un unico treap T, avendo che $x < y$ per ogni chiave $x \in A$ e $y \in B$; con una complessità di $O(logN)$.

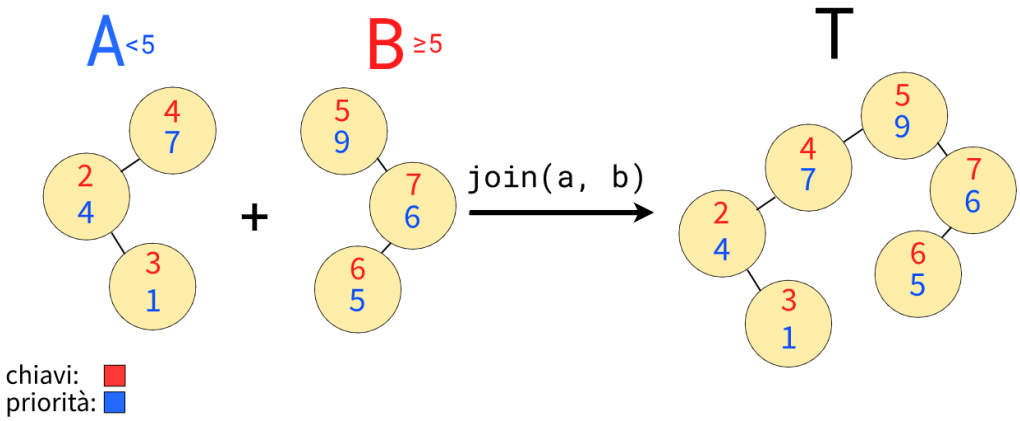

Figura 2.21: Rappresentazione dell'operazione di join.

Possiamo effettuare un *join* ricorsivamente in questo modo:

- Per prima cosa, occorre individuare la radice di T: siccome la radice di ciascun albero è già il nodo di priorità massima, basterà confrontare le priorità delle due radici e scegliere la maggiore delle due.

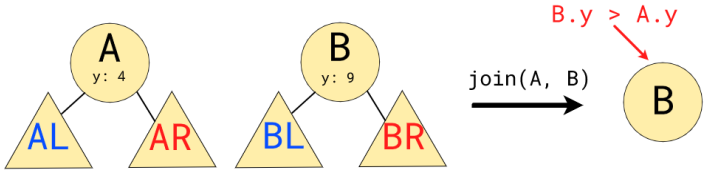

Figura 2.22: Scelta della radice.

- A questo punto, se abbiamo scelto l'albero con chiavi minori A, sappiamo già che il suo figlio sinistro non può che rimanere invariato, essendo le chiavi di B tutte maggiori di quelle in A. Viceversa, sarà il figlio destro di B a rimanere invariato qualora dovesse essere lui quello con maggiore priorità.

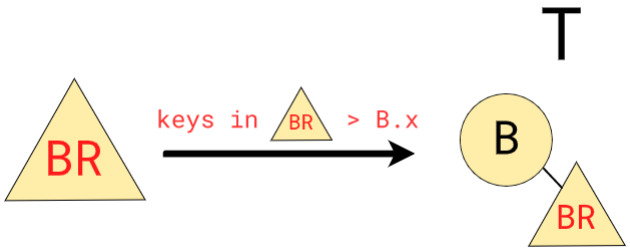

Figura 2.23: Uno dei due figli della radice non cambia.

- Per quel che riguarda l'altro figlio della radice di T però, non c'è ancora nulla di determinato, dato che oltre al secondo figlio della radice scelta c'è anche l'altro albero da considerare interamente. La cosa può però essere gestita con la ricorsione, unendo quindi con un altro join il figlio e l'albero rimanenti.

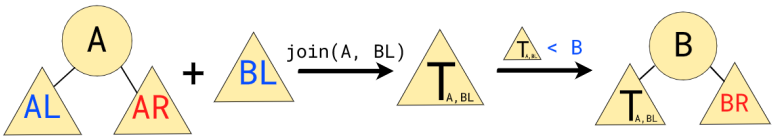

Figura 2.24: L'altro figlio si ottiene .

La situazione è inversa rispetto a uno split - questa operazione fa riferimento solo alle priorità, senza dover mai osservare i valori delle chiavi. Inoltre, join e split possono essere utilizzati in successione, per annullare l'uno l'effetto dell'altro.

Algorithm 7: Operazione di join.

```
JOIN(a, b)
    if a = null then
        return b
    if b = null then
        return a
    if a.priority > b.priority then
        a.right = JOIN(a.right, b);
        return a
    else
        b.left = JOIN(a, b.left);
        return b
```

Analogalmente a split, ad ogni chiamata della ricorsione aumenta di uno la distanza dalla radice - la complessità sarà anche qui $O(h) = O(logN)$.

Inserimento

Una volta che sono state implementate le operazioni di *split* e *join*, gli inserimenti possono essere effettuati utilizzando in sequenza 1 split e 2 join.

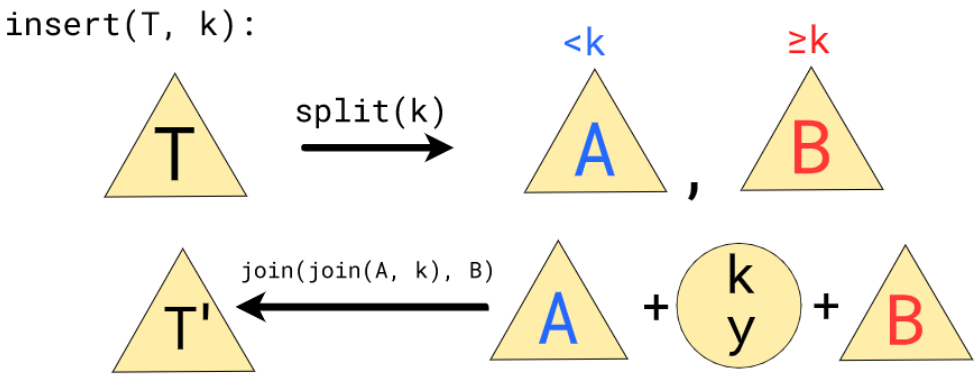

Figura 2.25: Operazione di inserimento con split e join.

Gli inserimenti seguiranno questo procedimento:

- Splittiamo in due il tree, dividendo i valori rispetto alla chiave k che stiamo inserendo.

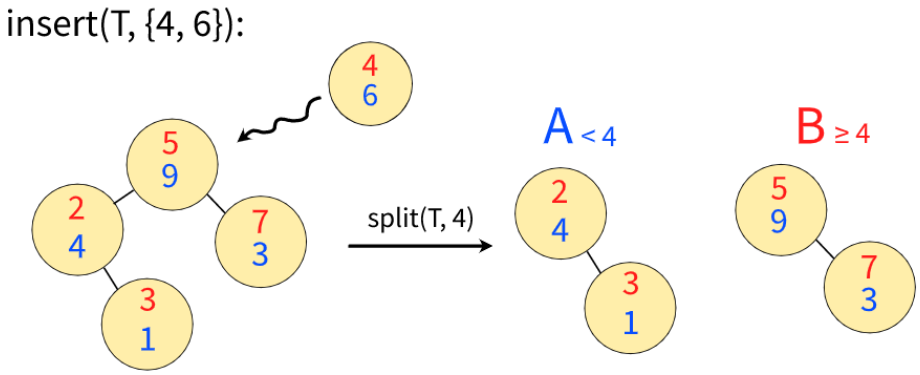

Figura 2.26: Split del tree rispetto a k

- Effettuiamo due *join*: con il primo, aggiungiamo ai valori minori il nodo da inserire; con il secondo aggiungiamo al risultato del primo anche i nodi maggiori, così da ripristinare l'intero albero.

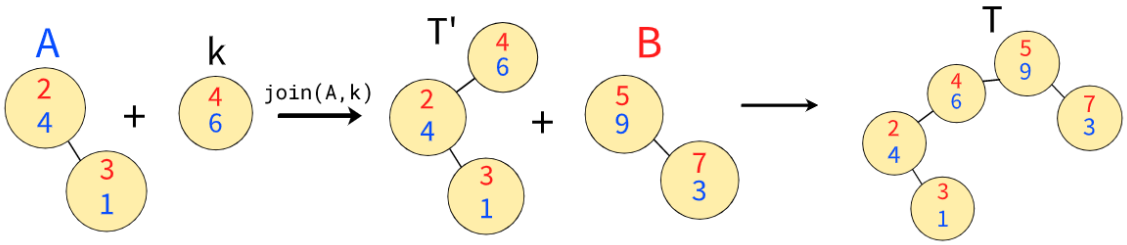

Figura 2.27: Doppio join per ripristinare l'albero.

Così facendo, avremo correttamente inserito la nuova chiave all'interno dell'albero, mantenendo sia l'ordine delle chiavi che la heap property per le priorità; mantenendo una complessità di $O(logN)$, dato che l'algoritmo utilizza 3 chiamate a funzioni di quella complessità.

Il codice risulterà particolarmente breve:

Algorithm 8: Inserimento su treap con split/join.

```
INSERT(root, x, y)
    {a, b} =SPLIT(root, x);
    return JOIN(JOIN(a, {x, y}), b)
```

Cancellazione

Come per l'inserimento, anche per la cancellazione possiamo basarci interamente su *split* e *join* - questa volta saranno necessari 2 split e 1 join.

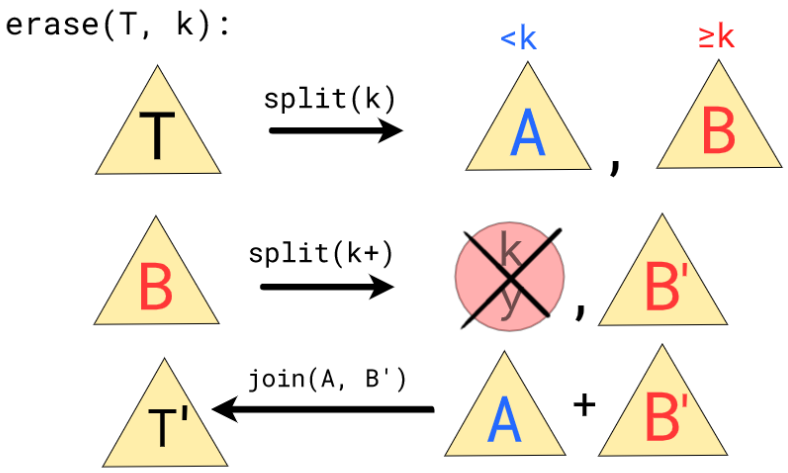

Figura 2.28: Operazione di cancellazione con split e join.

- Dividiamo in due il tree, dividendo i valori per la chiave k che stiamo inserendo.

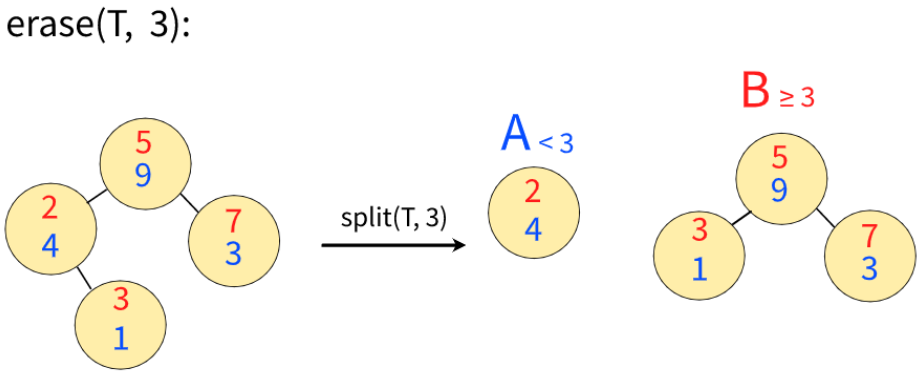

Figura 2.29: Primo split durante la rimozione.

- Con un secondo split, isoliamo la chiave K dall'albero con le chiavi maggiori o uguali a k; e se presente la eliminiamo.

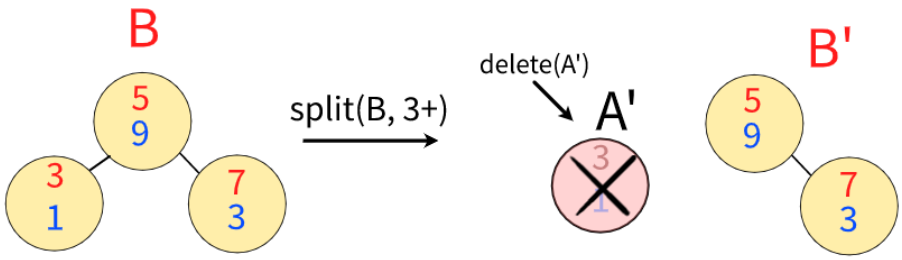

Figura 2.30: Secondo split durante la rimozione.

- Con un *join*, ricomponiamo l'albero, mettendo insieme i nodi con chiave minore di k a quelli con chiave maggiore di k.

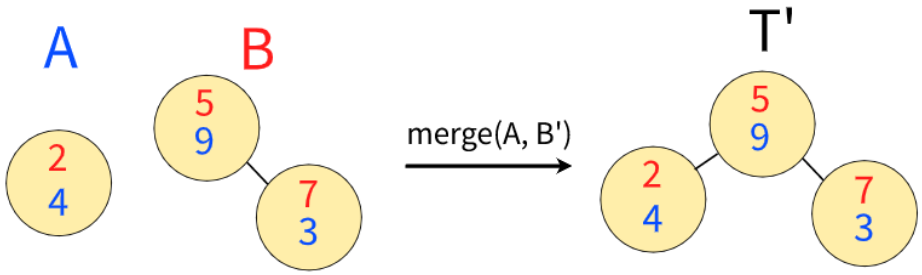

Figura 2.31: Join per ricomperre il treap.

A differenza dell'implementazione con le rotazioni, questa volta non dobbiamo preoccuparci di possibili casi speciali; l'algoritmo sarà comunque $O(logN)$, utilizzando solo funzioni di quella complessità.

Anche qui, il codice sarà piuttosto semplice:

Algorithm 9: Cancellazione su treap con split/join con chiavi intere.

```
ERASE(root, x)
    {a, b} =SPLIT(root, x);
    {to_erase, b2} =SPLIT(b2, x + 1);
    delete(to_erase);
    return JOIN(a, b2)
```

2.2 RBST

Nello stesso paper in cui Seidel e Aragon descrivono i *treap*, viene proposta anche una variante con il nome di "Randomized Search Tree" - successivamente chiamata anche "Randomized BST" (RBST)[2].

La differenza tra queste due strutture è nel calcolo della priorità: invece che assegnare a

ciascun nodo una priorità in fase di inizializzazione, l'*RBST* tratta la priorità di ciascun nodo non come un valore predeterminato, ma come una funzione casuale continua, uguale per ciascun nodo.

Per poter essere ancora in grado di confrontare due nodi, dovremmo però modificare i nodi, aumentando l'albero - non occorrerà più memorizzare il valore delle pioriorità, ma dovremmo mantenere aggiornato il numero di discendenti per ciascun nodo. Il numero di discedenti di un nodo T coincide con la dimensione del sottoalbero che si ottiene prendendo tale nodo come radice, di conseguenza indicheremo questo valore come $size(T)$.

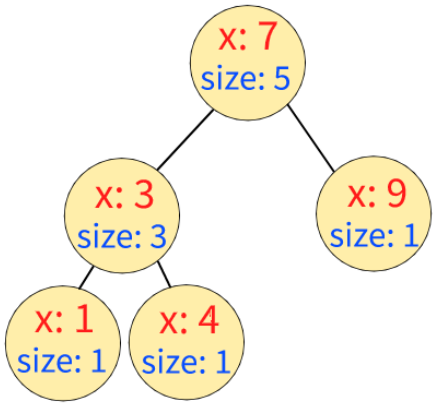

Figura 2.32: Esempio di RBST.

Noto il numero di discendenti di ciascun nodo (o la dimensione del sottoalbero relativo), ci serve sapere come confrontare le priorità tra le radici A e B di due sottoalberi disgiunti, così da poter implementare l'operazione di *join*.
Definiamo quindi una formula che ci indica se A ha priorità maggiore di B:

$$hasPriority(a, b) = random(0, size(a) + size(b)) < size(a)$$

Senza entrare nel merito di una dimostrazione diretta della formula utilizzata, una spiegazione intuitiva della sua correttezza può essere ricavata in questo modo:

- Se mettiamo insieme i nodi dei due sottoalberi A e B, un nodo di questi ha necessariamente priorità massima.

- Le priorità di ciascun nodo sono casuali, ne deriva che quindi ciascun nodo dovrebbe avere la stessa probabilità di essere quello di priorità massima: $\frac{1}{size(A)+size(B)}$.

- Tuttavia, dobbiamo sempre assumere che i nodi rispettino la heap property: di conseguenza quindi la priorità estratta da A va comunque posta maggiore di $size(A)-1$ nodi, mentre quella estratta da B di $size(B)-1$ nodi.

- Ne consegue quindi che A sarà il nodo di massima priorità solo se il nodo di massima priorità è uno tra quelli del suo sottoalbero, quindi con probabilità $\frac{size(A)}{size(A)+size(B)}$, e che analogamente per B varrà $\frac{size(B)}{size(A)+size(B)}$.

Per mantenere correttamente aggiornato questo valore, modifichiamo gli algoritmi di *join* e *split*, chiamando una funzione di *update* per ogni nodo a cui vengono modificati i figli.

Algorithm 10: Join aggiornato per RBST.

```
UPDATE(a)
    a.size = size(a.left) + size(b.left);
return SPLIT(root, k)
    if root = null then
        return null, null
    if root.key < k then
        T_1, T_2 =SPLIT(root.right, k);
        root.right = T_1;
        UPDATE(root);
        return root, T_2
    else
        T_1, T_2 =SPLIT(root.left, k);
        root.left = T_2;
        UPDATE(root);
        return T_1, root
```

Algorithm 11: Split aggiornato per RBST.

```
JOIN(a, b)
    if a = null then
        return b
    if b = null then
        return a
    if random()%(size(a) + size(b)) < size(a) then
        a.right = JOIN(a.right, b);
        UPDATE(a);
        return a
    else
        b.left = JOIN(a, b.left);
        UPDATE(b);
        return b
```

A primo impatto, potrebbe sembrare che lo spazio guadagnato nel non dover salvare le priorità sia comunque immediatamente perso in quanto diventa necessario salvare la dimensione di ciascun sottoalbero, senza portare quindi alcun tipo di vantaggio.

Nonostante questo sia vero nel caso generico di un BBST implementato semplicemente per supportare le operazioni viste fino ad'ora, memorizzare la dimensione di ciascun sottoalbero comporta alcuni vantaggi, che possiamo sfruttare per implementare più operazioni senza dover né modificare la struttura né gli algoritmi utilizzati per le altre operazioni.

Ad esempio, gli *RBST* sono già pronti per implementare le funzionalità degli *order statistic tree*[11], ovvero le due funzioni di *select* e *rank*, entrambe con complessità di $O(logN)$:

- *Select(i)* permette di trovare l'i-esimo elemento più piccolo dell'albero:
  partendo dalla radice, verrà scelto di volta in volta il sottoalbero in cui scendere confrontando l'indice dell'elemento cercato con la dimensione del sottoalbero sinistro, che se superato, andrà a diminuire l'indice da cercare nel sottoalbero destro.

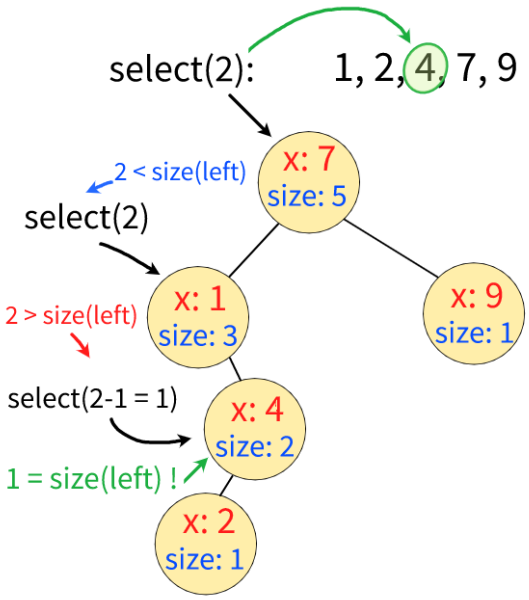

Figura 2.33: Operazione di select.

- *Rank(k)*, l'operazione inversa, che permette invece di calcolare la posizione occupata da una determinata chiave k, ovvero l'indice che avrebbe k in una lista ordinata contenente tutte chiavi dell'albero:
  viene simulata un'effettiva ricerca della chiave k, durante la quale manteniamo un contatore per il numero di chiavi che sono state saltate ogni volta che è stato scelto di spostarsi in un figlio destro.

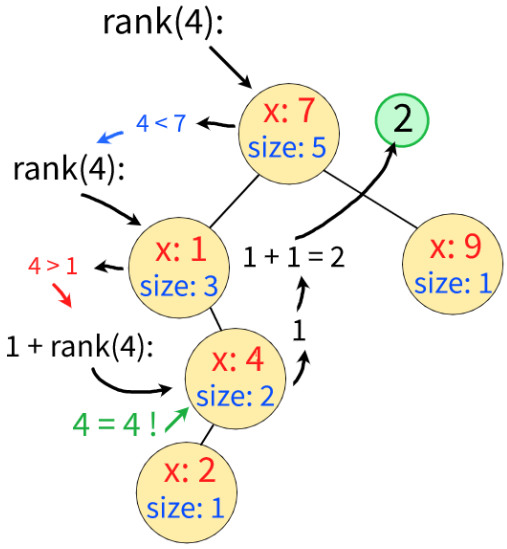

Figura 2.34: Operazione di rank.

Algorithm 12: Operazione per order statistic tree

```
SELECT(root, i)
    if root = null then
        return null
    if i = size(root.left) then
        return root
    else if i < size(root.left) then
        return SELECT(root.left, i)
    else
        return SELECT(root.right, i − 1 − size(root.left))
RANK(root, k)
    if root = null then
        return −1
    if k = root.key then
        return size(root.left
    else if k < root.key then
        return RANK(root.left, k)
    else
        return size(root.left) + 1+root.right, k
```

Un secondo vantaggio degli *RBST* sui *treap* è analizzato meglio nel terzo capitolo, e riguarda la possibilità di essere utilizzati per implementare strutture dati con *persistenza confluente*.

Un effettivo svantaggio degli *RBST* è invece la necessità di generare non più 1 solo numero casuale per inserimento, ma ben $O(logN)$ per operazione - questo può essere mitigato durante l'implementazione evitando l'utilizzo di generatori di numeri casuali poco efficienti, o possibilmente anche reiterando un'unica grande pool di numeri casuali.

2.3 Tree su chiavi implicite

Occasionalmente, e non così raramente nei contesti di programmazione competitiva, può capitare di trovarsi di fronte a problemi che potrebbero essere risolti facilmente, se solo avessimo a disposizione una struttura dati, in grado di gestire una lista ma anche di supportare operazioni più complesse con complessità sublineari, tipiche dei *BBST*.

Estendendo il concetto di *rank* visto nella sezione precedente per quanto riguardava gli order statistic tree, possiamo definire quindi una struttura dati in cui tramite un albero binario viene rappresentata una lista di valori, ordinati quindi per posizione, e non per chiave: questa struttura prende il nome di "albero su chiavi implicite"[3] e possiamo implementarla aumentando i nodi di un treap, memorizzando per ciascuno di essi anche il numero di discendenti.

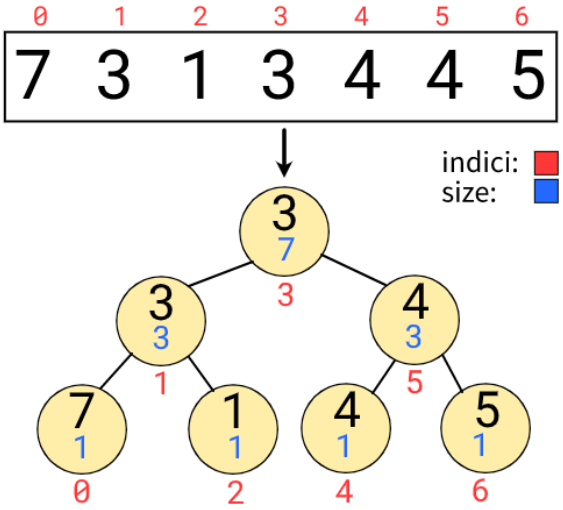

Figura 2.35: Esempio di RBST implicito.

Le operazioni di ricerca, non saranno più relative a chiavi "esplicite", ma agli indici della lista - analogamente a quello che abbiamo visto per l'operazione di *select* negli *order statistic tree*.

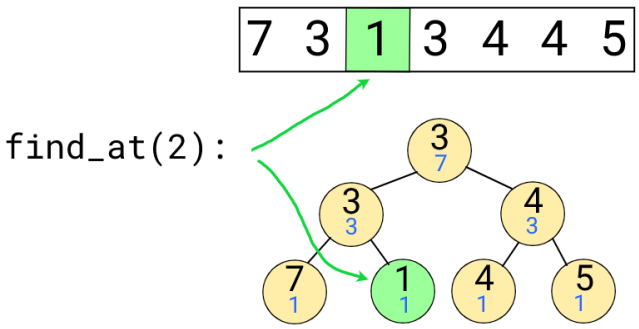

Figura 2.36: Ricerca in un RBST implicito.

L'operazione di inserimento dovrà ora accettare un secondo parametro: oltre al valore da inserire, andrà specificato anche l'indice in cui questo valore sarà posizionato.

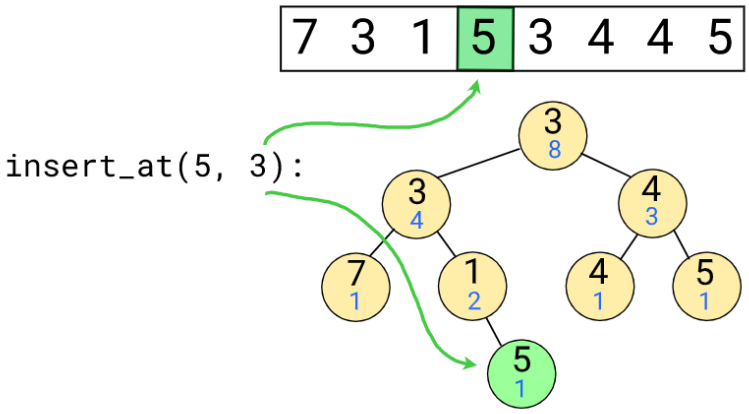

Figura 2.37: Inserimento in un RBST implicito.

La stessa cosa vale per l'operazione di rimozione, che ora utilizzerà un indice invece che una chiave.

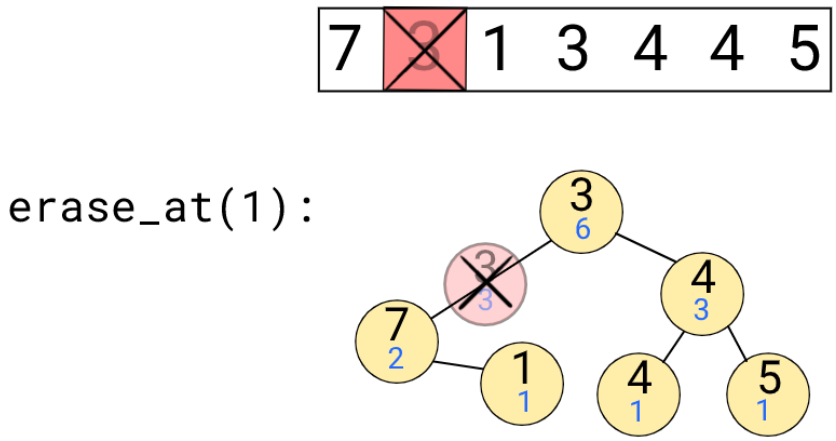

Figura 2.38: Cancellazione in un RBST implicito.

I treap - e ancora di più gli RBST - si prestano particolarmente bene per questa cosa, senza bisogno di intuizioni particolari: dopo aver modificato l'operazione di split per dividere gli elementi in base alla posizione e non l'indice, non ci sarà neanche bisogno di modificare la funzione di join.

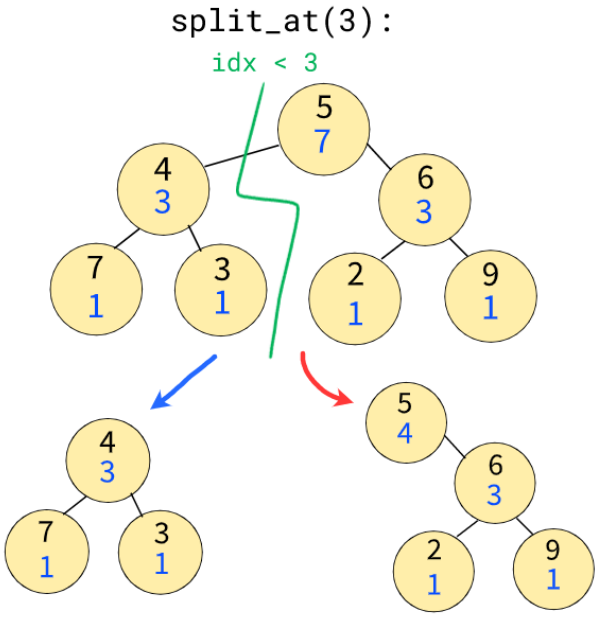

Figura 2.39: Split in un RBST implicito.

Inoltre, non solo abbiamo una struttura dati in grado di gestire una lista con accessi, inserimenti e rimozioni tutte implementabili in $O(log(N))$, ma avremo anche già disponibili operazioni di join e split, anche queste con complessità $O(log(N))$.

Algorithm 13: Split per RBST con chiavi implicite.

```
SPLIT(root, idx)
    if root = null then
        return null, null
    if size(root.left) < idx then
        T1, T2 = SPLIT(root.right, idx − size(root.left) − 1);
        root.right = T1;
        UPDATE(root);
        return root, T2
    else
        T1, T2 =SPLIT(root.left, idx);
        root.left = T2;
        UPDATE(root);
        return T1, root
```

Algorithm 14: Insert e erase aggiornati per RBST con chiavi implicite.

```
INSERT-AT(root, x, idx)
    {a, b} = SPLIT(root, i);
    return JOIN(JOIN(a, node(x)), b)

ERASE-AT(root, idx)
    {a, b} = SPLIT(root, idx);
    {to_erase, b2} = SPLIT(b2, 1);
    delete(to_erase);
    return JOIN(a, b2)
```

É importante notare che questo tipo di struttura dati non possa essere implementata banalmente utilizzando una semplice mappa o dizionario, in quanto non vogliamo che gli indici di ciascun elemento restino statici nel tempo; vogliamo che, come in una normale lista, crescano quando viene inserito un elemento prima di loro, e diminuiscano quando un elemento viene invece rimosso.

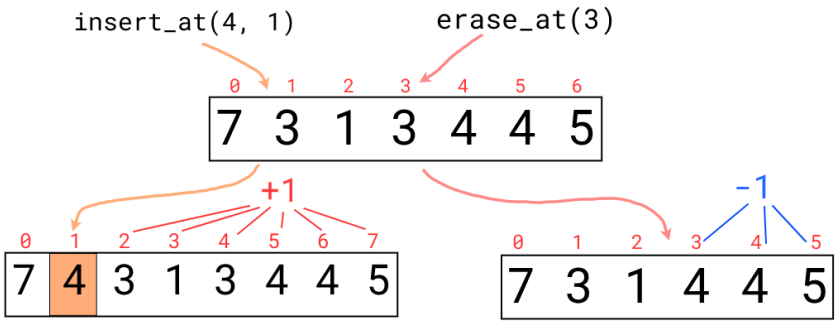

Figura 2.40: Cambiamento delle chiavi implicite durante inserimenti e rimozioni.

Nonostante gli alberi a chiavi implicite possano essere implementati anche aumentando altri BBST, la semplicità con cui i treap gestiscono split e join li rende una scelta molto diffusa per questa implementazione. Inoltre, utilizzando gli *RBST* si avrà l'ulteriore vantaggio di non dover aumentare ulteriormente i nodi dell'albero, evitando quindi l'utilizzo di memoria aggiuntiva.

2.4 Range queries

La semplicità di funzionamento dei treap rende particolarmente semplice anche aumentare l'albero per supportare il calcolo delle cosiddette "range query", ovvero funzioni calcolabili su intervalli di dati.
Questo vale anche sui treap con chiavi implicite, permettendoci quindi di implementare strutture sia per query su range di chiavi, che su effettivi intervalli di array o liste; il tutto con una complessità di $O(log(N))$.

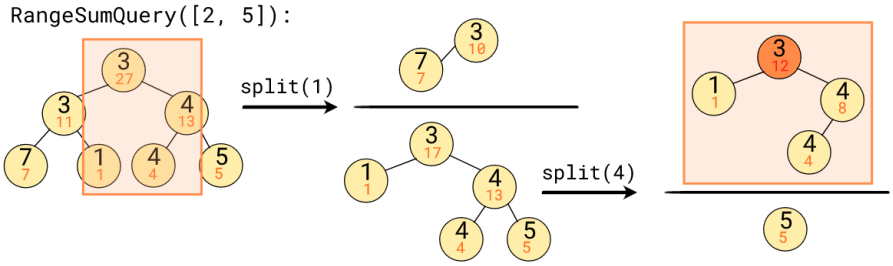

Figura 2.41: Esempi di range query.

Questo tipo di query trova molteplici utilizzi nella geometria computazionale, ma compare anche in algoritmi di altro tipo: ad esempio, strutture dati per calcolare "range minimum query" vengono utilizzate in algoritmi per il lowest common ancestor (LCA)[8] e per il longest common prefix (LCP)[9].

Possiamo utilizzare i treap per tutte le range query basate su funzioni che godono della proprietà transitiva: se basiamo l'implementazione su quella di un RBST, ci basterà aggiornare la funzione di *update*, aumentando il nodo così da poter memorizzare il risultato della funzione richiesta, riapplicandola ad ogni singola modifica effettuata sul nodo.

Algorithm 15: Update aggiornato per Range Sum Query.

```
UPDATE(a)
    a.size = size(a.left) + size(b.left);
    a.sum = a.left.x + a.right.x;
    return
```

Per rispondere ad una query, possiamo replicare le operazioni utilizzate nei range tree[4], una struttura dati statica utilizzata per RMQ.

Algorithm 16: Possibile implementazione per range query su chiavi implicite.

```
RangeSumQuery(t, a, b)
    if t = [null] then
     ∟ return 0
    if b < a then
     ∟ return 0
    if a = 0 & b = t.size() then
     ∟ return t.sum
    lsum = RangeSumQuery(t.left, a, min(b, size(t.left) − 1));
    rsum = RangeSumQuery(t.right, max(0, a − size(t.left) − 1, b − size(t.left) − 1));
    if size(t.left) ≥ a & size(t.left) ≤ b then
     ∟ lsum = lsum + t.val
    return lsum + rsum
```

Tuttavia, grazie all'operazione di split, ancora una volta possiamo semplificare notevolmente il codice necessario: ci basterà infatti eseguire due split per isolare l'intervallo interessato, la cui radice conterrà già il risultato della query richiesta. Dopodiché, possiamo ripristinare l'albero, eseguendo due join.

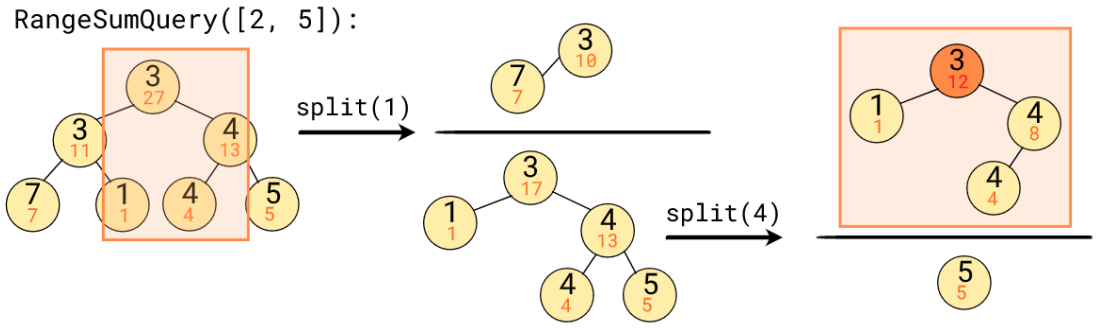

Figura 2.42: Range isolato tramite split.

Il codice così ottenuto non è altrettanto efficiente, ma mantiene la stessa complessità di $O(log(N))$ ed è sufficiente per comprendere il funzionamento della struttura.

Algorithm 17: Implementazione con split per range query su chiavi implicite.

```
RangeSumQuery(t, a, b)
    {t1, t2} = Split(t, a);
    {t3, t4} = Split(t2, b − a + 1);
    ans = t3.sum;
    root = Join(t1, Join(t3, t4));
    return ans
```

2.5 Range updates - lazy propagation

Un'ulteriore tecnica facilmente implementabile sui treap è la lazy propagation[11], che ci permette di applicare alcuni tipi di update ad interi intervalli dell'albero, il tutto con una complessità sempre di $O(log(N))$.
In questa sezione, viene assunto l'utilizzo di un albero a chiavi implicite.

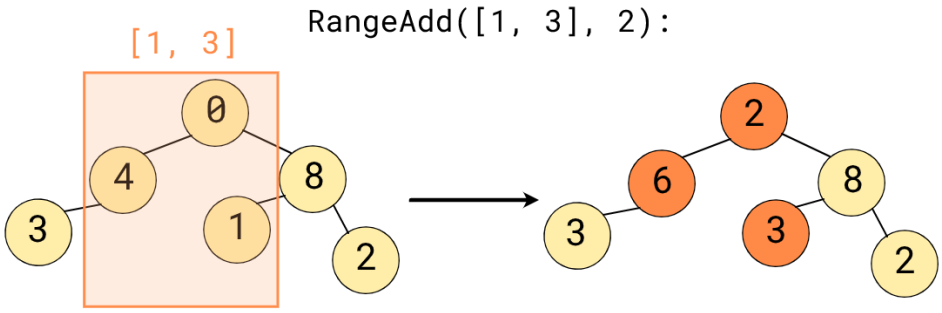

Figura 2.43: Esempio di range update.

Questa tecnica si basa sull'applicare gli aggiornamenti veri e propri solamente quando necessario, mantenendo le informazioni effettivamente aggiornate solamente per i nodi che sono stati esplicitamente visitati successivamente all'aggiornamento.
Per implementare questa tecnica, verrà aggiunto un campo addizionale in ogni nodo per indicare se l'informazione contenuta nel nodo è aggiornata o se c'è una qualche modifica da applicare.

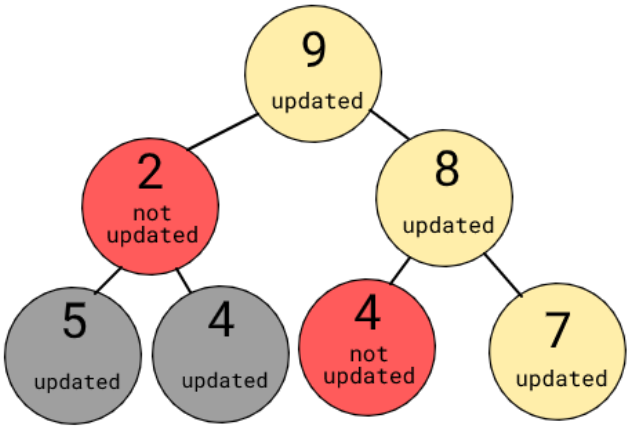

Figura 2.44: Esempio con nodi aggiornati, da aggiornare direttamente e indirettamente.

Un possibile esempio di range update implementabile con la lazy propagation possono essere le "range inversion": dato un intervallo $[l, r]$, invertire i valori al suo interno in modo che il primo diventi l'ultimo, il secondo il penultimo e così via.

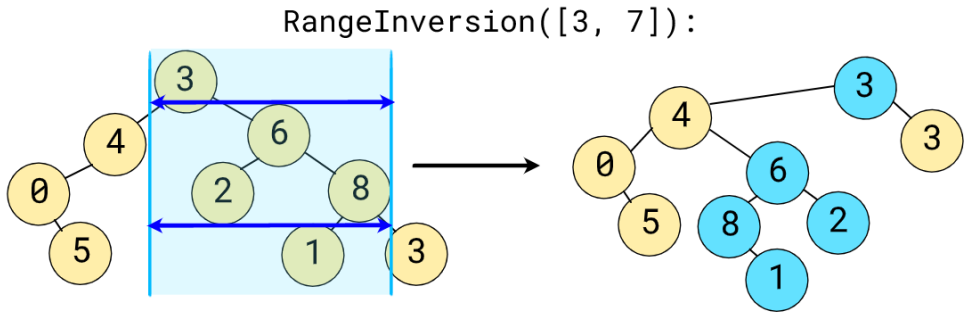

Figura 2.45: Rappresentazione di una query di inversione.

Come per le range query, per effettuare un update effettuiamo prima di tutto due split, così da isolare l'intervallo interessato dalle query.

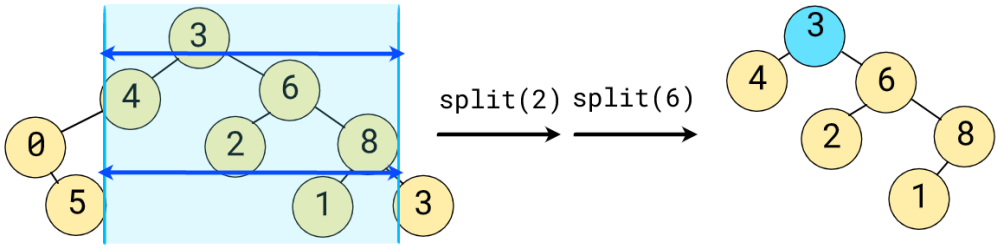

Figura 2.46: Isolamento del range da invertire.

Dopodiché, andiamo ad aggiornare la radice, scambiando tra loro i puntatori dei figli. Siccome l'intervallo da invertire potrebbe avere fino ad $O(N)$ elementi al suo interno, non proseguiamo con l'inversione: semplicemente, interiremo il valore del flag "da aggiornare" nei due figli della radice.

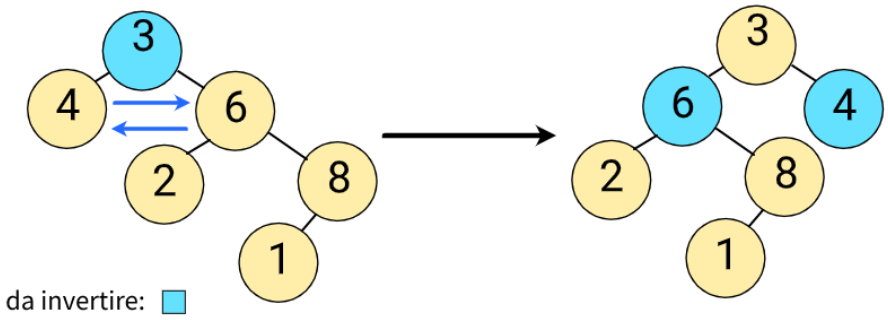

Figura 2.47: Inversione "pigra".

Ora, vorremmo poter effettuare due join così da poter ripristinare la situazione, tuttavia c'è un problema: i due nodi appena "segnati" potrebbero cambiare leggermente disposizione, e con loro anche i loro figli, evitando quindi di poter propagare correttamente i loro aggiornamenti.

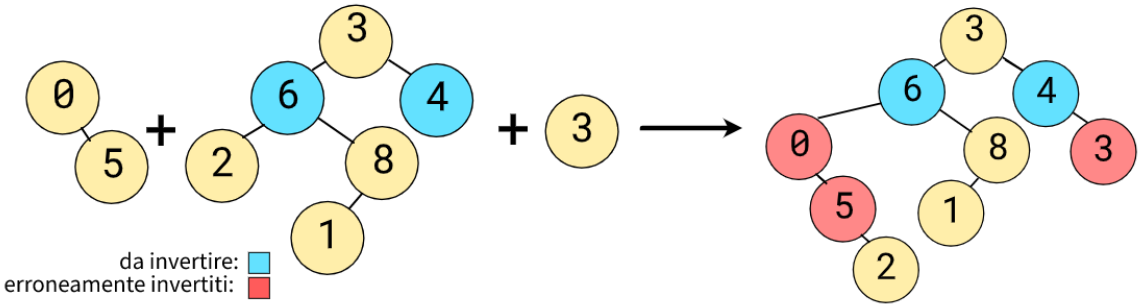

Figura 2.48: Effetti collaterali del join senza propagazione.

Tuttavia, questo può essere evitato semplicemente andando ad aggiornare le funzioni di split, join e search: prima di effettuare alcun tipo di modifica o lettura dei valori di un nodo, ciascuna funzione dovrà prima chiamare una funzione di supporto *propagate*, incaricata di applicare eventuali update arretrati su un nodo e di segnare eventuali update necessari da propagare ai suoi figli.

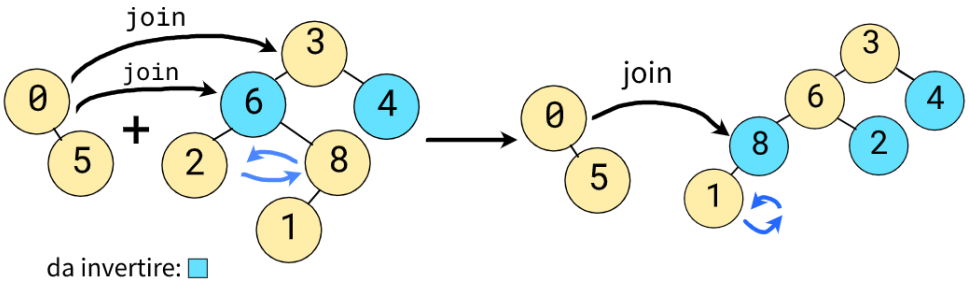

Figura 2.49: I nodi coinvolti nei join vengono man mano propagati.

Nel caso dell'inversione di un intervallo, questa funzione scambierà i figli sinistro e destro, dopodiché li marcherà come da aggiornare (prestando attenzione al fatto che ruotare due volte un intervallo coincida con il ripristinarne l'ordine originale).

Update di tipo diverso richiedono spesso accorgimenti leggermente diversi per poter essere correttamente applicati e adattati al problemi che si vuole risolvere; di conseguenza il codice della funzione *propagate* può cambiare notevolmente tra tipi diversi di query - rimane però sempre la necessità di chiamare la funzione ogni volta che si tocca un nodo, così come quella di aggiungere almeno un campo per gestire lo stato di aggiornamento dei nodi.

Algorithm 18: Funzione di propagate per range inversion.

```
PROPAGATE(t)
    if t.flip = [true] then
        swap(t.left, t.right);
        t.left.flip =!t.left.flip;
        t.right.flip =!t.right.flip;
```

Questa volta, potrebbe risultare meno ovvia la complessità computazionale dell'algoritmo, in quanto potrebbe sembrare che in qualche modo ciascun update vada effettivamente a cambiare il valore di ciascuno dei nodi. Tuttavia, ciascuna operazione continua a visitare al più $O(log(N))$ nodi - di conseguenza, un singolo aggiornamento aggiornerà al più $O(log(N))$ nodi, mantenendo così la complessità di ciascuna operazione $O(log(N))$ (assumendo di utilizzare una funzione propagate di complessità $O(1)$).

Capitolo 3

Persistenza

Sommario

In questo capitolo viene introdotto il concetto di persistenza e viene mostrato come utilizzando un RBST si possa ottenere un albero binario di ricerca bilanciato con persistenta totale, il tutto dovendo modificare solo poche righe di codice.

3.1 Persistenza

Quando si parla di strutture dati, generalmente si fa riferimento senza specificarlo a strutture *effimere*, ovvero strutture dati che se vengono modificate da una qualche operazione cambiano la loro forma e informazioni, senza preoccuparsi di preservare il loro stato precedente.

Tuttavia, esistono anche le strutture dati persistenti, ovvero strutture che quando vengono modificate non alterano la loro forma precedente, ma creano una nuova versione aggiornata, che riporterà le ultime modifiche applicate.

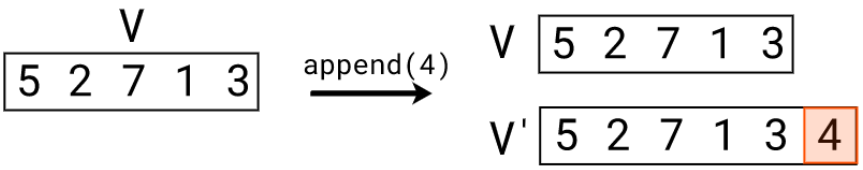

Figura 3.1: Esempio di operazione persistente.

Possiamo identificare 3 livelli di persistenza[5]:

- Una struttura *parzialmente persistente* ci permette di accedere a ciascuna delle versioni create nel tempo, tuttavia solo la più recente permetterà di applicare nuove modifiche, creando quindi nuove versioni.
  Ne consegue che il grafo formato dalle versioni sarà una lista semplice, avendo sempre solo un nodo da cui espandersi per nuove versioni.

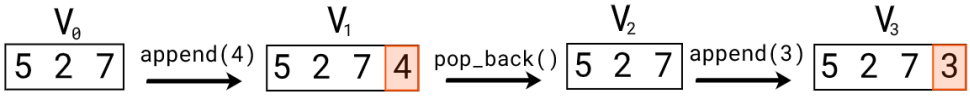

Figura 3.2: Grafo delle versioni di una struttura parzialmente persistente.

- Una struttura *completamente persistente* invece di permette di modificare liberamente ciascuna versione, indipendentemente dal numero di volte in cui è stata già modificata o da quante operazioni siano trascorse dalla sua creazione.
  Strutture con questo livello di persistenza sono tipicamente utilizzate nei linguaggi di programmazione funzionali, anche se è richiesto qualche altro requisito per poter definire una struttura come *puramente funzionale*.
  Siccome ciascuna versione può essere modificata, questa volta il grafo delle versioni sarà un albero.

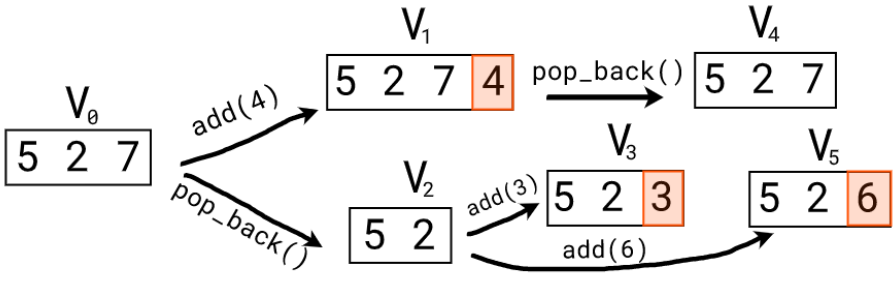

Figura 3.3: Grafo delle versioni di una struttura completamente persistente.

- Una struttura *confluentemente persistente* è una struttura che oltre ad essere completamente persistente supporta anche operazioni di *fusione* per unire diverse versioni

in una nuova.

Il grafo delle versioni sarà quindi un grafo diretto aciclico, siccome un singolo nodo potrebbe avere più di una versione come padre.

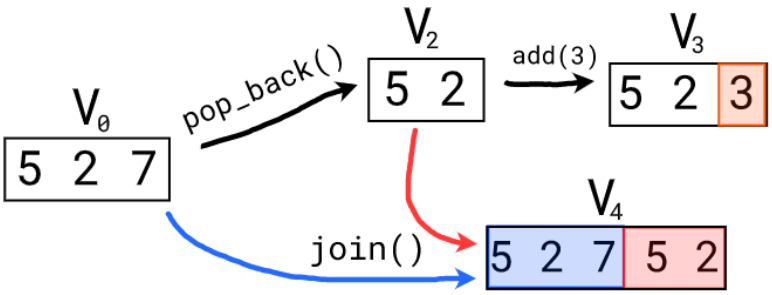

Figura 3.4: Grafo delle versioni di una struttura confluentemente persistente.

3.2 Path copying

La maggior parte delle strutture dati persistenti sono state ideate ad-hoc per avere implementazioni il più efficiente possibile per le strutture dati più comuni, normalmente utilizzate nelle loro versioni non persistenti.

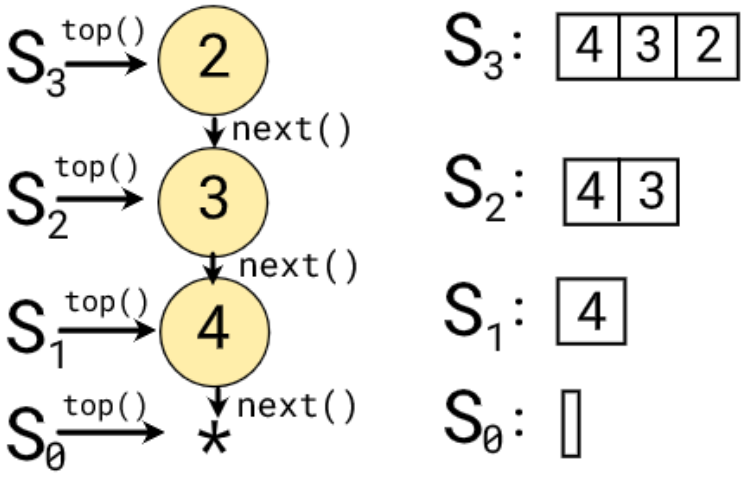

Figura 3.5: Rappresentazione di uno stack persistente.

Tuttavia, esistono anche alcune tecniche generiche[6] per rendere persistenti quasi qualsiasi tipo di struttura dati: tra queste, il *path copying* è una valida soluzione per le strutture ad albero.

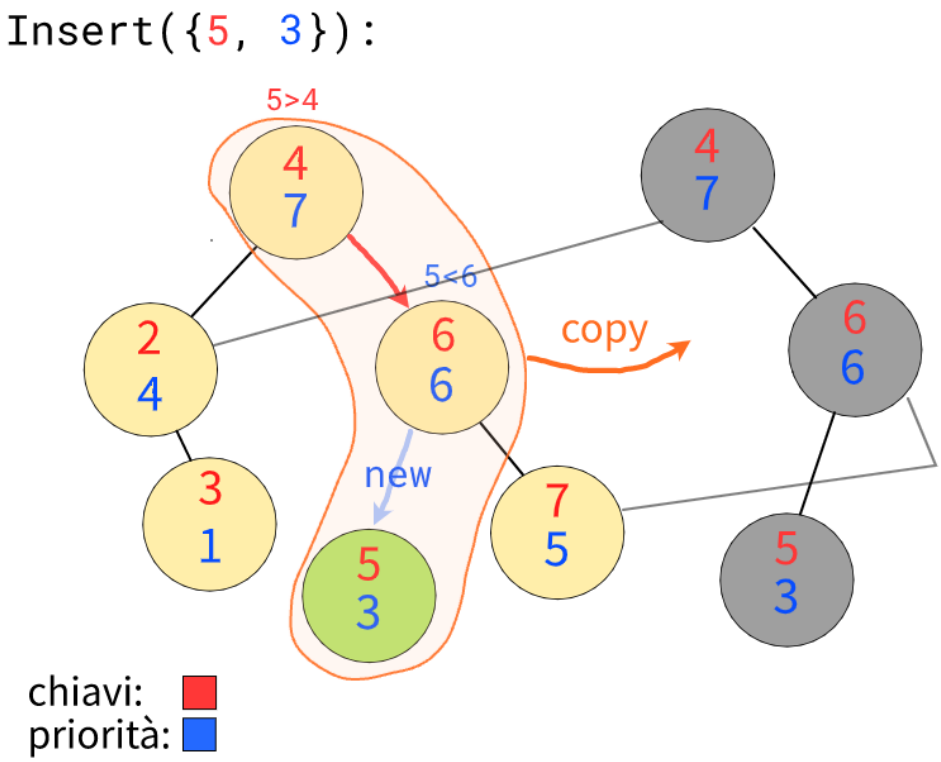

Figura 3.6: Path copying su un operazione di inserimento.

Il path copying ci permette di rendere completamente persistente una qualsiasi struttura ad albero (in cui sono utilizzati solamente puntatori verso i figli), creando una copia di tutti i nodi che vengono visitati durante ciascuna operazione di modifica - ovvero quei nodi che hanno almeno un discendente che è stato modificato.
Le parti di struttura non modificate vengono invece condivise tra le due versioni - non potendo avvenire modifiche dirette, non può succede che una versioni modifichi informazioni relative ad un'altra.

L'implementazione dell'albero viene modificata in questo modo:

- Ogni volta che un nodo o un suo campo sta per essere modificato, ne viene prima creata una copia esatta, che poi verrà modificata al suo posto.

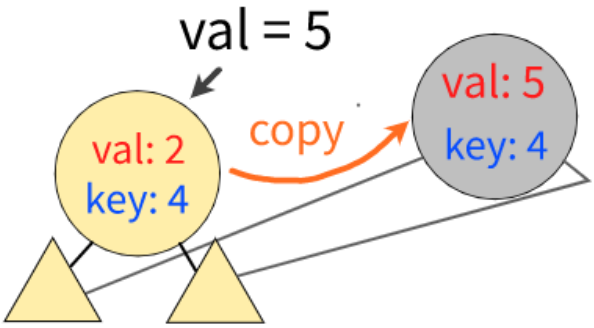

Figura 3.7: Copia del nodo prima della modifica.

- Ciascuna operazione che può modificare in qualche modo uno o più nodi dell'albero viene modificata per poter ritornare un puntatore alla radice della nuova versione ottenuta.

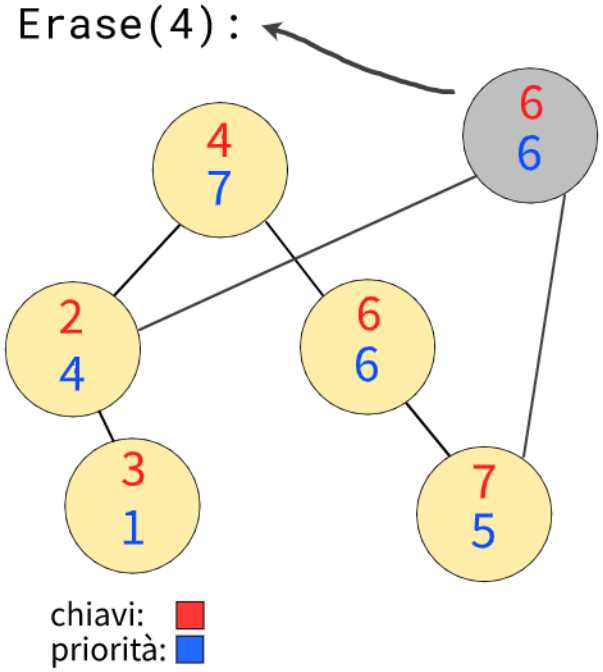

Figura 3.8: Esempio per operazione di rimozione.

- Memorizzando ciascuna nuova radice, sarà possibile mantenere accessi di lettura e modifica a ciascuna versione generata.

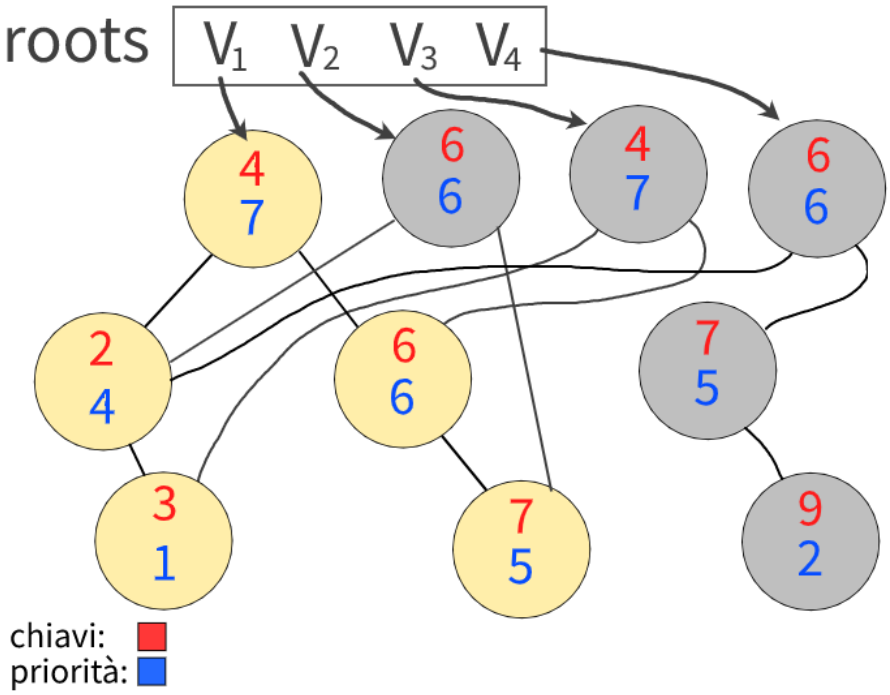

Figura 3.9: Array contenente le radici di ciascuna versione.

Creare una nuova versione con path copying avrà quindi un costo sia di memoria che di tempo di $O(k)$, dove k è il numero di nodi visitati. Il numero di nodi visitati non può mai essere maggiore del tempo impiegato dall'operazione che stiamo rendendo persistente, per cui nel caso di alberi binari bilanciati, avremo $k = O(log(N))$.

Il metodo del path copying può essere applicato anche su implementazioni iterative, ma risulterà più semplice adattare metodi ricorsivi. Quando si adatta una struttura dati, è inoltre importante notare che se la struttura non persistente ha delle operazioni la cui complessità è stata calcolata tramite analizi ammortizzata, la versione persistente non potrà garantire la stessa complessità computazionale, in quanto potrebbero venire create molteplici versioni di uno stato in cui eseguire una di quelle operazioni ha un alto costo. Per questo motivo, non tutti i tipi di albero binario si prestano bene per essere resi persistenti - ad esempio, splay tree e scapegoat tree.

3.2.1 Treap e RBST

Sia il treap che l'RBST possono essere modificati applicando il path copying per ottenere una struttura dati completamente persistente.

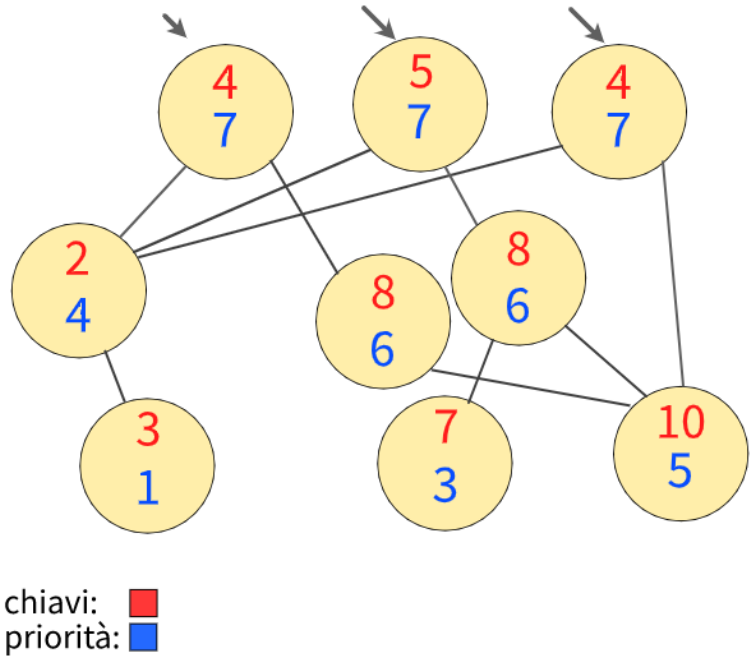

Figura 3.10: Esempio di treap con persistenza completa.

Il discorso è diverso se si parla di una struttura *confluentemente persistente*.
Per garantire un'altezza di $O(log(N))$, il treap sfruttava il fatto che essendo estratte casualmente da un grande dominio, le priorità hanno una bassisima probabilità di essere uguali tra loro, rendendo quindi poco necessario dover gestire il caso in cui ci fossero molte priorità uguali tra loro.

Tuttavia, la situazione cambia se vengono permesse operazioni di unione tra versioni diverse dello stesso albero, poiché in tal caso sarebbe garantito avere collisioni tra chiavi. Potendo quindi scegliere le operazioni da eseguire, sarebbe sempre possibile far degenerare l'albero ad un'altezza lineare.

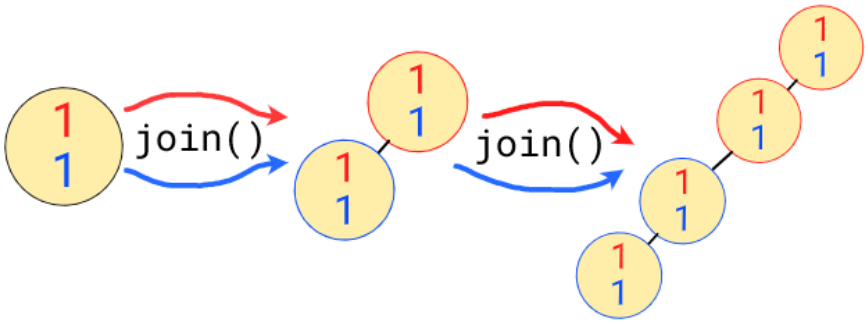

Figura 3.11: Esempio di operazioni che possono far degenerare un treap.

Ne consegue quindi che, senza modificarne l'implementazione, un treap non può essere utilizzato per ottenere una persistenza confluente. Gestendo le priorità come variabili aleatorie invece, questa cosa non rappresenta un problema per gli RBST, che garantiscono quindi le stesse complessità anche quando utilizzati come strutture confluentemente persistenti.

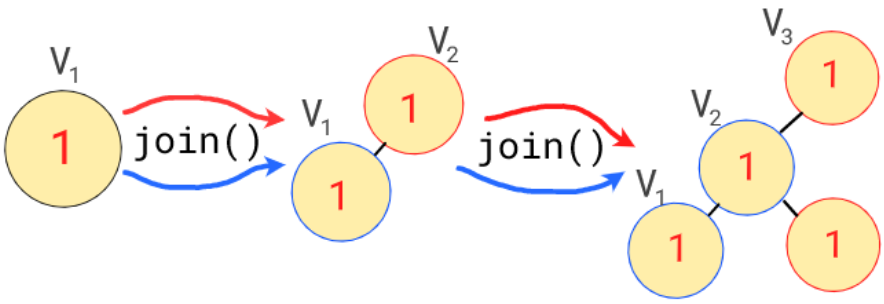

Figura 3.12: Possibile configurazione che può assumere un RBST.

Per implementare il path copying in un RBST, ci basterà aggiornare il codice in questo modo:

- Join dovrà creare una copia del nodo che viene scelto come radice, prima di modificarlo e poi restituirlo.

```
JOIN(a, b)
    if a = null then
        return copy(b)
    if b = null then
        return copy(a)
    if random()%(size(a) + size(b)) < size(a) then
        tmp = copy(a);
        tmp =JOIN(tmp.right, b);
        UPDATE(tmp);
        return tmp
    else
        tmp = copy(b);
        tmp.left =JOIN(a, tmp.left);
        UPDATE(tmp);
        return tmp
```

- Per split, bisogna creare una nuova copia del nodo su cui si sta ricorrendo, utilizzandola al posto della radice originale.

```
SPLIT(root, k)
    if root = null then
        return null, null
    tmp = copy(root);
    if tmp.key < k then
        T1, T2 =SPLIT(tmp.right, k);
        tmp.right = T1;
        UPDATE(tmp);
        return tmp, T2
    else
        T1, T2 =SPLIT(tmp.left, k);
        tmp.left = T2;
        UPDATE(tmp);
        return T1, tmp
```

- Le funzioni di inserimento e cancellazione rimangono invariate: essendo ricorsive, restituiscono già un puntatore a un nodo - siccome utilizzano split e join, il valore restituito sarà già una nuova copia. L'unica modifica riguarda la funzione di cancellazione: dovendo preservare le versioni precedenti, non ci sarà più bisogno di deallocare la memoria del nodo rimosso, che verrà quindi semplicemente ignorato nella versione restituita.

Capitolo 4

Benchmark

Sommario

In questo capitolo vengono riportati i risultati di una serie di test effettuati per confrontare le performance delle due versioni di treap proposte, degli RBST e delle operazioni che abbiamo esteso su di essi.
Tutti i benchmark fanno riferimento a codice C++, compilato con gcc ed eseguito su CPU i7 @ 3.8GHz. É stato inoltre scelto di non utilizzare ottimizzazioni in fase di compilazione, poiché diversi livelli corrispondevano a margini di miglioramento spesso significativamente diversi tra una struttura e l'altra. I tempi riportati sono stati calcolati come media dell'esecuzione di 10 test effettivi.

4.1 Benchmark generale

Per avere un'idea generale della performance delle varie strutture, sono stati misurati i tempi di diversi tipi di operazioni - oltre a treap e RBST, sono stati inclusi nei test anche un'implementazione di AVL tree e i set della standard library del C++.

Strutture dati	inserimenti			lookup		rimozioni		operazioni miste
	$n = 10^5$	$n = 10^6$	$n=10^7$	$n = 10^5$ #lookup = 10^5	$n = 10^6$ #lookup = 10^6	$n = 10^5$ #erase = 50.000	n = 500.000 #erase = 250.000	500.000 operazioni ripartite casualmente
Treap - rotazioni	0.038s	0.424s	4.747s	0.011s	0.160s	0.020s	0.168s	0.094s
Treap - split/join	0.048s	0.585s	6.102s	0.012s	0.158s	0.031s	0.242s	0.121s
RBST - split/join	0.086s	0.914s	10.059s	0.011s	0.171s	0.075s	0.409s	0.268s
AVL Tree	0.041s	0.511s	5.568s	0.007s	0.114s	0.031s	0.247s	0.114s
std::set<int>	0.047s	0.535s	5.883s	0.020s	0.274s	0.027s	0.224s	0.112s

Figura 4.1: Risultati del benchmark generale.

Possiamo osservare che il treap implementato con le rotazioni sia la più efficiente delle implementazioni testate per tutte le operazioni testate, ad eccezione degli inserimenti, dove a prendere il primo posto è l'AVL tree - nei test effettuati, l'altezza media dell'AVL era quasi la metà di quella dei treap, e questa caratteristica porta a tempi di lettura decisamente inferiori.

L'implementazione semplificata, che fa largo uso delle operazioni di split e join, performa effettivamente peggio della versione con le rotazioni - arrivando fino ad un 50% di tempo in più per le operazioni di rimozione. Tutto sommato però i tempi ottenuti si discostano comunque di molto poco da quelli ottenuti da AVL e dal set del C++, confermando quindi l'efficacia dell'implementazione.

Purtroppo invece non si può dire la stessa cosa degli RBST - le operazioni di inserimento e rimozione risultano circa il doppio più lente di quelle effettuate dalle altre implementazioni. Come si potrebbe già intuire dai tempi ottenuti dai lookup però, questo problema non deriva dall'effettiva altezza dell'albero, che si mantiene essenzialmente uguale a quella dei treap; ma dal maggiore costo causato dai frequenti aggiornamenti di dimensione applicati sui nodi e dal maggior numero di numeri casuali che devono essere generati.

4.2 Operazioni su intervalli

Per testare le performance delle operazioni su intervalli, è stata utilizzata un'implementazione di RBST a chiavi implicite aumentato per rispondere a Range Minimum Query, i cui risultati sono stati confrontati con quelli ottenuti dalle stesse query eseguite su array e su range tree, una struttura dati statica tipicamente utilizzata per rispondere a query di questo tipo.

Strutture dati	**Range Minimum Query**					
	$n = 10^4$		$n = 10^5$		$n = 10^6$	
	$q = 10^4$	$q = 10^5$	$q = 10^4$	$q = 10^5$	$q = 10^5$	$q = 10^6$
Array	0.055s	0.556s	0.559s	5.572s	54.92s	538.7s
RBST - split/join	0.002s	0.025s	0.004s	0.043s	0.097s	0.959s
Range Tree	0.001s	0.018s	0.003s	0.030s	0.075s	0.750s

Figura 4.2: Risultati del benchmark per range query.

I tempi misurati dall'RBST rispecchiano la complessità prevista, mantenendo un'effettiva crescita logaritmica con la dimensione dei dati; a differenza dell'implementazione su array, in cui rispondere ad una query richiede un tempo lineare.

Confrontando gli RBST ai Range Tree, possiamo notare un rallentamento delle operazioni di query del 30% circa - finché si tratta di operazioni su un numero di valori nell'ordine di grandezza dei milioni, la differenza non dovrebbe essere troppo rilevante. Il risultato ottenuto può comunque essere considerato molto buono, in quanto non ci siamo discostati di molto dai risultati ottenuti da una struttura dati statica, utilizzandone una completamente dinamica.

4.3 Persistenza

Per testare le performance della versione persistente, è stata utilizzata una versione persistente degli RBST, i cui risultati sono stati confrontati con la versione non persistente della stessa implementazione, la cui performance era stata già misurata nei test precedenti.

Strutture dati	Operazioni varie (~90% inserimenti, ~10% rimozioni)							
	$n = 10^5$		$n = 5 * 10^5$		$n = 10^6$		$n = 2 * 10^6$	
	tempo	memoria	tempo	memoria	tempo	memoria	tempo	memoria
RBST	0.071s	4.0 MB	0.467s	16.1 MB	1.127s	31.6 MB	2.739s	62.3 MB
RBST persistente	0.110s	51.2 MB	0.605s	267.3 MB	1.246s	535.4 MB	2.679s	1107 MB

Figura 4.3: Risultati del benchmark per RBST persistenti.

Salta subito all'occhio l'elevato consumo di memoria della versione persistente. Questo potrebbe sembrare insolito, ma riflette effettivamente l'analisi sul path copying - l'utilizzo di memoria passa da $O(N)$ a $O(Nlog(N))$, e quindi vedere l'utilizzo di memoria incrementato del 20% è accettabile per valori di N intorno a 10^6.

Al crescere di N notiamo che la versione persistente si avvicina sempre di più alla performance degli RBST standard - fino ad eventualmente superarla, di pochi decimi di secondo. Questo probabilmente è dovuto al fatto che gli inserimenti non vengono effettuati sempre sulla versione più recente - di conseguenza, alcune delle versioni avranno un numero di nodi minore e sarà meno dispendioso inserire un nodo al loro interno. D'altro canto però, questo vantaggio viene mitigato dal tempo speso per allocare il numero molto maggiore di nodi che vengono inizializzati.

Capitolo 5

Conclusioni

Sommario

Questa tesi cerca di presentare nel modo più semplice possibile il treap, una struttura dati che a mio parere si conferma un'ottima opzione per l'implementazione di alberi binari bilanciati. Per quanto non si possano consigliare i treap per scenari in cui il tempo di esecuzione è particolarmente critico, i benchmark effettuati hanno confermato performance comparabili a quelle di altri BBST, in alcune istanze anche migliori. La facilità con cui possono essere implementati, manipolati ed eventualmente aumentati li conferma come un'opzione più che valida per la maggior parte delle situazioni.

Sono state mostrate tecniche per estenderne le funzionalità ampliamente: il numero di operazioni che può essere implementato è davvero vasto e combinando queste funzionalità con la persistenza è possibile risolvere problemi mantenendo complessità computazionali logaritmiche. Le performance ottenute rispecchiano le aspettative e le assunzioni iniziali, ed è stato possibile implementare e verificare la correttezza di ciascuna delle operazioni proposte nella tesi.

5.1 Lavori futuri

Ci sono diversi aspetti di questa tesi su cui mi piacerebbe poter approfondire:

- Una delle caratteristiche che possono ancora essere sviluppate particolarmente è l'ottimizzazione delle implementazioni proposte, dato che in questa tesi ho cercato di evidenziare principalmente la semplicità dei treap, che pur essendo una struttura dati non particolarmente nota (al di fuori dell'ambito della programmazione competitiva) penso rimanga una delle più semplici implementazioni possibili per avere dei BBST performanti. Espandendo il lavoro maggiormente sull'efficienza del codice utilizzato, si potrebbero ottenere miglioramenti rilevanti per tutti quei casi in cui le performance effettive sono di maggiore importanza.

- Un ulteriore aspetto da approfondire riguarda la generazione dei numeri casuali, dato che gli RBST hanno ottenuto risultati peggiori delle implementazioni equivalenti con treap, nonostante l'altezza media mantenuta fosse effettivamente la stessa. Testando metodi diversi per genere i numeri utilizzati nella priorità in fase di benchmark, si sono notati miglioramenti anche significativi, soprattutto per le operazioni di inserimento - sono abbastanza incuriosito da questo aspetto e penso che analizzando meglio la cosa sia possibile riavvicinare la performance degli RBST a quella dei treap.

- L'implementazione persistente del treap è stata realizzata utilizzando il path copying, tuttavia esistono alternative più complesse che potrebbe valere la pena esplorare, per cercare di migliorare sia i tempi che l'utilizzo di memoria ottenuti nei benchmark.

Bibliografia

[1] Cecilia R. Aragon and Raimund G. Seidel. 1989. "Randomized Search Trees". *University of California Berkeley, Computer Science Division.*
http://faculty.washington.edu/aragon/pubs/rst89.pdf 5

[2] Conrado Martínez and Salvador Roura. January 31, 1997. "Randomized Binary Search Trees".
http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.17.243rep=rep1type=pdf 25

[3] Patrik Zajec. Ljubljana, Slovenia, 2019. "Treap with and without implicit keys". *Jozef Stefan Institute.*
http://salomon.ijs.si/Lectures/Seminars/Treap-2019/seminar.pdf 31

[4] Bentley, J. L. 1979. "Decomposable searching problems". *Information Processing Letters. 8.* https://doi.org/10.1016/0020-0190(79)90117-0 36

[5] Milan Straka. Prague, 2013." Functional Data Structures and Algorithms". *Doctoral thesis, Charles University in Prague, Faculty of Mathematics and Physics - Computer Science Institute.*
https://ufal.mff.cuni.cz/ straka/theses/doctoral-functional_data_structures_and_algorithms.pdf 44

[6] James R.Driscoll, Neil Sarnak, Daniel D.Sleator and Robert E.Tarjan. 1989. "Making data structures persistent". *Journal of Computer and System Sciences Volume 38, Issue 1, February 1989*
https://doi.org/10.1016/0022-0000(89)90034-2 45

[7] Cormen, T. H., Cormen, T. H. (2001). "Introduction to algorithms". Cambridge, Mass: MIT Press.
https://mitpress.mit.edu/books/introduction-algorithms-third-edition 9

[8] Bender, Michael A.; Farach-Colton, Martin (2000), "The LCA problem revisited", *Proceedings of the 4th Latin American Symposium on Theoretical Informatics* doi:10.1007/10719839_9 35

[9] Fischer, Johannes; Heun, Volker (2007). "A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array". Combinatorics, Algorithms, Probabilistic and Experimental Methodologies. https://link.springer.com/chapter/10.1007%2F978-3-540-74450-4_41 35

[10] Steven Halim and Felix Halim. 2013. Competitive Programming 3. The New Lower Bound of Programming Contests.
https://cpbook.net/ 2

[11] Antti Laaksonen. Draft 2018. Competitive Programmers Handbook. https://cses.fi/book/book.pdf 28, 38

[12] Mike Mirzayanov,
Codeforces: Results of 2020 2

[13] File:AVL-tree-wBalance K.svg. (2020, July 11). Wikimedia Commons, the free media repository. Retrieved 16:49, May 10, 2021 from
https://commons.wikimedia.org/w/index.php?title=File:AVL-tree-wBalance_K.svgoldid=432587143 1

[14] File:Red-black tree example.svg. (2021, March 25). Wikimedia Commons, the free media repository. Retrieved 16:48, May 10, 2021 from
https://commons.wikimedia.org/w/index.php?title=File:Red-black_tree_example.svgoldid=546371221. 1

Per tutti i link elencati è stato verificato il corretto funzionamento in data 10/05/2021.