Van Emde Boas tree

Alex Falzone

2020/2021

1. Introduzione

Creato nel 1975 dall'informatico Peter van Emde Boas e successivamente affinato da van Emde Boas, Kaas e Zijlstra.
Questo albero supporta le operazioni di inserimento, cancellazione, successore, predecessore, massimo, minimo e ricerca tutte nel tempo O(lg(lg(u))). Un albero di van Emde Boas (abbreviato anche come vEB) deve avere delle caratteristiche ben precise:

- Un attributo **u** che rappresenta la dimensione dell'albero
- Un attributo **min** che memorizza l'elemento minimo.
- Un attributo **max** che memorizza l'elemento massimo.
- Un array **cluster** di dimensione $\sqrt{u}$
- Un array **summary**[0 ... $\sqrt{u}$ - 1], dove summary[i] contiene 1 se e soltanto se l'i-esimo cluster[i$\sqrt{u}$ ... (i + 1)$\sqrt{u}$ - 1] contiene un 1.

Inoltre non sono ammesse chiavi duplicate.

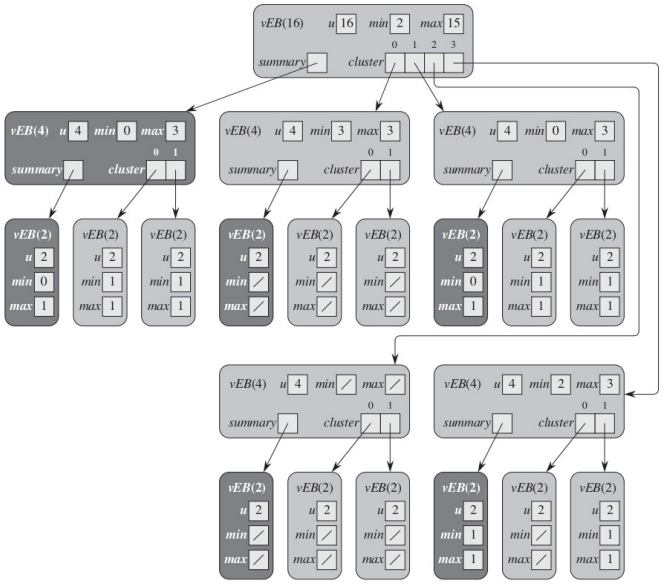


Figure 1: Esempio albero di van Emde Boas (immagine tratta da "Introduction to Algorithm")

2. Struttura dell'albero

Indichiamo con $\sqrt[\downarrow]{u}$ la radice quadrata **inferiore**, ovvero $2^{\lfloor lg(u)/2 \rfloor}$.
Mentre con $\sqrt[\uparrow]{u}$ la radice **superiore**, ovvero $2^{\lceil lg(u)/2 \rceil}$.
Denotiamo con vEB(u) un albero vEB con dimensione dell'universo pari a **u** e, a meno che u non sia uguale alla dimensione base 2, l'attributo summary punta a un albero vEB($\sqrt[\uparrow]{u}$) e l'array cluster[0 ... $\sqrt[\uparrow]{u}$ - 1] punta ai $\sqrt[\uparrow]{u}$ alberi vEB($\sqrt[\downarrow]{u}$).
Successivamente è necessario stabilire come accedere sia al numero di cluster di un determinato valore(x), sia alla posizione del valore(x) all'interno del cluster; Esso avviene usando rispettivamente:

$$high(x) = \lfloor x / \sqrt[\downarrow]{u} \rfloor \tag{1}$$

$$low(x) = x mod \sqrt[\downarrow]{u} \tag{2}$$

$$index(x, y) = (x \sqrt[\downarrow]{u}) + y \tag{3}$$

Dove index rappresenta il numero(x) e la posizione(y) del cluster.
Inoltre è importante fissare una ricorrenza che tornerà utile nei costi delle singole operazioni.

$$T(u) \leq T(\sqrt[\uparrow]{u}) + O(1) \tag{4}$$

Ponendo $m = lg(u)$ abbiamo:

$$T(2^m) \leq T(2^{\lceil m/2 \rceil} + O(1)$$

Poiché $\lceil$ m/2 $\rceil \leq 2m/3$ per ogni $m \geq 2$, si ha

$$T(2^m) \leq T(2^{2m/3} + O(1)$$

Ponendo S(m) = T(2^m), riscriviamo l'occorrenza come

$$S(m) \leq S(2m/3) + O(1)$$

Grazie al secondo caso del teorema master la ricorrenza ha soluzione S(m) = O(lg m).
Dunque abbiamo T(u) = T(2^m) = S(m) = O(lg m) = O(lg lg u)

3. Operazioni

4.0.1 Minimo e massimo

MINIMO(V)

1. **return** V.min

MASSIMO(V)

1. **return** V.max

Poiché il massimo e il minimo sono memorizzati negli attribuiti **min** e **max** essi richiedono tempo costante, ovvero O(1).

4.0.2 Successore e Predecessore

SUCCESSORE(V, key)

1. **if** V.universe == 2
2. **if** key == 0 and $key < V.min$
3. **return** 1
4. **else return** NIL
5. **else if** $V.min \neq NIL$ and $key < V.min$
6. **return** V.min
7. **else** max-low = MASSIMO(V.cluster[high(key)])
8. **if** $max - low \neq NIL$ and $low(key) < max - low$
9. offset = SUCCESSORE(V.cluster[high(key)], low(key))
10. **return** index(high(key), offset)
11. **else** succ-cluster = SUCCESSORE(V.summary, high(key))
12. **if** succ-cluster == NIL
13. **return** NIL
14. **else** offset = MINIMO(V.cluster[succ-cluster])
15. **return** index(succ-cluster, offset)

Iniziamo dal caso base, in cui la dimensione dell'albero è pari a 2. In questo caso se key è uguale a 0 e 1, ovvero il nostro ipotetico successore, si trova nel medesimo cluster ritorniamo quest'ultimo. Altrimenti non sarà presente nessun successore e ritorniamo NIL.
Alla riga 5 verifichiamo se esiste il minimo e se la key è minore di quest'ultimo, in caso affermativo la riga 6 restituisce il minimo.
Se ci troviamo alla riga 7 sappiamo che non ci troviamo in un caso base e che key è maggiore del minimo. In questo caso settiamo max-low al massimo valore del cluster di key e (alla riga 8) verifichiamo che il cluster di key abbia qualche elemento maggiore di key stesso, infatti in caso di esito positivo sappiamo che il successore di key si trova da qualche parte nel cluster di key. Se il successore di key si trova all'interno del cluster di key, la riga 9 determina dove si trova e la riga 10 restituisce il successore.
Se arriviamo alla riga 11 allora il cluster di key non ha nessun elemento maggiore di key e dobbiamo cercarlo in cluster successivi. Assegniamo a succ-cluster il cluster successivo di key e se esso è diverso da NIL troviamo il minimo valore del cluster successivo a key (riga 14) e lo restituiamo (riga 15), altrimenti torniamo NIL (riga 13).
La ricorrenza (4) caratterizza il tempo di esecuzione di SUCCESSOR, infatti la procedura chiama se stessa alla riga 9 (su un albero con dimensione **u** pari a $\sqrt[\downarrow]{u}$) o alla riga 11(su un albero con dimensione **u** pari a $\sqrt[\uparrow]{u}$). Quindi controllerà al più un albero vEB con dimensione dell'universo pari $\sqrt[\uparrow]{u}$. Le procedure MINIMO e MASSIMO richiedono tempo O(1), quindi SUCCESSOR viene eseguita nel tempo O(lg(lg(u))) nel caso peggiore.

La procedura PREDECESSORE è simmetrica alla procedura SUCCESSORE, ma con un caso aggiuntivo.

PREDECESSORE(V, key)

1. **if** V.universe == 2
2. **if** key == 1 and $key > V.max$
3. **return** 0
4. **else return** NIL
5. **else if** $V.max \neq NIL$ and $key > V.max$
6. **return** V.max
7. **else** min-low = MINIMO(V.cluster[high(key)])
8. **if** $min - low \neq NIL$ and $low(key) > min - low$
9. offset = PREDECESSORE(V.cluster[high(key)], low(key))

10. **return** index(high(key), offset)
11. **else** pred-cluster = PREDECESSORE(V.summary, high(key))
12. **if** pred-cluster == NIL
13. **if** $V.min \neq NIL$ and $key > V.min$
14. **return** V.min
15. **else return** NIL
16. **else** offset = MINIMO(V.cluster[succ-cluster])
17. **return** index(succ-cluster, offset)

Il caso aggiuntivo (alle righe 13-14) si verifica quando il predecessore di key, se esiste, non si trova nel cluster di key. Quindi se il predecessore di key è il valore minimo nell'albero vEB V, allora il predecessore non si trova in alcun cluster. Questo caso extra non influisce sul tempo di esecuzione dell'algoritmo, di conseguenza il costo sarà uguale a quello di SUCCESSOR, ovvero O(lg lg u).

4.0.3 Inserimento

Empty-Insert(V, key)

1. V.min = key
2. V.max = key

Questa funzione ci tornerà utile in INSERIMENTO, per cercare di smaltire il codice.

INSERIMENTO(V, key)

1. **if** V.min == NIL
2. Empty-Insert(V, key)
3. **else if** $x < V.min$
4. scambia key con V.min
5. **if** $V.universe > 2$
6. **if** MINIMO(V.cluster[high(key)]) == NIL
7. INSERIMENTO(V.summary, high(key))
8. Empty-Insert(V.cluster[high(key)], low(key))

9. **else** INSERIMENTO(V.cluster[high(key)], low(key))
10. **if** $key > V.max$
11. V.max = key

Innanzitutto è necessario verificare se l'albero è vuoto, se lo è settiamo min e max a key(riga 1-2).
Successivamente alla riga 3 verifichiamo se key è minore del valore minimo, in tal caso scambiamo min e key. Di conseguenza la nostra key originale diverrà il minimo e il nostro minimo originale diverrà key.
Alla riga 5 se l'albero non si trova nel caso base allora dobbiamo determinare se il cluster dove andrà key è vuoto (riga 6), se lo è, inseriamo il numero di cluster di key nel summary, creiamo un nuovo cluster e inseriamo key come min e max (riga 7-8).
Se invece il cluster non è vuoto inseriamo key nel cluster corrispondente (riga 9).
Le righe 10-11 hanno il compito di aggiornare il campo max se e solo se esso è minore di key.
Possiamo notare come la ricorrenza (4) caratterizzi anche questo algoritmo, infatti viene eseguita la chiamata ricorsiva o nella riga 8 (su un albero vEB con dimensione dell'universo $\uparrow\sqrt{u}$) o nella riga 12 (su un albero vEB con dimensione dell'universo $\downarrow\sqrt{u}$). La parte restante dell'algoritmo richiede tempo O(1), di conseguenza applicando la ricorrenza (4) abbiamo che il tempo di esecuzione di questo algoritmo è pari a O(lg(lg(u))).

4.0.4 Cancellazione

CANCELLAZI0NE(V, key)

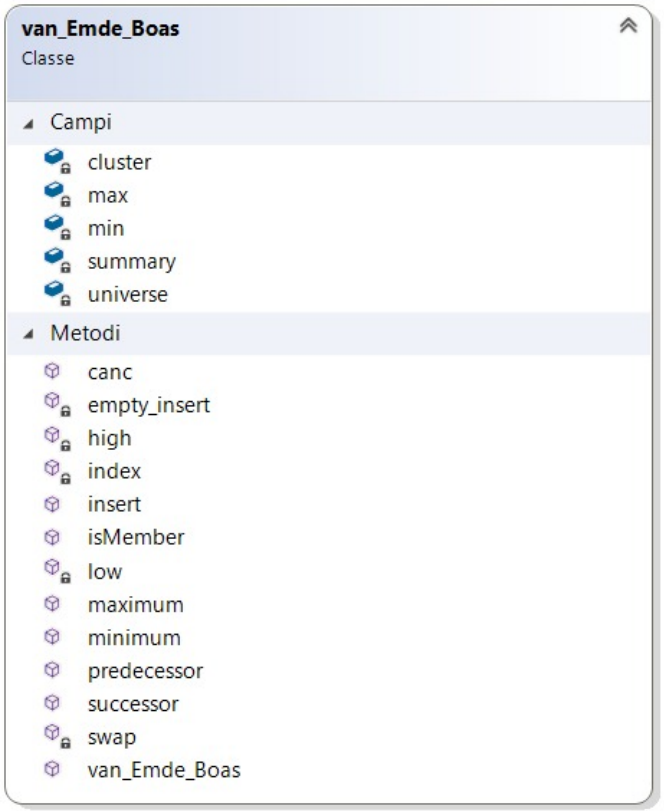
1. **if** V.min == V.max
2. V.min = NIL
3. V.max = NIL
4. **else if** V.universe == 2
5. **if** key == 0
6. V.min = 1
7. **else** V.min = 0
8. V.max = V.min
9. **else if** key == V.min
10. first-cluster = MINIMO(V.summary)

```
11.         x = index(first-cluster, MINIMO(V.cluster[first-cluster])
12.         V.min = key
13.     CANCELLAZIONE(V.cluster[high(key)], low(key))
14.     if MINIMO(V.cluster[high(key)]) == NIL
15.         CANCELLAZIONE(V.summary, high(key))
16.         if x == V.max
17.             summary-max = MASSIMO(V.summary)
18.             if summary-mx == NIL
19.                 V.max = V.min
20.             else V.max = index(summary-max,
                                   MASSIMO(V.cluster[summary-max]))
21.     else if key == V.max
22.         V.max = index(high(key), MASSIMO(V.cluster[high(key)]))
```

Se l'albero vEB V contiene un solo elemento, allora impostiamo min e max a NIL. Le righe 1-3 gestiscono questo caso.
Altrimenti l'albero ha almeno 2 elementi. La riga 4 controlla se è un albero nel caso base, se lo è, le righe 5-8 impostano min e max all'elemento rimasto. Successivamente sappiamo che l'albero avrà due o più elementi e che la dimensione 'universe' sarà maggiore o uguale a 4; Di conseguenza dovremmo cancellare un elemento da un cluster. Tuttavia, l'elemento che cancelliamo da un cluster potrebbe non essere key, perché se key è uguale a min, allora una volta cancellato key stesso, qualche altro elemento all'interno di uno dei cluster dell'albero vEB diventa il nuovo min, e dovremo cancellare tale elemento dal suo cluster. Se il test nella riga 9 rivela che ci troviamo in questo caso, allora la riga 10 imposta first-cluster al numero del cluster che contiene il più piccolo elemento diverso da min, e la riga 11 imposta key al valore del più piccolo elemento in tale cluster. Questo elemento diventa il nuovo min nella riga 12 e, poiché impostiamo key al valore di tale elemento, esso è l'elemento che sarà cancellato dal suo cluster.
Quando arriviamo alla riga 13, sappiamo che dobbiamo cancellare l'elemento key dal suo cluster, sia che esso sia il valore originariamente passato a Cancellazione e sia che sia l'elemento che deve diventare il nuovo minimo. La riga 13 cancella key dal suo cluster. Questo cluster adesso potrebbe essere vuoto; la verifica di ciò avviene nella riga 14. Se il cluster è vuoto, occorre rimuovere il numero del cluster di key dal summary; questo compito è svolto dalla riga 15. Dopo avere aggiornato il summary, occorre aggiornare max. La riga 16 verifica se stiamo cancellando l'elemento massimo in vEB e, in caso

affermativo, la riga 17 imposta summary-max al numero del cluster non vuoto con il numero più grande. Se tutti i cluster di vEB sono vuoti, allora l'unico elemento rimasto è min; la riga 18 verifica questo caso, e la riga 19 aggiorna max in modo appropriato. Altrimenti, la riga 20 imposta max al massimo elemento nel cluster con il numero più grande. (Se questo è il cluster in cui abbiamo cancellato l'elemento, ci affidiamo di nuovo alla chiamata ricorsiva nella riga 13 che ha già corretto l'attributo max di tale cluster.)
Infine, dobbiamo gestire il caso in cui il cluster di key non sia vuoto dopo la cancellazione di key stesso. Sebbene non occorra aggiornare il summary in questo caso, potrebbe essere necessario aggiornare max. La riga 21 verifica questo caso, e se occorre aggiornare max, ciò avviene nella riga 22.
Anche questo metodo ha tempo di esecuzione nel caso peggiore O(lg lg u)

4. UML


van_Emde_Boas
Classe
Campi
cluster
max
min
summary
universe
Metodi
canc
empty_insert
high
index
insert
isMember
low
maximum
minimum
predecessor
successor
swap
van_Emde_Boas