Дървета на ван Емде Боас

(van Emde Boas (vEB) trees)

Introduction to Algorithms - 3rd Edition, THOMAS H. CORMEN CHARLES E. LEISERSON RONALD L. RIVEST CLIFFORD STEIN

До този момент сме разглеждали структури от данни, които поддържат операциите на приоритетна опашка - бинарният хийп ($binary\ heap$) или така наречената пирамида, червено-черни дървета, AVL балансирани дървета и др. Във всяка една от тези структури, поне една основна операция отнема $O(\log n)$ време (навсякъде по-долу където пишем „време“ ще имаме предвид времева сложност) в най-лошия сценарий или амортизирано. В действителност, тъй като всяка една от изброените по-горе структури от данни базира решението си на сравняването на ключове, долната граница от $\Omega(n \log n)$ за сортиране ни казва, че поне една операция ще отнеме $\Omega(\log n)$ време. Ако може да изпълняваме и двете операции $INSERT$ и $EXTRACT–MIN$ на минимален елемент за $o(\log n)$ време, то тогава ще може да сортираме n ключа за $o(n \log n)$ време като първо изпълним n операции $INSERT$, последвани от n операции от $EXTRACT–MIN$.

Въпреки това, видяхме че понякога може да използваме допълнителна информация относно ключовете, които сортираме, за да го направим за $o(n)$ време. Именно с метода на броене ($counting$) може да сортираме n ключа, всеки от които е в интервала от 0 до k, за време $\Theta(n + k)$, което е $\Theta(n)$, когато $k = O(n)$.

Тъй като може да заобиколим долната граница от $\Omega(n \log n)$ за сортиране, когато ключовете са цели числа в ограничен интервал, възниква въпроса дали може да заобиколим и операциите на приоритетната опашка за $o(\log n)$ време по подобен начин. Ще видим, че това е възможно с дърветата на ван Емде Боас, които поддържат операциите на приоритетна опашка и няколко други, всяка от които е за $O(\log \log n)$ време в най-лошия случай. Допълнителната информация е, че ключовете трябва да са в интервала от 0 до $n - 1$, без да се позволяват дубликати.

По-конкретно, дърветата на ван Емде Боас поддържат всяка една от динамичните операции $SEARCH$, $INSERT$, $DELETE$, $MINIMUM$, $MAXIMUM$, $SUCCESSOR$ и $PREDECESSOR$ за $O(\log \log n)$ време.

До тук използвахме параметъра n за две цели: да бележим с него броя на елементите на динамично множество и обхвата на възможните стойности. За да избегнем бъдещи объркавания, от сега нататък ще използваме n за да бележим броя на елементите, които се намират в множеството и с u обхвата на възможните стойности. Така всяка една от ван Емде Боас операциите се изпълнява за $O(\log \log u)$ време. Ще наричаме множеството $\{0, 1, 2, \ldots, u - 1\}$ Вселена ($universe$) от стойности, които могат да се съхраняват и u ще е размера на Вселената. Ще допускаме, че u е някаква точна степен на двойката, т.е. $u = 2^k$ за някое цяло число $k \geq 1$.

Преди да стигнем до желаната сложност на тези операции ще разгледаме множество подходи, които да ни напътят в правилната посока на мислене. След това ще подобряваме тези подходи докато стигнем до прототип на дърво на ван Емде Боас, което е рекурсивно, но все още не достига желаното време за изпълнение на операциите. Накрая ще покажем как може да подобрим, развием и имплементираме прототипа по начин, който ни позволява времева сложност от $O(\log \log n)$ за желаните операции.

1. Предварителни подходи

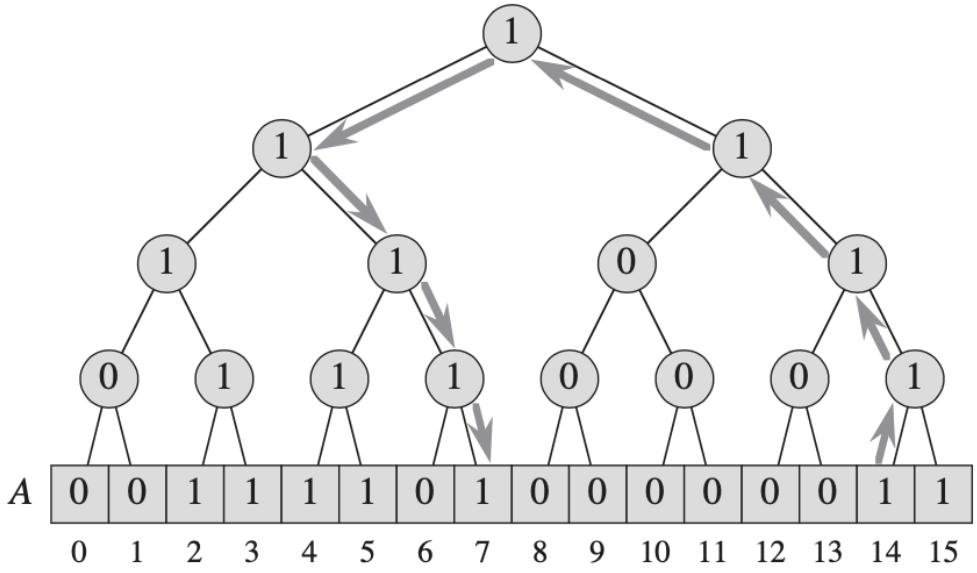
1.1. Директно адресиране

Директното адресиране осигурява най-простия подход за съхраняване на динамично множество. Тъй като ще се занимаваме само със съхраняване на ключове, може да опростим подхода с директно адресиране, за да съхраним динамичното множество като битов вектор. За да съхраняваме динамично множество от стойности от Вселената

$\{0, 1, 2, \ldots, u-1\}$, ни е необходим масив $A[0..u-1]$ от u бита. На клетката $A[x]$ има вдигнат бит 1, ако стойността x е в динамичното множество, а в противен случай съдържа 0. Въпреки, че може да изпълним всяка операция $INSERT$, $DELETE$ и $MEMBER$ за $O(1)$ време с бит вектор, останалите операции - $MINIMUM$, $MAXIMUM$, $SUCCESSOR$, $PREDECESSOR$ отнемат $\Theta(u)$ време в най-лошия сценарии, тъй като може да ни се наложи да обикаляме през $\Theta(u)$ елементи. Например, ако множеството съдържа единствено стойностите 0 и $u-1$, то тогава за да намерим наследника (следващия по-голям от него елемент - $successor$) на 0, е необходимо да обиколим всички клетки от 1 до $u-2$ преди да намерим, че на позиция $A[u-1]$ има 1.

1.2. Наслагване на двоична дървесна структура

Може да прекъснем дългите обхождания в битовия вектор, като насложим двоично дърво от битове върху него. На фигурата по-долу е показан пример.



Фиг. 1. Двоично дърво от битове, насложено върху битов вектор, представляващ множеството $\{2, 3, 4, 5, 7, 14, 15\}$*, когато* $u = 16$*. Всеки връх, който не е листо (всеки вътрешен връх) съдържа* 1 *тогава и само тогава, когато в поддървото му някой лист съдържа* 1*. Стрелките показват пътя, следван за определяне на предшественика на* 14 *в множеството.*

Записите на битовия вектор образуват листата на двочното дърво и всеки вътрешен връх съдържа 1, тогава и само тогава когато, който и да е лист в поддървото му съдържа 1. С други думи, битът, съхраняван във вътрешния връх, е *логическото-или* на двете си деца.

Операциите, които отнемаха $\Theta(u)$ в най-лошия случай без наставенена битова дървесна структура, сега може да използват тази структура, като:

- За да намерим минималната стойност в множеството, започваме от корена и се насочваме надолу към листата, като винаги вземаме най-левия възел съдържащ 1.
- За да намерим максималната стойност в множеството, започваме от корена и се насочваме надолу към листата, като винаги вземаме най-десния възел, съдържащ 1.
- За да намерим наследника на x, започваме от листа, индексиран с x, и се насочваме нагоре към корена, *докато не влезем във връх отляво* и този връх има 1 в дясното си дете z. Тогава се насочваме надолу през върха z, като винаги взиаме най-левия връх съдържащ 1 (т.е. намираме минималната стойност в поддървото, с корен дясното дете z).

- За да намерим предшественика на x, започваме от листа, индексиран с x, и се насочваме нагоре към корена, *докато не влезем във връх отдясно* и този връх има 1 в лявото си дете z. Тогава се насочваме надолу през върха z, като винаги взимаме най-десния връх съдържащ 1 (т.е. намираме максималната стойност в поддървото, с корен лявото дете z).

По същия начин може да аргументираме операциите $INSERT$ и $DELETE$.

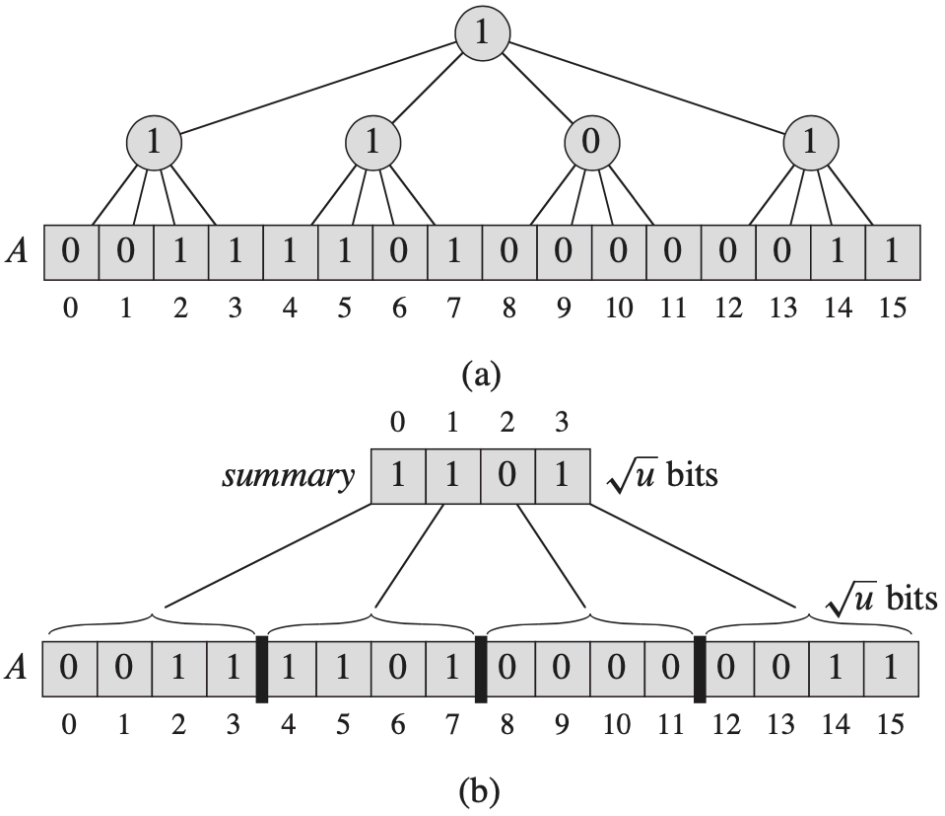
- Когато вмъкваме стойност, съхраняваме 1 във всеки връх по простия път от съответния лист нагоре до корена.
- Когато изтриваме стойност, преминаваме от съответния лист нагоре към корена, преизчислявайки бита във всеки вътрешен възел по пътя като *логическото-или* на двете му деца.

Тъй като височината на дървото е $\log u$ и всяка от горните операции прави най-много едно преминаване нагоре по дървото и най-много едно преминаване надолу, то всяка операция отнема $O(\log n)$ време в най-лошия случай.

Този подход е само незначително по-добър от използването на червено-черно дърво. Все още може да извършим операцията $MEMBER$ за $O(1)$ време, където при търсенето в червено-черно дърво тя отнема $O(\log n)$ време. Но пък от друга страна, ако броят на съхраняваните елементи е много по-малък от размера u на Вселената, червено-черното дърво ще бъде по-бързо за всички останали операции.

1.3. Налагане на дърво с константна височина

Какво ще се случи, ако наложим дърво от по-голяма степен? Нека приемем, че размера на Вселената е $u = 2^{2k}$ за някое цяло положително число k, така че $\sqrt{u}$ също е цяло положително число. Вместо да наслагваме двоично дърво върху битовия вектор, ние наслагваме дърво със степен $\sqrt{u}$.



Фиг. 2.(a) Дърво от степен $\sqrt{u}$ насложено над същия битов вектор от фиг. 1. Всеки вътрешен връх съхранява *логическото-или* на битовете от всичките си деца. (b) Поглед над същата структура, но

като третираме вътрешните върхове на височина 1 като масив $summary[0..\sqrt{u}-1]$, където $summary[i]$ е *логическото-или* на подмасива $A[i\sqrt{u}\,..\,(i+1)\sqrt{u}-1]$.

Височината на резултатното дърво при тази логика винаги ще е равна на 2.

Както и преди, всеки вътрешен връх съхранява *логическото-или* на битовете намиращи се в поддървото му, така че $\sqrt{u}$ вътрешни върха на дълбочина 1 обобщават всяка група от $\sqrt{u}$ стойности. Както е показано на фиг. 2 (b), може да възприемем тези възли като масив $summary[0..\sqrt{u}-1]$, където $summary[i]$ съдържа 1, тогава и само тогава, когато подмасива му $A[i\sqrt{u}\,..\,(i+1)\sqrt{u}-1]$ съдържа 1. Ще наричаме този подмасив с $\sqrt{u}$ бита i-ти ***клъстер***. За дадена стойност на x, битът $A[x]$ се появява в клъстера под номер $\lfloor x/\sqrt{u} \rfloor$. Сега, $INSERT$ става операция за $O(1)$ време: за да вмъкнем x, актуализираме и $A[x]$ и $summary[\lfloor x/\sqrt{u} \rfloor]$ на 1. Може да използваме $summary$ масива (обобщения масив), за да изпълним всяка една от операциите $MINIMUM$, $MAXIMUM$, $SUCCESSOR$, $PREDECESSOR$ и $DELETE$ за $O(\sqrt{u})$ време:

- За да намерим минималната (максималната) стойност, намираме най-левия (най-десния) запис в масива $summary$, който има 1, да речем $summary[i]$ и правим линейно търсене в i-тия за да намерим най-лявата (най-дясната) 1-ца.
- За да намерим наследника (предшественика) на x, първо търсим надясно (наляво) в неговия клъстер. Ако намерим 1, то тая позицията на тази (първо намерена) единица ни дава резултата. В противен случай, нека $i = \lfloor x/\sqrt{i} \rfloor$ и търсим надясно (наляво) в $summary$ масива от индекс i. Първото срещане на позиция, която съдържа 1 ще даде индекс на клъстер, в който ще търсим най-дясната (най-лявата) 1. Тази позиция съдържа наследника (предшественика).
- За да изтриеме стойност x, нека $i = \lfloor x/\sqrt{u} \rfloor$. Вмъкваме 0 в $A[x]$ и след това актуализираме $summary[i]$ да е равно на логическото-или на битовете от i-тия клъстер.

Във всяка една от горните операции търсим в най-много два клъстера с дължина $\sqrt{u}$ плюс веднъж в $summary$ масива, и следователно всяка операция отнема $O(\sqrt{u})$ време.

На пръв поглед изглежда, че сме постигнали отрицателен напредък. Налагането на двоично дърво ни даде $O(\log n)$ - времеви операции, които са асимптотично по-бързи от $O(\sqrt{u})$. Обаче, използването на дърво от степен $\sqrt{u}$ ще се окаже ключова идея при дърветата на ван Емде Боас.

Наблюдения:

- Използвайки структурите в горния раздел, начина по който намираме наследника или предшественика на стойност x, не зависи от това дали x се намира по това време в множеството или не.
- Нека допуснем, че вместо наслагване на дърво от степен $\sqrt{u}$, насложим дърво от степен $\sqrt[k]{u} = u^{1/k}$, където $k > 1$ е константа. Дървото ще е с височина k, а за всяка от таргетираните операция ще отнема най-много $k\sqrt[k]{u}$ прости операции.

2. Рекурсивна структура

В този раздел ще модифицираме идеята за наслагване на дърво със степен $\sqrt{u}$ над битов вектор с размер u. В предишния раздел използвахме обобщена структура под формата на

$summary$ масив от размер $\sqrt{u}$, в който всяка клетка сочеше към друга структура (масив) с размер $\sqrt{u}$. Сега ще направим тази структура рекурсивна като намаляваме размера на Вселената с коефициент на свиване корен квадратен на всяко ниво на рекурсията. Стартирайки от Вселена с размерност u образуваме структури с размерност $\sqrt{u} = u^{1/2}$, които от своя страна притежават структури с размерност $u^{1/4}$, които притежават структури с $u^{1/8}$ и т.н.

За простота в този раздел ще приемаме, че $u = 2^{2^k}$ за някакво цяло число k, така че $u, u^{1/2}, u^{1/4}, \ldots$ са цели числа. Това ограничение е доста тежко на практика, тъй като позволява само стойности на u в последователността $2, 4, 16, 256, 65536, \ldots$ Ще видим в следващия раздел как да смекчим това предположение и да приемем само, че $u = 2^k$ за някое цяло положително число k. Тъй като структурата, която разглеждаме в този раздел, е само предшественик на истинската дървесна структура на ван Емде Боас, то ще толерираме това ограничение само в полза на нашето разбиране.

Като припомним, че целта ни е да постигнем време за работа от порядъка на $O(\log \log n)$ за таргетираните операции, нека помислим как бихме могли да получим такава времева сложност. Да разгледаме следната рекурентна зависимост:

$$T(u) = T(\sqrt{u}) + O(1) \qquad (\star)$$

Тогава ще имаме:

$$T(n) = T(\sqrt{n}) + 1 = T(n^{1/2}) + 1 \qquad (1)$$

$$T(n^{1/2}) = T(n^{1/4}) + 1 \qquad (2)$$

Заместваме с (2) в (1) и получаваме:

$$T(n) = T(n^{1/4}) + 2$$

$$\ldots$$

$$T(n) = T(n^{(1/2)^k}) + k.$$

Очевидно тази рекурсия трябва да спре някъде. Нека изберем това да е в числото 2. Тогава $n^{(1/2)^k} = 2$. Логаритмуваме и двете страни и получаваме:

$$\log_2 n^{(1/2)^k} = \log_2 2 \Rightarrow \frac{1}{2^k} log_2 n = 1 \Rightarrow \log_2 n = 2^k \Rightarrow k = \log_2 \log_2 n.$$

Тази рекурентна зависимост ще напътства търсенето ни на структура от данни, която да дава желаната сложност за таргетираните операции. Ще проектираме рекурсивна структура от данни, която се свива с коефициент от $\sqrt{u}$ във всяко ниво на своята рекурсия. Когато операция се придвижва в тази структура от данни, на нея ще и е необходимо константно време за преминаване от едно ниво към друго в рекурсията. Следователно рекурентната зависимост, която разгледахме по-горе ще даде желаната сложност.

Поглеждайки назад към структурата на данните на фигура 2. (b), дадена стойност x се намира в клъстер номер $\lfloor x/\sqrt{u} \rfloor$. Ако разгледаме x като $\log_2 u$ - битово бинарно число, този клъстер под номер $\lfloor x/\sqrt{u} \rfloor$ задава най-значимите $\log_2 u/2$ бита на x (тъй като $u = 2^{2^k}$, $\sqrt{u} = 2^{2^{k-1}}$). В неговия клъстер, x се появява на позиция $x \mod \sqrt{u}$, което също така задава $\log_2 u/2$ най-малко значими бита на x. Ще се наложи да индексираме по точи начин и за това нека дефинираме някои функции, които ще ни помогнат да направим това:

$$high(x) = \lfloor x/\sqrt{u} \rfloor,$$
$$low(x) = x \mod \sqrt{u},$$
$$index(x, y) = x\sqrt{u} + y$$

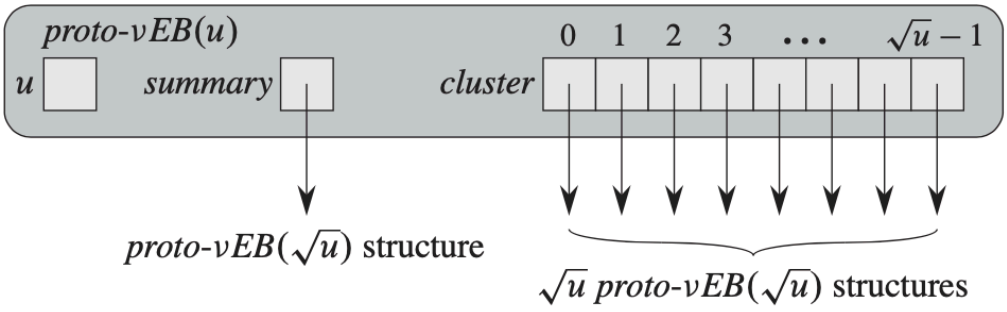
Функцията $high(x)$ дава най-значимите $\log_2 u/2$ бита на x и номера на клъстера на x. Функцията $low(x)$ дава най-малко значимите $\log_2 u/2$ бита на x и позицията на x в своя клъстер. Функцията $index(x, y)$ изгражда номер на елемент от x и y, третирайки x като най-значимите $\log_2 u/2$ бита от номера на елемента и y като най-малко значимите $\log_2 u/2$ бита от номера на елемента. Имаме идентичността $x = index(high(x), low(x))$.
Стойността на u използвана във всяка една от тези функции винаги ще бъде размера на Вселената на структурата на данните, в която извикваме функцията, който се променя докато се спускаме в рекурсивната структура.

2.1. Прототип на дървото на ван Емде Боас

Имайки предвид подсказката от рекурентното уравнение ($\star$), което разгледахме, нека проектираме рекурсивна структура от данни за да подсигурим времевата сложност на операциите. Въпреки, че тази структура от данни ще се провали да постигне целта от $O(\log \log n)$ време за някой операции, тя служи като основа за дървесната структура на ван Емде Боас, която ще видим в предстоящ раздел.

За Вселената $\{0, 1, \ldots, u-1\}$, дефинираме прототип на дървото на ван Емде Боас или proto-vEB структура, която ще означаваме с $proto\text{–}vEB(u)$, рекурсивно както следва. Всяка $proto\text{–}vEB(u)$ структура съдържа атрибут u, който задава размера на Вселената. В допълнение на това, тази структура от данни съдържа още:

- Ако $u = 2$, то тогава имаме размера необходим за базата за рекурсията и съдържа масив $A[0..1]$ от два бита.
- В противен случай, $u = 2^{2^k}$ за някое цяло число $k \geq 1$, така че $u \geq 4$. В допълнение на размера на Вселената u, структурата от данни $proto\text{–}vEB(u)$ съдържа и следните атрибути илюстрирани на фигурата по-долу:



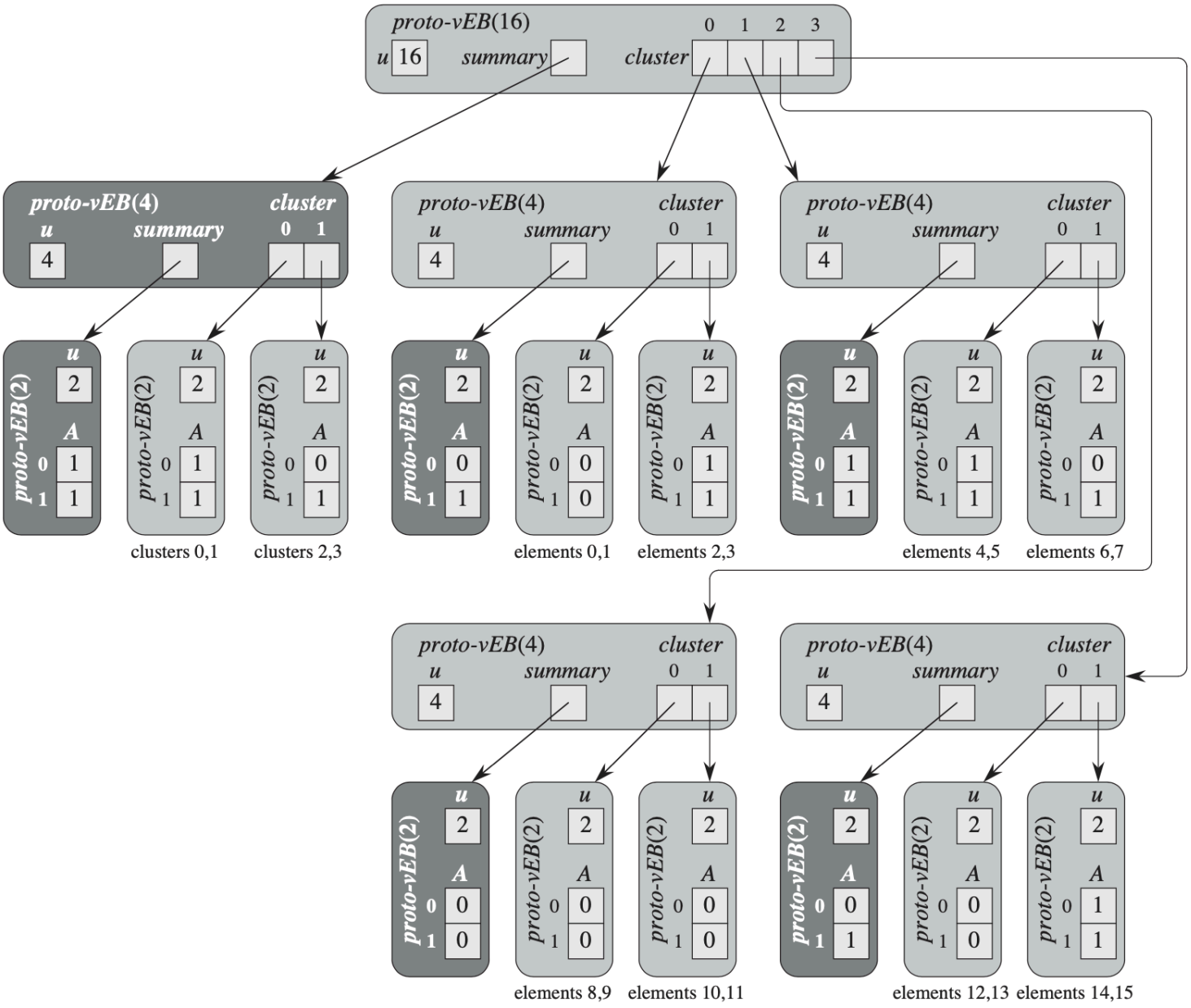
Фиг. 3. Информацията в $proto\text{–}vEB(u)$ структура от данни, когато $u \geq 4$. Структурата съдържа размер на вселената u, указател $summary$ към структура $proto\text{–}vEB(\sqrt{u})$ и масив, който $cluster[0..\sqrt{u}-1]$ от $\sqrt{u}$ поинтъри към $proto\text{–}vEB(\sqrt{u})$ структури.

  - указател с име $summary$ към $proto\text{–}vEB(\sqrt{u})$ структура и
  - масив $cluster[0..\sqrt{u}]-1$ от $\sqrt{u}$ указатели, всеки от които сочи към $proto\text{–}vEB(\sqrt{u})$ структура.

Елементът x, където $0 \leq x < u$ се съхранява рекурсивно в клъстера с номер $high(x)$ като елемент под номер $low(x)$ в този клъстер.

В двустепенната структура от предишния раздел, всеки връх се съхраняваше в $summary$ масив с размер $\sqrt{u}$, в който всяка клетка съдържаше бит. От индекса на всяка клетка може да пресметнем стартовата позиция на подмасива с размер $\sqrt{u}$, който бита обобщава. В структурата $proto\text{–}vEB$ използваме явни указатели вместо индекс

изчисления. Масива $summary$ съдържа обобщените битове, съхранявани рекурсивно в $proto–vEB$ структура, а масивът на клъстера ($cluster$) съдържа $\sqrt{u}$ указатели.



Фиг. 4. $proto–vEB(16)$ структура, представляваща множеството $\{2, 3, 4, 5, 7, 14, 15\}$. Тя сочи към четири $proto–vEB(4)$ структури чрез клъстера $cluster[0..3]$ и до $summary$ структура, която също е $proto–vEB(4)$. Всяка $proto–vEB(4)$ структура сочи до други две $proto–vEB(2)$ структури чрез масива $cluster[0..1]$ и до $proto–vEB(2)$ $summary$. Всяка $proto–vEB(2)$ структура съдържа просто един масив от два бита $A[0..1]$. $proto–vEB(2)$ структурата над „елементи i и j“, съхранява битовете i и j на действителното динамично множество и $proto–vEB(2)$ структурата над „клъстерите i и j“ съхраняват $summary$ битовете на клъстерите i и j в най-горната $proto–vEB(16)$ структура. За по-голяма яснота, тъмно сивите блокчета показват най-горното ниво на $proto–vEB$ структурата, която съхранява обобщена информация за родителската си структура. Такава $proto–vEB$ структура е идентична с всяка друга $proto–vEB$ структура със същия размер на Вселената.

На фигурата по-горе (фиг. 4.) е показана напълно „разгърната“ $proto–vEB(16)$ структура, която представлява множеството $\{2, 3, 4, 5, 7, 14, 15\}$. Ако стойността i е в структурата

$proto–vEB$, посочена от $summary$, тогава i-тия клъстер съдържа някаква стойност в множеството което представлява. Както в дървото с постоянна височина, $cluster[i]$ представлява стойностите от $i\sqrt{u}$ до $(i+1)\sqrt{u}-1$, които формират i-тия клъстер.

На базовото ниво, елементите от дествителното динамично множество се съхраняват в някои от $proto–vEB(2)$ структурите, а останалите $proto–vEB(2)$ структури съхраняват обобщени битове. Под всяка от базовите структури, които не са обобщаващи (не са $summary$), цифрата показва кои битове съхранява. Например $proto–vEB(2)$ структурата номерирана с „$elements\ 6, 7$“ съхранява бит 6 (0, тъй като елемент 6 не се съдържа в множеството) в клетка $A[0]$ и бит 7 (1, тъй като елемент 7 е в множеството) в клетка $A[1]$.

Подобно на клъстерите, всеки $summary$ масив е просто динамично множество с размер на Вселената $\sqrt{u}$ и за това представяме всеки $summary$ като $proto–vEB(\sqrt{u})$ структура. Четирите $summary$ бита за главната $proto–vEB(16)$ структура са в най-лявата $proto–vEB(4)$ структура и в крайна сметка се появяват в две $proto–vEB(2)$ структури. Например, $proto–vEB(2)$ структурата обозначена с „$elements\ 2, 3$“ има $A[0]=0$, което показва, че клъстер 2 от $proto–vEB(16)$ структурата (съдържащ елементи 8, 9, 10, 11) е изцяло празен, а $A[1]=1$, което показва, че клъстер 3 (съдържащ елементи 12, 13, 14, 15) има поне един елемент. Всяка $proto–vEB(4)$ структура сочи към свой собствен масив $summary$, който е съхранен като $proto-vEB(2)$ структура. Например, да разгледаме $proto–vEB(2)$ която се намира от ляво на $proto–vEB(2)$ структурата, която е номерирана с „$elements\ 0, 1$“.Тъй като $A[0]=0$, то това ни показва, че „$elements\ 0, 1$“ структурата е изцяло от битове 0, а тъй като $A[1]=1$, то това ни показва, че „$elements\ 2, 3$“ структурата съдържа поне един бит 1.

2.2. Операции върху $proto-vEB$ (прототип на vEB tree) структура от данни

Ще опишем как да изпълняваме операции върху $proto–vEB$ структури от данни. Първо ще разгледаме операциите за заявки - $MEMBER$, $MINIMUM$, $MAXIMUM$ и $SUCCESSOR$, които не променят $proto–vEB$ структурата. След това ще разгледаме операциите $INSERT$, $DELETE$. Ще оставим $MAXIMUM$ и $PREDECESSOR$, които са симетрични на $MINIMUM$ и $SUCCESSOR$, съответно.

Всяка една от операциите $MEMBER$, $SUCCESSOR$, $PREDECESSOR$, $INSERT$ и $DELETE$ взима параметър x, заедно с $proto–vEB$ структурата V. За всяка от тези операции без ограничение на общността ще считаме, че $0 \le x \le V.u$.

2.2.1. Определяне на това дали даден елемент е в множествсото

За да изпълним $MEMBER(x)$, трябва да намерим бита, съответстващ на x в съответната $proto–vEB(2)$ структура. Може да го направим за $O(\log\log n)$ време като преминаваме през $summary$ структурите. Следващата процедура взема $proto–vEB$ структурата V и стойност x и връща бит, който показва дали x е в динамичното множество представлявано от V или не.

$proto–vEB–MEMBER(V, x)$
1. **if** $V.u == 2$
2. **return** $V.A[x]$
3. **else** $return\ proto–vEB–MEMBER(V.cluster[high(x)], low(x))$

Процедурата $proto–vEB–MEMBER$ работи по следния начин. Първия ред проверява дали сме в базовия случай, където V е $proto–vEB(2)$ структура. Втория ред обработва базовия случай, като просто връща съответния бит в масива A. Третия ред се занимава с рекурсивния случай „пробивайки“ в подходящата по-малка структура $proto–vEB$. Стойността $high(x)$ казва коя $proto–vEB(\sqrt{u})$ структура да посетим, а $low(x)$ определя за кой елемент в тази $proto–vEB(\sqrt{u})$ структура е заявката.

Нека да видим какво се случва, когато извикаме $proto–vEB–MEMBER(V, 6)$ на $proto–vEB(16)$ структурата (на фиг. 3). Тъй като $high(6) = 1$, когато $u = 16$, рекурсивно влизаме в $proto–vEB(4)$ структурата отгоре в дясно и търсим елемента $low(6) = 2$ в тази структура. В това рекурсивно извикване $u = 4$ и следователно отново влизаме навътре в рекурсията. С $u = 4$ имаме $high(2) = 1$ и $low(2) = 0$ и следователно търсим елемент елемента под номер 0 в $proto–vEB(2)$ структурата отгоре в дясно. Това рекурсивно извикване се оказва базово за рекурсията и следователно връща $A[0] = 0$ нагоре по веригата от рекурсивни извиквания. По този начин получаваме резултата 0, който $proto–vEB–MEMBER(6)$ функцията връща и с него показва, че 6 не е в множеството.

За да определим времето за работа на $proto–vEB–MEMBER$, нека с $T(u)$ означим времето за работа върху $proto–vEB(u)$ структура. Всяко рекурсивно извикване отнема константно време, без да се включва времето, необходимо за рекурсивните повиквания. Когато $proto–vEB–MEMBER$ прави рекурсивно повикване, тя се обръща към (обажда на) $proto–vEB(\sqrt{u})$ структура. По този начин може да характеризираме времето за изпълнение чрез рекурантната зависимост $T(u) = T(\sqrt{u}) + O(1)$, която вече разглеждахме в ($\star$). Нейното решение е $T(u) = O(\log \log n)$ и от тук стигаме до извода, че $proto–vEB–MEMBER$ функцията работи за време $O(\log \log n)$.

2.2.2. Намиране на минимален елемент

Сега разглеждаме как да извършим операцията $MINIMUM$. Процедурата $proto–vEB–MINIMUM(V)$ връща минималния елемент в $proto–vEB$ структурата V или NIL, ако V представлява празно множество ($V = \emptyset$).

```
proto–vEB–MINIMUM(V)
1.  if V.u == 2
2.      if V.A[0] == 1
3.          return 0
4.      elseif V.A[1] == 1
5.          return 1
6.      else return NIL
7.  else min–cluster = proto–vEB–MINIMUM(V.summary)
8.      if min–cluster == NIL
9.          return NIL
10.     else offset = proto–vEB–MINIMUM(V.cluster[min–cluster])
11.         return index(min–cluster, offset)
```

Тази процедура работи по следния начин. Първия ред проверява дали сме в базовия случай, който редовете от $2 - 6$ обработват (brute force). Редовете $7 - 11$ обработват рекурсивния случай. Първо ред номер 7 намира номера на първия клъстер, който съдържа елемент от множеството. Това става чрез рекурсивно извикване на

$proto–vEB–MINIMUM$ на $V.summary$, което е $proto–vEB(\sqrt{u})$ структура. Ред 7 присвоява този номер на клъстера на променливата $min–cluster$. Ако множеството е празно, тогава рекурсивното извикване връща NIL и ред номер 9 връща NIL. В противен случай минималният елемент на множеството е някъде в клъстера с номер $min–cluster$. Рекурсивното извикване в ред 10 намира отместването в рамките на клъстера на минималния елемент в този клъстер. И накрая, ред номер 11 конструира стойността на минималния елемент от номера на клъстера и отместването и връща тази стойност.

Въпреки, че заявката върху $summary$ информацията ни позволява бързо да намерим клъстера, съдържащ минималния елемент, тъй като тази процедура прави две рекурсивни извиквания на $proto–vEB(\sqrt{u})$ структурата, тя не работи за $O(\log\log n)$ време в най-лошия случай. Нека с $T(u)$ означим най-лошия случай за $proto–vEB–MINIMUM$ на $proto–vEB(u)$ структурата. Имаме рекурентната зависимост:
$T(u) = 2T(\sqrt{u}) + O(1).$ ⊛
Ще използваме промяна на променливите, за да решим това рекурентно уравнение.
Полагаме $m = \log_2 n$, което води до
$T(2^m) = 2T(2^{m/2}) + O(1).$
Преименуване на $S(m) = T(2^m)$ ни дава
$S(m) = 2S(m/2) + O(1)$, което от случай 1 от основния метод (master method), има решение $S(m) = \Theta(m)$. Сменяйки обратно от $S(m)$ на $T(u)$, получаваме
$T(u) = T(2^m) = S(m) = \Theta(m) = \Theta(\log u)$. По този начин виждаме, че поради второто рекурентно извикване $proto–vEB–MINIMUM$ работи за $\log n$ време, а не за желаното $O(\log\log u)$ време.

За решаване на ⊛ може да подходим и по следния начин:
$T(u) = 2T(\sqrt{u}) + O(1)$
Тогава ще имаме:
$T(n) = 2T(\sqrt{n}) + 1 = 2T(n^{\frac{1}{2}}) + 1 \qquad (1)$
$T(n^{1/2}) = 2T(n^{\frac{1}{4}}) + 1 \qquad (2)$
Заместваме с (2) в (1) и получаваме:
$T(n) = 4T(n^{1/4}) + 2$
$\dots$
$T(n) = 2^k T(n^{(1/2)^k}) + k.$
Очевидно тази рекурсия трябва да спре някъде. Нека изберем това да е в числото 2.
Тогава $n^{(1/2)^k} = 2$. Логаритмуваме и двете страни и получаваме:
$\log_2 n^{(1/2)^k} = \log_2 2 \Rightarrow \frac{1}{2^k} log_2 n = 1 \Rightarrow \log_2 n = 2^k \Rightarrow k = \log_2\log_2 n.$
Сега, $T(n) = 2^{k+1} + k = 2^{\log_2\log_2 n+1} + \log_2\ \log_2 n = 2.2^{\log_2\log_2 n} + \log_2\ \log_2 n =$
$= 2.\log_2 n + log_2\log_2 n = O(\log n).$

2.2.3. Намиране на наследник

Операцията за намиране на наследник ($SUCCESSOR$) е дори по-лоша. В най-лошия случай, тя прави две рекурсивни извиквания, заедно с извикване към $proto–vEB–MINIMUM$. Процедурата $proto–vEB–SUCCESSOR(V, x)$ връща най-малкия елемент в $proto–vEB$ структурата V, който е по-голям от x или NIL, ако нито един елемент във V не е по-голям от x. Не изисква x да бъде член на множеството, но предполага, че $0 \leq x < V.u$.

```
proto–vEB–SUCCESSOR(V, x)
1.  if V.u == 2
2.      if x == 0 and V.A[1] == 1
3.          return 1
4.      else return NIL
5.  else offset = proto–vEB–SUCCESSOR(V.cluster[high(x)], low(x))
6.      if offset ≠ NIL
7.          return index(high(x), offset)
8.      else succ–cluster = proto–vEB–SUCCESSOR(V.summary, high(x))
9.          if succ–cluster == NIL
10.             return NIL
11.         else offset = proto–vEB–MINIMUM(V.cluster[succ–cluster])
12.             return index(succ–cluster, offset)
```

Процедурата $proto–vEB–SUCCESSOR$ работи по следния начин. Както обикновено първия ред проверява дали сме стигнали до дъното на рекурсията (базовия случай), който редовете $2-4$ обработват: единственият начин x да има наследник в $proto–vEB(2)$ структура е когато $x = 0$ и $A[1]$ е 1. Редовете $5-12$ обработват рекурсивния случай. Ред 5 търси наследник на x в клъстера, в който x се намира; ако има, тогава ред 7 изчислява и връща стойността на този наследник. В противен случай трябва да търсим в други клъстери. Ред 8 присвоява на $succ–cluster$ номера на следващия непразен клъстер, като използва $summary$ информацията, за да го намери. Ред 9 проверява дали $succ–cluster$ е NIL, а ред 10 връща NIL, ако всички следващи клъстери са празни. Ако $succ–cluster$-ът не е NIL, ред 11 присвоява първия елемент от този клъстер на променливата $offset$ и ред 12 изчислява и връща минималния елемент в този клъстер.

В най-лошия случай, $proto–vEB–SUCCESSOR$ се извиква рекурсивно два пъти на $proto–vEB(\sqrt{u})$ структури и прави допълнително извикване на $proto–vEB–MINIMUM$ на $proto–vEB(\sqrt{u})$ структура. По този начин, рекурентната зависимост за най-лошия случай $T(u)$ на $proto–vEB–SUCCESSOR$ е $T(u) = 2T(\sqrt{u}) + \Theta(\log\sqrt{u}) = 2T(\sqrt{u}) + \Theta(\log u)$. Може да използваме същата техника, която използвахме за рекурентните зависимости в ($\star$) и $\circledast$, за да покажем, че това уравнение има решението $T(u) = \Theta(\log u \log\log u)$. Следователно $proto–vEB–SUCCESSOR$ е асимптотично по-бавна от $proto–vEB–MINIMUM$.

$$T(u) = 2T(\sqrt{u}) + O(\log u) \qquad (\clubsuit)$$

От рекурентната зависимост следва, че:

$$T(n) = 2T(n^{\frac{1}{2}}) + \log_2 n \quad (1)$$

$$T(n^{\frac{1}{2}}) = 2T(n^{\frac{1}{4}}) + \log_2 n^{\frac{1}{2}} \quad (2)$$

Заместваме с (2) в (1) и получаваме:

$T(n) = 2T(n^{\frac{1}{2}}) + \log_2 n = 2\left(2T(n^{\frac{1}{4}}) + \log_2 n^{\frac{1}{2}}\right) + \log_2 n = 2^2 T(n^{\frac{1}{2^2}}) + 2\log_2 n.$

$\ldots$

$T(n) = 2^k T(n^{\frac{1}{2^k}}) + k\log_2 n.$

Очевидно тази рекурсия трябва да спре някъде. Нека изберем това да е в числото 2. Тогава $n^{(1/2)^k} = 2$. Логаритмуваме и двете страни и получаваме:

$\log_2 n^{(1/2)^k} = \log_2 2 \Rightarrow \frac{1}{2^k} log_2 n = 1 \Rightarrow \log_2 n = 2^k \Rightarrow k = \log_2 \log_2 n.$

Следователно $T(n) = 2^{\log_2 \log_2 n}.2 + \log_2 n \log_2 \log_2 n = 2\log_2 n + \log_2 n \log_2 \log_2 n =$
$= O(\log_2 n \log_2 \log_2 n).$

2.2.4. Вмъкване на елемент

За да вмъкнетм елемент, трябва да го направим в съответния клъстер и също така да зададем $summary$ бит за този клъстер на 1. Процедурата $proto\text{–}vEB\text{–}INSERT(V, x)$ вмъква стойността x в $proto\text{–}vEB$ структурата V.

$proto\text{–}vEB\text{–}INSERT(V, x)$
1. **if** $V.u == 2$
2. $\quad V.A[x] = 1$
3. **else** $proto\text{–}vEB\text{–}INSERT(V.cluster[high(x)], low(x))$
4. $\quad proto\text{–}vEB\text{–}INSERT(V.summary, high(x))$

В базовия случай, ред 2 задава подходящия бит в масива A на 1. В рекурсивния случай, рекурсивното извикване на ред 3 вмъква x в съответния клъстер, а ред 4 задава $summary$ бит-а на този клъстер на 1.

Тъй като $proto\text{–}vEB\text{–}INSERT$ прави две рекурсивни извиквания в най-лошия случай, рекурентната зависимост, която характеризира времевата сложност за изпълнение на функцията, в най-лошия случай отнема $\Theta(\log u)$ време.

2.2.5. Изтриване на елемент

Операцията $DELETE$ е по-сложна от вмъкването. Докато винаги може да зададем $summary$ бит 1 при вмъкване, не винаги може да нулираме същия $summary$ бит на 0 при изтриване. Трябва да определим дали има друг бит в съответния клъстер, който да е 1. Така както дефинирахме $proto\text{–}vEB$ структурите, ще трябва да изследваме всички битове в клъстера, за да определим дали някой от тях не е 1. Алтернативно бихме могли да добавим атрибут n към $proto\text{–}vEB$ структурата, който показва колко елемента има в нея (брояч на елементите). Ще оставим имплементацията на $proto\text{–}vEB\text{–}DELETE$ процедура за последващите раздели.

Очевидно, трябва да модифицираме структурата $proto\text{–}vEB$, за да сведем всяка операция до извършване на най-много едно рекурсивно повикване. Ще видим в следващия раздел как да го направим.

3. Дърво на ван Емде Боас

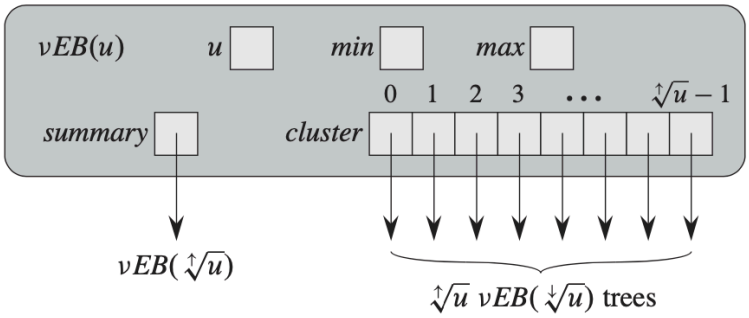
Структурата $proto\text{–}vEB$, която разглеждахме като прототип на дърво на ван Емде Боас в предишния раздел е близка до това, което ни е необходимо, за да постигнем $O(\log \log u)$ време на работа. Проваляме се малко преди да достигнем желаното време, защото трябва да извикаме рекурсията повече пъти в повечето от операциите. В този раздел ще проектираме структура на данни, която е подобна на $proto\text{–}vEB$ структурата, но съхранява малко повече информация, като по този начин премахва необходимостта от част от рекурсията.

В предишния раздел забелязахме, че предположението, което направихме за размера на Вселената - $u = 2^{2^k}$ за някое цяло положително k е твърде ограничаващо и свиващо възможните стойности на u до твърде малко. От този момент нататък, ще позволяваме на размера на Вселената u да има стойности само точните степени на двойката и когато $\sqrt{u}$ не е цяло число, т.е. когато u е нечетна степен на двойката ($u = 2^{2k+1}$, за някое цяло неотрицателно $k \geq 0$) - тогава ще разделим $\log_2 u$ битовете на числото на най-значимите $\lceil (\log_2 u)/2 \rceil$ бита и най-малко значимите $\lfloor (\log_2 u)/2 \rfloor$ бита. За удобство обочначаваме $2^{\lceil (\log_2 u)/2 \rceil}$ („горния корен квадратен“ на u) с $\sqrt[\uparrow]{u}$ и $2^{\lfloor (\log_2 u)/2 \rfloor}$ („долния корен квадратен“ на u) с $\sqrt[\downarrow]{u}$, така, че $u = \sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u}$, а когато u е четна степен на двойката ($u = 2^{2k}$, за някое цяло k), $\sqrt[\uparrow]{u} = \sqrt[\downarrow]{u} = \sqrt{u}$. Тъй като сега позволяваме на u да взима стойности равни на нечетни степени на двойката, то трябва да предефинираме помощните функции от предишната секция:

$$high(x) = \lfloor x / \sqrt[\downarrow]{u} \rfloor,$$
$$low(x) = x \mod \sqrt[\downarrow]{u},$$
$$index(x, y) = x \sqrt[\downarrow]{u} + y$$

3.1. Дървета на ван Емде Боас

Дърветата на **ван Емде Боас** или **vEB tree**, модифицира прототипа на структурата на ван Емде Боас $proto\text{–}vEB$, която разгледахме по-горе. Обозначаваме vEB дърво с размер на Вселената u с $vEB(u)$ и, освен ако u е равно на базовия размер 2, атрибута $summary$ сочи към $vEB(\sqrt[\uparrow]{u})$ дърво и масива $cluster[0..\sqrt[\uparrow]{u} - 1]$ сочи до $\sqrt[\uparrow]{u}$ на брой $vEB(\sqrt[\downarrow]{u})$ дървета.



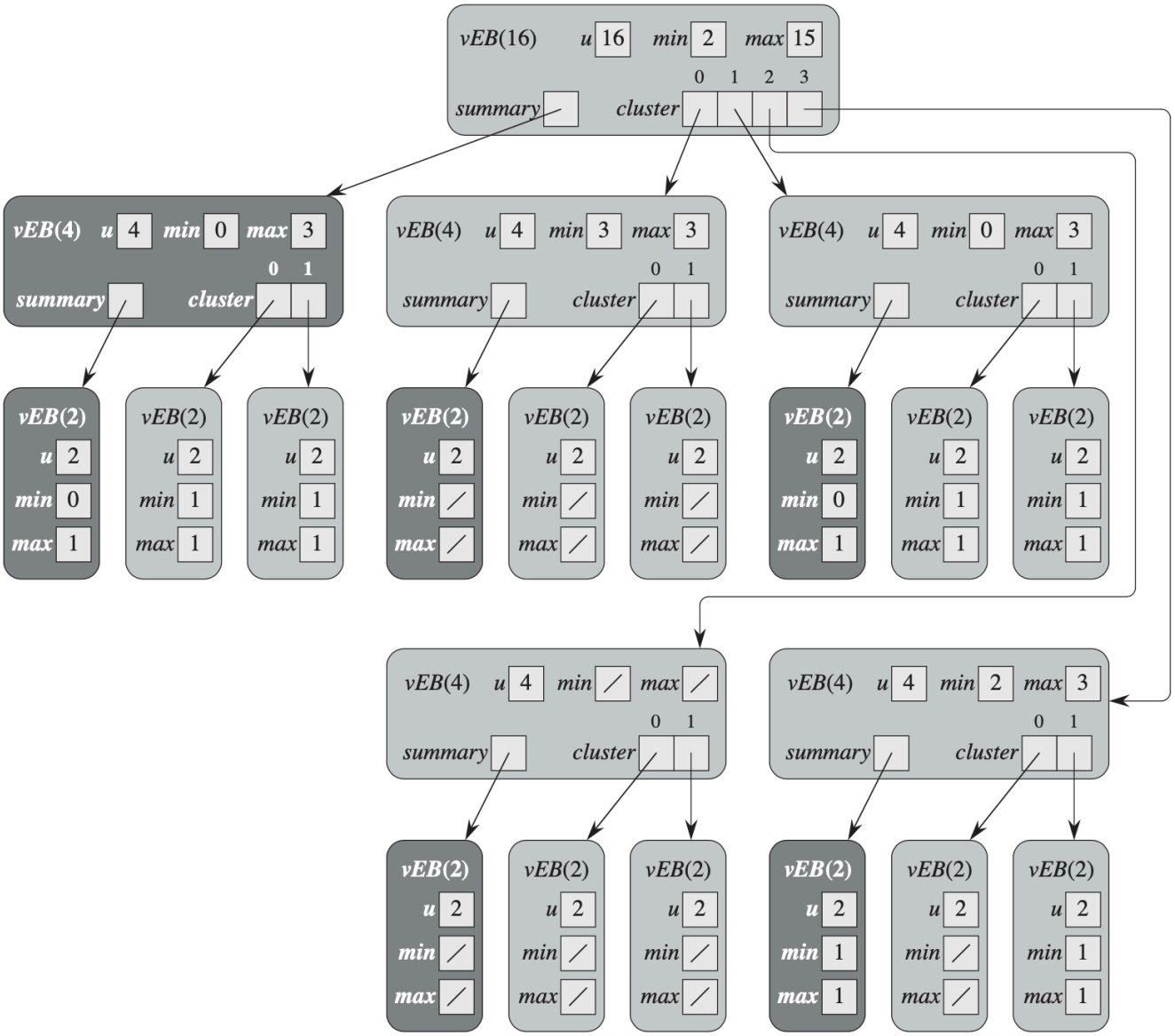
Фиг. 5. Информацията, която $vEB(u)$ дърво съхранява, когато $u > 2$. Структурата съдържа размера на Вселената u, елементите $\min$ и $\max$, пойнтър $summary$ до $vEB(\sqrt[\uparrow]{u})$ дърво и масив $cluster[0..\sqrt[\uparrow]{u} - 1]$ от $\sqrt[\uparrow]{u}$ на брой указатели към $vEV(\sqrt[\downarrow]{u})$ дървета.

Както фигура 5 илюстрира, vEB дървото съдържа два атрибута, който в $proto\text{–}vEB$ структурата не фигурират:

- $\min$ съхранява минималния елемент във vEB дървото, а
- $\max$ съхранява максималния елемент във vEB дървото.

Освен това, елементът съхранен в $\min$, не се появява в нито една от рекурсивните $vEB(\sqrt[\downarrow]{u})$ дървета, към които масива от указатели $cluster$ сочи. Следователно, елементите съхранени във $vEB(u)$ дървото V са $V.\min$ плюс всички елементи рекурсивно съхранени във $vEB(\sqrt[\downarrow]{u})$ дърветата соени от масива от указатели $V.cluster[0...\sqrt[\uparrow]{u}-1]$. Отбелязваме, че когато vEB дърво съдържа два или повече елемента, ние третираме $\min$ и $\max$ различно: елемента съхранен в $\min$ не фигурира в нито един от клъстерите, но елемента съхранен в $\max$ се появява в някой от клъстерите.

Тъй като базовият размер е 2, $vEB(2)$ дървото не се нуждае от масив A, който съответната $proto\text{–}vEB(2)$ структура имаше. Вместо това може да определим неговите елементи само чрез атрибутите $\min$ и $\max$. Във vEB дърво, в което няма елементи, независимо от размера на неговата Вселена u, и $\min$, и $\max$ са равни на NIL.



Фиг. 6. $vEB(16)$ дърво съответстващо на $proto\text{–}vEB$ дървото от фиг. 4. То съхранява множеството от числа $\{2,3,4,5,7,14,15\}$. Наклонените черти отбелязват NIL стойностите. Стойността съхранена в атрибута $\min$ на vEB дървото не се появява в нито един от клъстерите му. Силното потъмняване на сивия цвят има същата цел както на фиг. 4.

Фигура 6 по-горе показва $vEB(16)$ дърво V, което съдържа множеството от числа $\{2,3,4,5,7,14,15\}$. Тъй като най-малкия елемент от това множество е 2, то $V.\min = 2$ и въпреки, че $high(2) = 0$, елемента 2 не се появява във $vEB(4)$ дървото сочено от указателя $V.cluster[0]$: забележете, че $V.cluster[0].\min$ е равен на 3 и следователно 2 не е в това vEB дърво. По подобен начин, тъй като $V.cluster[0].\min$ е равно на 3, и 2 и 3 са единствените елементи във $V.cluster[0]$, то $vEB(2)$ клъстерите във $V.cluster[0]$ са празни.

Атрибутите $\min$ и $\max$ ще се окажат ключови за намаляването на броя на рекурсивните извиквания в рамките на операциите върху vEB дърветата. Тези атрибути ще ни помогнат по четири начина:

1. $MINIMUM$ и $MAXIMUM$ операциите, дори не трябва да извикват каквато и да е рекурсия, тъй като могат просто да върнат стойностите на $\min$ и $\max$ член данните.

2. Операцията $SUCCESSOR$ може да избегне извършването на рекурсивно извикване, за да определи дали наследникът на стойност x се намира в $high(x)$. Това е така, защото наследникът на x се намира в неговия клъстер, тогава и само тогава, когато x е строго по-малък от атрибута $\max$ на неговия клъстер. Симетричния аргумент е в сила и за операцията $PREDECESSOR$ заедно с атрибута $\min$.

3. Може да разберем дали vEB дървото няма елементи, има точно един елемент или има поне два елемента за константно време, чрез стойностите на атрибутите му $\min$ и $\max$. Тази способност ще помогне при операциите $INSERT$ и $DELETE$. Ако $\min$ и $\max$ са едновременно NIL, то тогава vEB дървото няма елементи. Ако $\min$ и $\max$ не са NIL, но са равни помежду си, то тогава vEB дървото има точнчо един елемент. В противен случай, и $\min$ и $\max$ не са NIL, но не са и равни помежду си и следователно vEB дървото има два или повече елемента.

4. Ако знаем, че vEB дървото е празно, може да вмъкнем елемент в него, като актуализираме само неговите атрибути $\min$ и $\max$. Следователно може да вмъкнем елемент в празно дърво на vEB за константно време. По същия начин, ако знаем, че vEB дървото има само един елемент, може да го изтрием за константно време, като актуализираме само $\min$ и $\max$. Тези свойства ще ни позволят да прекъснем по-рано веригата от рекурсивни извиквания.

Дори ако размерът на Вселената u е нечетна степен на 2, разликата в размерите на обобщеното vEB дърво и клъстерите няма да окажат влияние върху асимптотичните времена на работа на операциите на vEB дървото. Рекурсивните процедури, които изпълняват операциите на vEB дървото, ще имат времена за работа, характерризиращи се със следната рекурентна зависимост: $T(u) \leq T(\sqrt[\uparrow]{u}) + O(1)$. ◆

Тази рекурентна зависимост изглежда близка до зависимостта, която разглеждахме в ($\star$) и ще я решим по следния начин:
Нека $m = \lg u$, презаписваме зависимостта по следния начин:
$T(2^m) \leq T(2^{\lceil m/2 \rceil}) + O(1)$.
Забелязвайки, че $\lceil m/2 \rceil \leq 2m/3$ за всяко $m \geq 2$, получаваме, че
$T(2^m) \leq T(2^{2m/3}) + O(1)$.

Полагаме $S(m) = T(2^m)$ и презаписваме последната рекурентна зависимост по следния начин:
$S(m) \leq S(2m/3) + O(1)$,
което от случай 2 на $master\ method$-а, има решение $S(m) = \lg m$ (По отношение на асимптотичното решение, дробта $2/3$ не оказва никаква разлика в сравнение с дробта $1/2$, тъй като прилагайки $master\ method$-а стигаме до $\log_{3/2} 1 = \log_2 1 = 0$.) Следователно $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$.

Преди да използваме дървото на ван Емде Боас, трябва да знаем размера на Вселената u, за да може да създадем дърво на ван Емде Боас с подходящ размер, което първоначално представлява празно множество. Това празно дърво ще се създаде за линейно време по размера на Вселената u. Следователно, ако знаем, че ще имаме малко на брой операции, ще е по-удачно да създадем червено-черно дърво, тъй като такава празна структура ще се създаде за константно време. Това е така, защото реално може времето за създаване на структурата от данни да надвиши времето, необходимо за изпълнението на отделните операции. Този недостатък обикновено не е съществен, тъй като обикновено използваме проста структура от данни като масив или свързан списък за да представим множество само с няколко елемента.

3.2. Операции върху дърво на ван Емде Боас

Вече сме готови да видим как да извършваме операции върху дърво на ван Емде Боас. Както направихме за структурата на прототипа на дърво на ван Емде Боас ($proto\text{–}vEB$), първо ще разгледаме операциите за заявки и след това операциите $INSERT$ и $DELETE$. Поради малката асиметрия между минималния и максималния елемент във vEB дървото - когато vEB дървото съдържа поне два елемента, минималния елемент не се появява в клъстера, но максималният елемент го прави - ние ще представим псевдокод за всички пет заявки. Както при операциите върху прототипа на дърво на ван Емде Боас, и тук операциите вземат като параметри V и x, където V е дърво на ван Емде Боас, а x е елемент, като допускаме, че е изпълнено неравенството $0 \leq x < V.u$.

3.2.1. Намиране на минималния и максималния елемент

Тъй като съхраняваме минимума и максимума в атрибутите $\min$ и $\max$, две от операциите са едноредови, отнемащи константно време:

$vEB\text{–}TREE\text{–}MINIMUM(V)$
1. **return** $V.\min$

$vEB\text{–}TREE\text{–}MAXIMUM(V)$
1. **return** $V.\max$

3.2.2. Определяне дали дадена стойност е в множеството

Процедурата $vEB\text{–}TREE\text{–}MEMBER(V, x)$ има рекурсивен случай като тази в $proto\text{–}vEB\text{–}MEMBER$, но дъното на рекурсията е малко по-различно. Също така проверяваме директно дали x е равен на минималния или максималния елемент. Тъй като vEB дървото не съхранява битове, както прави $proto\text{–}vEB$ структурата, ние проектираме $vEB\text{–}TREE\text{–}MEMBER$ да връща TRUE или FALSE, вместо 0 или 1.

$vEB\text{–}TREE\text{–}MEMBER(V, x)$
1. **if** $x == V.\min\ \ or\ x == V.\max$

```
2.      return TRUE
3.  elseif V.u == 2
4.      return FALSE
5.  else return vEB–TREE–MEMBER(V.cluster[high(x)], low(x))
```

Ред 1 проверява дали x е равен или на минималния, или на максималния елемент. Ако го направи, ред 2 връща TRUE. В противен случай, ред 3 тества за базовия случай. Тъй като дървото $vEB(2)$ няма елементи, различни от тези в min и max, ако това е базовият случай, ред 4 връща FALSE. Другата възможност - това не е базов случай и x не е равно нито на min, нито на max - се обработва от рекурсивното извикване в ред 5.

Рекурентната зависимост в ♦ характеризира времето за работа на процедурата $vEB–TREE–MEMBER$ и следователно тази процедура се изпълнява за време $O(\lg \lg u)$.

3.2.3. Намиране на предшественик и наследник

Сега да видим как да релаизираме операцията $SUCCESSOR$. Спомнете си, че процедурата $proto–vEB–SUCCESSOR(V, x)$ имаше възможност да направи две рекурсивни извиквания: едното, зада се определи дали наследникът на x се намира в същия клъстер като на x и ако не се окаже, че не се намира - още едно рекурсивно извикване, за да намери клъстера, съдържащ наследника на x. Тъй като може да получим бърз достъп до максималната стойност във vEB дървото, то може да избегнем извършването на две рекурсивни извиквания и вместо това да извършим едно рекурсивно извикване или на $cluster$ или на $summary$, но не и на двете.

```
vEB–TREE–SUCCESSOR(V, x)
1.  if V.u == 2
2.      if x == 0 and V.max == 1
3.          return 1
4.      else return NIL
5.  elseif V.min ≠ NIL and x < V.min
6.      return V.min
7.  else max–low = vEB–TREE–MAXIMUM(V.cluster[high(x)])
8.      if max–low ≠ NIL and low(x) < max–low
9.          offset = vEB–TREE–SUCCESSOR(V.cluster[high(x)], low(x))
10.         return index(high(x), offset)
11.     else succ–cluster = vEB–TREE–SUCCESSOR(V.summary, high(x))
12.         if succ–clustrr = NIL
13.             return NIL
14.         else offset = vEB–TREE–MINIMUM(V.cluster[succ–cluster])
15.             return index(succ–cluster, offset)
```

Тази процедура има шест **return** изявления и няколко случая. Започваме с базовия случай на линии 2-4, който връща 1 на ред 3, ако се опитваме да намерим наследника на 0 и 1 е в множеството от два елемента; в противен случай, базовия случай връща NIL в ред 4.

Ако не сме в базовия случай, следващата проверка на ред 5 е дали x е строго по-малък от минималния елемент. Ако това е така, тогава просто връщаме минималния елемент в ред 6.

Ако стигнем до ред 7, то тогава знаем, че не сме в базовия случай и че x е по-голямо или равно на минималната стойност на vEB дървото V. Ред 7 присвоява на $max\text{–}low$ максималния елемент в клъстера на x. Ако клъстерът на x съдържа някакъв елемент, който е по-голям от x, тогава знаем, че наследникът на x се намира някъде в клъстера на x. Ред 8 тества за това условие. Ако наследникът на x е в клъстера на x, тогава ред 9 определя къде в клъстера се намира, а ред 10 връща наследника по същия начин както ред 7 на $proto\text{–}vEB\text{–}SUCCESSOR$.

Стигаме до ред 11, ако x е по-голямо или равно на най-големия елемент в неговия клъстер. В този случай редовете 11-15 намират наследника на x по същия начин, както редовете 8-12 на $proto\text{–}vEB\text{–}SUCCESSOR$.

Лесно е да се види как рекурентната зависимост ◆ характеризира времето за работа на $vEB\text{–}TREE\text{–}SUCCESSOR$. В зависимост от резултата от теста на ред 7, процедурата се извиква рекурсивно или на ред 9 (върху vEB дърво с размер на вселената $\sqrt[\downarrow]{u}$) или на ред 11 (върху vEB дърво с размер на вселената $\sqrt[\uparrow]{u}$). И в двата случая имаме едно рекурсивно извикване върху vEB дърво с размер на вселената, който е най-много $\sqrt[\uparrow]{u}$. Остатъка от процедурата, включително извикванията към $vEB\text{–}TREE\text{–}MINIMUM$ и $vEB\text{–}TREE\text{–}MAXIMUM$, отнемат $O(1)$ време. Следователно, $vEB\text{–}TREE\text{–}SUCCESSOR$ се изпълнява за време $O(\lg \lg u)$ в най-лошия случай.

Процедурата $vEB\text{–}TREE\text{–}PREDECESSOR$ е симетрична на процедурата $vEB\text{–}TREE\text{–}SUCCESSOR$, но с един допълнителен случай:

```
vEB–TREE–PREDECESSOR(V, x)
1.  if V.u == 2
2.      if x == 1 and V.min == 0
3.          return 0
4.      else return NIL
5.  esleif V.max ≠ NIL and x > V.max
6.      return V.max
7.  else min–low = vEB–TREE–MINIMUM(V.cluster[high(x)])
8.      if min–low ≠ NIL and low(x) > min–low
9.          offset = vEB–TREE–PREDECESSOR(V.cluster[high(x)], low(x))
10.         return index(high(x), offset)
11.     else pred–cluster = vEB–TREE–PREDECESSOR(V.summary, high(x))
12.         if pred–cluster == NIL
13.             if V.min ≠ NIL and x > V.min
14.                 return V.min
15.             else return NIL
16.         else offset = vEB–TREE–MAXIMUM(V.cluster[pred–cluster])
17.             return index(pred–cluster, offset)
```

Редовете 13-14 образуват допълнителния случай. Този случай възниква, когато предшественикът на x, ако съществува, не се намира в клъстера на x. Във $vEB\text{–}TREE\text{–}SUCCESSOR$ бяхме уверени, че ако наследникът на x се намира извън клъстера на x, тогава той трябва да се намира в клъстер с по-голям номер. Но ако предшественикът на x е минималната стойност на vEB дървото V, то наследникът изобщо

не се намира в клъстер. Ред 13 проверява за това условие, а ред 14 връща минималната стойност според случая.

Този допълнителен случай не засяга асимптотично времето за работа на $vEB\text{–}TREE\text{–}PREDECESSOR$ в сравнение с $vEB\text{–}TREE\text{–}SUCCESSOR$, така че $vEB\text{–}TREE\text{–}PREDECESSOR$ работи за $O(\lg \lg n)$ време в лошия случай.

3.2.4. Добавяне на елемент

Сега разглеждаме как да вмъкнем елемент във vEB дърво. Спомнете си, че $proto\text{–}vEB\text{–}INSERT$ правеше две рекурсивни извиквания: едно за вмъкване на елемента и едно за вмъкване на номера на клъстера на елемента в $summary$. Процедурата $vEB\text{–}TREE\text{–}INSERT$ ще извърши само едно рекурсивно извикване. Как може да се измъкнем само с едно рекурсивно извикване? Когато вмъкваме елемент, или клъстерът, в който влиза, вече има друг елемент или не. Ако клъстерът вече има друг елемент, тогава номерът на клъстера вече е в $summary$ и затова **не** е обходимо да правим това рекурсивно извикване. Ако клъстерът все още няма друг елемент, тогава елементът, който се вмъква, става единственият елемент в клъстера и **не** е нужно да се връщаме, за да вмъкнем елемент в празно дърво на vEB:

$vEB\text{–}EMPTY\text{–}TREE\text{–}INSERT(V, x)$
1. $V.\min = x$
2. $V.\max = x$

С тази процедура под ръка, вече може да представим псевдокода за $vEB\text{–}TREE\text{–}INSERT(V, x)$ процедурата, която приема, че x не се намира в множеството, представено от vEB дървото V:

$vEB\text{–}TREE\text{–}INSERT(V, x)$
1. **if** $V.\min == NIL$
2. $\quad vEB\text{–}EMPTTY\text{–}TREE\text{–}INSERT(V, x)$
3. **else**
4. $\quad$ **if** $x < V.\min$
5. $\quad\quad exchange\ x\ with\ V.\min$
6. $\quad$ **if** $V.u > 2$
7. $\quad\quad$ **if** $vEB\text{–}TREE\text{–}MINIMUM(V.cluster[high(x)]) == NIL$
8. $\quad\quad\quad vEB\text{–}TREE\text{–}INSERT(V.summary, high(x))$
9. $\quad\quad\quad vEB\text{–}EMPTY\text{–}TREE\text{–}INSERT(V.cluster[high(x)], low(x))$
10. $\quad\quad$ **else** $vEB\text{–}TREE\text{–}INSERT(V.cluster[high(x)], low(x))$
11. **if** $x > V.\max$
12. $\quad V.\max = x$

Тази процедура работи по следния начин. Ред 1 проверява дали V е празно vEB дърво и ако е, тогава ред 2 обработва този лесен случай. Редове 3-10 предполагат, че V не е празно и следователно някои елементи ще бъдат вмъкнати в един от клъстерите на V. Но този елемент може да не е непременно елементът x, предаден на $vEB\text{–}TREE\text{–}INSERT$. Ако $x < \min$, както е тествано в ред 4, тогава x трябва да стане новото $\min$. Не искаме обаче да загубим оригиналния $\min$ и затова трябва да го вмъкнем в един от клъстерите на V. В този случай ред 5 разменя x с $\min$, така че да вмъкнем оригиналния $\min$ в един от клъстерите на V.

Изпълняваме редове 7-10 само ако V не е в базовото vEB дърво. Ред 6 определя дали клъстерът, в който ще влезе x, в момента е празен. Ако е така, тогава ред 8 вмъква номера на клъстера на x в $summery$, а ред 9 обработва лесния случай на вмъкване на x в празен клъстер. Ако клъстерът на x понастоящем не е празен, тогава ред 10 вмъква x в неговия клъстер. В този случай **не** е обходимо да актуализираме $summary$, тъй като номерът на клъстера на x вече е член на $summary$.

Накрая, редове 11-12 се грижат за актуализиране на $\max$, ако $x > \max$. Обърнете внимание, че ако V е базово vEB дърво, което не е празно, тогава редове 4-5 и 11-12 актуализират правилно $\min$ и $\max$.

За пореден път може лесно да видим как рекурентната зависимост от ◆ характеризира времето за работа. В зависимост от резултата от теста в ред 6, се изпълняват или рекурсивното извикване на ред 8 (изпълнява се върху vEB дърво с размер на Вселената $\sqrt[\uparrow]{u}$) или рекурсивното извикване на ред 10 (изпълнява се върху vEB дърво с размер на Вселената $\sqrt[\downarrow]{u}$). И в двата случая имаме едно рекурсивно извикване върху vEB дървовидна структура с размер на Вселената най-много $\sqrt[\uparrow]{u}$. Тъй като останалата част от $vEB\text{–}TREE\text{–}INSERT$ отнема $O(1)$ време, рекурентната зависимост от ◆ е в сила и за тази функция и следователно цялото време за изпълнението на процедурата е от порядъка на $O(\lg \lg u)$ в лошия случай.

3.2.5. Изтриване на елемент

Накрая, нека разгледаме как да изтрием елемент от vEB дърво. Процедурата $vEB\text{–}TREE\text{–}DELETE(V, x)$ допуска, че в момента на извикването ѝ, елемента x се намира в множеството представено от vEB дървото V.

```
vEB–TREE–DELETE(V, x)
1.  if V.min == V.max
2.      V.min = NIL
3.      V.max = NIL
4.  elseif V.u == 2
5.      if x == 0
6.          V.min = 1
7.      else V.min = 0
8.      V.max = V.min
9.  else if x == V.min
10.         first–cluster = vEB–TREE–MINIMUM(V.summary)
11.         x = index(first–cluster,
                      vEB–TREE–MINIMUM(V.cluster[first–cluster]))
12.         V.min = x
13.     vEB–TREE–DELETE(V.cluster[high(x)], low(x))
14.     if vEB–TREE–MINIMUM(V.cluster[high(x)]) == NIL
15.         vEB–TREE–DELETE(V.summary, high(x))
16.         if x == V.max
17.             summary–max = vEB–TREE–MAXIMUM(V.summary)
18.             if summary–max == NIL
19.                 V.max = V.min
```

20. **else** $V.max = index(summary\text{–}max,$
$vEB\text{–}TREE\text{–}MAXIMUM(V.cluster[summary\text{–}max]))$
21. **elseif** $x == V.\max$
22. $V.\max = index(high(x),$
$vEB\text{–}TREE\text{–}MAXIMUM(V.cluster[high(x)]))$

Процедурата $vEB\text{–}TREE\text{–}DELETE$ работи по следния начин. Ако vEB дървото V съдържа само един елемент, тогава е също толкова лесно да го изтрием, колкото и да вмъкнем елемент в празно vEB дърво: просто присвояваме на $\min$ и $\max$ атрибутите стойността NIL. Редовете 1-3 се справят с този случай. В противен случай V има поне два елемента. Ред 4 тества дали V е базово vEB дърво и ако е такова, редове 5-8 задават $\min$ и $\max$ на единствения оставащ елемент.

Редове 9-22 предполагат, че V има два или повече елемента и че $u \geq 4$. В този случай ще трябва да изтрием елемент от клъстер. Елементът, който изтриваме от клъстер, може да не е x, защото, ако x е равно на $\min$, тогава след като изтрием x, някой друг елемент в един от клъстерите на V става новият $\min$ и трябва да изтрием този друг елемент от неговия клъстер. Ако тестът на ред 9 разкрие, че сме в тои случай, тогава ред 10 присвоява на $first\text{–}cluster$ променливата - номера на клъстера, който съдържа най-малкия елемент различен от $\min$, а ред 11 присвоява на x - стойността на най-малкия елемент на този клъстер. Този елемент се превръща в новия $\min$ на ред 12 и тъй като присвояваме x на неговата стойност, той е елементът, който ще бъде изтрит от неговия клъстер.

Когато достигнем ред 13, знаем че трябва да изтрием елемент x от неговия клъстер, независимо дали x е стойността, първоначално предадена на $vEB\text{–}TREE\text{–}DELETE$ или x е елемента, който се превръща в новия минимум. Ред 13 изтрива x от своя клъстер. Този клъстер вече може да стане празен, а ред 14 тества именно за това, и ако това се случи, тогава трябва да премахнем номера на клъстера на x от $summary$, което се обработва от ред 15. След актуализиране на $summary$ може да се наложи да актуализираме $\max$. Ред 16 проверява дали изтриваме максималния елемент във V и ако е така, тогава ред 17 присвоява на $summary\text{–}max$ променливата - номера на най-голямия непразен клъстер. (Извикването $vEB\text{–}TREE\text{–}MAXIMUM(V.summary)$ работи, защото вече рекурсивно сме извикали $vEB\text{–}TREE\text{–}DELETE$ на $V.summary$ и следователно $V.summary.\max$ вече е актуализирано, ако е било необходимо.) Ако всички клъстери на V са празни, тогава единственият оставащ елемент във V е $\min$; ред 18 проверява за този случай и ред 19 актуализира $\max$ по подходящ начин. В противен случай ред 20 задава $\max$ на максималния елемент в клъстера с най-голям номер. (Ако този клъстер е там, където елементът е бил изтрит, ние отново разчитаме на рекурсивното извикване на ред 13, който вече е коригирал максималния атрибут на този клъстер.)

Накрая, трябва да се справим със случая, в който клъстерът на x не е станал празен поради изтриването на x. Въпреки, че в този случай не се налага да актуализираме $summary$, може да се наложи да актуализираме $\max$. Ред 21 тества за този случай и ако трябва да актуализираме $\max$, ред 22 го прави (отново разчитайки на рекурсивното извикване за коригиране на $\max$ в клъстера).

Сега, остана да покажем, че операцията $vEB\text{–}TREE\text{–}DELETE$ се изпълнява за $O(\lg \lg u)$ време в лошия случай. На пръв поглед може да си помислим, че рекурентната зависимост от ◆ не винаги се прилага, тъй като едно извикване на $vEB\text{–}TREE\text{–}DELETE$ може да направи две рекурсивни извиквания: едно на ред 13 и едно на ред 15. Въпреки че процедурата може да направи и двете рекурсивни извиквания, нека помислим какво се

случва когато го направи. За да се извика рекурсията на ред 15, е необходимо теста на ред 14 да покаже, че клъстерът на x е празен. Единственият начин клъстерът на x да бъде празен е, ако x е бил единственият елемент в неговия клъстер, когато направихме рекурсивното извикване на ред 13. Но ако x беше единственият елемент в неговия клъстер, тогава това рекурсивно извикване е отнело $O(1)$ време, тъй като се е обработила само в редовете от 1-3. Следователно имаме две взаимно изключващи се възможности:

- Рекурсивното извикване на ред 13 се изпълнява за константно време.
- Рекурсивното извикване на ред 15 не се случва.

И в двата случая, рекурентната зависимост от ♦ характеризира времето за работа на $vEB\text{–}TREE\text{–}DELETE$ и следователно в лошия случай тази процедура ще струва $O(\lg \lg u)$ време.