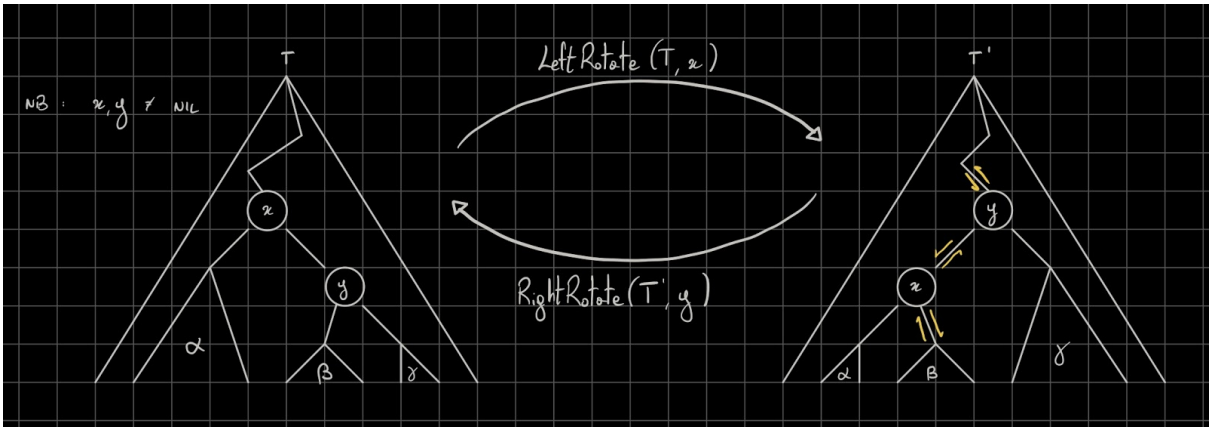
RBT

PROPRIETÀ

- Proprietà dei BST
- Ogni foglia (NIL) è BLACK
- Ogni nodo è RED o BLACK
- ogni nodo RED ha due figli BLACK
- per ogni nodo x lungo ogni cammino x-foglia si trovano sempre lo stesso numero di BLACK (l'altezza nera è uguale per ogni figlio)
- la radice è BLACK
- Ogni nodo ha il puntatore al parent (la radice punta a NIL)
- L'altezza è $\Theta(log\ n)$

OPERAZIONI:

- **ROTAZIONI** : $\Theta(1)$



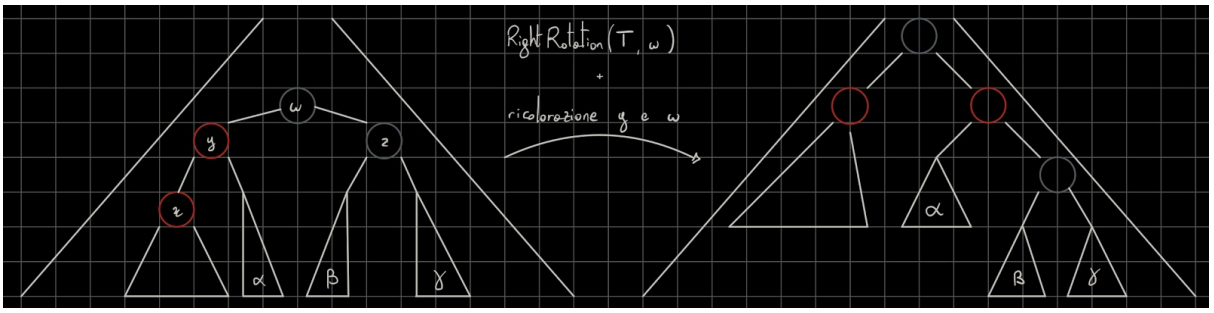
- **RICERCA** : $O(logn)$
  Come nei BST

- **INSERIMENTO** : $\Theta(logn)$
  Inserisco come nei BST, ma coloro il nuovo nodo di RED → Se anche il padre è RED devo rimediare con la procedura **REDFixUP** (rotazioni e ricolorazioni)

- **REDFixUP**: $O(logn)$
  Termino quando risolvo il problema (non ho più due nodi rossi in cascata) o quando arrivo alla radice.

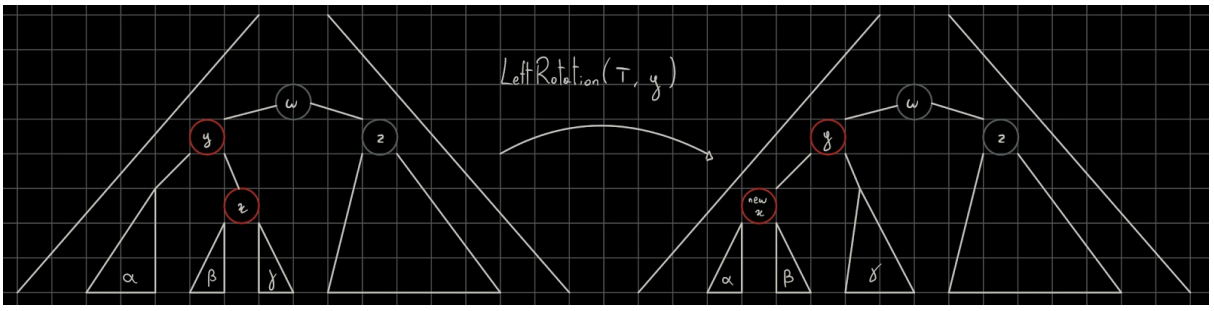
  3 Casi: (più altri tre speculari nel caso in cui il peso sia sbilanciato nel senso opposto)

1. Caso **fortunato**: zio di x BLACK e opposto a x



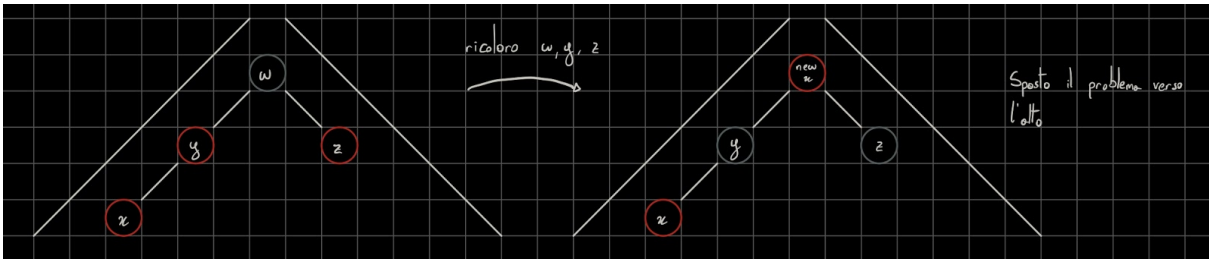
È sufficiente una rotazione nella direzione dello zio (z) e la ricolorazione del parent (y) e del nonno (w). Termino subito.

2. Caso **quasi fortunato**: zio di x BLACK e dalla stessa parte di x



Con una rotazione nella direzione opposta a quella dello zio mi riconduco al caso 1. Termino subito dopo.

3. Caso **sfortunato**: zio di x RED

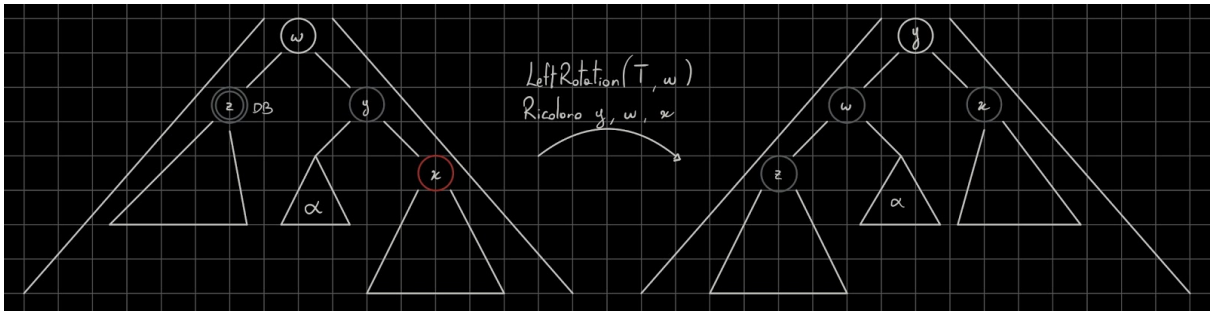


Ricoloro il nonno (w), il parent (y) e lo zio (z). Sposto il problema verso l'alto (il nonno diventa il nuovo x), finché non mi riconduco al caso 1 o 2. Se arrivo alla radice, la ricoloro di nero e termino.

- **CANCELLAZIONE**: $\Theta(log\ n)$
  Rimuovo come nei BST e ricoloro il nodo con cui viene sostituito il nodo cancellato (il predecessore/successore se aveva 2 figli, NIL altrimenti) di BLACK. Se questo nodo era già BLACK ho causato una violazione delle proprietà dei RBT → vale momentaneamente come nodo **DOUBLE-BLACK** e uso la procedura **BLACKFixUp** per risolvere.

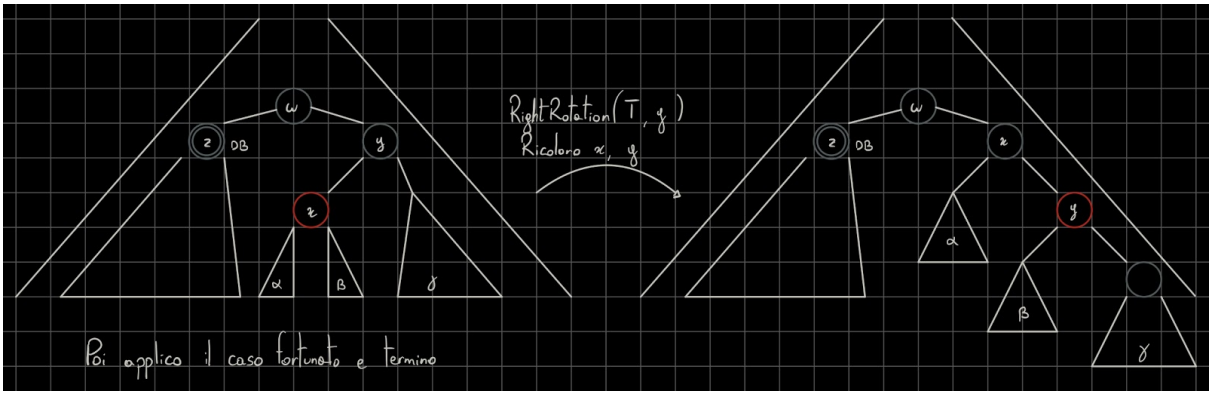
- **BLACKFixUp**: $O(log\ n)$
  Termino quando risolvo il problema (“ricolorando” un nodo DOUBLE-BLACK in BLACK). Se arrivo alla radice termino semplicemente. La procedura consiste concettualmente nell’opposto di REDFixUP: se prima lo zio aveva meno peso perché in x era stato inserito un nodo che aveva sbilanciato l’albero, ora in x è presente meno peso in quanto il nodo è stato cancellato, quindi il nodo **DOUBLE-BLACK è** essenzialmente “**lo zio**” di REDFixUP.

  4 Casi :
  1. Caso **fortunato**: il DOUBLE-BLACK ha un nipote RED opposto a lui



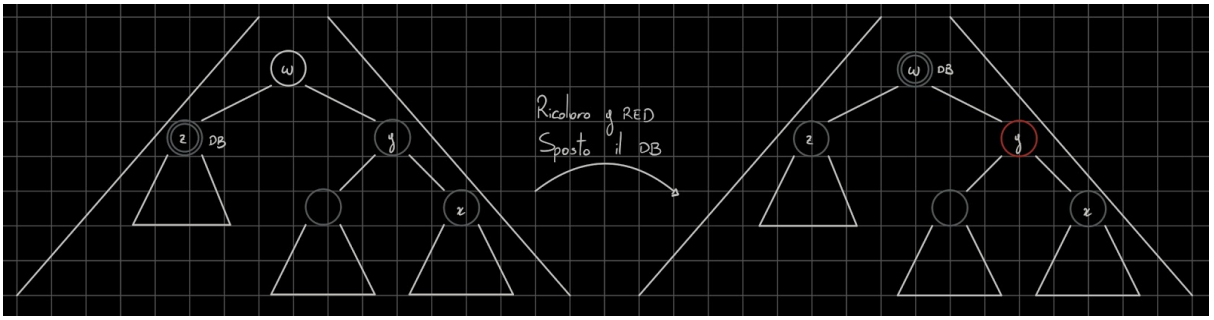
     Effettuo una rotazione nella direzione del DOUBLE-BLACK rispetto al suo parent (w) e ricoloro il parent, il fratello (y) e il nipote (x). Termino subito.

  2. Caso **quasi fortunato**: il DOUBLE-BLACK ha un nipote RED dallo stesso lato



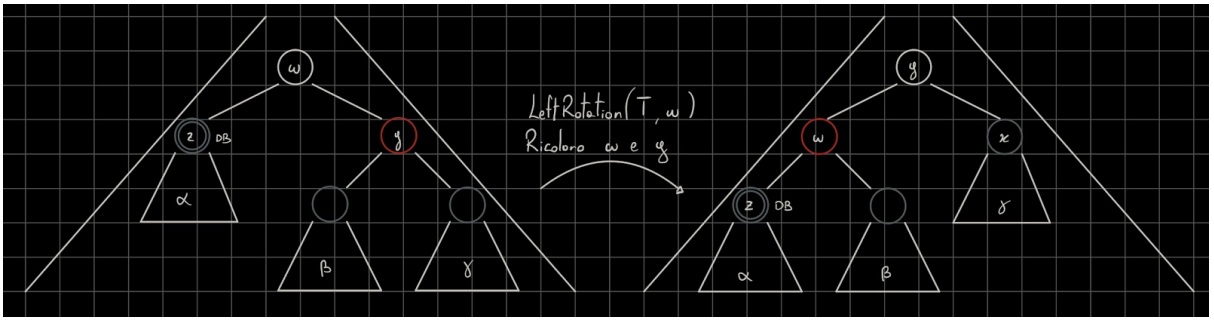
     Effettuo una rotazione nella direzione opposta del nipote (x) rispetto al fratello (y) e ricoloro nipote e fratello. Mi riconduco quindi al caso 1 e termino dopo un’ulteriore iterazione della procedura.

  3. Caso **sfortunato**: il DOUBLE-BLACK ha sia il fratello che i nipoti BLACK



Ricoloro di RED il fratello e sposto il problema verso l'alto: il nuovo DOUBLE-BLACK è il parent (w). Se prima era RED, ho risolto, se era BLACK devo proseguire nell'interazione della procedura.

4. Caso **antipatico**: i nipoti del DOUBLE-BLACK sono neri, ma il fratello è RED

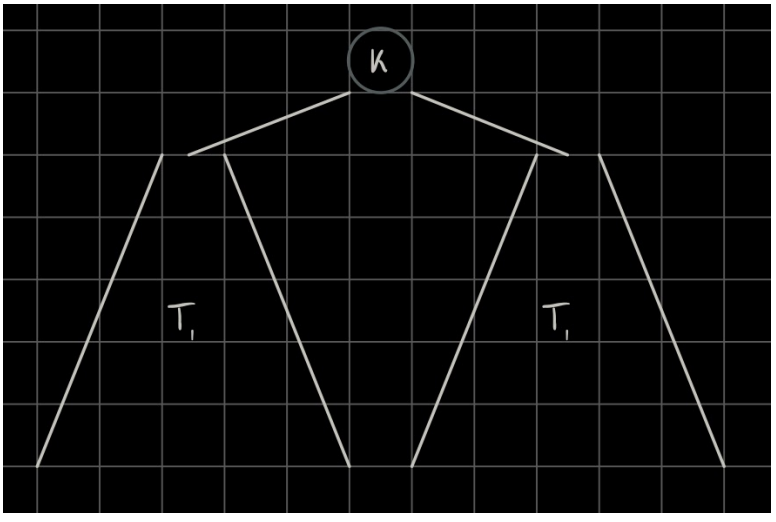


Effettuo una rotazione nel senso del DOUBLE-BLACK rispetto al parent (w). Ricoloro il parent e il fratello (y). Risolvo con uno dei tre casi precedenti: se ricado nel caso 3, comunque termino subito perché il parent è RED e quindi è sufficiente ricolorarlo di BLACK.

- **JOIN** : $O(log\ n)$
  Le precondizioni alla procedura sono che i due alberi T e T' forniti in input siano RBT validi e la chiave K sia maggiore a tutte le chiavi contenute in T e minore a tutte le chiavi contenute in T'.

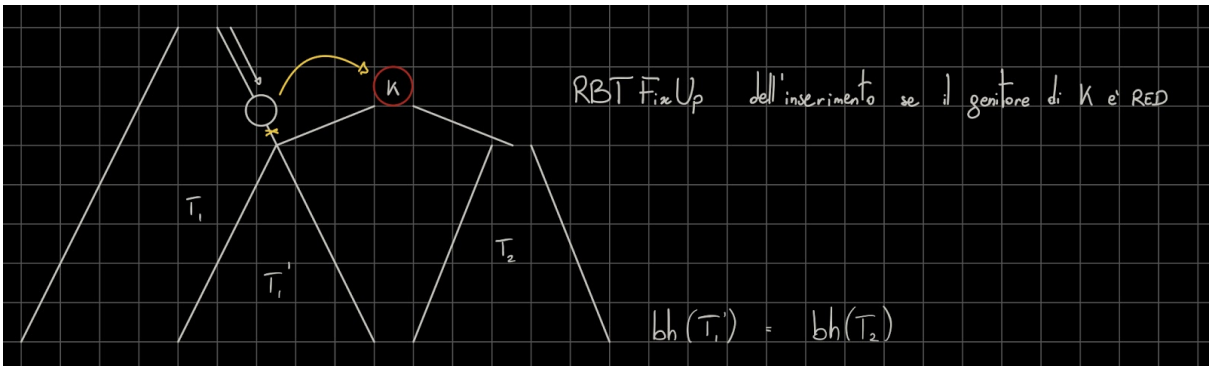
  3 Casi (il 3° è speculare rispetto al secondo) :

  1. bh(T1) = bh(T2)



Pongo K come chiave di un nuovo nodo BLACK radice dell'albero risultante, T come figlio sinistro e T' come figlio destro.

2. bh(T1) > bh(T2)



Scendo a destra nell'albero T1 finché l'altezza nera del sottoalbero è pari a bh(T2). Inserisco quindi K come figlio destro RED dell'albero T in quel punto. Il sottoalbero che prima era radicato al posto di K viene posto come figlio sinistro di K. Se il parent di K è rosso ho causato una violazione delle proprietà dei RBT, quindi uso REDFixUp per risolvere.

3. bh(T1) < bh(T2) : speculare al precedente

BTREE

Sono una generalizzazione di RBT e BST. I nodi sono salvati in RAM.

PROPRIETÀ

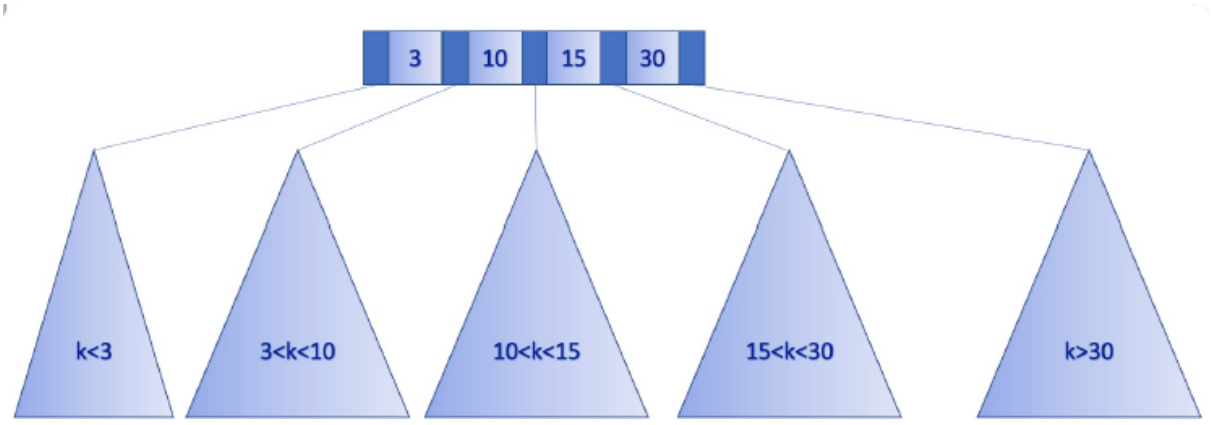
Un B TREE di grado t e' un albero in cui ogni nodo x:

contiene le informazioni:

- x.n (numero di chiavi nel nodo)
- x.c[] (vettore con i puntatori ai child che ha il nodo, sono n+1)
- x.leaf (boolean che mi dice se il nodo e foglia)
- x.key[] (vettore contenente le chiavi nel nodo)

ha le seguenti proprietà:

- se x non e la radice allora $t-1 \leq x.n \leq 2t-1$
- se x e radice allora $1 \leq x.n \leq 2t-1$
- x.key[1] < x.key[2] < x.key[3] < .. < x.key[n]
- tutte le foglie si trovano allo stesso livello
- tutte le chiavi del sotto albero radicato in x.c[i] sono maggiori di x.key[i-1]e minori della chiave x.key[i+1]



L'altezza di un B TREE e' $O(log\ t\ n)$
Il grado t lo decido in base alla grandezza della RAM.

OPERAZIONI (costi divisi in CPU e R/W)

- **RICERCA**: ricerco nel nodo corrente la chiave, quando trovo un nodo più grande allora devo spostarmi nel figlio, ogni volta che mi devo spostare su un figlio lo carico dalla memoria secondaria in RAM e ricorsivamente ricerco in esso il valore.

Dato T B-Tree di grado t e k chiave intera, trovare il nodo x di T che contiene k, se esiste.

- Partiamo da x=T.root
- Scandiamo le chiavi del vettore x.key fino a trovare x.key[i-1]<k≤x.key[i]
- Se x.key[i]=k restituiamo x, altrimenti
- Se x.key[i]>k e x è una foglia restituiamo NIL, altrimenti
- Se x.key[i]>k e x non è una foglia, leggiamo x.c[i] dalla memoria secondaria e procediamo ricorsivamente su x.c[i]

costi:

- R/W: $O(h) = O(log\ t\ n)$
- CPU: $O(h) * O(t) = O(t * log\ t\ n)$

- **SPLIT**: operazione eseguita per creare spazio in un nodo. Dato un nodo contenente 2t-1 chiavi essa prende l'elemento centrale e lo inserisce nel nodo padre.
  La split alza l'altezza nel BTREE solo nel caso venga eseguita nella root.
  Sposto il problema verso l'alto.

  - sia y un nodo pieno e sia x il genitore di y (non abbiamo il puntatore al genitore); y è l'i-esimo figlio di x
  - x (il genitore di y) non è pieno (altrimenti lo avrei «splittato» prima)
    il nodo y ha 2t-1 chiavi e 2t figli
  - «splitto» in 2 parti y:
  - y mantiene le prime t-1 chiavi e i primi t figli;
  - la chiave centrale di y viene portata su in x (spostando chiavi verso dx);
  - viene creato un nuovo nodo z che prende le restanti t-1 chiavi e t figli;
  - il nuovo nodo z diventa l'i+1-esimo figlio di x (spostando figli verso dx);

  costi:

  - R/W: $\Theta(1)$
  - CPU: $O(t)$ → deve eseguire gli shift degli elementi

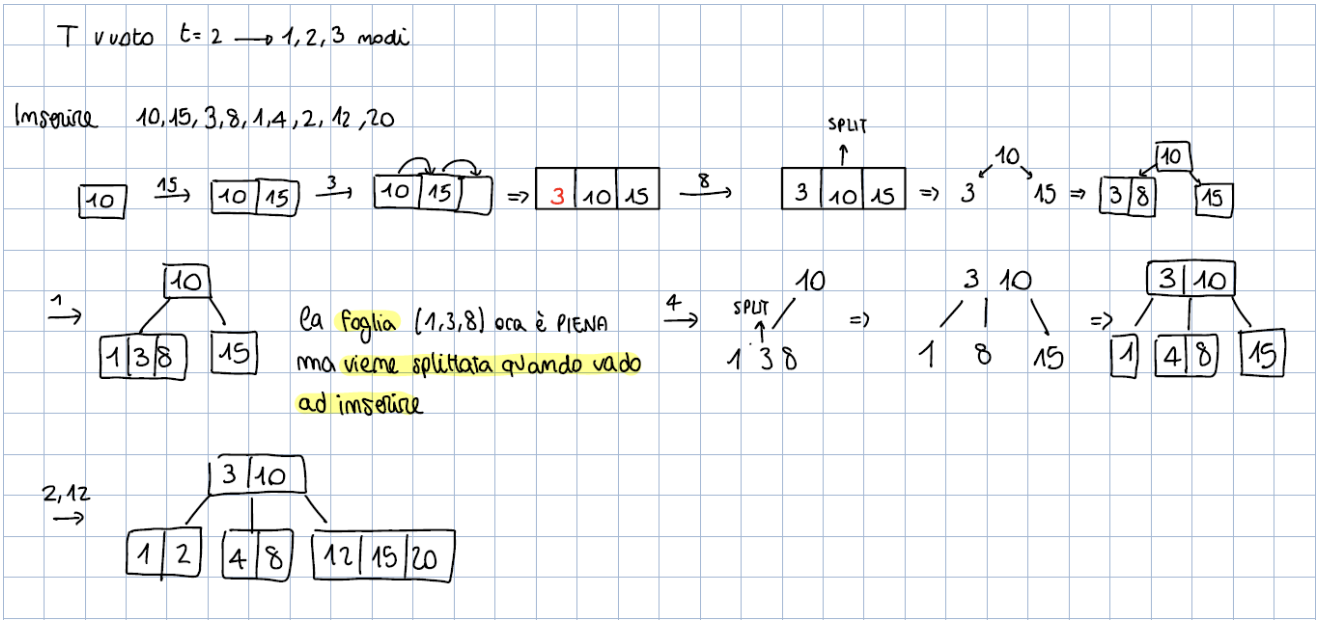
- **INSERIMENTO**: inserisco sempre in una foglia come nei BST. Il problema e' quando un nodo e' pieno → eseguo una **SPLIT** ogni volta che, inserendo, incontro lungo il cammino un nodo con esattamente **2t - 1** chiavi.

  - Si parte dalla radice (se è piena, si splitta la radice aumentando l'altezza dell'albero)

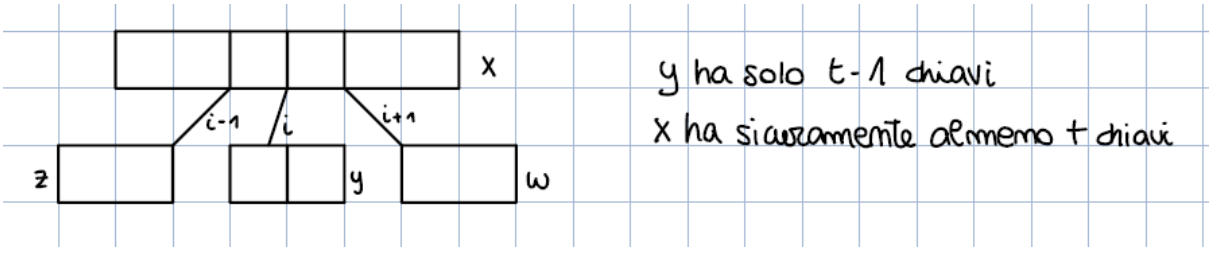
- Si scende come nella ricerca, assicurandosi di splittare i nodi pieni prima di scendere
- Arrivati ad una foglia si inserisce la nuova chiave spostando quelle più grandi verso dx

costi:

- R/W: $O(h) = O(log\ t\ n)$
- CPU: $O(h) * O(t) = O(t * log\ t\ n)$

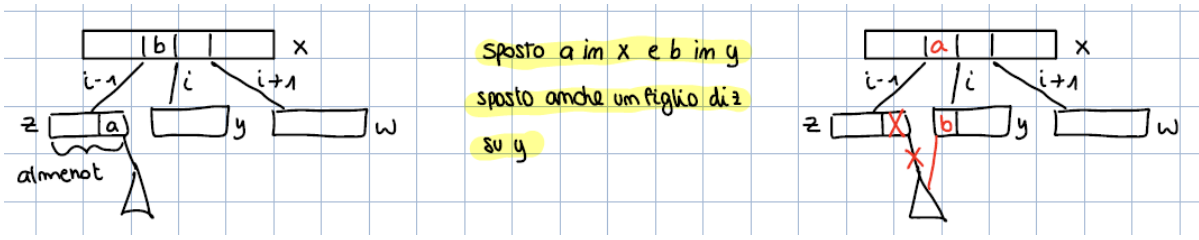


- **MERGE**: operazione eseguita per evitare che un nodo abbia meno di t-1 chiavi.

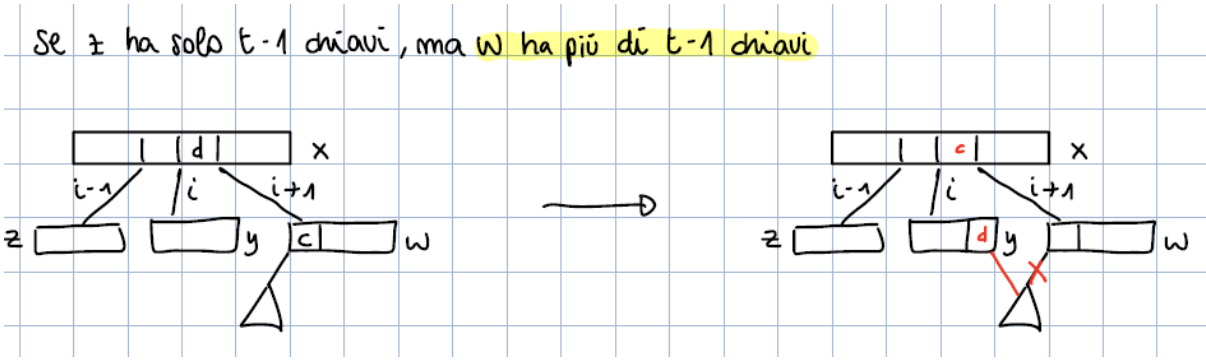


Controllo z e w:

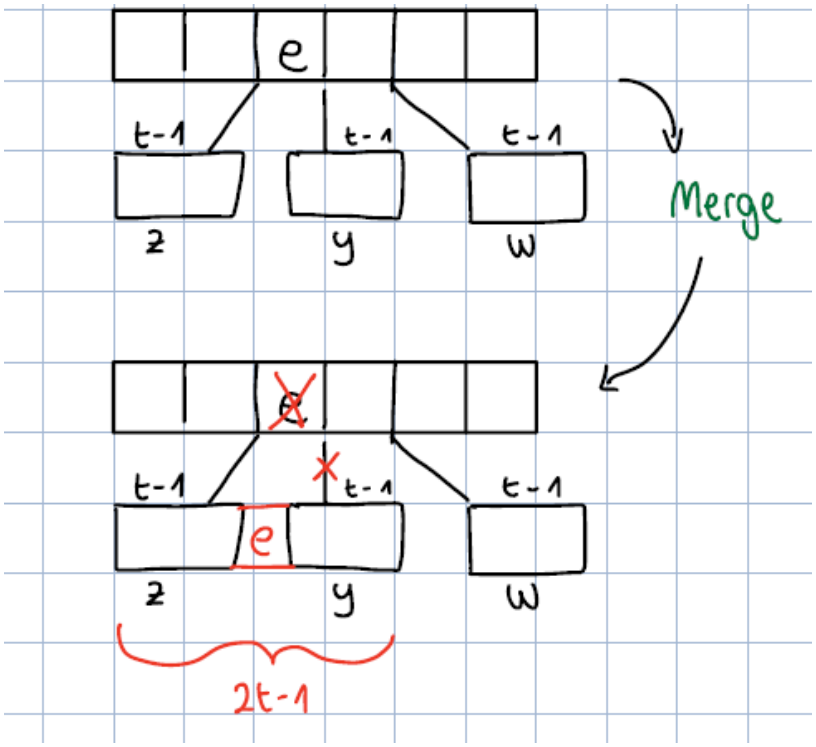
- se **z ha più di t-1 chiavi**: rubo una chiave a z. Sposto in x l'ultima chiave di z (b) e sposto in y la chiave che prima era in x (a) e collego i figli.



- se **w ha più di t-1 chiavi**: rubo una chiave a w. Sposto in x la prima chiave di z (c) e sposto in y la chiave che prima era in x (d) e collego i figli.



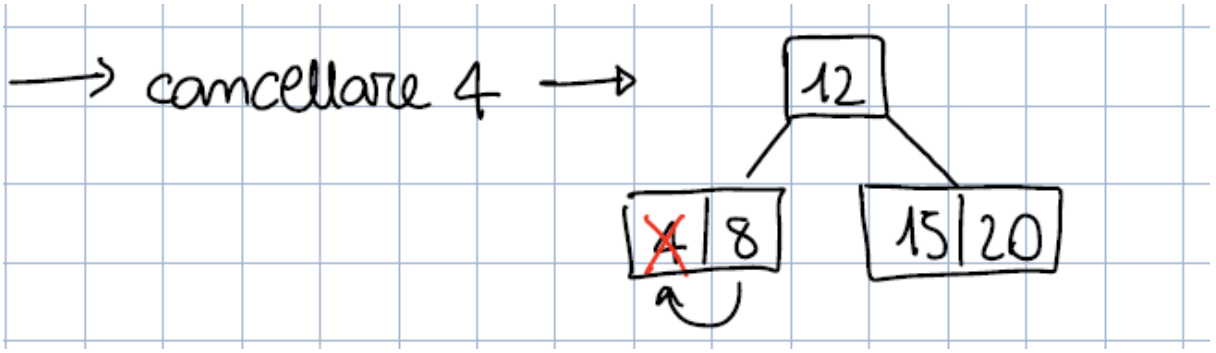
- se **sia w che z hanno t-1 chiavi**: prendo la chiave i-esima del padre e la fondo con i figli y e z (ottenendo quindi un nodo con 2t -1 chiavi).



- **CANCELLAZIONE :** elimino come nei BST, ovvero solo dai nodi foglia.
  Ogni volta che scendo controllo sempre di avere nodi con almeno t-1 chiavi ed eventualmente faccio il Merge.

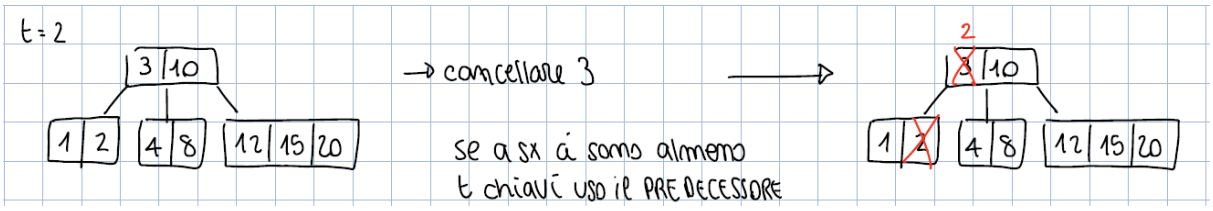
  x e' il nodo corrente, k chiave da eliminare

  - se **x e' una foglia e k e' in x** (ho già eventualmente fatto il Merge se aveva t-1 chiavi) : elimino la chiave x e riordino il vettore key[] shiftando i figli.

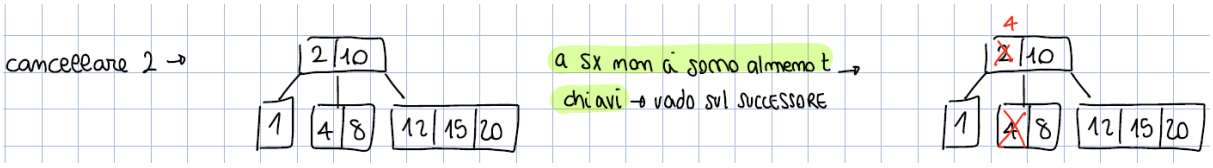


  - se **x non e' una foglia e k e' in x**:

    - se **z ha almeno t chiavi** : sostituisco k con il **predecessore** ed elimino k.

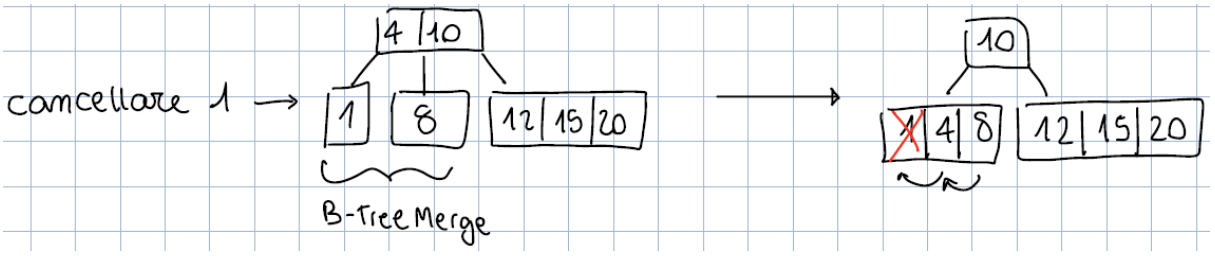


    - se **w ha almeno t chiavi** : sostituisco k con il **successore** ed elimino k.



    - se **z e w hanno t-1 chiavi** : faccio un Merge con la chiave i-esima di x ed unisco i due figli, poi elimino k.

- se **x non e' una foglia e k non e' in x**: individuo il figlio y in cui scendere
    - se **y ha almeno t chiavi** : scendo in t.
    - se **y ha t-1 chiavi** : rubo una chiave ad uno dei due fratelli o Merge di fratello sinistro, y e chiave i-esima del padre.



costi:

- R/W: $O(h) = O(log\ t\ n)$
- CPU: $O(h) * O(t) = O(t * log\ t\ n)$

Posso creare un BTREE da un RBT? si, basta che creo nodi contenenti tutti le chiavi (nodi rbt) rosse e una chiave nera per livello in modo che tutti i nodi siano alla stessa altezza (bh).
→ esiste un SOLO RBT associato ad un BTREE, il viceversa non vale