Esercizio

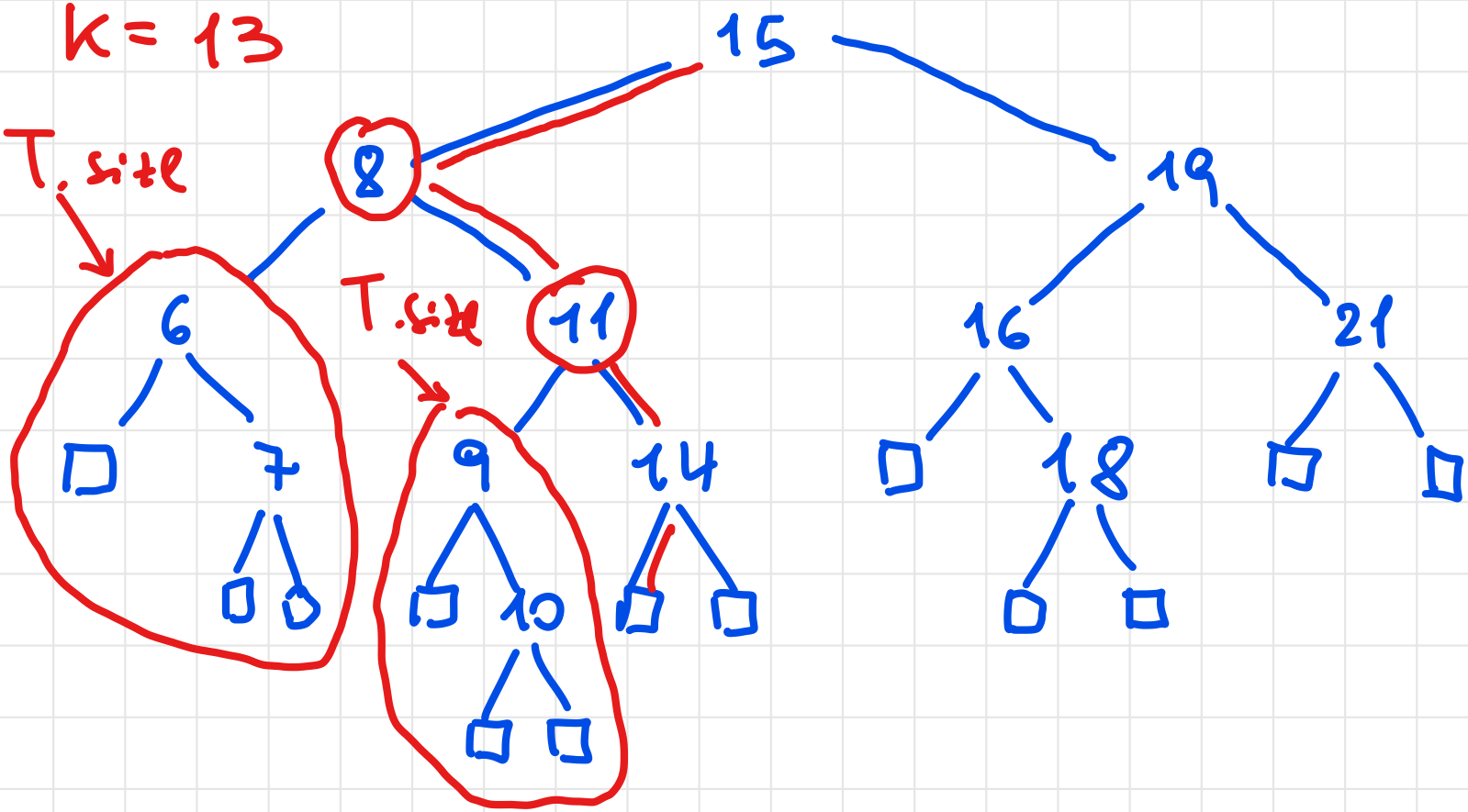
Analizzare la complessità dell'algoritmo sviluppato per l'esercizio della slide 73 della Parte 1, assumendo che al posto della variabile `v.size` si abbia a disposizione un metodo T`.size(v)` che restituisce il numero di entry in T_v, con complessità lineare nel valore restituito. (**Suggerimento:** esprimere la complessità come somma di due termini, uno dei quali è il costo aggregato di tutte le invocazioni del metodo `size`.)

Algoritmo CountLE(k, v)

input
output } stessa specifica del caso con variabili size

ESEMPIO

$k = 13$



if (T.isExternal(v)) then return 0

if (v.getElement().getKey() > k) then

return CountLE(k, T.left(v))

else return 1+T.size(T.left(v))+ CountLE(k, T.right(v))

Complessità di CountLE(k, T.root())

X = # operazioni eseguite, escludendo quelle relative alle chiamate a T.size

Y = # operazioni eseguite in TUTTE le chiamate a T.size

$\Rightarrow$ Complessità $\in \Theta(X+Y)$

So già che $X \in \Theta(h)$ dall'analisi dell'algoritmo che usa la variabile T.size

Sia m = valore finale restituito da CountLE(k, T.root())

Sia m_u = valore restituito da T.size(u)

Sia $U = \{u : \text{l'algoritmo invoca T.size}(u)\}$

nell'esempio $U = \{6, 9\}$

Si ha che $\sum_{u \in U} m_u \leq m$ e al caso pessimo $\sum_{u \in U} m_u \in \Theta(m)$

Ed è facile vedere che $Y \in \Theta\left(\sum_{u \in U} m_u\right)$

$\Rightarrow Y \in \Theta(m)$ al caso pessimo

$\Rightarrow$ Complessità di CountLE(k, T.bot())

è $\Theta(h+m)$

h: X, m: Y

Esercizio

Come cambia l'algoritmo sviluppato per l'esercizio della slide 73 della Parte 1 se non si hanno a disposizione né le variabili né il metodo `size`?

Suggerimento: Sostituire T.size(T.left(v))
con CountLE(k, T.left(v))

Il costo aggregato di tutte le invocazioni di CountLE fatte in sostituzione di T.size
sarà $\Theta(m)$ (m = valore finale restituito)

$\Rightarrow$ Complessità $\in \Theta(h+m)$

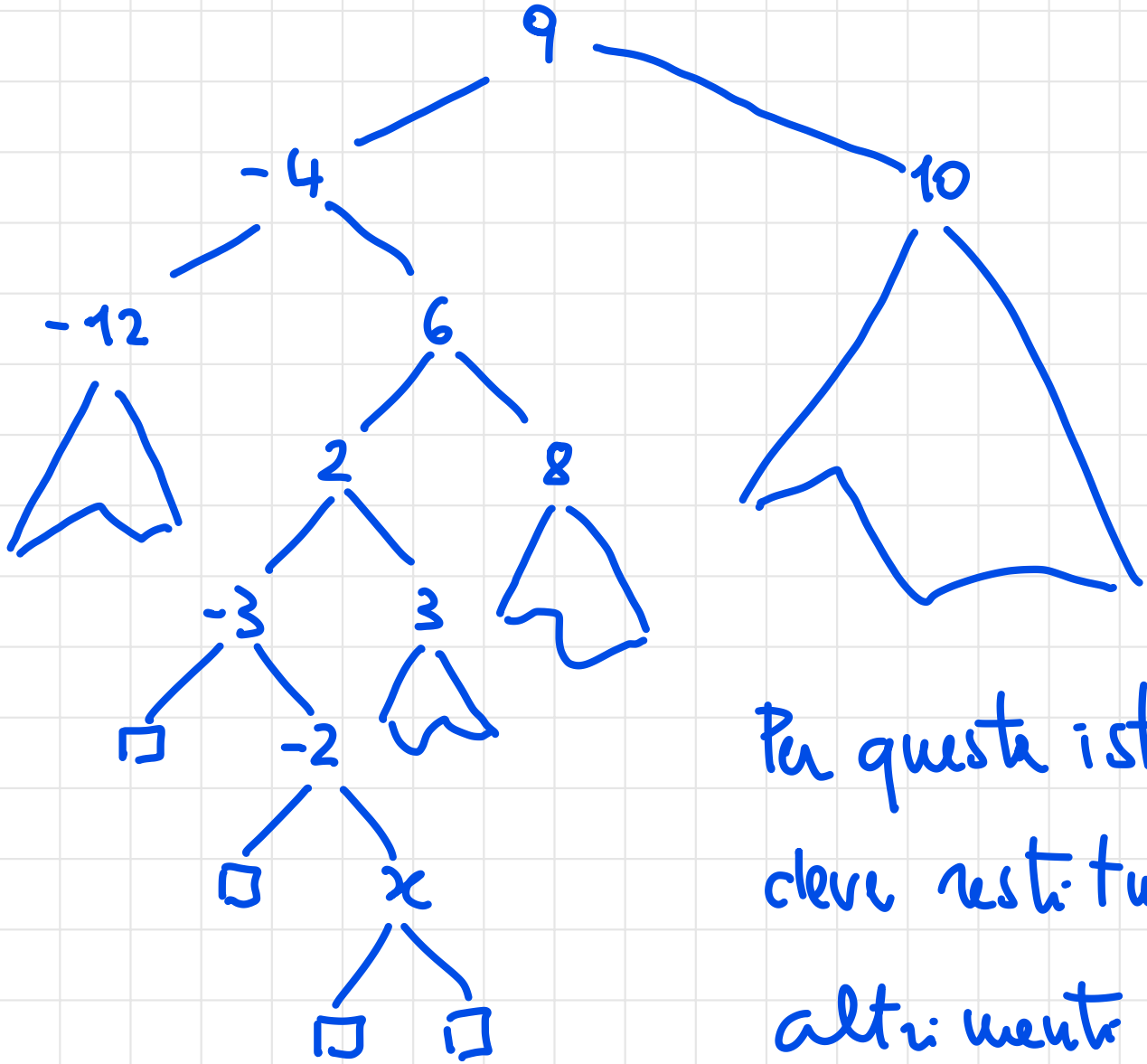
Esercizio: Argomentare bene l'analisi

Esercizio

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

Algoritmo MinPositive(T)
input ABR T con chiavi reali distinte
output minima chiave positiva (se esiste)
o "no positive key" altrimenti

ESEMPIO



Per questa istanza l'algoritmo deve restituire 2, se $x \leq 0$, altrimenti x

IDEA per l'algoritmo

* Si parte dalla radice
* Per ogni nodo incontrato v con chiave k
  - Se k > 0 : salva k come candidato e vai sul figlio Sx
  - Se k ≤ 0 : vai sul figlio dx

Vediamo lo pseudo codice

```
v ← T.root(); minPk ← +∞;
while (T.isInternal(v)) do {
  k ← v.getElement().getKey()
  if (k>0) then {minPk ← k; v ← T.left(v)}
  else v ← T.right(v)
}
if (minPk < +∞) then return minPk
else return "no positive key"
```

Complessità: $\Theta(h)$ h = altezza di T

È dominata dal while

* $\leq h$ iterazioni

* $\Theta(1)$ operazioni per iterazione

Esercizio: Trovare un opportuno invariante per il while, utile per provare la correttezza

Esercizio

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

1. Descrivere tramite pseudocodice la generica invocazione di `MinMatStraniero`, specificandone con attenzione l'input e l'output.
2. Analizzare la complessità di `MinMatStraniero`

Algoritmo MinMatStraniero(T, v)
input ABR T e v ∈ T
output minima matricola di uno straniero in T_v,
se ne esiste almeno uno, o null altrimenti

Prima invocazione: $v = T.root()$

IDEA

* se v è foglia o v.numStr = 0 ⇒ return null
* se v è interno
  - se u (figlio sx d. v) ha u.numStr > 0
    ⇒ return RicRicStringa(T, u)
  - se u.numStr = 0 e la entry in v ha flag 1
    ⇒ return chiave di v (figlio dx di v)
  - altrimenti return RicRicStringa(T, w)

Pseudocode

```
if (T.isExternal(v) OR v.numSt2 = 0) then
    return null
m ← T.left(v).numSt2; x ← v.getElement().
                          getValue()
if (m>0) then return MinNetStream(T,T.left(v))
else {
    if (x=1) then return v.getElement().getKey()
    else return MinNetStream(T,T.right(v))
}
```

Complessità: $\Theta(h)$ con h altezza di T

Stessa analisi di TreeSearch

Esercizio: Fare una versione iterativa dell'algoritmo

Esercizio (Slide 5 della Parte 1 e Slide 89 della Parte 2)

Sia D un documento di N parole, rappresentato da un array $D[0], D[1], \ldots, D[N-1]$.

1. Progettare un algoritmo che, usando una Mappa d'appoggio, restituisca la parola con il massimo numero di occorrenze in D. Se ne esistono più di una, l'algoritmo ne restituisce una arbitraria tra esse.
2. Analizzare la complessità dell'algoritmo scegliendo una opportuna implementazione per la Mappa.

Algoritmo MostFrequentWord(D)

Input: Documento D = D[0] ... D[N-1] di N parole

Output: Una parola w con il massimo numero di occorrenze

IDEA

* Scansione lineare di D
* Mantenere in una mappa di appoggio M coppie (parole, # occorrenze) tenendo anche traccia delle parole più frequenti

Pseudocode

```
M ← mappe vuote; maxCount ← 0;
for i ← 0 to N-1 do {
    count ← M.get(D[i]);
    if (count = null) then count ← 0;
    M.put(D[i], ++count);
    if (count > maxCount) then {
        maxCount ← count
        maxWord ← D[i]
    }
```

}

return maxWord

Complessità

* $\Theta(N)$ operazioni (escluse le get e le put sulle mappe)
* N get e N put sulle Mappe M le cui complessità dipende dalla implementazione scelta per M

Hash Table: N get e N put richiedono

$\Theta(N(1+\lambda))$ operazioni al caso medio dove

λ = load factor della Tabella

(Splay Tree / Red Black Tree): N get e N put richiedono $\Theta(N \cdot \lg m)$ operazioni al caso pessimo, con m = # parole distinte in D

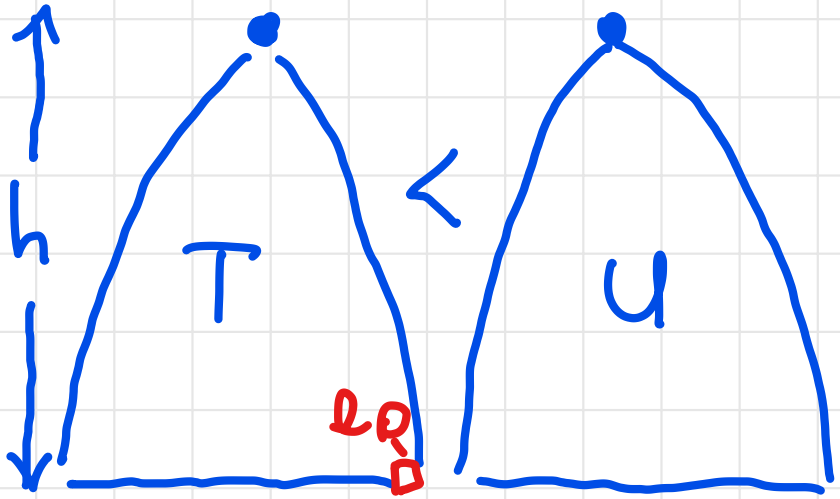
Esercizio

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.

1. Progettare un algoritmo di complessità $O(h)$ per fondere T e U in un unico (2,4)-Tree TU.
2. Dire quali valori può assumere l'altezza di TU e se la radice contiene una o più entry.

① Algoritmo 24TreeMergeEq(T, U)
input (2,4)-Tree T e U d altezza h ciascuno
e max chiave in T < min chiave in U
output (2,4)-Tree TU fusione di T e U

IDEA



e = entry con chiave max in T

metto e in un nuovo nodo r

che diventa la radice del

nuovo (2,4)-Tree TU

e le rimuovo da T

invoco nodo Delete

v ← T.root(); z ← figlio più a dx di v;
while (T.isInternal(z)) do {
  v ← z; z ← figlio più a dx di v
}
e ← entry in v con chiave massima
crea un nuovo nodo radice r contenente e,
  con figlio sx T e figlio dx U
TU ← (2,4)-Tree con radice r
TU.Delete(e, v)

Return TV

Complessità:

* ricerca di e : $\Theta(h)$
* costruzione di TV e x : $\Theta(1)$
* Delete : $\Theta(h)$

$\Rightarrow \Theta(h)$

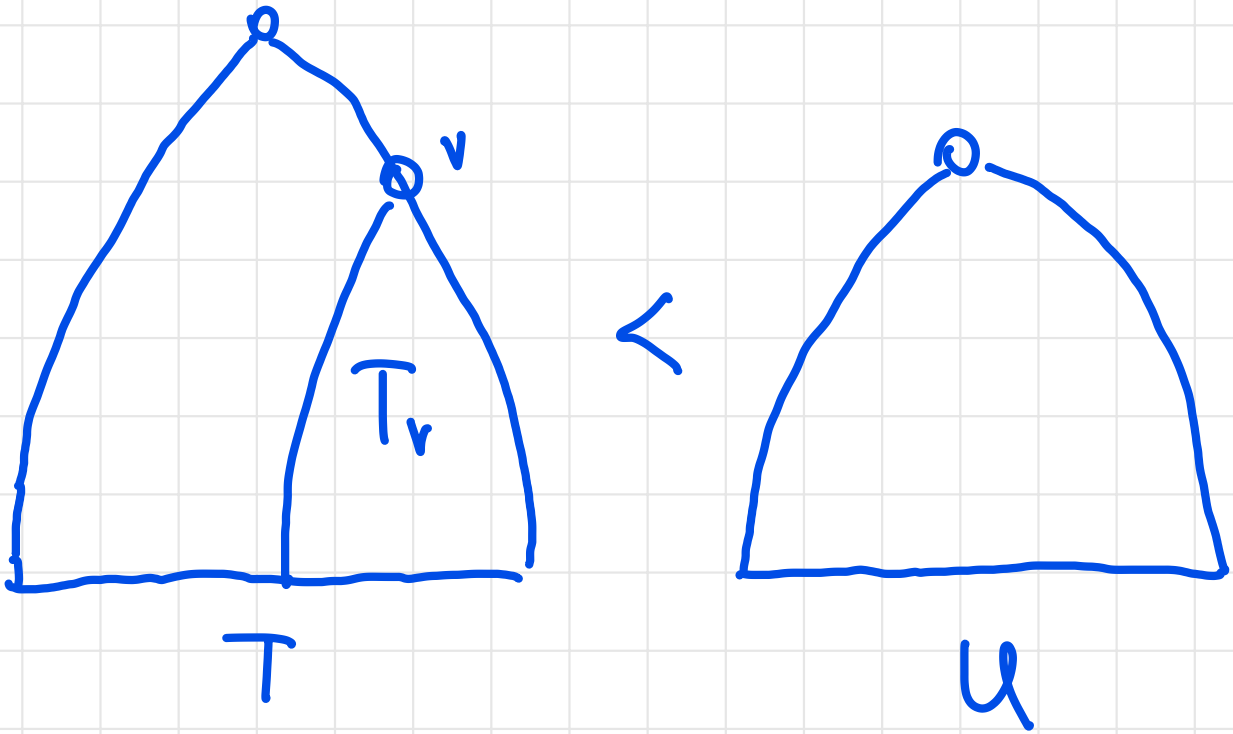
② Se Delete propaga l'underflow alla radice allora l'altezza di TV rimane h, altrimenti l'altezza di TV è h+1 e la radice cambierà

Esercizio

Siano T e U due (2,4)-Tree contenenti rispettivamente n ed m entry e tali che la massima chiave in T è *minore* della minima chiave in U. Progettare un algoritmo di complessità $O(\log n + \log m)$ per fondere T e U in un unico (2,4)-Tree TU.

Suggerimento: Se i due alberi hanno altezze diverse, fondere l'albero di altezza minore con un opportuno sottoalbero dell'altro di uguale altezza (utilizzando l'algoritmo sviluppato per l'esercizio precedente).

CASO 1: Altezza di T > Altezza di U



IDEA

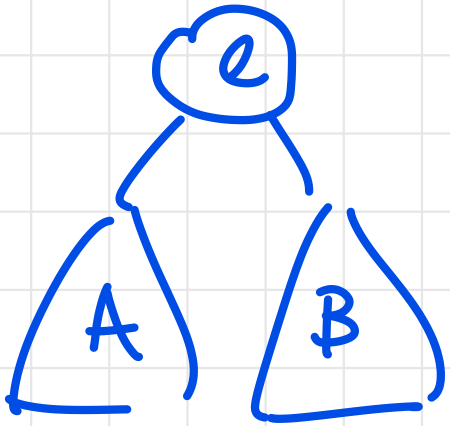
* Identifica il nodo v alla dx di T con la stessa altezza di U
* Fond. T_v e U in un unico (2,4)-Tree T'

con il metodo 24TreeMerge

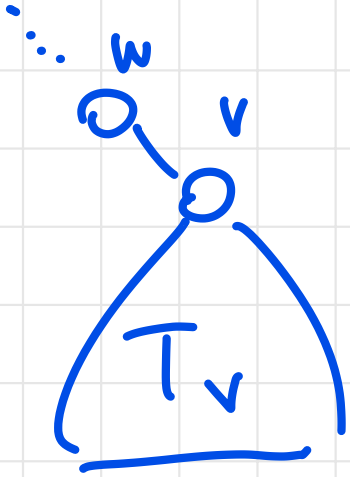
* Se T' ha la stessa altezza di T_v, allora sostituisco T_v con T'
* Altrimenti T' avrà l'altezza di $T_r + 1$ e dell'esercizio precedente so che in questo caso la sua radice ha 1 entry. Allora metto T' al posto di T_v fondendo la sua radice con il padre di v e invocando split se il padre di v va in overflow

In questo secondo caso devo fare attenzione

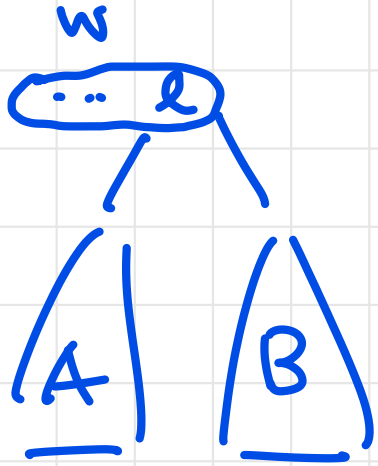
T' =



T =



$\Rightarrow$



Se w va in overflow con e, si invoca Split

Aggiungere i dettagli come esercizio.