Алгоритми та складність

II семестр

Лекція 7

Реалізації черги з пріоритетами

Процедура	Бинарная пирамида (наихудший случай)	Биномиальная пирамида (наихудший случай)	Пирамида Фибоначчи (амортизированное время)
MAKE_HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT_MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Omega(\lg n)$	$\Theta(1)$
DECREASE_KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

- Також можлива реалізація на червоно-чорних деревах.
- Обов'язково знайдеться важлива операція, що виконується за O(log *n*), чи у найгіршому випадку, чи за амортизований час.

Реалізації черги з пріоритетами

- Кожна зі згаданих структур фактично базується на порівнянні ключів.
- Нижня оцінка часу сортування $\Omega(n \log n)$, тому хоча б одна з операцій має виконуватися за $\Omega(\log n)$.
- Інакше, якщо б операції INSERT та EXTRACT_MIN могли виконуватися за $o(\log n)$, то ми би відсортували n ключів за $o(n \log n)$ після спочатку n операцій INSERT, а потім n операцій EXTRACT_MIN.
- Однак існують сортування, які використовують додаткову інформацію про природу ключів, і тому працюють швидше (те ж сортування підрахунком).
- Чи не можна створити швидшу реалізацію черги з пріоритетами, якщо ключі будуть цілими числами з обмеженого діапазону?

Дерева ван Емде Боаса

- Van Emde Boas tree – vEB-дерево.
- Підтримують операції над динамічними множинами з найгіршим часом O(log log u).
- Умова: ключі цілі з діапазону 0..(u–1), повтори ключів не дозволяються.
- Зосередимось на питанні зберігання ключів (опускаючи моменти щодо супутніх даних).
- Замість операції SEARCH буде реалізована простіша булева операція MEMBER(S,x): чи містить динамічна множина S значення x.
- Позначимо n – кількість елементів множини в поточний момент, u (причому $u=2^k$, ціле $k \geq 1$) – розмір універсума значень, що можуть зберігатися у дереві ({0,1,…,(u–1)}).
- Тоді час виконання операцій складе O(log log u).

Попередні підходи

Розглянемо деякі підходи до зберігання динамічної множини, які приведуть до ідей в основі дерев ван Емде Боаса.

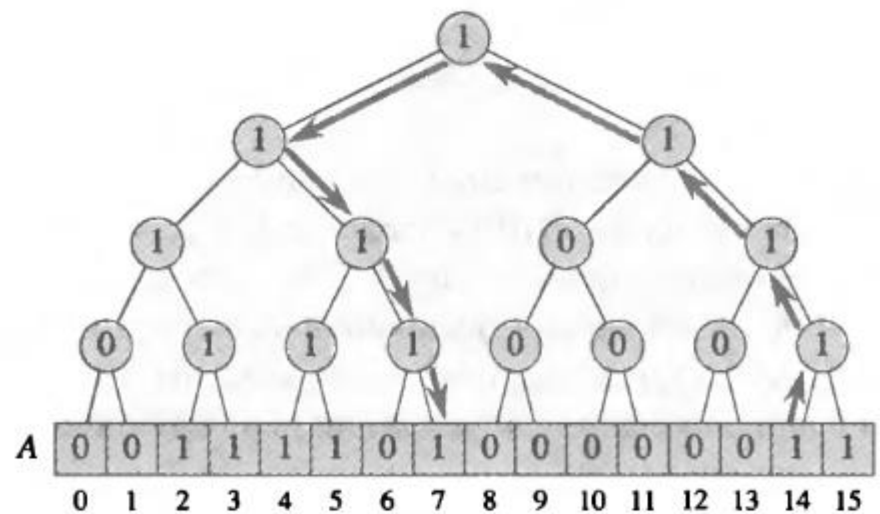
Пряма адресація

- Найпростіший варіант – динамічна множина {0,1,…,(u–1)} зберігається у бітовому векторі A[0.. u–1].
- Елемент A[x] містить 1, якщо значення x належить множині, і 0 – якщо не належить.
- Операції INSERT, DELETE і MEMBER виконуються за час O(1).
- Операції MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR виконуються в найгіршому випадку за $\Theta(u)$, оскільки може знадобитися перегляд $\Theta(u)$ елементів масиву.

Попередні підходи

Накладання структури бінарного дерева

- Елементи бітового вектора утворюють листя бінарного дерева; кожен внутрішній вузол містить 1 тоді й тільки тоді, коли деякий лист його піддерева містить 1 (логічний OR дочірніх вузлів).



Бінарне дерево бітів, накладене на бітовий вектор для множини {2,3,4,5,7,14,15} при *u* = 16.

За стрілками – пошук попередника ключа 14 у множині.

Попередні підходи

Пошук використовує структуру дерева.

- Мінімум: рух від кореня до листа, завжди вибираючи *найлівіший* вузол, що містить 1.
- Максимум: рух від кореня до листа, завжди вибираючи *найправіший* вузол, що містить 1.
- Наступник x: спочатку рух вгору від листа x, поки не зайдемо *зліва* у вузол, *правий син* якого z містить 1; потім пошук мінімуму в піддереві z.
- Попередник x: спочатку рух вгору від листа x, поки не зайдемо *справа* у вузол, *лівий син* якого z містить 1; потім пошук максимуму в піддереві z.

Попередні підходи

- Вставка: зберігається 1 у кожному вузлі на простому шляху від вставленого у лист значення догори до кореня.
- Видалення: аналогічно рух вгору від листа до кореня з новим обчисленням бітів кожного внутрішнього вузла шляху як логічного OR синів.

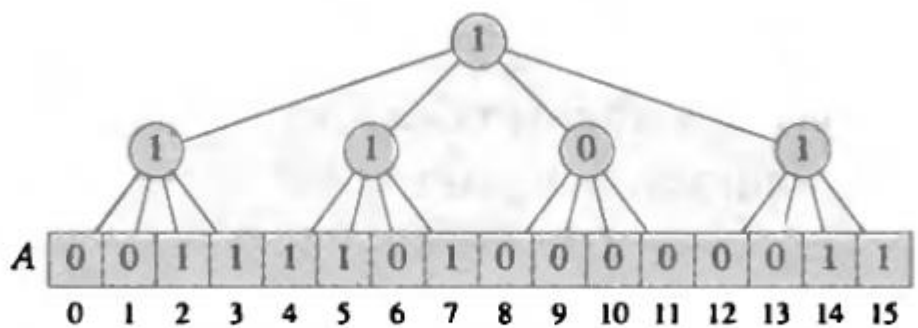
Кожна з розглянутих операцій виконується не більш як за два проходи по дереву висоти $\log u$, тому їх найгірша часова оцінка $O(\log u)$.

Отриманий підхід кращий за використання червоно-чорних дерев лише константним часом роботи операції MEMBER, але у випадку, коли кількість елементів множини n суттєво менша за u, червоно-чорні дерева працюватимуть швидше на всіх інших операціях.

Попередні підходи

Накладання дерева константної висоти

- Припустимо $u = 2^{2k}$ для деякого цілого k, що $\sqrt{u}$ – ціле.
- Накладемо на бітовий вектор дерево степеня $\sqrt{u}$.
- Висота такого дерева завжди дорівнюватиме 2.

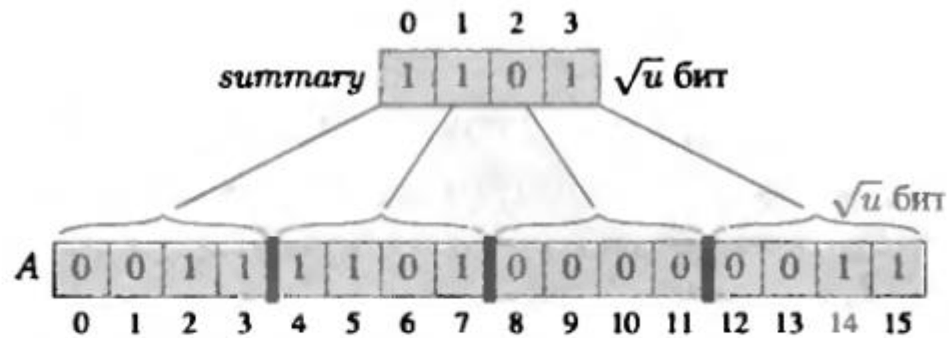


Дерево степеня $\sqrt{u}$, накладене на бітовий вектор для множини {2,3,4,5,7,14,15} при u = 16.

Кожен внутрішній вузол зберігає побітове OR своїх синів.

Попередні підходи

- $\sqrt{u}$ внутрішніх вузлів на глибині 1 можна розглядати як масив *summary*[0..$\sqrt{u}-1$], де *summary*[i] містить 1 ⇔ підмасив A[$i\sqrt{u}..(\mathrm{i}+1)\sqrt{u}-1$] містить 1.



- Такий $\sqrt{u}$-бітовий підмасив A назвемо *i*-м *кластером*.
- Для заданого значення x біт A[x] міститься у кластері номер $\lfloor x/\sqrt{u} \rfloor$.
- Операція INSERT виконується за O(1): для вставки x присвоюємо 1 коміркам A[x] та *summary*[$\lfloor x/\sqrt{u} \rfloor$].

Попередні підходи

Використаємо масив *summary* для інших операцій.

- MINIMUM (MAXIMUM): знаходиться крайній зліва (справа) елемент *summary*, що містить 1, потім лінійний пошук у відповідному кластері найлівішої (найправішої) 1.
- SUCCESSOR (PREDECESSOR) x: спочатку пошук направо (наліво) в межах відповідного кластера номер $i = \lfloor x/\sqrt{u} \rfloor$. Якщо не знайшли, пошук в масиві *summary* правіше (лівіше) індекса *i*; перша позиція, що містить 1, дає індекс кластера, в ньому шукаємо найлівішу (найправішу для попередника) позицію 1.
- DELETE x: покладемо $i = \lfloor x/\sqrt{u} \rfloor$; встановимо A[x] рівним 0, а summary[i] – значенню логічного OR бітів в *i*-му кластері.

Попередні підходи

В кожній з описаних операцій виконується пошук не більш ніж у двох кластерах по $\sqrt{u}$ бітів, а також по масиву *summary*.

Тому час роботи вказаних операцій O($\sqrt{u}$).

Хоча попередній варіант з накладанням бінарного дерева і давав нам логарифмічний час виконання операцій, саме застосування дерева степені $\sqrt{u}$ виявляється ключовою ідеєю дерев ван Емде Боаса.

Протоструктури ван Емде Боаса

- Модифікуємо ідею накладання дерева степені $\sqrt{u}$ на бітовий вектор.
- Зробимо структуру рекурсивною, кожен раз зменшуючи розмір універсуму в квадратний корінь разів: для універсуму розміром *u* робимо структури, що зберігають $\sqrt{u} = u^{1/2}$ елементів, які зберігають структури по $u^{1/4}$ елементів, що зберігають структури по $u^{1/8}$ елементів, – і так до базового розміру 2.
- Для простоти вважаємо $u = 2^{2^k}$ для деякого цілого *k* – так розмір структур на всіх рівнях буде цілим. (Таке обмеження дуже жорстке, в деревах ван Емде Боаса буде достатньо $u = 2^k$.)

Протоструктури ван Емде Боаса

- Спробуємо уявити, звідки можна отримати час роботи O(log log u).
- Розглянемо рекурентне співвідношення
$$T(u) = T(\sqrt{u}) + O(1).$$
- Його розв’язок O(log log u).
- Маємо отримати рекурсивну структуру даних, яка на кожному рівні рекурсії буде зменшуватися в $\sqrt{u}$ раз. Коли операція обходить цю структуру, на кожному рівні вона повинна витрачати константний час.

Працюватимемо з ключем x як (log *u*)-бітним цілим числом і введемо допоміжні функції для роботи над таким поданням.

Протоструктури ван Емде Боаса

- Для полегшення індексації визначимо функції:

$$\mathrm{high}(x) = \lfloor x/\sqrt{u} \rfloor,$$

дає (log *u*)/2 старших бітів числа x,
повертаючи номер кластера x;

$$\mathrm{low}(x) = x \bmod \sqrt{u},$$

дає (log *u*)/2 молодших бітів числа x,
вказуючи позицію x в його кластері;

$$\mathrm{index}(x, y) = x\sqrt{u} + y,$$

будує номер елемента зі значень x та y,
де x – це (log *u*)/2 старших бітів,
а y – (log *u*)/2 молодших бітів номера елемента.

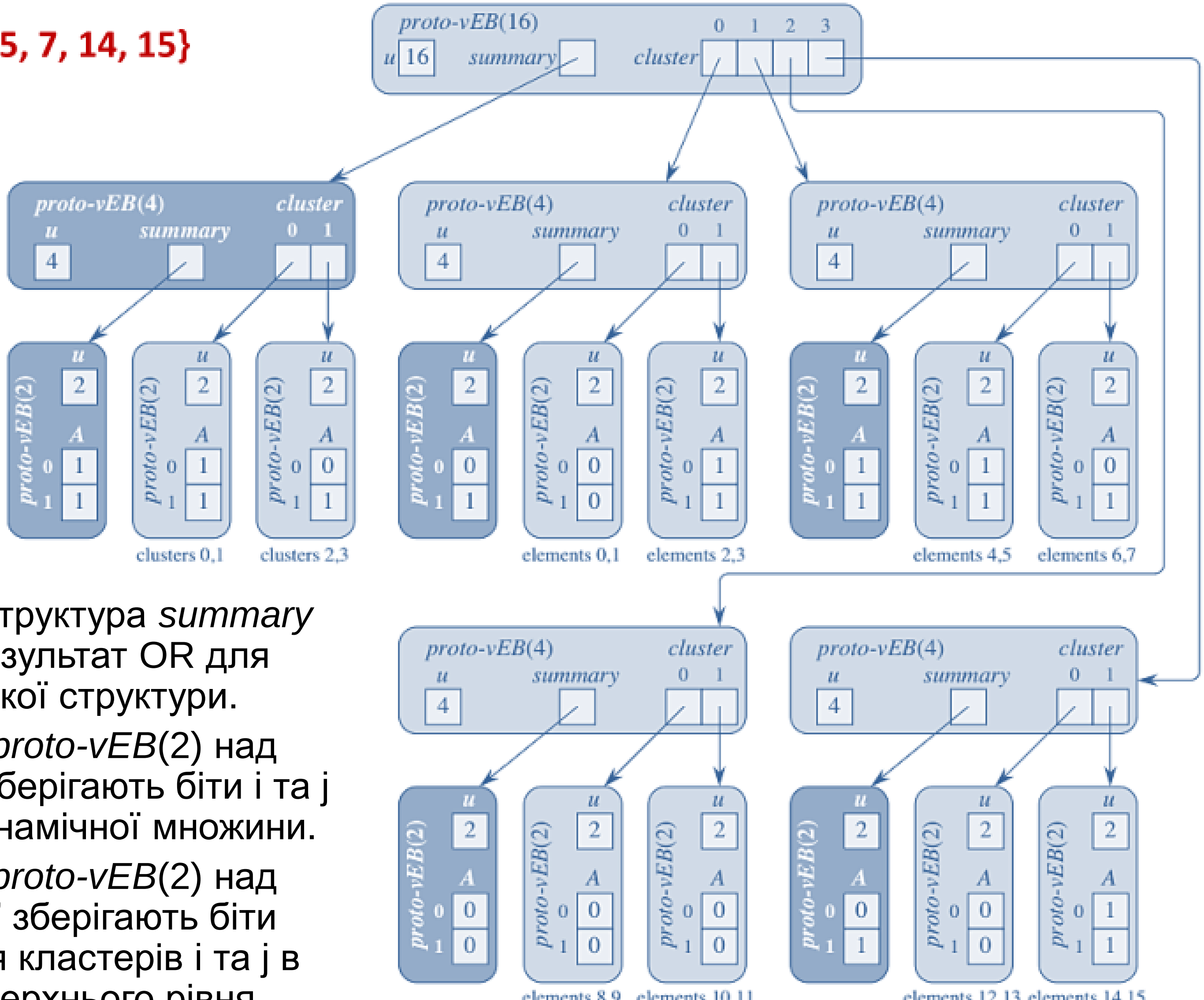
- Справедливо $x = \mathrm{index}(\mathrm{high}(x), \mathrm{low}(y))$.
- За *u* береться розмір універсуму структури даних на поточному рівні.

Протоструктури ван Емде Боаса

- *Протоструктура ван Емде Боаса* (proto van Emde Boas structure, *proto-vEB*-структура) служитиме основою для структури дерева ван Емде Боаса.
- Зберігає ключі з універсума {0,1,…,(*u*–1)}.
- Кожен вузол дерева *proto-vEB* містить розмір універсума *u*.
- При *u* = 2:
  - базовий розмір, структура містить масив з двох бітів A[0..1].
- Інакше:
  - вказівник *summary* на структуру *proto-vEB*($\sqrt{u}$);
  - масив *cluster*[0.. $\sqrt{u}-1$] з $\sqrt{u}$ вказівників на структури *proto-vEB*($\sqrt{u}$).

Протоструктури ван Емде Боаса

U = {2, 3, 4, 5, 7, 14, 15}
u = 16



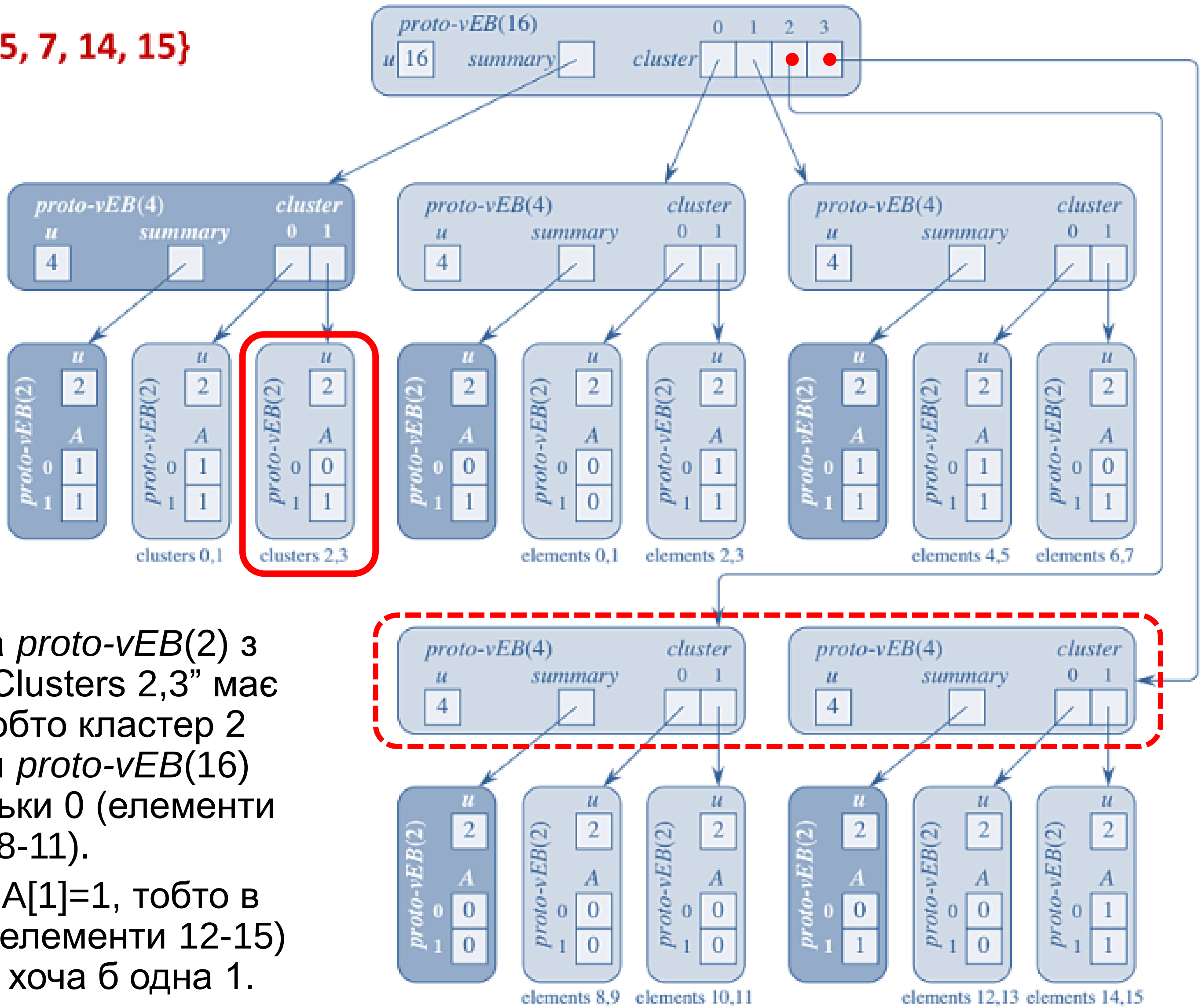
Резюмуюча структура *summary* зберігає результат OR для батьківської структури.

Структури *proto-vEB*(2) над “Elements i,j” зберігають біти i та j фактичної динамічної множини.

Структури *proto-vEB*(2) над “Clusters i,j” зберігають біти *summary* для кластерів i та j в структурі верхнього рівня.

Протоструктури ван Емде Боаса

$U = \{2, 3, 4, 5, 7, 14, 15\}$
$u = 16$

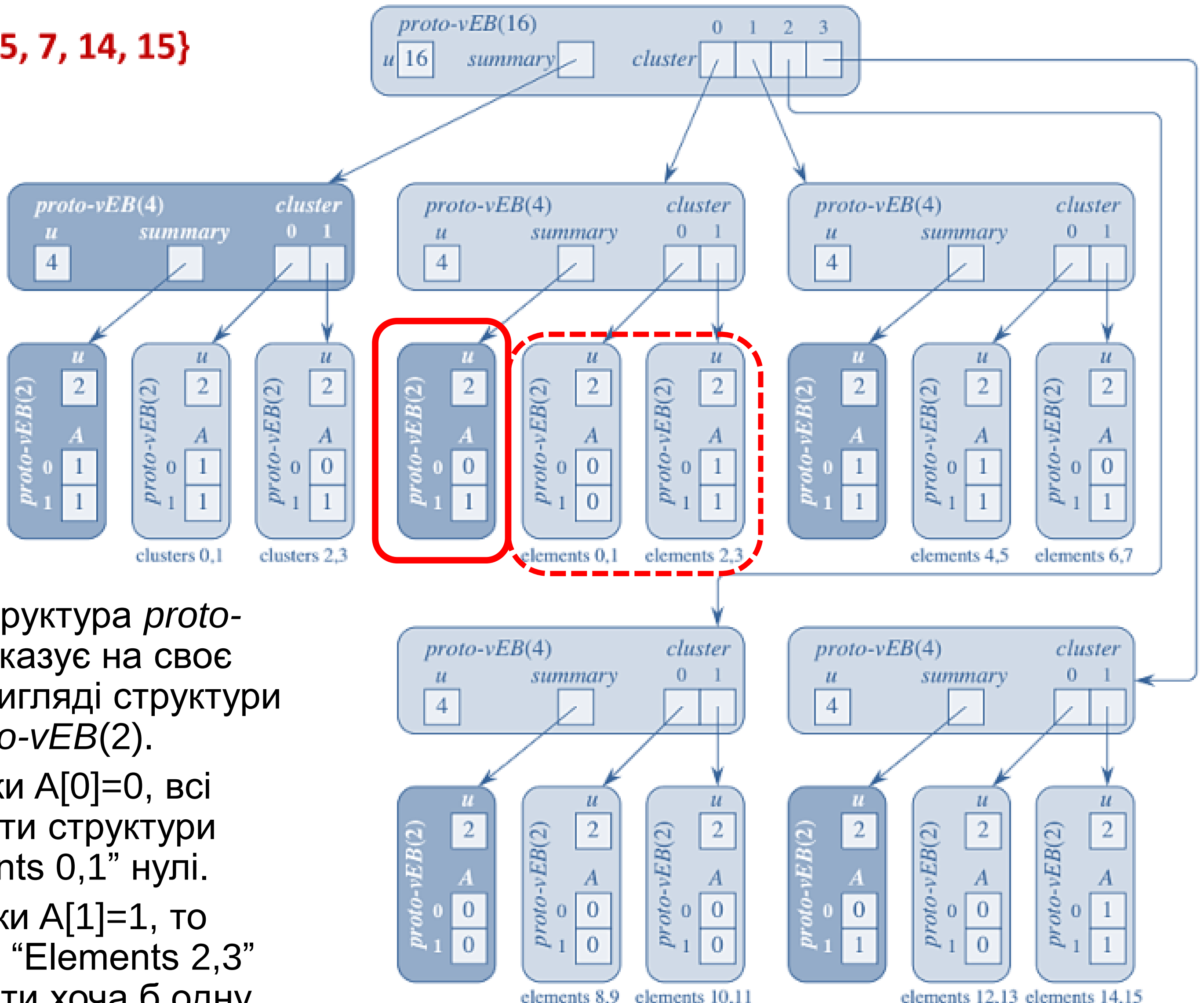


Структура *proto-vEB*(2) з поміткою “Clusters 2,3” має A[0]=0, тобто кластер 2 структури *proto-vEB*(16) містить тільки 0 (елементи 8-11).

Водночас A[1]=1, тобто в кластері 3 (елементи 12-15) міститься хоча б одна 1.

Протоструктури ван Емде Боаса

U = {2, 3, 4, 5, 7, 14, 15}
u = 16



Кожна структура *proto-vEB*(4) вказує на своє резюме у вигляді структури *proto-vEB*(2).

Оскільки A[0]=0, всі елементи структури “Elements 0,1” нулі.

Оскільки A[1]=1, то структура “Elements 2,3” має містити хоча б одну одиницю.

Операції над *proto-vEB*-структурами

- *Перевірка наявності елемента в множині* MEMBER.

```
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))
```

Рядок 2 – базовий випадок, повертається відповідний біт масиву.

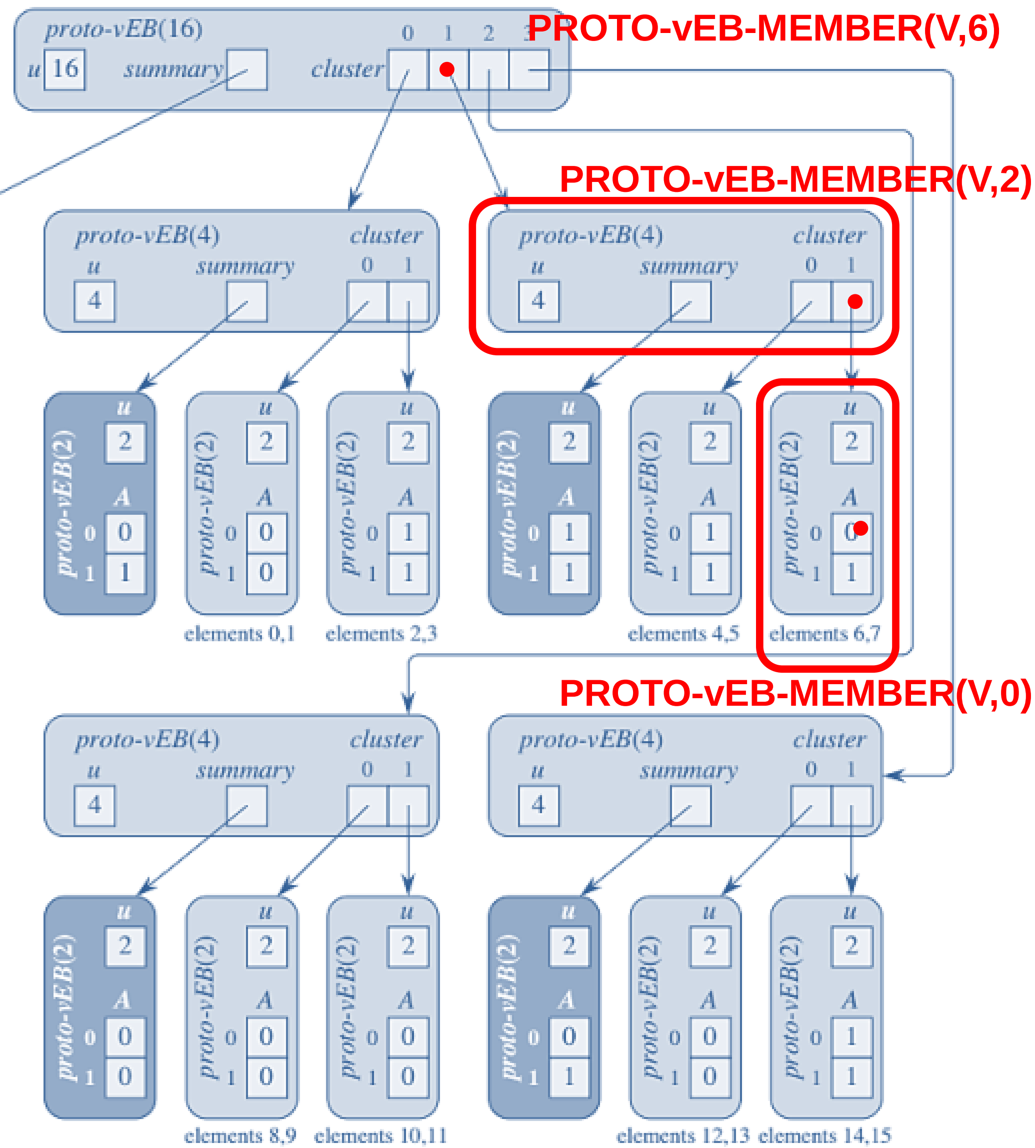
Інакше рекурсивний пошук у кластері номер high(*x*) елемента зі значенням low(*x*).

- Час роботи T(u): O(1) плюс рекурсивний виклик в структурі *proto-vEB*($\sqrt{u}$): $T(u) = T(\sqrt{u}) + O(1)$.
- Тобто складність операції O(log log u).

Протоструктури ван Емде Боаса

$U = \{2, 3, 4, 5, 7, 14, 15\}$

$u = 16$



PROTO-vEB-MEMBER(V,6) ($u = 16$)

high(6) = 1 0110

low(6) = 2

PROTO-vEB-MEMBER(V,2) ($u = 4$)

high(2) = 1 10

low(2) = 0

PROTO-vEB-MEMBER(V,0) ($u = 2$)

Базовий випадок: A[0] = 0
(елемент відсутній)

Операції над *proto-vEB*-структурами

- *Пошук мінімального елемента* MINIMUM.

```
PROTO-vEB-MINIMUM(V)
if V.u == 2
    if V.A[0] == 1
        return 0
    elseif V.A[1] == 1
        return 1
    else return NIL
else min-cluster = PROTO-vEB-MINIMUM(V.summary)
    if min-cluster == NIL
        return NIL
    else offset = PROTO-vEB-MINIMUM(V.cluster[min-cluster])
        return index(min-cluster, offset)
```

Операції над *proto-vEB*-структурами

Пошук мінімального елемента MINIMUM (далі)

- Спочатку безпосередньо перевіряється базовий випадок.
- Інакше в рядку 7 рекурсивно знаходиться номер *min-cluster* першого кластера, який містить 1 (через атрибут *summary*).
- Рекурсивно шукається зсув *offset* мінімального елемента в межах кластера *min-cluster*.
- За отриманими значеннями номера кластера і зсуву визначається мінімальний елемент.
- Маємо два рекурсивних виклики над *proto-vEB*($\sqrt{u}$), тому: $T(u) = 2T(\sqrt{u}) + O(1)$.
- Складність операції Θ(log *u*).

Операції над *proto-vEB*-структурами

- *Пошук наступника* SUCCESSOR.

```
PROTO-vEB-SUCCESSOR(V, x)
if V.u == 2
    if x == 0 и V.A[1] == 1
        return 1
    else return NIL
else offset = PROTO-vEB-SUCCESSOR(V.cluster[high(x)], low(x))
    if offset ≠ NIL
        return index(high(x), offset)
    else succ-cluster = PROTO-vEB-SUCCESSOR(V.summary, high(x))
        if succ-cluster == NIL
            return NIL
        else offset = PROTO-vEB-MINIMUM(V.cluster[succ-cluster])
            return index(succ-cluster, offset)
```

Операції над *proto-vEB*-структурами

Пошук наступника SUCCESSOR (далі)

- Базовий випадок при наявності наступника однозначний.
- Потім робиться спроба (рядок 5) рекурсивно знайти наступника *x* в його кластері.
- Інакше по резюмуючій інформації відбувається пошук першого наступного непорожнього кластера. Якщо такий знайшовся, обчислюється його мінімальний елемент.
- Рекурентне співвідношення для найгіршого випадку:

$$T(u) = 2T(\sqrt{u}) + \Theta(\lg \sqrt{u}) = 2T(\sqrt{u}) + \Theta(\lg u).$$

- Його розв’язок Θ(log *u* log log *u*).

Операції над *proto-vEB*-структурами

- *Вставка елемента* INSERT.

```
PROTO-vEB-INSERT(V, x)
if V.u == 2
    V.A[x] = 1
else PROTO-vEB-INSERT(V.cluster[high(x)], low(x))
    PROTO-vEB-INSERT(V.summary, high(x))
```

- В базовому випадку відповідний біт масиву A стає 1.
- В рекурсивному випадку вставляємо *x* у відповідний кластер та встановлюємо його резюмуючий біт = 1.
- Рекурентне співвідношення для операції

$$T(u) = 2T(\sqrt{u}) + O(1).$$

- Розв’язок Θ(log *u*).

Операції над *proto-vEB*-структурами

- *Видалення елемента DELETE*.
- Операція складніша за вставку – при видаленні не можна просто скинути резюмуючий біт в 0.
- Слід перевірити, чи містить кластер одиничні біти – дослідити всі його $\sqrt{u}$ біт.

Загалом необхідно модифікувати протоструктуру ван Емде Боаса так, щоб кожна операція виконувала не більше одного рекурсивного виклику.

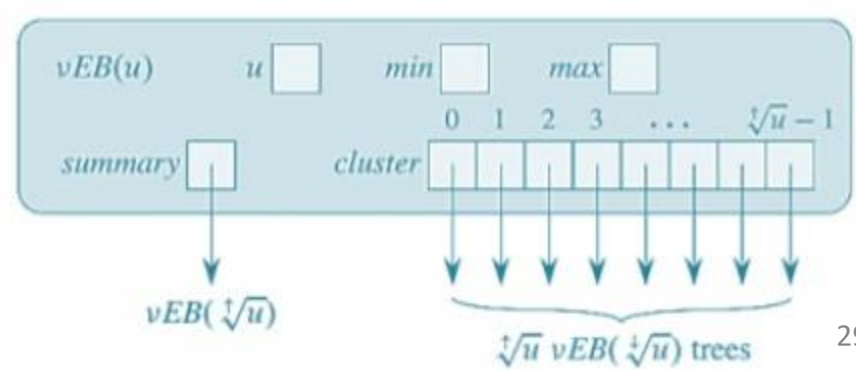
Дерево ван Емде Боаса

- Послабимо припущення щодо розміру універсуму до $u = 2^k$.
- У випадку нецілого значення $\sqrt{u}$ ділитимемо ($\log u$) біт числа на старші $\lceil(\log u)/2\rceil$ та молодші $\lfloor(\log u)/2\rfloor$ біт.
- Позначимо $2^{\lceil(\log u)/2\rceil} = {\uparrow}\sqrt{u}$, $2^{\lfloor(\log u)/2\rfloor} = {\downarrow}\sqrt{u}$.
- Тоді $u = {\uparrow}\sqrt{u} \cdot {\downarrow}\sqrt{u}$.
- При цьому ${\uparrow}\sqrt{u} = {\downarrow}\sqrt{u} = \sqrt{u}$ при парному k.
- Перевизначимо допоміжні функції:

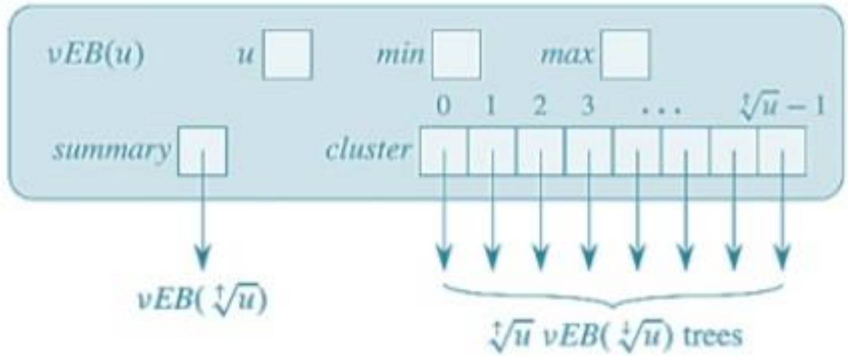
$$\mathrm{high}(x) = \lfloor x/{\uparrow}\sqrt{u} \rfloor,$$
$$\mathrm{low}(x) = x \bmod {\downarrow}\sqrt{u},$$
$$\mathrm{index}(x, y) = x{\downarrow}\sqrt{u} + y.$$

Дерево ван Емде Боаса

- *Дерево ван Емде Боаса* (van Emde Boas tree, *vEB*-дерево) матиме наступну структуру.
- Зберігає ключі з універсума {0,1,…,(*u*–1)} // vEB(*u*)
- Кожен вузол дерева *vEB* містить розмір *u*, значення мінімального елемента *min* та максимального елемента *max*.
- В небазовому випадку (*u* > 2):
  - вказівник *summary* на дерево *vEB*($\uparrow\sqrt{u}$);
  - масив *cluster*[0.. $\uparrow\sqrt{u}-1$] з $\uparrow\sqrt{u}$ вказівників на корені дерев *vEB*($\downarrow\sqrt{u}$).



Дерево ван Емде Боаса



- Елемент, що зберігається в *min*, вже не з'явиться в рекурсивних деревах масиву *cluster*.
- Елемент, що зберігається в *max*, також наявний і в кластері (крім випадку єдиного елемента у дереві).
- Елементи vEB-дерева базового розміру визначаються за атрибутами *min* та *max*.
- В дереві без елементів атрибути *min* та *max* дорівнюють NIL незалежно від розміру універсуму.

Дерево ван Емде Боаса

Атрибути *min* та *max* дозволяють зменшити кількість рекурсивних викликів при роботі з vEB-деревом.

- Операції MINIMUM (MAXIMUM) повертають значення відповідних атрибутів.
- При визначенні наступника операцією SUCCESSOR можна позбутися рекурсії: наступник x буде міститися в цьому ж кластері, тільки якщо x < *max* (аналогічно для PREDECESSOR та *min*).
- За значеннями *min* та *max* можна за константний час визначити, скільки елементів у vEB-дереві: 0, 1 чи мінімум 2.
- Простим оновленням атрибутів можна вставити елемент у порожнє vEB-дерево чи видалити його єдиний елемент.

Дерево ван Емде Боаса

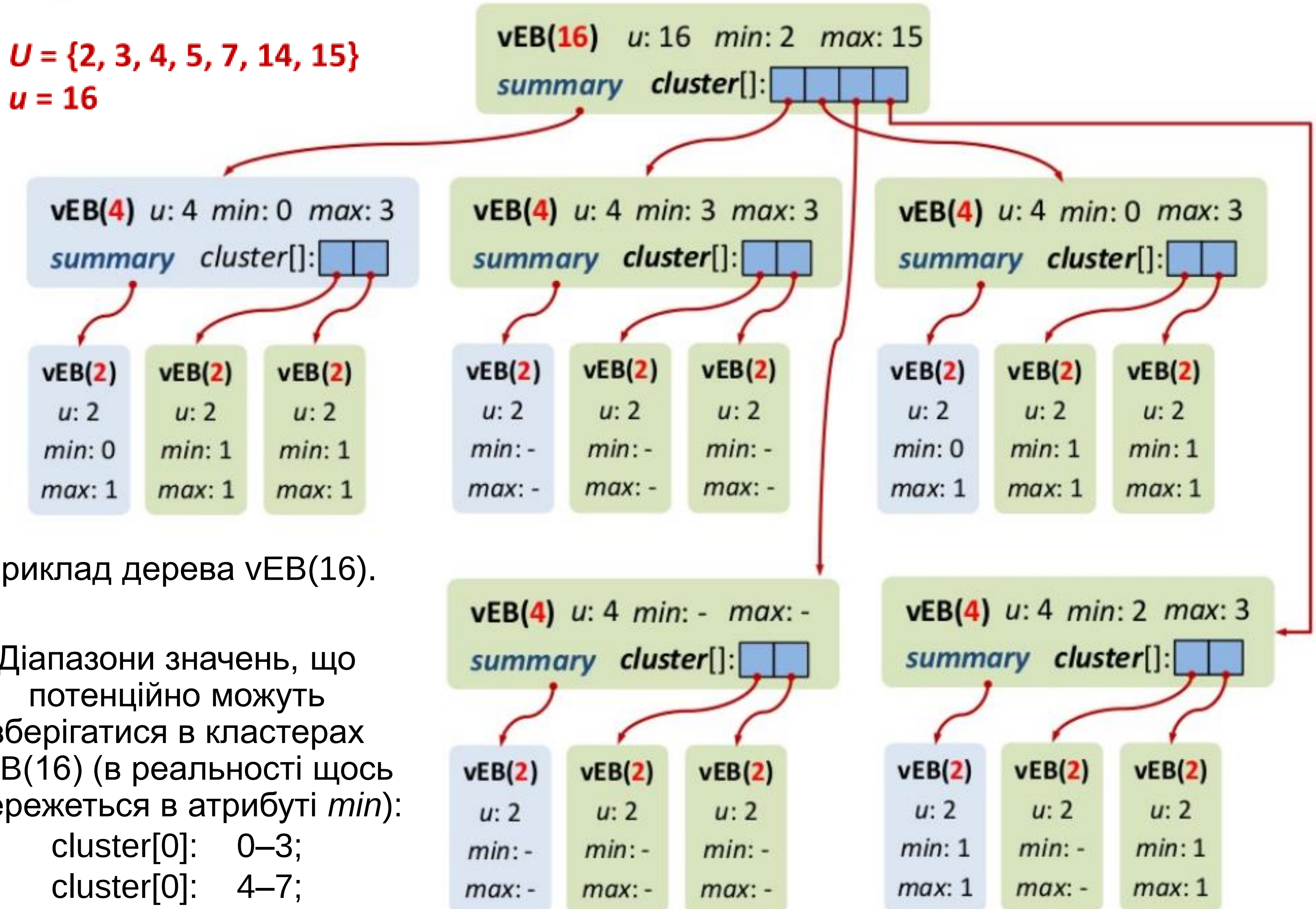
- Всі рекурсивні процедури, які реалізують операції над vEB-деревом, описуються рекурентним співвідношенням

$$T(u) \leq T\left(\uparrow\sqrt{u}\right) + O(1).$$

- Наявність операції взяття «верхнього квадратного кореня» не вплине на його розв'язок O(log log *u*).

- Для представлення vEB-дерева з розміром універсуму *u* потрібна пам'ять O(*u*), тому час на створення порожнього дерева складе $\Theta(u)$.

Дерево ван Емде Боаса

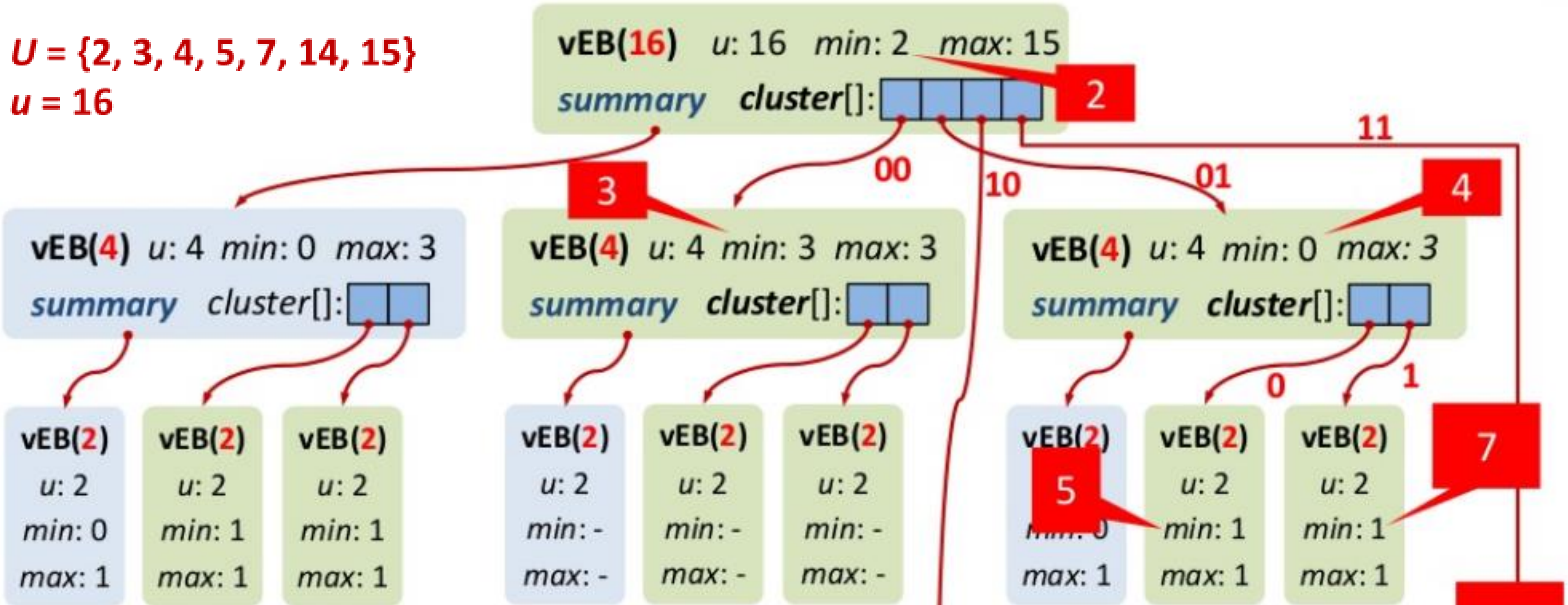


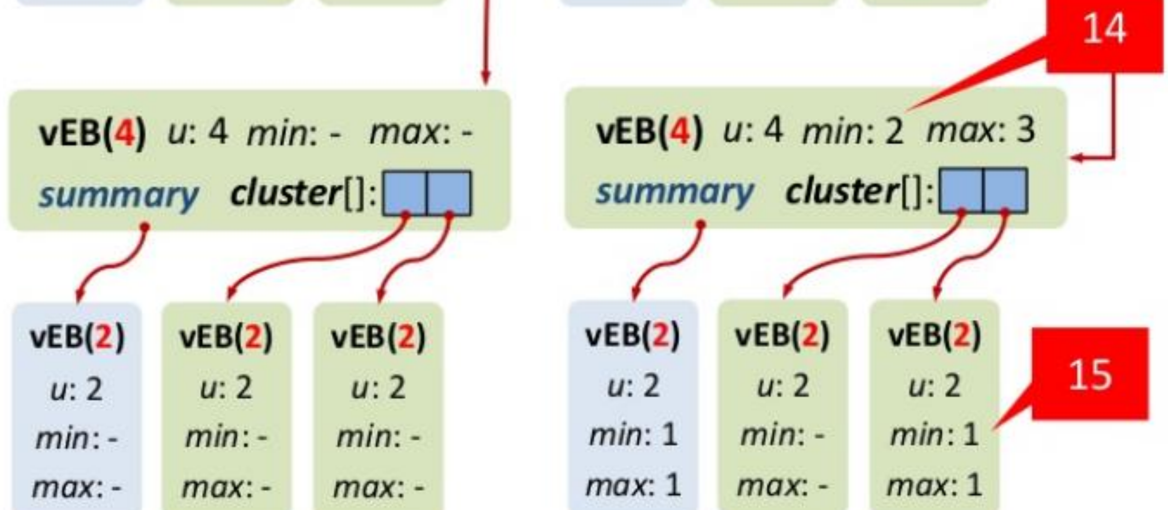
Приклад дерева vEB(16).

Діапазони значень, що потенційно можуть зберігатися в кластерах vEB(16) (в реальності щось збережеться в атрибуті *min*):

cluster[0]: 0–3;
cluster[0]: 4–7;
cluster[0]: 8–11;
cluster[0]: 12–15.

Дерево ван Емде Боаса





Приклад дерева vEB(16).

Показано, де зберігається кожен з ключів.

Значення, що зберігається в атрибуті *min* vEB-дерева, не з’являється в жодному з його кластерів.

Операції над деревом ван Емде Боаса

- *Пошук мінімального та максимального елемента.*

```
vEB-TREE-MINIMUM(V)
1  return V.min

vEB-TREE-MAXIMUM(V)
1  return V.max
```

- Виконуються за константний час.

Операції над деревом ван Емде Боаса

- *Перевірка наявності елемента в множині* MEMBER.

```
vEB-TREE-MEMBER(V, x)
if x == V.min или x == V.max
    return TRUE
elseif V.u == 2
    return FALSE
else return vEB-TREE-MEMBER(V.cluster[high(x)], low(x))
```

- vEB-дерево не зберігає біти, тому процедура повертає булеві значення.
- Значення x звіряється з *min* та *max*, потім перевіряється можливість базового випадку, інакше відбувається рекурсивний виклик.

Операції над деревом ван Емде Боаса

- *Пошук наступника* SUCCESSOR.

```
vEB-TREE-SUCCESSOR(V, x)
if V.u == 2
    if x == 0 и V.max == 1
        return 1
    else return NIL
elseif V.min ≠ NIL и x < V.min
    return V.min
else max-low = vEB-TREE-MAXIMUM(V.cluster[high(x)])
    if max-low ≠ NIL и low(x) < max-low
        offset = vEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
        return index(high(x), offset)
    else succ-cluster = vEB-TREE-SUCCESSOR(V.summary, high(x))
        if succ-cluster == NIL
            return NIL
        else offset = vEB-TREE-MINIMUM(V.cluster[succ-cluster])
            return index(succ-cluster, offset)
```

Операції над деревом ван Емде Боаса

Пошук наступника SUCCESSOR (далі)

- Базовий випадок при наявності наступника однозначний: наступником 0 є 1.
- Далі (рядок 5) перевіряємо, чи x строго менший за мінімум у vEB-дереві.
- Потім робиться спроба (рядок 7) рекурсивно знайти наступника *x* в його кластері.
- Інакше (рядок 11) по резюмуючій інформації рекурсивно шукаємо перший наступний непорожній кластер і в ньому мінімальний елемент.

Операції над деревом ван Емде Боаса

- *Пошук попередника* PREDECESSOR.

```
vEB-TREE-PREDECESSOR(V, x)
if V.u == 2
    if x == 1 и V.min == 0
        return 0
    else return NIL
elseif V.max ≠ NIL и x > V.max
    return V.max
else min-low = vEB-TREE-MINIMUM(V.cluster[high(x)])
    if min-low ≠ NIL и low(x) > min-low
        offset = vEB-TREE-PREDECESSOR(V.cluster[high(x)], low(x))
        return index(high(x), offset)
    else pred-cluster = vEB-TREE-PREDECESSOR(V.summary, high(x))
        if pred-cluster == NIL
            if V.min ≠ NIL и x > V.min
                return V.min
            else return NIL
        else offset = vEB-TREE-MAXIMUM(V.cluster[pred-cluster])
            return index(pred-cluster, offset)
```

Операції над деревом ван Емде Боаса

Пошук попередника PREDECESSOR (далі)

• Процедура симетрична пошуку наступника.

• Однак слід врахувати факт, що мінімальний елемент не продубльований у кластері.

• Тому з’являється окреме порівняння зі значенням *min* (рядок 13).

• Обидві процедури можуть здійснити лише один рекурсивний виклик себе (або пошук по кластеру, або по резюме).

• Оскільки пошук мінімуму та максимуму відбувається за O(1), це дає загальний час O(log log *u*).

Операції над деревом ван Емде Боаса

- *Вставка елемента* INSERT.

```
vEB-EMPTY-TREE-INSERT(V, x)
V.min = x
V.max = x
```

```
vEB-TREE-INSERT(V, x)
if V.min == NIL
    vEB-EMPTY-TREE-INSERT(V, x)
else if x < V.min
        Обменять x с V.min
    if V.u > 2
        if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
            vEB-TREE-INSERT(V.summary, high(x))
            vEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
        else vEB-TREE-INSERT(V.cluster[high(x)], low(x))
    if x > V.max
        V.max = x
```

Операції над деревом ван Емде Боаса

Вставка елемента INSERT (далі)

- Вводиться допоміжна процедура вставки в порожнє vEB-дерево.
- Якщо x менший за *min* (рядок 3), відбувається їх обмін значеннями і колишній мінімальний елемент вставляється в один з кластерів.
- Якщо кластер, куди треба помістити елемент x, порожній (рядок 6), вставка туди елементарна, також номер елемента вставляється в резюме.
- У випадку непорожнього кластера його номер вже міститься в резюме, достатньо провести вставку елемента.
- Нарешті, за потреби оновлюється значення *max*.
- Рекурсивний виклик можливий лише один (відбудеться вставка або до резюме, або до кластера).

Операції над деревом ван Емде Боаса

- *Видалення елемента* DELETE.

```
vEB-TREE-DELETE(V, x)
if V.min == V.max
    V.min = NIL
    V.max = NIL
elseif V.u == 2
    if x == 0
        V.min = 1
    else V.min = 0
    V.max = V.min
else if x == V.min
        first-cluster = vEB-TREE-MINIMUM(V.summary)
        x = index(first-cluster,
            vEB-TREE-MINIMUM(V.cluster[first-cluster]))
        V.min = x
    vEB-TREE-DELETE(V.cluster[high(x)], low(x))
    if vEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
        vEB-TREE-DELETE(V.summary, high(x))
        if x == V.max
            summary-max = vEB-TREE-MAXIMUM(V.summary)
            if summary-max == NIL
                V.max = V.min
            else V.max = index(summary-max,
                        vEB-TREE-MAXIMUM(V.cluster[summary-max]))
    elseif x == V.max
        V.max = index(high(x),
            vEB-TREE-MAXIMUM(V.cluster[high(x)]))
```

Операції над деревом ван Емде Боаса

Видалення елемента DELETE(далі)

- Прості випадки: vEB-дерево містить єдиний елемент чи маємо базовий випадок.
- Якщо треба видалити поточне значення *min* (рядок 9), спочатку маємо знайти нове мінімальне значення у vEB і видалити його з кластера.
- В рядку 13 значення *x* видаляється з його кластера.
- Якщо внаслідок видалення елемента кластер стає порожнім (рядок 14), потрібно видалити його номер з резюме (рядок 15). При цьому може знадобитися оновлення *max*, в тому числі з урахуванням значення *min* у випадку всіх порожніх кластерів (рядок 18).
- Якщо кластер після видалення елемента не став порожнім, зміни в резюме не потрібні, але може знадобитися корегування *max* (рядок 21).

Переваги та недоліки vEB-дерев

Переваги

• Висока швидкодія (O(log log *u*)).

• Час роботи операцій не залежить від кількості елементів, що зберігаються.

Використання

• Сортування *n* цілочисельних ключів за час $O(n \log_2 \log_2 u)$ – швидше, ніж порозрядне сортування (radix sort).

• Реалізація купи в алгоритмі Дейкстри побудови найкоротшого шляху в графі.

Переваги та недоліки vEB-дерев

Недоліки

- Великі значення констант в оцінках часу виконання.
- Дозволяють працювати лише з цілими невід’ємними числами.
- Вимагають дуже багато пам’яті ($\Theta(2^{\log u})$), що робить їх непопулярними на практиці. Для великих дерев існують оптимізовані модифікації з меншими затратами пам’яті.