Van Emde Boas tree

Alex Falzone

2020/2021

1. Introduzione

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

1. Un attributo **u** che rappresenta la dimensione dell'albero
2. Un attributo **min** che memorizza l'elemento minimo.
3. Un attributo **max** che memorizza l'elemento massimo.
4. Un array **cluster** di dimensione $\sqrt{u}$
5. 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.

2. Struttura dell'albero

Indichiamo con $\sqrt[\downarrow]{u}$ la radice quadrata **inferiore**, ovvero

$$2^{\lfloor lg(u)/2 \rfloor}. \tag{1}$$

Mentre con $\sqrt[\uparrow]{u}$ la radice **superiore**, ovvero

$$2^{\lceil lg(u)/2 \rceil} \tag{2}$$

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($\sqrt[\downarrow]{u}$).
Inoltre all'interno di un cluster è presente un valore **min** e un valore **max**. Banalmente memorizzano rispettivamente l'elemento minimo e massimo nell'albero vEB. Inoltre è importante notare che l'elemento memorizzato in **min** non appare in nessuno degli albero di ricorsione vEB($\sqrt[\downarrow]{u}$) cui punta l'array cluster (diversamente accade per **max**).
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:

$$high(x) = \lfloor x / \sqrt[\downarrow]{u} \rfloor \tag{3}$$

$$low(x) = x mod \sqrt[\downarrow]{u} \tag{4}$$

$$index(x, y) = (x \sqrt[\downarrow]{u}) + y \tag{5}$$

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.

$$T(u) \leq T(\sqrt[\uparrow]{u}) + O(1) \tag{6}$$

Ponendo $m = lg(u)$ abbiamo:

$$T(2^m) \leq T(2^{\lceil m/2 \rceil} + O(1) \tag{7}$$

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

$$T(2^m) \leq T(2^{2m/3} + O(1) \tag{8}$$

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

$$S(m) \leq S(2m/3) + O(1) \tag{9}$$

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)

3.5 Operazioni vEB

3.5.1 Minimo e massimo

Minimum(V) return V.min Maximum(V) return V.max

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

3.5.2 Successore e Predecessore

Successore

Successor(V, key) ($V.universe == 2$)
(($key == 0$) **and** ($V.max == 1$)) **return** 1 **return** NIL (($V.min \neq NIL$) **and** ($key < V.min$)) **return** V.min $max - low = Maximum(V.cluster[high(key)]$ (($max–low \neq NIL$) **and** ($low(key) < max–low$)) offset = $Successor(V.cluster[high(key)], low(key))$ **return** $index(high(key), offset)$ succ-cluster = $Successor(V.summary, high(key))$ (succ-cluster == NIL) **return** NIL offset = Minimum(V.cluster[succ-cluster]) **return** index(succ-cluster, offset)

Iniziamo dal caso base, ovvero la dimensione di **u** è uguale a 2; In questo caso se la chiave da ricercare si trova nella posizione 0 del determinato cluster, ed esiste **max** di quel cluster, allora il successore sarà proprio **max**, perché non ci saranno altri elementi se non key e **max**.
Altrimenti ritorniamo NIL, perchè non ci sarà nessun successore.
Se la dimensione di **u** è diversa da 2, quindi abbiamo saltato la prima condizione, ed esiste **min** ed esso è strettamente maggiore di key allora ritorniamo semplicemente **min**, che sarà appunto il nostro successore.
Se arriviamo alla terza condizione (else) allora sappiamo che non ci troviamo in un caso base e che key è maggiore o uguale all'elemento minimo. Assegniamo a max-low l'elemento massimo nel cluster di key.
Se il cluster di key contiene qualche elemento che è maggiore di key, allora sappiamo che il successore di key si trova proprio nel cluster di key. Se il successore di key è all'interno del cluster di key, allora il primo offset determina dove si trova e successivamente restituiamo il successore.
Altrimenti assegniamo a succ-cluster il numero del successivo cluster non vuoto, aiutandoci con summary. Se succ-cluster è non vuoto assegniamo al secondo offset il primo elemento all'interno di tale cluster e successivamente restituiamo l'elemento minimo di quel cluster.

Per quanto riguarda la complessità possiamo notare che l'algoritmo richiama se stesso, in modo ricorsivo, nella riga 12 (su un albero vEB con dimensione dell'universo $\sqrt[\downarrow]{u}$) o nella riga 15 (su un albero vEB con dimensione dell'universo $\sqrt[\uparrow]{u}$). In ogni caso, la chiamata ricorsiva riguarda un albero vEB con dimensione dell'universo al più $\sqrt[\downarrow]{u}$. La parte restante incluso Minimum e Maximum richiede tempo costante, ovvero O(1). Quindi applicando la ricorrenza (6), Successor viene eseguita nel tempo O(lg lg u).

Predecessore

La procedura Predecessor è simmetrica a quella di Successor, ma con un caso aggiuntivo: Predecessor(V, key) ($V.universe$ == 2)

((key == 1) **and** ($V.min$ == 0)) **return** 0 **return** NIL (($V.max \neq NIL$) **and** ($key > V.max$)) **return** V.max $min - low = Minimum(V.cluster[high(key)]$ (($min–low \neq NIL$) **and** ($low(key) > min–low$)) offset = $Predecessor(V.cluster[high(key)], low(key))$ **return** $index(high(key), offset)$ pred-cluster = $Predecessor(V.summary, high(key))$ (pred-cluster == NIL) (($V.min \neq NIL$) and ($x > V.min$)) **return** V.min **return** NIL offset = Maximum(V.cluster[pred-cluster]) **return** index(pred-cluster, offset)

Il caso aggiuntivo (alle righe 17-18) 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).

3.5.3 Insert

Vediamo come inserire un elemento in un albero vEB. Quando inseriamo un elemento, il cluster dove va inserito l'elemento può contenere già un elemento oppure no. Se il cluster ha già un altro elemento, allora il numero di cluster è già nel summary. Se il cluster non ha un altro elemento, allora l'elemento da inserire diventa l'unico elemento nel cluster.

Insert(V, key) ($V.min == NIL$) $V.min = key$ $V.max = key$ ($key < V.min$) swap(key, V.min) ($V.universe > 2$) Minimum(V.cluster[high(key)]) == NIL Insert(V.summary, high(key)) $V.cluster[high(key)].min = low(key)$ $V.cluster[high(key)].max = low(key)$ Insert(V.cluster[high(key)], low(key)) ($key > V.max$) $V.max = key$

Innanzitutto è necessario verificare se l'albero è vuoto, se lo è settiamo **min** e **max** a key.
Successivamente alla riga 4 verifichiamo se key è minore del minimo, in tal caso scambiamo min e key. Di conseguenza la nostra key originale diverrà il minimo e il nostro minimo originale diverrà key.
Se l'albero non si trova nel caso base (ovvero $V.universe > 2$, riga 6) allora dobbiamo determinare se il cluster dove andrà key è vuoto, se lo è, inseriamo il numero di cluster di key nel summary e alle righe 9 e 10 creiamo un nuovo cluster e inseriamo key come **min** e **max**.
Se invece il cluster non è vuoto inseriamo key nel cluster corrispondente.
Le righe 15-16 hanno il compito di aggiornare il campo **max** se e solo se esso è minore di key.

Possiamo notare come la ricorrenza (6) 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 (6) abbiamo che il tempo di esecuzione di questo algoritmo è pari a O(lg lg u).

3.5.4 Delete