

ЗМІСТ:

- **ЛЕКЦІЯ 1 (2ст.)**
 - Типові операції на динамічних множинах (2ст.)
 - Елементарні структури даних(2ст.)
 - Реалізація вказівників і об'єктів(3ст.)
 - Управління пам'яттю(4ст.)
 - Представлення дерев з коренем(4ст.)
 - Хешування і хеш-таблиці(5ст.)
 - Хеш-таблиці(6ст.)
 - Хеш-функції(7ст.)
 - Універсальне Хешування(7ст.)
 - Ідеальне Хешування(8ст.)
- **ЛЕКЦІЯ 2 (9ст.)**
 - Дерево пошуку(9ст.)
 - Бінарні дерева пошуку(10ст.)
 - Прошиті, збалансовані, червоно-чорні дерева(14ст.)
- **ЛЕКЦІЯ 3 (22ст.)**
 - АВЛ-дерева(22 ст.)
 - АА-дерева(23 ст.)
 - В-дерева(26ст.)
 - Розширювані дерева(27ст.)
 - Декартові дерева(28ст.)
 - Дерево порядкової статистики(32 ст.)
- **Персистентні динамічні множини(33ст.)**
- **ЛЕКЦІЯ 4 (34ст.)**
 - В-дерева(34ст.)
 - Структури даних у вторинній пам'яті(34ст.)
 - Означення В-дерева(35ст.)
 - Висота В-дерева(36ст.)

- Основні операції над В-деревами(36ст.)
 - В+-дерева(40 ст.)
 - 2-3-4-дерева(40 ст.)
- ЛЕКЦІЯ 5 (41ст.)
 - Піраміди злиття(41ст.)
 - Біноміальна піраміда(43ст.)
 - Амортизаційний аналіз(47ст.)
 - Груповий аналіз(47ст.)
 - Метод бухгалтерського обліку(47ст.)
 - Метод Потенціалів(49ст.)
- ЛЕКЦІЯ 6 (49 ст.)
 - Піраміди Фібоначі(49ст.)
 - Піраміди Фібоначчі: функція потенціалу(50ст.)
 - Операції над пірамідами злиття(51ст.)
 - Додаткові операції над пірамідами злиття(57ст.)
 - Оцінка максимальної степені вузла(60ст.)
 - Список з пропусками(61ст.)
- ЛЕКЦІЯ 7 (64 ст.)
 - Реалізації черги з пріоритетами(64ст.)
 - Дерева ван Емде Боаса(65ст.)
 - Попередні підходи(65ст.)
 - Операції над деревом ван Емде Боаса(76ст.)
 - Протоструктури ван Емде Боаса(67ст.)
 - Операції над proto-vEB-структурями(72ст.)
 - Переваги та недоліки vEB-дерев(81ст.)

ЛЕКЦІЯ 1

Типові операції на динамічних множинах

- **SEARCH(S,k):** запит повертає вказівник на елемент x заданої множини S , для якого $\text{key}[x]=k$ або NIL , якщо такого елемента в S немає.
- **INSERT(S,x):** модифікуюча операція, що поповнює множину S одним елементом, на який вказує x ; вважають, що всі поля x , необхідні для реалізації множини вже попередньо ініціалізовані.

- **DELETE(S,x):** модифікуюча операція, що видаляє з множини S елемент, на який вказує x .
- **MINIMUM(S):** запит до цілком впорядкованої множини S , що повертає вказівник на елемент з найменшим ключем.
- **MAXIMUM(S):** запит до цілком впорядкованої множини S , що повертає вказівник на елемент з найбільшим ключем.
- **SUCCESSOR(S,x):** запит до цілком впорядкованої множини S , що повертає вказівник на елемент множини, ключ якого є найближчим більшим сусідом ключа елементу x ; якщо x – максимальний елемент, повертається NIL.
- **PREDECESSOR(S,x):** запит до цілком впорядкованої множини S , що повертає вказівник на елемент множини, ключ якого є найближчим меншим сусідом ключа елементу x ; якщо x – мінімальний елемент, повертається NIL.

Елементарні структури даних.

- **Стеки**
 - Стратегія «останній прийшов – перший вийшов» (LIFO)
 - PUSH: поміщення елементу в вершину стеку
 - POP: зняття елементу з вершини стеку
 - Всі операції виконуються за час $O(1)$
- **Черги**
 - Стратегія «перший прийшов – перший вийшов» (FIFO)
 - ENQUEUE: поміщення у хвіст черги
 - DEQUEUE : зняття елементу з голови черги
- **Деки**
 - Дек (deque) – це черга з двостороннім доступом.
 - Допустимі операції вставки і видалення елементів з обох кінців.
 - Англійська назва deque приходить від слів: написання "d-e-que" як скорочення "double-ended queue" і вимова як у "deck" через схожість з колодою карт, звідки можна здавати як зверху, так і знизу.
 - Всі операції виконуються за час $O(1)$.
- **Зв'язані списки**
 - Зв'язаний список – структура даних, в якій елементи розташовані в лінійному порядку, що визначається вказівниками на кожен об'єкт
 - Однобічно зв'язаний список: поле-ключ key та вказівник на наступний елемент $next$.
 - Двобічно зв'язаний список: поле-ключ key та вказівники на наступний ($next$) та попередній ($prev$) елементи.
 - Кільцевий список: перший та останній елементи зв'язані: $next$ останнього елемента вказує на голову, а $prev$ першого на хвіст.
 - Відсортований список: лінійний порядок списку відповідає лінійному порядку його ключів: мінімальний елемент міститься в голові, максимальний – в хвості.
 - *Пошук в списку L за ключем* В найгіршому випадку час $\Theta(n)$ для списку в n елементів.

- Вставка в голову списку L . Час роботи $O(1)$.
- Видалення за вказівником. Час роботи $O(1)$, але при видаленні за ключем спочатку треба викликати LIST_SEARCH (найгірший час $\Theta(n)$).
- Обмежувач (sentinel) – фіктивний об'єкт, що спрощує обробку граничних умов.

Реалізація вказівників і об'єктів

Представлення об'єктів кількома масивами

- Кожному полю відповідає свій масив.
- Для представлення двозв'язного списку потрібно три масиви: *key* для збереження ключів, *next* та *prev* для вказівників на наступний та попередній елементи. В ролі вказівників – індекси елементів, NIL позначають числом, яке не може бути індексом масиву, змінна L зберігає індекс голови.

Представлення об'єктів одним масивом

- Об'єкт займає неперервну ділянку пам'яті.
- Вказівником є індекс першої комірки пам'яті, де починається об'єкт.
- Індекс інших полів визначається за величиною зміщення.
- Таке представлення більш гнучке, оскільки дозволяє в одному масиві зберігати об'єкти різної довжини. Але одночас управління такими масивами є більш складною задачею.

Управління пам'яттю

- Використовуються масиви довжиною m . В певний момент часу динамічна множина містить $n \leq m$ елементів. Тому $(m-n)$ елементів вільні, їх можна використати в майбутньому для представлення нових елементів множини.
- Вільні елементи зберігаються в списку вільних позицій
 - однозв'язному списку, що використовує масив *next*. Індекс голови зберігає глобальна змінна *free*.
- Список вільних позицій – це стек: під кожен новий об'єкт виділяється остання звільнена пам'ять.
- Часто один список вільних позицій обслуговує декілька зв'язаних списків, що зберігаються в одному масиві:

<i>free</i>	10	1	2	3	4	5	6	7	8	9	10
<i>L₂</i>	9	5	/	6	8	/	2	1	/	7	4
<i>L₁</i>	3	<i>key</i>	<i>k₁</i>	<i>k₂</i>	<i>k₃</i>		<i>k₅</i>	<i>k₆</i>	<i>k₇</i>		<i>k₉</i>
		<i>next</i>	5	/	6	8	/	2	1	/	7
		<i>prev</i>	7	6	/		1	3	9		/

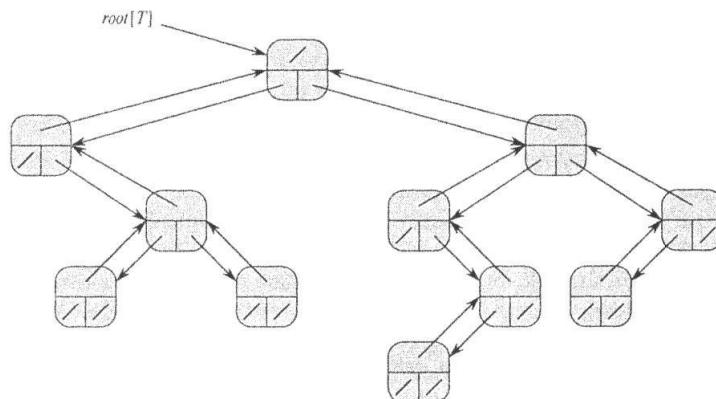
- Час виконання обох процедур $O(1)$.
- Їх можна модифікувати для роботи з довільним набором однорідних об'єктів, щоб тільки одне з полів працювало в якості поля $next$ списку вільних позицій.

Представлення дерев з коренем

- В найзагальнішому випадку (*вільним*) деревом називають зв'язний неорієнтований ацикличний граф.
- Але для широкого практичного застосування така структура має бути більш структурованою: вводяться напрямленість та (можливо) впорядкування.
- *Напрямленість*: наявність кореня.
- *Впорядкування*: піддерева-сини однієї вершини мають певний порядок

Бінарні дерева:

- поле-ключ key ;
- вказівники на батька, лівого та правого синів p , $left$, $right$;
- x – корінь: $p[x]=NIL$;
- x – лист: $left[x]=right[x]=NIL$;
- $root[T]$ – вказівник на корінь дерева T ;
- $root[T] = NIL$ – порожнє дерево.



Довільна кількість вузлів (через бінарне дерево):

- представлення з лівим дочірнім і правим сестринським вузлами;
- для дерева з n вузлів потрібно $O(n)$ пам'яті;
- поле $left_child[x]$ зберігає вказівник на найлівішого сина x ; $left_child[x]=NIL$ – лист;
- поле $right_sibling[x]$ зберігає вказівник на правого брата; $right_sibling[x]=NIL$ – x є найправішим сином.

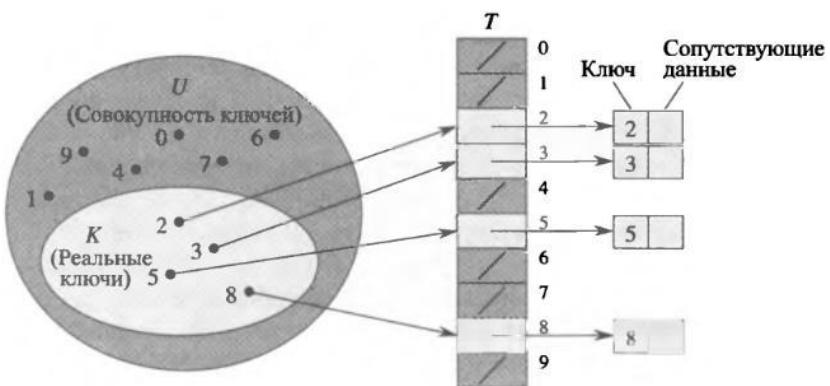
Хешування і хеш-таблиці

- Ефективна структура даних для реалізації словників.
- Узагальнення звичайного масиву.
- В середньому всі базові операції словника вимагають час $O(1)$:

- пошук,
- вставка,
- видалення.

Таблиці з прямою адресацією

- Кожен елемент множини має ключ з множини $U = \{0, 1, \dots, m-1\}$, де m невелике.
- Всі елементи мають різні ключі.
- Використовується масив $T[0..m-1]$, кожна позиція (комірка) якого відповідає ключу з простору ключів U .
- Комірка k вказує на елемент множини з ключем k .
- $T[k] = \text{NIL}$: в множині елемент з ключем k відсутній.
- Іноді елементи зберігаються прямо в таблиці, що економить пам'ять. При цьому потрібний інший механізм позначення порожніх комірок.
- В середньому всі базові операції словника вимагають час $O(1)$



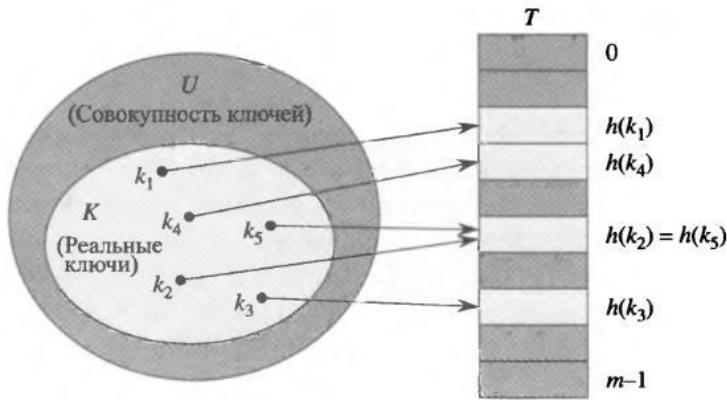
DIRECT_ADDRESS_SEARCH(T, k)
return $T[k]$

DIRECT_ADDRESS_INSERT(T, x)
 $T[\text{key}[x]] \leftarrow x$

DIRECT_ADDRESS_DELETE(T, x)
 $T[\text{key}[x]] \leftarrow \text{NIL}$

- Кожна з операцій виконується за час $O(1)$.
- Кількість реально збережених ключів може бути мала відносно простору можливих ключів U або кількість елементів в U завелика.
- При хешуванні елемент з ключем k зберігатиметься в комірці $h(k)$: елемент з ключем k хешується в комірку $h(k)$. Величина $h(k)$ – хеш-значення ключа k .
- Хеш-функція h відображає простір ключів U на комірки хеш-таблиці $T[0..m-1]$: $h : U \rightarrow \{0, 1, \dots, m-1\}$.
- Мета хеш-функції – зменшити робочий діапазон масиву з $|U|$ до m значень.
- Хеш-функція детермінована: для однакових k має давати те саме хеш-значення $h(k)$.

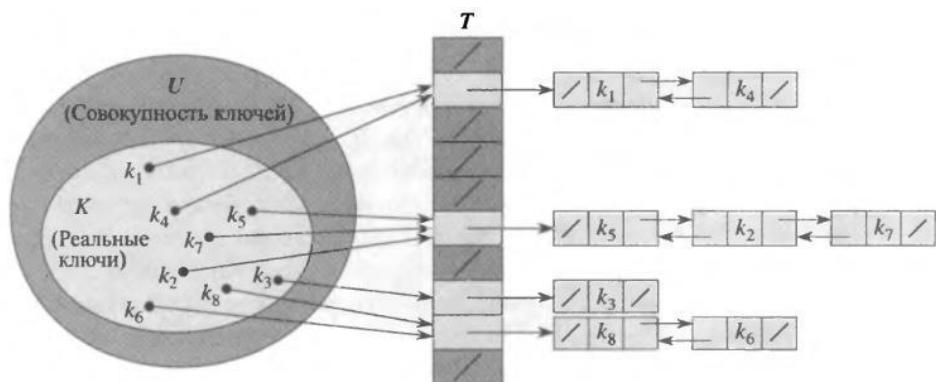
Хеш-таблиці



- Колізія: коли два ключі потрапили в одну комірку.
- Колізії будуть існувати «за побудовою», однак хороша хеш-функція може мінімізувати їх кількість.

Розв'язання колізій за допомогою ланцюжків

- Елементи з одинаковими хеш-значеннями організовуються в список



- CHAINED_HASH_INSERT(T,x): вставити x в голову списку $T[h(key[x])]$.
- CHAINED_SEARCH(T,k): пошук елемента з ключем k в списку $T[h(k)]$.
- CHAINED_HASH_DELETE(T,x): видалення x зі списку $T[h(key[x])]$

Хеш-функції

- Хороша хеш-функція має рівномірно поміщати ключ в одну з m комірок незалежно від хешування інших ключів.
- Функція має бути підібрана так, щоб не корелювала з можливими закономірностями вхідних даних.
- Для більшості хеш-функцій простір ключів представляється множиною натуральних чисел (або може бути певним чином так проінтерпретований).

Метод поділу

- Хеш-функція має вигляд $h(k) = k \bmod m$.
- Деякі значення m будуть невдалими.
- Зокрема, значеннями m не беруть степені двійки.
- Хороші результати дають значення m , що є простими числами, далекими від степені 2.

Метод множення

1. Ключ k множиться на деяку константу $0 < A < 1$ і береться дробова частина.
2. Отримане значення домножується на m і відкидається $h(k) = \lfloor m(kA \bmod 1) \rfloor$
- Тепер значення m стає некритичним. Для зручності реалізації за m можна взяти степінь 2.

Універсальне хешування

- Будь-яка фіксована хеш-функція має ризик стати вразливою у випадку підбору таких значень, які будуть хешуватися в одну комірку, призводячи до лінійного часу вибірки.
- Вихід – довільний вибір хеш-функції з деякої множини.
- Нехай H – скінченна множина хеш-функцій з простору ключів U в діапазон $\{0, 1, \dots, m-1\}$. Така множина *універсальна*, якщо для довільної пари хеш-ключів $k, l \in U$ кількість функцій $h \in H$, для яких $h(k) = h(l)$, не перевищує $|H|/m$.
- Тобто ймовірність колізії не перевищує ймовірності співпадіння двох серед m різних елементів.
- Універсальні хеш-функції мають хорошу ефективність.

Відкрита адресація

- Всі елементи зберігаються безпосередньо в хештаблиці: записи таблиці містять або елемент динамічної множини, або значення NIL.
- Шукаючи елемент, систематично перевіряються комірки, поки він не знайдеться або не впевнимося в його відсутності.
- Можлива ситуація заповнення таблиці і неможливості вставки нового елемента.
- Вказівники відсутні. Обчислюється послідовність комірок, які переглядаються.
- При пошуку і вставці ключа послідовно перевіряються комірки хеш-таблиці, поки не знаходиться порожня комірка, куди вставляється новий елемент (або комірка з шуканим елементом).
- Послідовність перевірки залежить від *конкретного ключа*.

Лінійне дослідження

Візьмемо звичайну хеш-функцію $h': U \rightarrow \{0, 1, \dots, m-1\}$. Назвемо її *допоміжною хеш-функцією*. Для методу задамо хеш-функцію так: $h(k, i) = (h'(k) + i) \bmod m$, де i приймає значення від 0 до $m-1$. Проблема первинної кластеризації: утворюються довгі ланцюжки зайнятих комірок. Метод дає m перестановок.

Приклад виникнення кластеризації при лінійному дослідженні (показані перші 6 комірок).



Допоміжна хеш-функція має вигляд $h'(k) = k \bmod 10$.

Квадратичне дослідження

- Хеш-функція задається так: $h(k,i) = (h'(k) + c_1i + c_2i^2) \bmod m$, де h' – допоміжна хеш-функція, $c_1, c_2 \neq 0$ – допоміжні константи, а i приймає значення від 0 до $m-1$.
- Необхідно вибирати значення c_1, c_2 та m .
- Проблема вторинної кластеризації: якщо два ключі мають одну й ту саму початкову позицію, то послідовності дослідження також співпадатимуть.
- Метод дає m перестановок.

Подвійне хешування

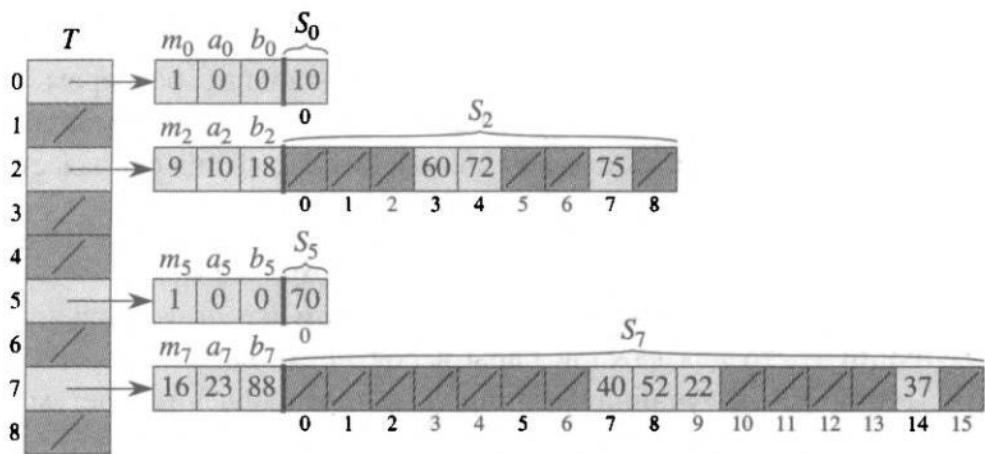
- Один з найкращих способів відкритої адресації.
 - Метод дає m^2 перестановок.
 - Хеш-функція задається так: $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$, де h_1, h_2 – допоміжні хеш-функції.
 - Таким чином, початкова позиція і крок обчислюються окремо.
 - Щоб обійти всю таблицю необхідно, щоб значення $h_2(k)$ було взаємно простим з розміром хеш-таблиці m (m – степінь 2 і тільки непарні значення $h_2(k)$ або m просте, а $h_2(k)$ повертає значення, менші за m).
- Приклад вибору хеш-функцій:
- $h_1(k) = k \bmod m$
- $h_2(k) = 1 + (k \bmod m')$, де m просте, а m' – трохи менше за m (як варіант, $m' = m-1$)
- Приклад вставки ключа 14 в таблицю
- $h_1(k) = k \bmod 13$,
- $h_2(k) = 1 + (k \bmod 11)$.

Ідеальне хешування

- $O(1)$ звертань до пам'яті в найгіршому випадку.
- Множина ключів статична – не змінюється після збереження в таблицю.
- Перший рівень хешування: n ключів хешуються в m комірок за допомогою універсальної хеш-функції h .
- Другий рівень хешування: для кожної комірки своя вторинна хеш-таблиця зі своєю універсальною хешфункцією, вибраною так, щоб уникнути колізій; її розмір – квадрат кількості ключів, захешованих в комірку.
- Очікувана загальна пам'ять під таку структуру $O(n)$.

Приклад використання ідеального хешування. Множина ключів

$$K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\} \quad h(k) = ((ak + b) \bmod p) \bmod m, \text{ де } a=3, b=42, p=101, m=9$$



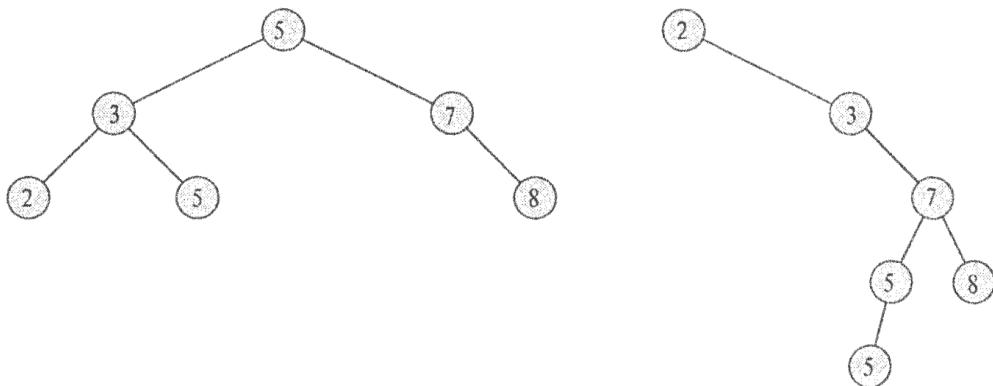
ЛЕКЦІЯ 2

Дерево пошуку

- Структура даних, що підтримує ряд операцій з динамічними множинами:
 - пошук елемента;
 - пошук мінімального та максимального значення;
 - пошук попереднього і наступного елемента;
 - вставка та видалення елемента.
- Набір операцій дозволяє використовувати дерево пошуку для реалізації як словника, так і черги з пріоритетами.
- Основні операції виконуються за час, пропорційний висоті дерева.

Бінарні дерева пошуку

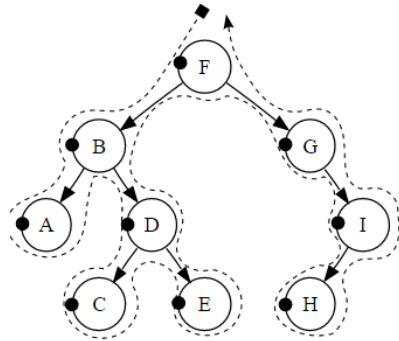
- Поле-ключ *key*.
- Вказівники на батька, лівого та правого синів *p*, *left*, *right*.
- *x* – корінь: *p[x]=NIL*.
- *x* – лист: *left[x]=right[x]=NIL*.
- Для кожного вузла *x* виконується *властивість бінарного дерева пошуку*:
 - якщо вузол *y* знаходитьться в лівому піддереві вузла *x*, то *key[y] ≤ key[x]*;
 - якщо вузол *y* знаходитьться в правому піддереві вузла *x*, то *key[x] ≤ key[y]*.



Обходи дерев

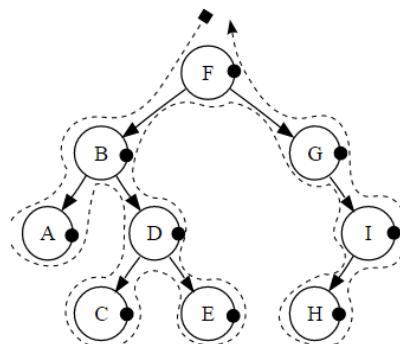
- Прямий порядок (preorder tree walk): корінь, ліве піддерево, праве піддерево.

F, B, A, D, C, E, G, I, H



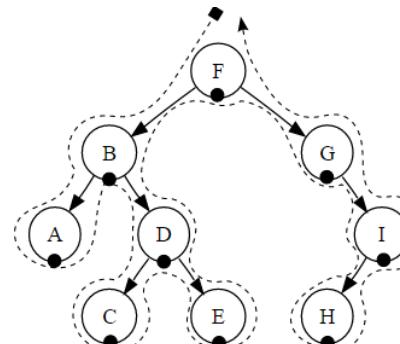
- Обернений порядок (postorder tree walk): ліве піддерево, праве піддерево, корінь.

A, C, E, D, B, H, I, G, F



- Симетричний порядок (inorder tree walk): ліве піддерево, корінь, праве піддерево.

A, B, C, D, E, F, G, H, I



INORDER_TREE_WALK(x)

```

1  if  $x \neq \text{NIL}$ 
2    then INORDER_TREE_WALK( $\text{left}[x]$ )
3      print  $\text{key}[x]$ 
4      INORDER_TREE_WALK( $\text{right}[x]$ )

```

- Коректність алгоритму INORDER_TREE_WALK($\text{root}[T]$) безпосередньо випливає з властивості бінарного дерева пошуку.

- Після початкового виклику процедура викликається для кожного вузла дерева рівно двічі: для лівого та правого сина, тому дерево обходиться за лінійний час: **Теорема.** Якщо x – корінь піддерева з n вузлами, то INORDER_TREE_WALK(x) виконується за час $\Theta(n)$.

Доведення. Нижня границя $\Omega(n)=n$ – бо відвідуються всі n вершини. Нехай $T(n)$ – час виконання процедури INORDER_TREE_WALK для параметра – кореня дерева з n вузлами. Якщо параметр порожній, покладемо $T(0)=c$, де c – деяка додатна константа (час перевірки $x \neq \text{NIL}$).

При $n > 0$ вважатимемо, що процедура викличеться перший раз для піддерева з k вузлами, а другий – для піддерева з $(n-k-1)$ вузлами. Тоді загальний час $T(n) = T(k) + T(n-k-1) + d$, де d – час роботи за винятком рекурсивних викликів.

Покажемо методом підстановок, що $T(n) = O(n)$. Доведемо $T(n) = (c+d) \cdot n + c$.

При $n=0$: $T(0) = (c+d) \cdot 0 + c = c$.

При $n > 0$:

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d = \\ &= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d = \\ &= (c+d)n + c - (c+d) + c + d = (c+d)n + c, \end{aligned}$$

що і треба довести.

Пошук елемента

Хвостову рекурсію можна розгорнути в цикл

TREE_SEARCH(x, k)

```
1  if  $x = \text{NIL}$  или  $k = \text{key}[x]$ 
2    then return  $x$ 
3  if  $k < \text{key}[x]$ 
4    then return TREE_SEARCH( $\text{left}[x], k$ )
5    else return TREE_SEARCH( $\text{right}[x], k$ )
```

ITERATIVE_TREE_SEARCH(x, k)

```
1  while  $x \neq \text{NIL}$  и  $k \neq \text{key}[x]$ 
2    do if  $k < \text{key}[x]$ 
3      then  $x \leftarrow \text{left}[x]$ 
4      else  $x \leftarrow \text{right}[x]$ 
5  return  $x$ 
```

Відвідані вузли утворюють шлях від кореня донизу, тому час виконання $O(h)$, де h – висота дерева.

Пошук мінімуму і максимуму

TREE_MINIMUM(x)

```
1  while  $\text{left}[x] \neq \text{NIL}$ 
2    do  $x \leftarrow \text{left}[x]$ 
3  return  $x$ 
```

TREE_MAXIMUM(x)

```
1  while  $\text{right}[x] \neq \text{NIL}$ 
2    do  $x \leftarrow \text{right}[x]$ 
3  return  $x$ 
```

- Властивість бінарного дерева пошуку гарантує коректність обидвох процедур.
- Відвідані вузли утворюють шлях від кореня донизу, тому час виконання $O(h)$, де h – висота дерева.

Попередній і наступний елементи

```

TREE_SUCCESSOR( $x$ )
1 if  $right[x] \neq \text{NIL}$ 
2   then return TREE_MINIMUM( $right[x]$ )
3    $y \leftarrow p[x]$ 
4   while  $y \neq \text{NIL}$  и  $x = right[y]$ 
5     do  $x \leftarrow y$ 
6      $y \leftarrow p[y]$ 
7   return  $y$ 

```

- Відвідані вузли утворюють шлях від вузла або донизу, або вгору, тому час виконання $O(h)$, де h – висота дерева.
- Процедура пошуку попередника TREE_PREDECESSOR буде симетричною і також матиме час $O(h)$.
- У випадку вузлів з однаковими ключами попереднім і наступним вузлами можна визначити ті, що повертаються процедурами TREE_PREDECESSOR та TREE_SUCCESSOR відповідно.

Теорема. Операції пошуку, визначення мінімального і максимального елемента, а також елементапопередника та наступника в бінарному дереві пошуку висоти h можуть бути виконані за час $O(h)$.

Вставка елемента

Процедура вставляє в дерево T вузол z , в якого $key[z]=v$, $left[z]=\text{NIL}$, $right[z]=\text{NIL}$.

```

TREE_INSERT( $T, z$ )
1    $y \leftarrow \text{NIL}$ 
2    $x \leftarrow \text{root}[T]$ 
3   while  $x \neq \text{NIL}$ 
4     do  $y \leftarrow x$ 
5       if  $key[z] < key[x]$ 
6         then  $x \leftarrow left[x]$ 
7         else  $x \leftarrow right[x]$ 
8    $p[z] \leftarrow y$ 
9   if  $y = \text{NIL}$ 
10    then  $\text{root}[T] \leftarrow z$             $\triangleright$  Дерево  $T$  – пустое
11    else if  $key[z] < key[y]$ 
12      then  $left[y] \leftarrow z$ 
13      else  $right[y] \leftarrow z$ 

```

- Час виконання $O(h)$ для дерева висоти h .

Видалення елемента

```

TREE_DELETE( $T, z$ )
1 if  $left[z] = \text{NIL}$  или  $right[z] = \text{NIL}$ 
2   then  $y \leftarrow z$ 
3   else  $y \leftarrow \text{TREE_SUCCESSOR}(z)$ 
4   if  $left[y] \neq \text{NIL}$ 
5     then  $x \leftarrow left[y]$ 
6     else  $x \leftarrow right[y]$ 
7   if  $x \neq \text{NIL}$ 
8     then  $p[x] \leftarrow p[y]$ 
9   if  $p[y] = \text{NIL}$ 
10    then  $\text{root}[T] \leftarrow x$ 
11    else if  $y = left[p[y]]$ 
12      then  $left[p[y]] \leftarrow x$ 
13      else  $right[p[y]] \leftarrow x$ 
14   if  $y \neq z$ 
15     then  $key[z] \leftarrow key[y]$ 
16     Копирование сопутствующих данных в  $z$ 
17   return  $y$ 

```

Випадок однакових ключів

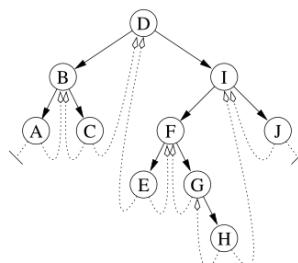
- (Оцініть асимптотичний час роботи процедури TREE_INSERT при вставці п однакових ключів в порожнє дерево.)
- Модифікація алгоритму TREE_INSERT:
 - перед рядком 5 додається перевірка $key[z] = key[x]$;
 - перед рядком 11 – перевірка $key[z] = key[y]$;
 - якщо рівності виконуються, реалізується одна з наступних стратегій (описи для порівняння ключів z та x)

Варіанти стратегій

- У вузлі x зберігається логічний прапорець $b[x]$,
значення якого визначає, вправо чи вліво
вставляється ключ зі значенням x . Значення прапорця
при кожній такій вставці змінюється на протилежне.
- Всі елементи з однаковими ключами зберігаються в
одному вузлі за допомогою списку.
- Дочірній вузол $left[x]$ або $right[x]$ вибирається
випадковим чином.

Прошиті дерева

- При всіх обходах бінарних дерев використовується рекурсія, що призводить до явного чи неявного (залежно від реалізації) використання додаткової пам'яті пропорційно до висоти дерева.
- У випадку незбалансованих дерев це може мати значення.
- Існують способи здійснити обхід без рекурсії і стека.
- Один варіант – "розвертати" вказівники при спуску вниз з відновленням структури дерева на зворотному шляху, використовуючи "розвернуті" вказівники для підйому.
- Інший спосіб – використання прошитих дерев.
- Прошивка дерева прив'язана до конкретного обходу.
- Для спрощення обходу нульові вказівники замінюються вказівниками (нитками) на попередній і/або наступний елементи при обході.
- При цьому у вузлі мають додатково з'явитися бітимаркери характера вказівника (чи це нитка?).
- Так, шукаючи наступника заданого вузла в прошитому дереві, спочатку перевіряємо, чи є його правий вказівник ниткою. Якщо так, переходимо по ній, інакше спускаємося відь правого сина по лівим вказівникам до самого низу.



Збалансовані дерева

- Дерево збалансоване, якщо його висота гарантовано не перевищує $\log n$ для n елементів.
- Для збалансованих дерев час виконання операцій над динамічними множинами навіть в найгіршому випадку $O(\log n)$.
 - AVL-дерева (придумані в 1962 році в СРСР);
 - 2-3 дерева;
 - 2-3-4 дерева;
 - AA-дерева;
 - червоно-чорні дерева (RB, Red-Black);

- списки з пропусками (Skip lists);
 - декартові дерева (Treaps = Tree + Heap).

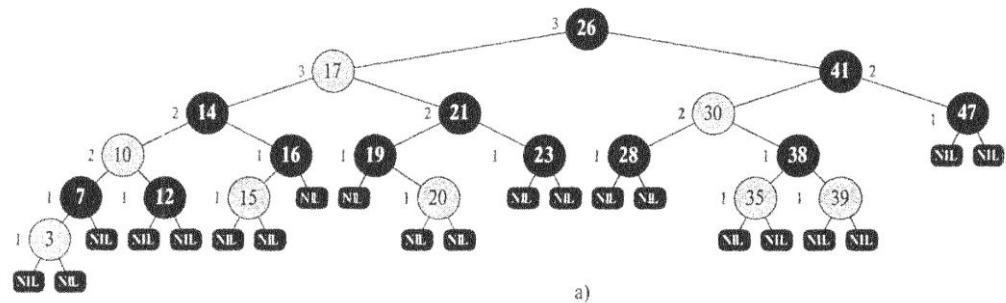
Насправді 2-3 і 2-3-4 дерева є різновидами загальнішої структури – B-дерев.

Червоно-чорні дерева

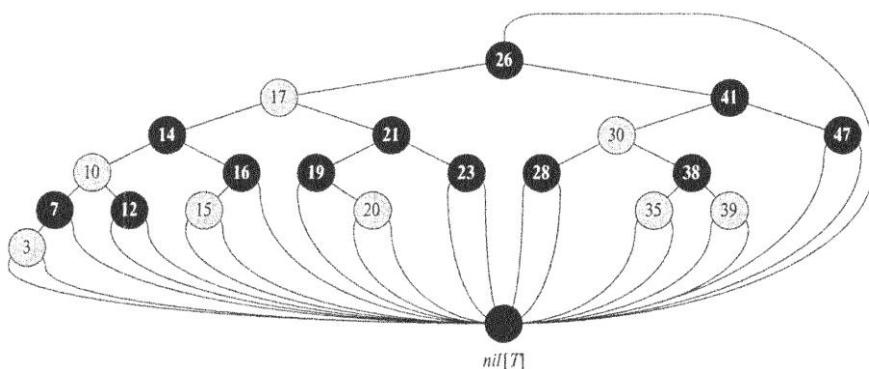
- Бінарне дерево пошуку з додатковим бітом кольору в кожному вузлі: червоний чи чорний.
 - Дерева наближено збалансовані: жоден шлях в червоно-чорному дереві не відрізняється від іншого більше ніж удвічі.
 - Кожен вузол містить поля *key*, *color*, *p*, *left*, *right*.
 - Всі вузли, що містять ключ, – внутрішні.
 - Розглядатимемо значення NIL як вказівники на листя (зовнішні вузли) бінарного дерева пошуку.
 - Бінарне дерево пошуку буде червоно-чорним, якщо задовольнятиме червоно-чорні властивості.

Червоно-чорні властивості

1. Кожен вузол є або червоним, або чорним.
 2. Корінь дерева – чорний.
 3. Кожен лист дерева (NIL) – чорний.
 4. Якщо вузол червоний, то обидва його сини чорні.
 5. Для кожного вузла всі шляхи від нього до листівпотомків містять однакову кількість чорних вузлів.
 - Чорна висота вузла x ($bh(x)$, black-height) – кількість чорних вузлів на шляху від вузла x (не рахуючи його самого) до листів; вона визначається однозначно.
 - Чорна висота дерева – чорна висота його кореня.



- Для зручності роботи з червоно-чорним деревом Т замінимо всі листи одним вузлом-обмежувачем $nil[T]$, що представляє значення NIL.
 - Значення його поля *color* буде чорне (BLACK).

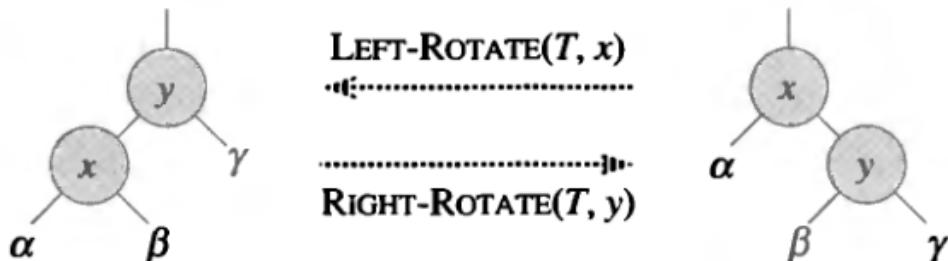


Твердження. Висота червоно-чорного дерева з n внутрішніми вузлами не перевищує $2\lg n + 1$.

Доведення. Спочатку покажемо, що піддерево довільного вузла x містить не менше $2bh(x)-1$ внутрішніх вузлів. Доводимо за індукцією. Нехай висота x дорівнює 0. Тоді вузол є листом ($nil[T]$) і його піддерево містить не менше $2bh(x)-1=20-1=0$ внутрішніх вузлів. Нехай висота x додатна і вузол має два потомки. Чорна висота кожного з потомків $bh(x)$ або $(bh(x)-1)$, залежно від того, має він червоний чи чорний колір відповідно. Оскільки висота потомків менша за висоту самого вузла x , користуємося припущенням індукції: дерево з коренем у вершині x міститиме принаймні $(2bh(x)-1-1)+(2bh(x)-1-1)+1=(2bh(x)-1)$ внутрішніх вузлів. Нехай h – висота дерева. За властивістю 4, як мінімум половина вузлів на шляху від кореня до листа, не враховуючи сам корінь, повинна бути чорними. Тоді чорна висота кореня має бути не меншою за $h/2$, звідси $n \geq 2h/2 - 1$. Переносимо одиницю наліво і логарифмуємо: $\lg(n + 1) \geq h/2$, тобто $h \leq 2\lg(n + 1)$.

- Отже, операції пошуку, взяття мінімуму, максимуму, попереднього та наступного елемента мають час $O(h)$.

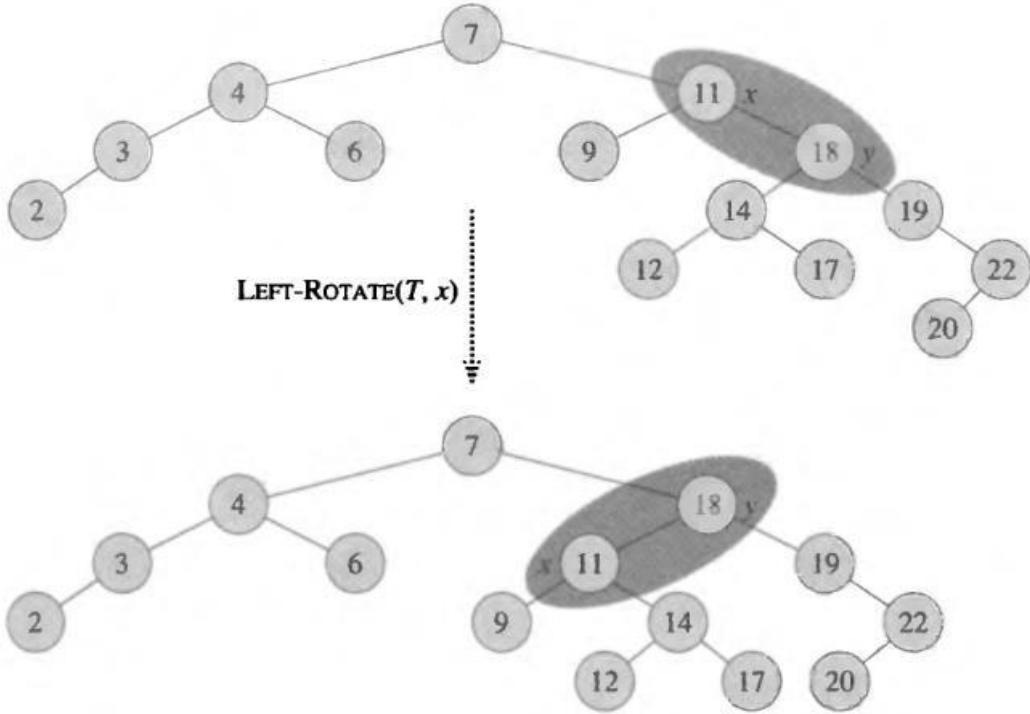
Повороти



LEFT_ROTATE(T, x)

- 1 $y \leftarrow right[x]$ ▷ Устанавливаем y .
- 2 $right[x] \leftarrow left[y]$ ▷ Левое поддерево y становится
- ▷ правым поддеревом x
- 3 **if** $left[y] \neq nil[T]$
- 4 **then** $p[left[y]] \leftarrow x$
- 5 $p[y] \leftarrow p[x]$ ▷ Перенос родителя x в y
- 6 **if** $p[x] = nil[T]$
- 7 **then** $root[T] \leftarrow y$
- 8 **else** **if** $x = left[p[x]]$
- 9 **then** $left[p[x]] \leftarrow y$
- 10 **else** $right[p[x]] \leftarrow y$
- 11 $left[y] \leftarrow x$ ▷ x – левый дочерний y
- 12 $p[x] \leftarrow y$

Приклад виконання лівого повороту:



Властивості дерева пошуку зберігаються.

Вставка вузла

1. Спочатку відбувається звичайна вставка як в бінарнедерево пошуку.
2. Вузол розфарбовується червоним.
3. Викликається допоміжна процедура, що відновлює червоно-чорні властивості (рух знизу вгору).

Прості випадки:

- новий вузол є коренем – перефарбовується в чорний;
- батьківський вузол чорний – властивості не порушені.

Модифікована вставка:

```

RB_INSERT( $T, z$ )
1    $y \leftarrow \text{nil}[T]$ 
2    $x \leftarrow \text{root}[T]$ 
3   while  $x \neq \text{nil}[T]$ 
4       do  $y \leftarrow x$ 
5           if  $\text{key}[z] < \text{key}[x]$ 
6               then  $x \leftarrow \text{left}[x]$ 
7               else  $x \leftarrow \text{right}[x]$ 
8    $p[z] \leftarrow y$ 
9   if  $y = \text{nil}[T]$ 
10      then  $\text{root}[T] \leftarrow z$ 
11      else if  $\text{key}[z] < \text{key}[y]$ 
12          then  $\text{left}[y] \leftarrow z$ 
13          else  $\text{right}[y] \leftarrow z$ 
14    $\text{left}[z] \leftarrow \text{nil}[T]$ 
15    $\text{right}[z] \leftarrow \text{nil}[T]$ 
16    $\text{color}[z] \leftarrow \text{RED}$ 
17   RB_INSERT_FIXUP( $T, z$ )

```

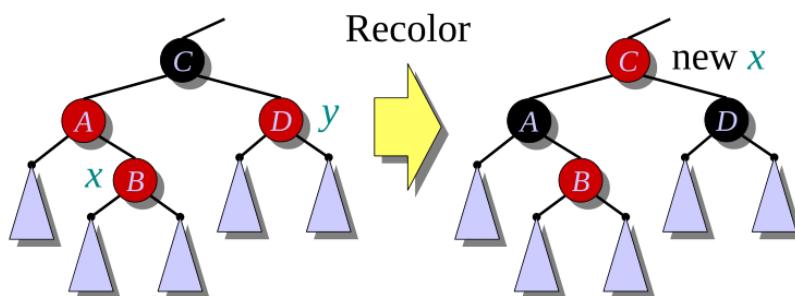
Відновлення властивостей (загалом не більше 2 обертань та $\lg n$ кроків):

```

RB_INSERT_FIXUP( $T, z$ )
1 while  $color[p[z]] = \text{RED}$ 
2   do if  $p[z] = \text{left}[p[p[z]]]$ 
3     then  $y \leftarrow \text{right}[p[p[z]]]$ 
4       if  $color[y] = \text{RED}$ 
5         then  $color[p[z]] \leftarrow \text{BLACK}$            ▷ Случай 1
6            $color[y] \leftarrow \text{BLACK}$            ▷ Случай 1
7            $color[p[p[z]]] \leftarrow \text{RED}$            ▷ Случай 1
8            $z \leftarrow p[p[z]]$            ▷ Случай 1
9         else if  $z = \text{right}[p[z]]$ 
10        then  $z \leftarrow p[z]$            ▷ Случай 2
11          LEFT_ROTATE( $T, z$ )           ▷ Случай 2
12           $color[p[z]] \leftarrow \text{BLACK}$            ▷ Случай 3
13           $color[p[p[z]]] \leftarrow \text{RED}$            ▷ Случай 3
14        RIGHT_ROTATE( $T, p[p[z]]$ )           ▷ Случай 3
15      else (то же, что и в “then”, с заменой
            $left$  на  $right$  и наоборот)
16   $color[\text{root}[T]] \leftarrow \text{BLACK}$ 

```

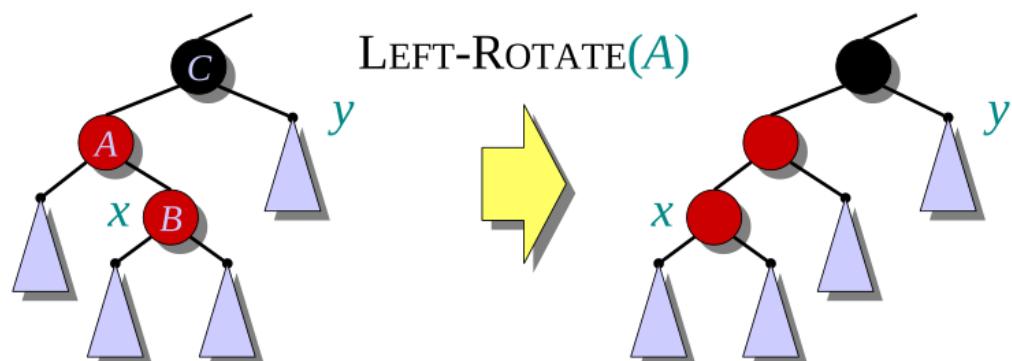
- Випадок 1 (або симетричний відносно А) «Дідусь» чорний, «дядько» у червоний.



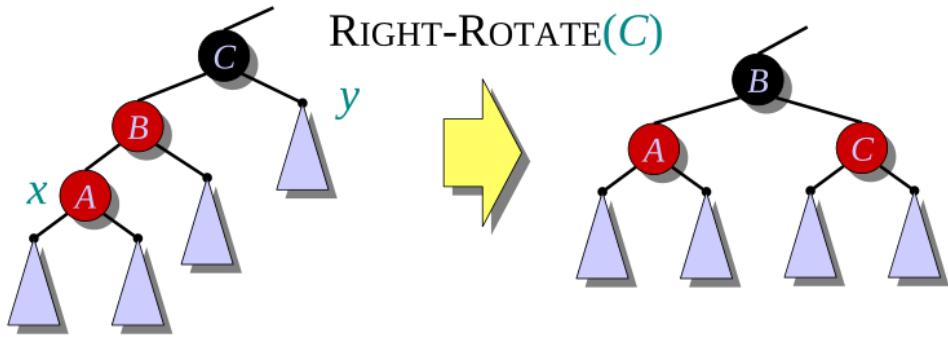
«Батько» і «дядько» стають чорними, «дідусь» червоним. Перевірка продовжується, бо предок С може виявитися червоним.

- Випадок 2 «Дядько» чорний, x є правим потомком.

Лівий поворот відносно А. Зведення до випадку 3.



- Випадок 3 «Дядько» чорний, x є лівим потомком. Правий поворот відносно С та зміна кольорів. Вихід з алгоритму



Видалення вузла

RB_DELETE(T, z)

```

1  if  $left[z] = nil[T]$  или  $right[z] = nil[T]$ 
2    then  $y \leftarrow z$ 
3    else  $y \leftarrow TREE\_SUCCESSOR(z)$ 
4  if  $left[y] \neq nil[T]$ 
5    then  $x \leftarrow left[y]$ 
6    else  $x \leftarrow right[y]$ 
7   $p[x] \leftarrow p[y]$ 
8  if  $p[y] = nil[T]$ 
9    then  $root[T] \leftarrow x$ 
10   else if  $y = left[p[y]]$ 
11     then  $left[p[y]] \leftarrow x$ 
12     else  $right[p[y]] \leftarrow x$ 
13  if  $y \neq z$ 
14    then  $key[z] \leftarrow key[y]$ 
15    Копируем сопутствующие данные  $y$  в  $z$ 
16  if  $color[y] = BLACK$ 
17    then RB_DELETE_FIXUP( $T, x$ )
18  return  $y$ 

```

Можливі три випадки залежно від наявності потомків:

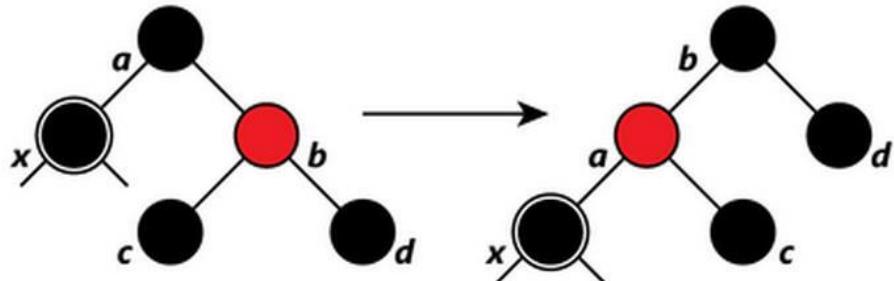
- Якщо у вершини немає дітей, то змінюється вказівник на неї у батька на NIL.
- Якщо у неї тільки один потомок, то у батька робиться посилання на нього замість цієї вершини.
- Якщо ж є обидва потомки, то знаходимо вершину з наступним значенням ключа. У такої вершини немає лівого дитини. Видаляємо вже цю вершину описаним вище способом, скопіювавши її ключ в початкову.

```

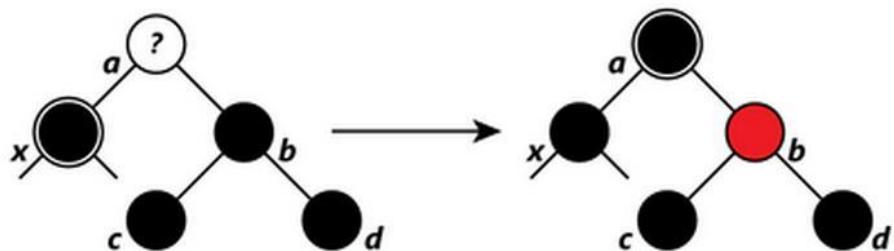
RB_DELETE_FIXUP( $T, x$ )
1  while  $x \neq \text{root}[T]$  и  $\text{color}[x] = \text{BLACK}$ 
2    do if  $x = \text{left}[p[x]]$ 
3      then  $w \leftarrow \text{right}[p[x]]$ 
4        if  $\text{color}[w] = \text{RED}$ 
5          then  $\text{color}[w] \leftarrow \text{BLACK}$   $\triangleright \text{Случай 1}$ 
6             $\text{color}[p[x]] \leftarrow \text{RED}$   $\triangleright \text{Случай 1}$ 
7             $\text{LEFT\_ROTATE}(T, p[x])$   $\triangleright \text{Случай 1}$ 
8             $w \leftarrow \text{right}[p[x]]$   $\triangleright \text{Случай 1}$ 
9        if  $\text{color}[\text{left}[w]] = \text{BLACK}$  и  $\text{color}[\text{right}[w]] = \text{BLACK}$ 
10       then  $\text{color}[w] \leftarrow \text{RED}$   $\triangleright \text{Случай 2}$ 
11        $x \leftarrow p[x]$   $\triangleright \text{Случай 2}$ 
12     else if  $\text{color}[\text{right}[w]] = \text{BLACK}$ 
13       then  $\text{color}[\text{left}[w]] \leftarrow \text{BLACK}$   $\triangleright \text{Случай 3}$ 
14          $\text{color}[w] \leftarrow \text{RED}$   $\triangleright \text{Случай 3}$ 
15          $\text{RIGHT\_ROTATE}(T, w)$   $\triangleright \text{Случай 3}$ 
16          $w \leftarrow \text{right}[p[x]]$   $\triangleright \text{Случай 3}$ 
17          $\text{color}[w] \leftarrow \text{color}[p[x]]$   $\triangleright \text{Случай 4}$ 
18          $\text{color}[p[x]] \leftarrow \text{BLACK}$   $\triangleright \text{Случай 4}$ 
19          $\text{color}[\text{right}[w]] \leftarrow \text{BLACK}$   $\triangleright \text{Случай 4}$ 
20          $\text{LEFT\_ROTATE}(T, p[x])$   $\triangleright \text{Случай 4}$ 
21          $x \leftarrow \text{root}[T]$   $\triangleright \text{Случай 4}$ 
22   else (то же, что и в “then”, с заменой left на right и наоборот)
23    $\text{color}[x] \leftarrow \text{BLACK}$ 

```

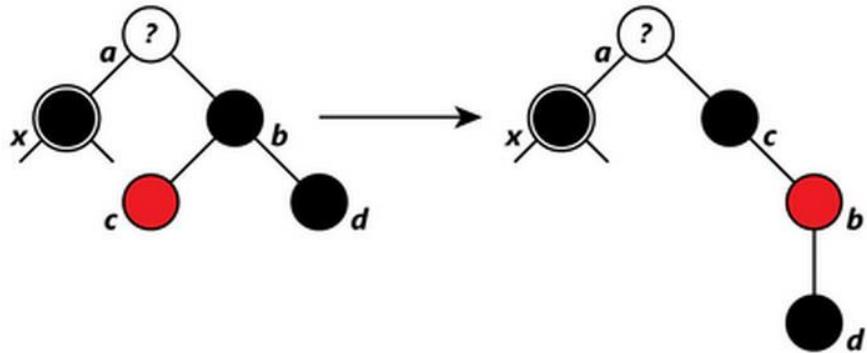
- Випадок 1. «Брат» вершини x червоний. Ліве обертання відносно а та перефарбування. Зведення до випадків 2, 3, 4.



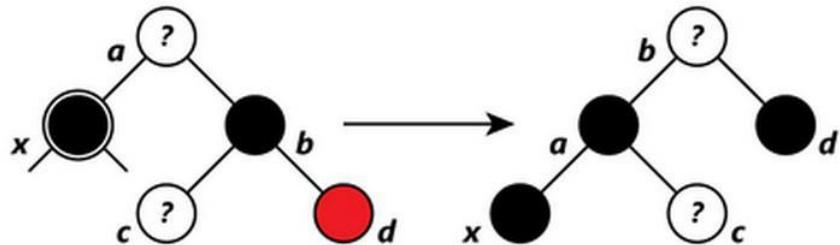
- Випадок 2. «Брат» вершини x чорний. Обидва потомки «брата» чорні. Перефарбовуємо «брата». Далі розглядаємо вузла-батька



- Випадок 3. «Брат» вершини x чорний. Лівий потомок «брата» червоний, правий чорний. Перефарбовуємо «брата» і його лівого сина. Правий поворот відносно b . Зведення до випадку 4.

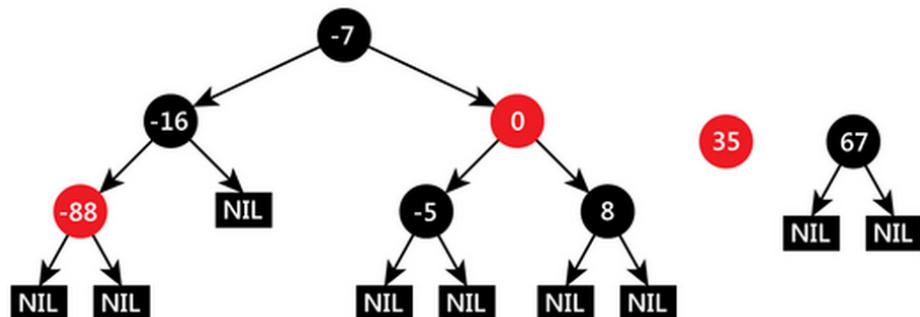


- Випадок 4. «Брат» вершини x чорний. Правий потомок «брата» червоний. Перефарбовуємо і робимо лівий поворот відносно a . Вихід з алгоритму.



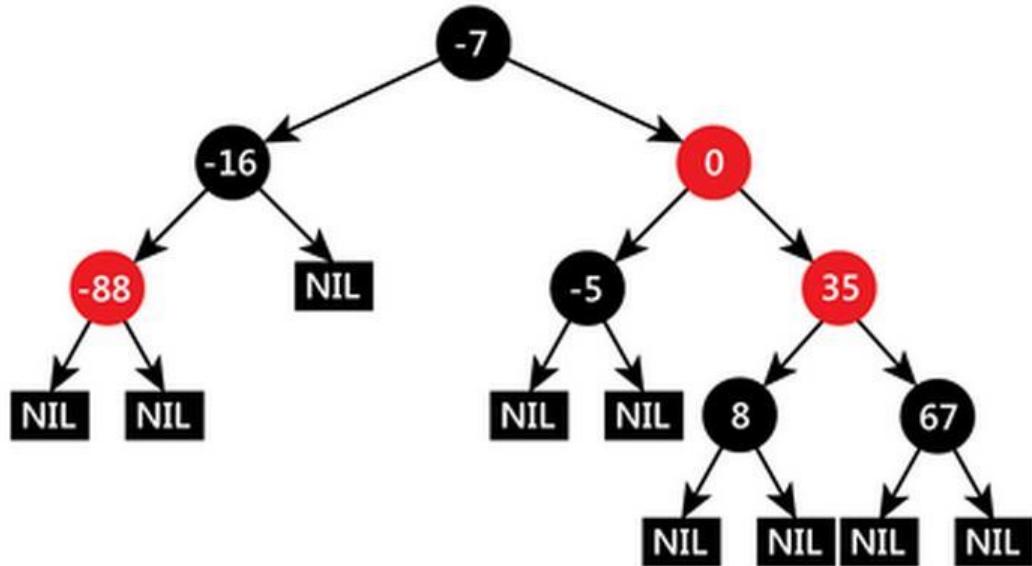
Об'єднання червоно-чорних дерев

- Операція об'єднання застосовується до двох динамічних множин T_1 і T_2 та елемента x такого, що для довільних $x_1 \in T_1$, $x_2 \in T_2$ виконується $key[x_1] \leq key[x] \leq key[x_2]$.
- Результатом операції є множина $T = T_1 \cup \{x\} \cup T_2$.
- Припустимо, що $bh(T_1) \geq bh(T_2)$. В дереві T_1 серед чорних вершин з чорною висотою $bh(T_2)$ шукаємо вузол у з найбільшим ключем.
- Нехай T у – піддерево з коренем u . Об'єднуємо T_u з T_2 в одне дерево з червоним коренем x . Тобто батьком x стає колишній батько u .
- Відновлюємо властивості червоно-чорного дерева, як при вставці вузла.
- Всі операції виконуються за час $O(\lg n)$.



Чорна висота лівого дерева більша. Шукаємо в ньому найбільшу чорну вершину з чорною висотою 1. Це буде вузол 8.

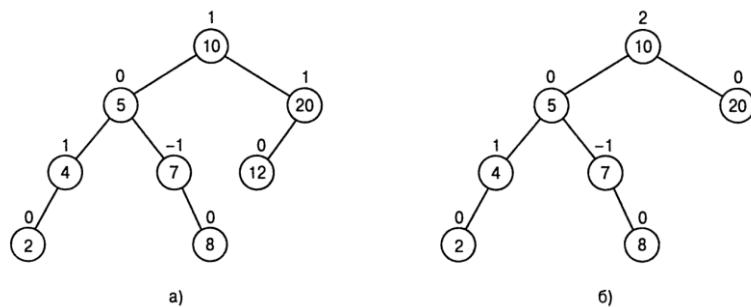
Об'єднаємо дерева і вершину:



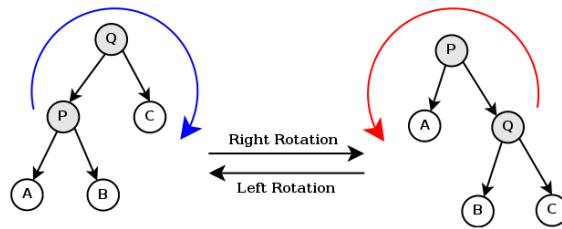
ЛЕКЦІЯ 3

АВЛ-дерева

- Адельсон-Вельський Георгій Максимович, Ландіс Євген Михайлович (1962 р.)
- Бінарне дерево пошуку, збалансоване по висоті: для кожної вершини висота її двох піддерев відрізняється не більше ніж на 1.
- Висота порожнього дерева рівна -1.
- Вводиться додаткове поле для значення балансу вершини (або висоти піддерева).

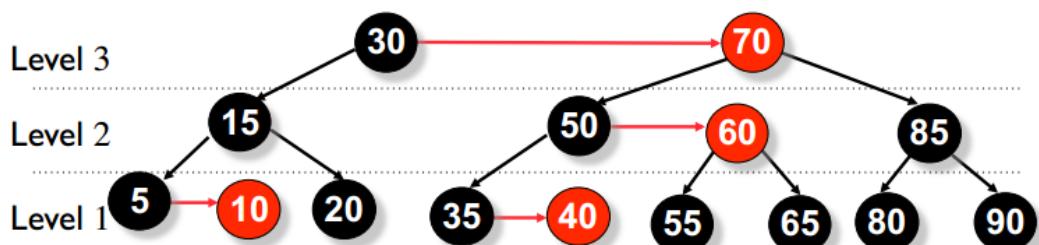
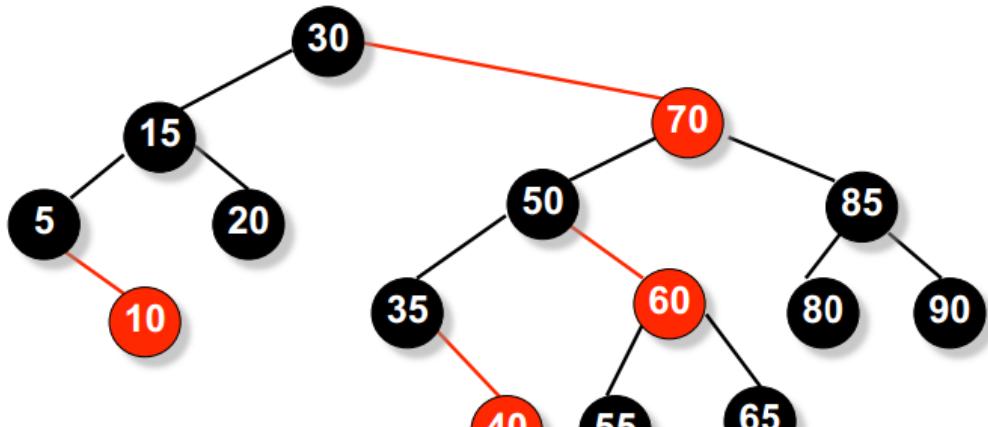


- Висота АВЛ-дерева з n вузлів обмежена знизу $\log_2 n$ і не перевищує $\log \varphi (5(n + 2)) - 2 \approx 1.44 \log_2 n + 2 - 0.328$, де φ – золотий перетин.
- Експериментально показано, що середня висота АВЛдерева з n вузлів для немаленьких n складає $1.011 \log_2 n + 0.1$.
- Операції вставки та видалення вузла займають час $O(\lg n)$, причому при відновленні балансу може відбутися до $O(\lg n)$ обертань.

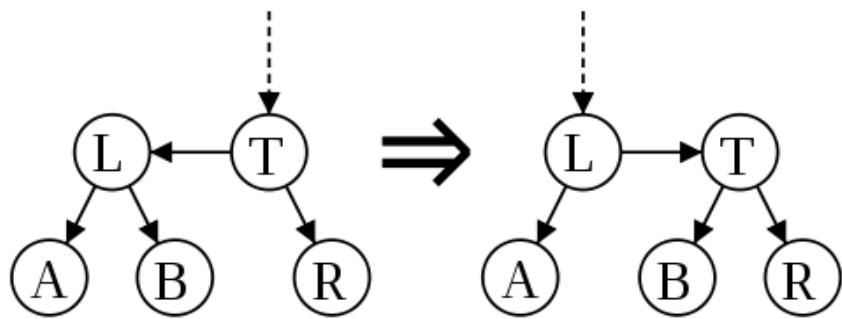


AA-дерева

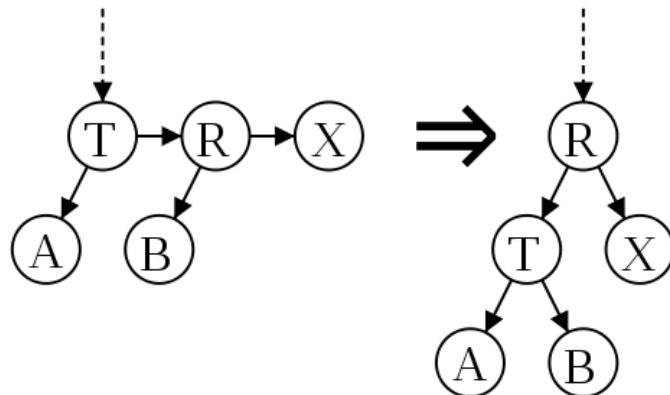
- AA = Arne Andersson (1993).
- Варіація червоно-чорного дерева: червоні вершини можуть бути лише правими потомками.
- *Рівень* вершини: збільшується від листів до кореня; рівень листа 1.
- *Коротке правило AA-дерева*: до однієї вершини можна приєднати іншу вершину того ж рівня лише одну і тільки справа.
- Для балансування використовуються дві операції: SKEW (перекрут) та SPLIT (розділення).
- Всі операції потребують часу $O(\lg n)$.



- З'являється поняття рівня і напрямку зв'язку (правийлівий).
- Зв'язки можуть бути вертикальні і горизонтальні.
- SKEW (усунення лівого зв'язку на одному рівні)

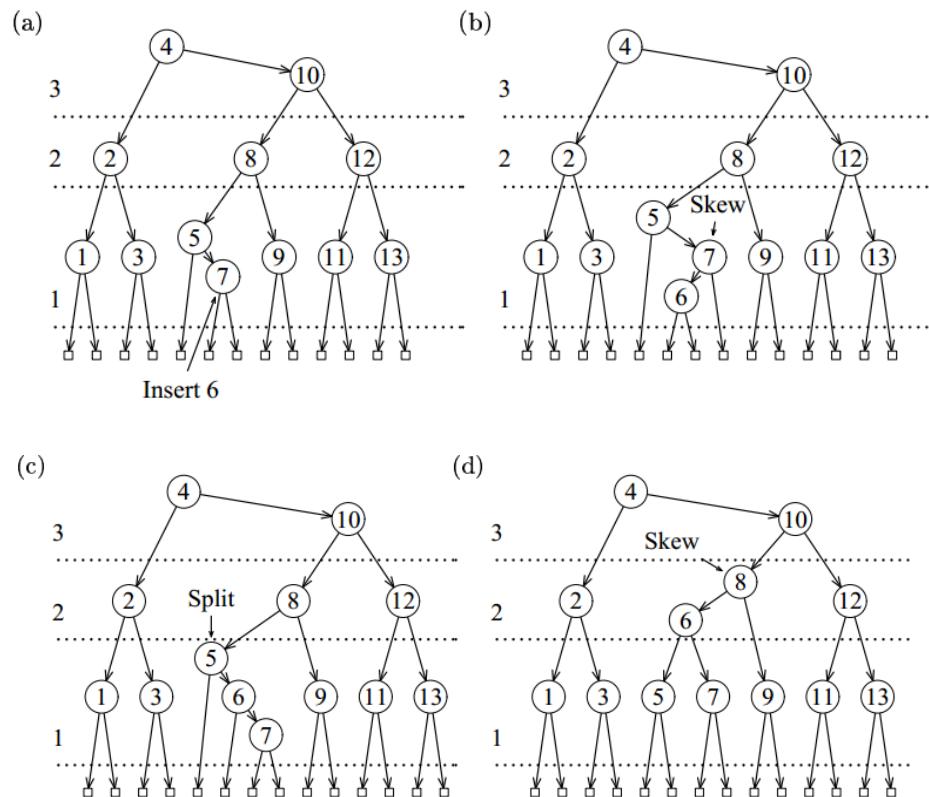


- SPLIT (усунення двох правих зв'язків на одному рівні).

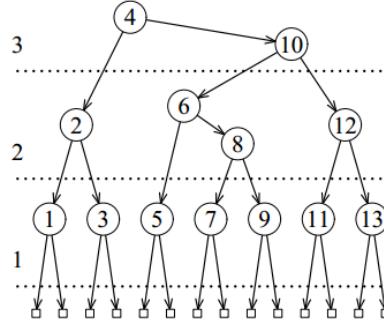


Додавання вузла

- Після звичайного додавання вершини піднімаємося вгору, виконуючи для кожного вузла SKEW, а потім SPLIT.

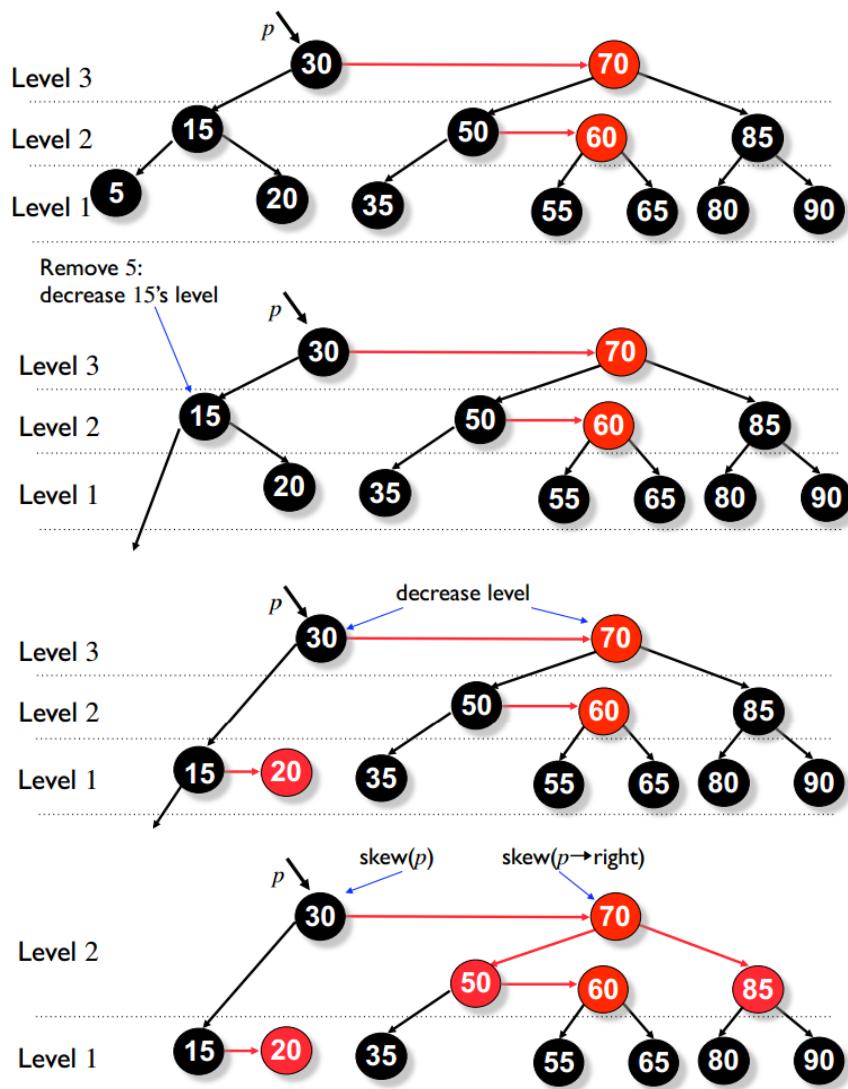


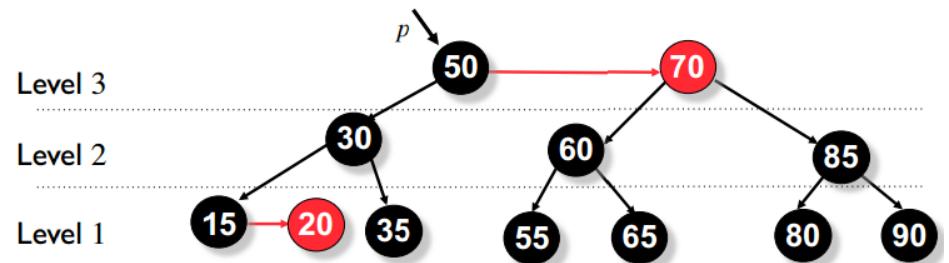
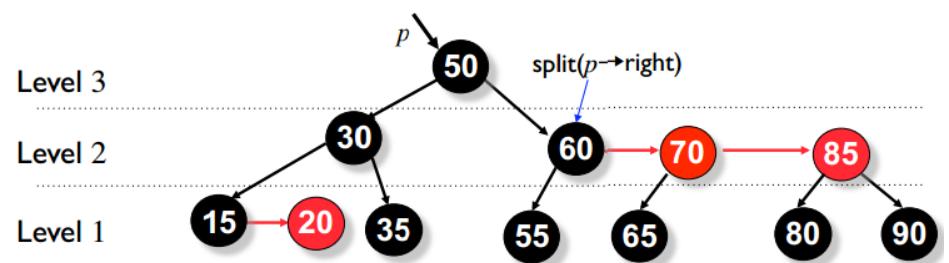
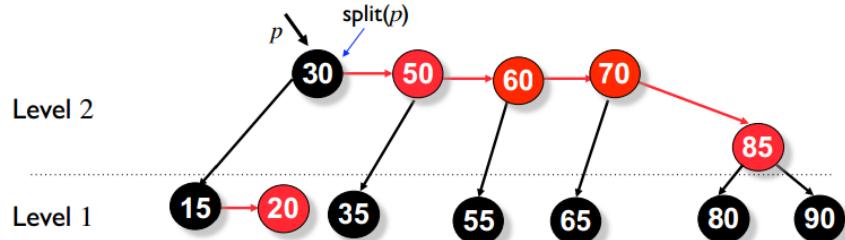
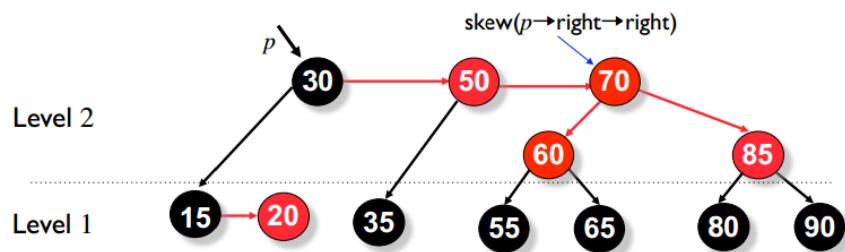
(e)



Видалення вузла

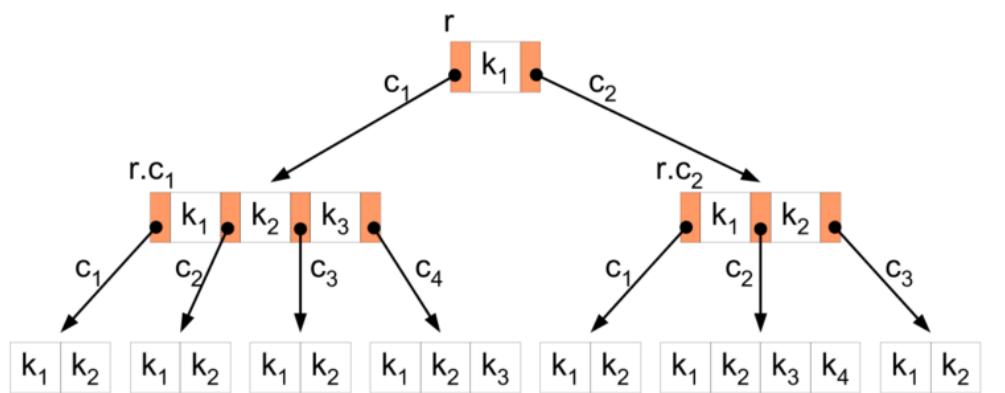
- Після видалення листа при русі вгору спочатку за необхідності відбувається пониження рівня, потім тричі викликається SKEW і двічі SPLIT:





В-дерева

- Дерево пошуку, збалансоване по висоті.
- Кожен вузол може мати від 2 до m потомків для дерева порядку m .
- Основні операції виконуються за час $O(\lg n)$.
- 2-3 дерева – частковий випадок В-дерев.



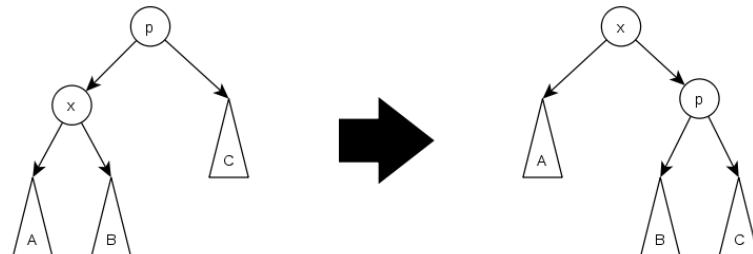
Розширювані дерева (splay trees)

- Двійкове дерево пошуку з підтримкою збалансованості.
- Не потребує додаткових полів у вузлі.
- Явні функції балансування відсутні.
- При кожному звертанні до дерева виконується «операція розширення» (splay operation).
- В результаті вузли, до яких звертаються частіше, зберігаються більше близче до кореня, а до яких рідше – більше близче до листків.
- Всі операції потребують часу в середньому $O(\lg n)$.

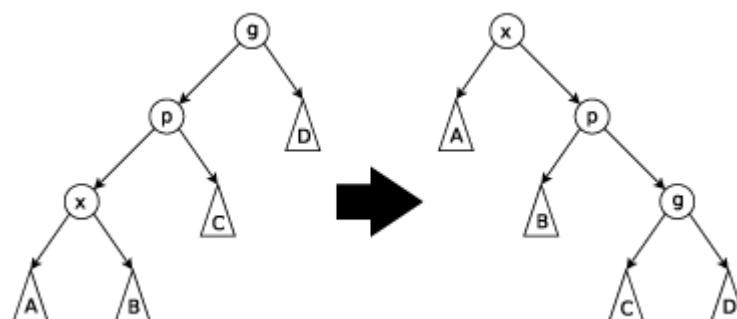
Операція SPLAY

Переміщує вершину x в корінь за допомогою операцій Zig, Zig-Zig та Zig-Zag.

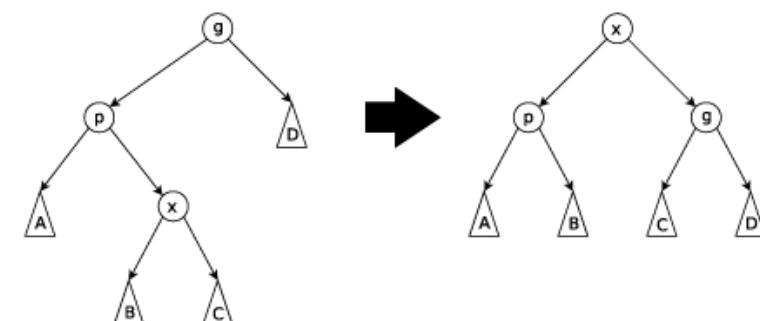
- Zig



- Zig-Zig



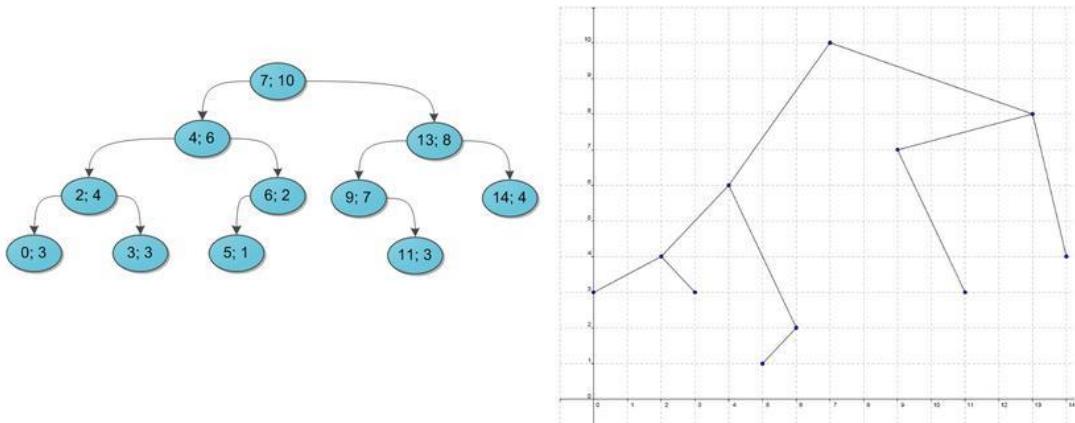
- Zig-Zag



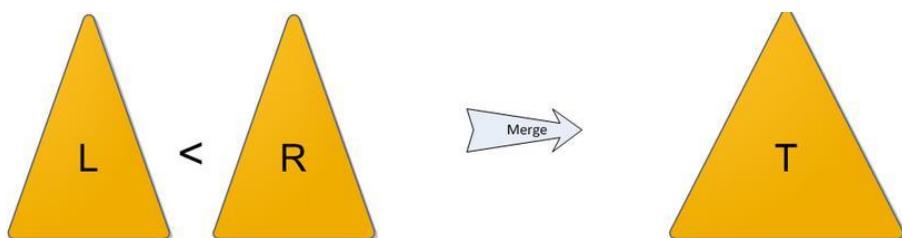
- *Merge* (об'єднання двох дерев). Для злиття дерев T_1 і T_2 , в яких всі ключі T_1 менше ключів в T_2 , робимо Splay для максимального елементу T_1 , тоді біля кореня T_1 не буде правого дочірнього елемента. Після цього робимо T_2 правим дочірнім елементом T_1 .
- *Split* (розділення дерева на дві частини). Для розділення дерева знаходиться найменший елемент, більший або рівний x і для нього робиться Splay. Після цього відрізаємо ліве піддерево у якості другого дерева.
- *Search* (пошук елемента). Спочатку звичайний пошук. При знаходженні елементу запускаємо Splay для нього.
- *Insert* (додавання елемента). Запускаємо Split від елементу, що додається, і підвішуємо дерево, що вийшли, за нього.
- *Delete* (видалення елемента). Знаходимо елемент в дереві, робимо Splay для нього, робимо поточним деревом Merge його дітей.

Декартові дерева (treaps)

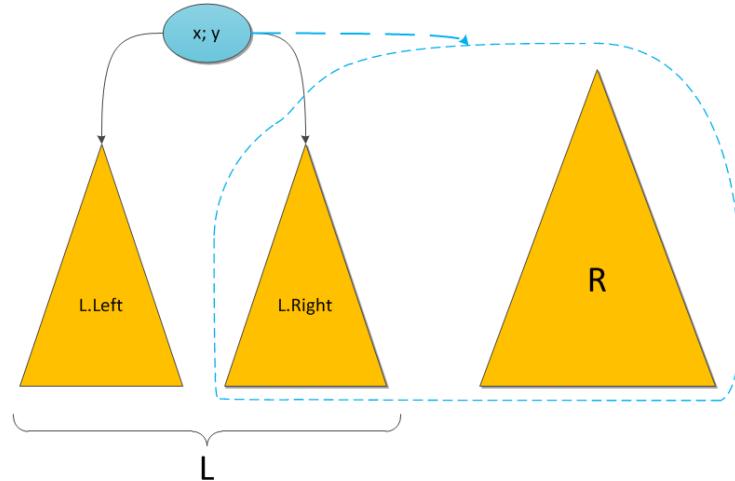
- Поєднує в собі бінарне дерево пошуку і купу.
- Вузол містить значення *ключа* та *пріоритету*.
- Структура є деревом пошуку за ключами та купою (пірамідою) за пріоритетами.
- Ключі можуть повторюватися, але дублі мають знаходитися однозначно тільки справа чи зліва.
- Пріоритетом є випадкове число з заданого проміжку (наприклад, $(0, 1)$). Повторів пріоритетів слід уникати.
- Висота дерева з дуже високою ймовірністю $\leq 4 \log_2 n$.



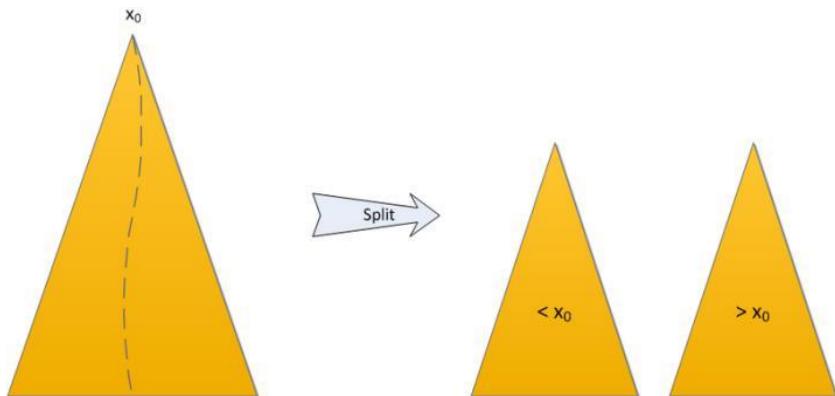
- Злиття двох дерев *Merge*. Умова: всі ключі одного дерева не перевищують ключі іншого.



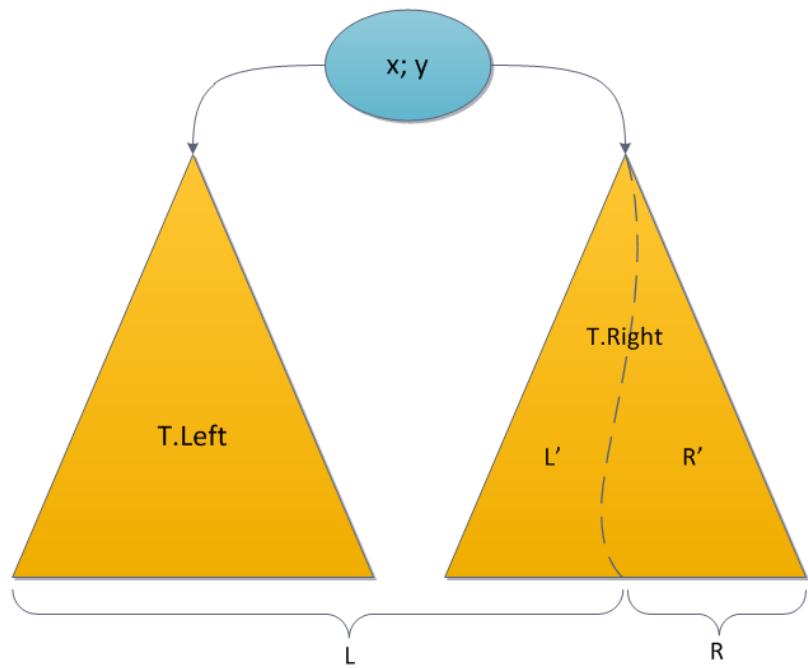
В якості нового кореня береться корінь дерева з більшим пріоритетом. Одне з піддерев зрозуміле, друге отримується рекурсивно. Пріоритет лівогокореня більший



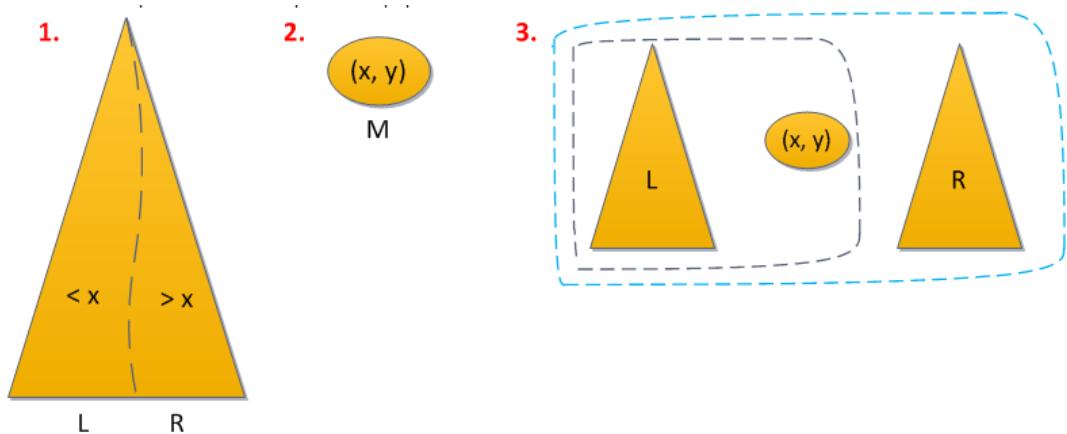
- Розбиття дерева за ключем *Split*. Умова: всі ключі одного дерева не перевищують ключів іншого.



Звіряємо корінь з ключем. В залежності від результату бачимо, до якого з нових дерев належатиме корінь з одним із піддерев. Друге дерево рекурсивно виділяється з іншого піддерева. Ключ кореня менший за x_0

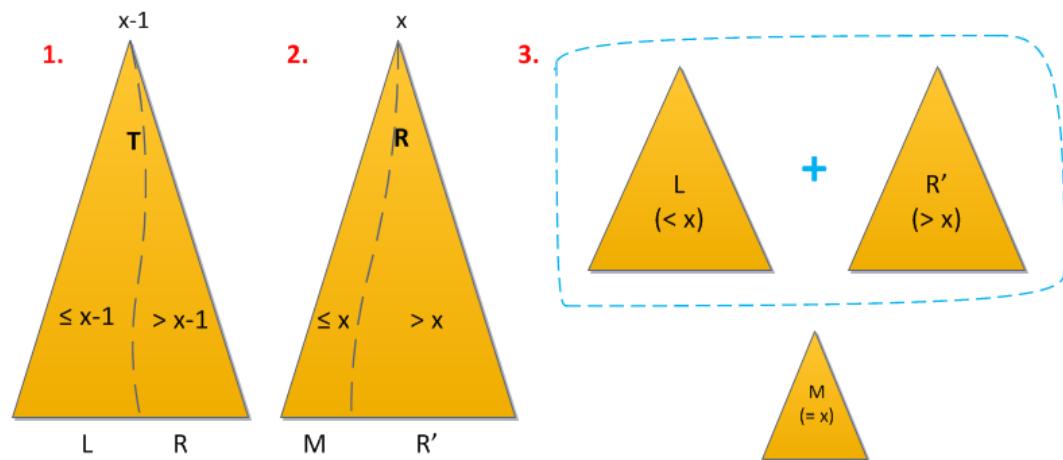


- Вставка елемента



Час виконання операцій *Merge* та *Split* складає $O(\lg n)$. Такий же буде час при виконанні операцій, що складаються зі скінченої кількості їх викликів.

- Видалення елемента



Інший спосіб вставки та видалення елемента (x, y)

- Вставка

1. Спуск як по дереву пошуку, поки не знайдемо перший елемент (x_1, y_1) з пріоритетом $y_1 < y$.
2. Розбиття піддерева з коренем (x_1, y_1) по ключу x .
3. На це місце вставляємо дерево з коренем (x, y) і отриманими після розбиття деревами в якості синів.

- Видалення

1. Спуск як по дереву пошуку, поки не знайдемо елемент (x, y) .
2. Злиття синів піддерева з коренем (x, y) .
3. Результат цього злиття ставиться на місце (x, y) .

Теорема (розширення червоно-чорних дерев). Нехай f – поле, яке розширює червоно-чорне дерево T з n вузлів, і нехай вміст поля f вузла x може бути обчислений з використанням лише інформації, що зберігається у вузлах x , $left[x]$ і $right[x]$, включаючи $f[left[x]]$ та $f[right[x]]$. Тоді можливо підтримувати актуальність інформації f у всіх вузлах дерева T в процесі вставки і видалення без впливу на асимптотичний час роботи цих процедур $O(\lg n)$.

Доведення.

Ідея доведення.

Зміна поля f вузла x може вплинути тільки на значення поля f предків вузла x . Тобто, зміна $f[x]$ може зумовити зміну лише $f[p[x]]$, зміна $f[p[x]]$ – зміну тільки $f[p[p[x]]]$, і так до кореня. При модифікації $f[root[T]]$ від цього значення ніякі інші вже не залежать, процес оновлення завершується. Висота червоно-чорного дерева $O(\lg n)$, тому зміна поля f у вузлі вимагатиме часу $O(\lg n)$ для оновлення всіх залежних від нього вузлів.

Вставка вузла.

Перша фаза. Вузол x вставляється як дочірній деякого існуючого вузла $p[x]$.

Значення $f[x]$ обчислюється за час $O(1)$, бо за умовою залежить тільки від

інформації в інших полях x та його синах (в даному випадку це обмежувачі $ni/[T]$).

Після цього зміни «піднімаються» по дереву, що дає час першої фази $O(\lg n)$.

Друга фаза. Структурні зміни можуть зумовити лише повороти, яких буде не більше двох. За один поворот зміни відбуваються у двох вузлах, тому на оновлення піде час $O(\lg n)$. Отже, вставка відбувається за час $O(\lg n)$.

Вставка вузла.

Перша фаза. Вузол x вставляється як дочірній деякого існуючого вузла $p[x]$.

Значення $f[x]$ обчислюється за час $O(1)$, бо за умовою залежить тільки від

інформації в інших полях x та його синах (в даному випадку це обмежувачі $ni/[T]$).

Після цього зміни «піднімаються» по дереву, що дає час першої фази $O(\lg n)$.

Друга фаза. Структурні зміни можуть зумовити лише повороти, яких буде не більше двох. За один поворот зміни відбуваються у двох вузлах, тому на оновлення піде час $O(\lg n)$. Отже, вставка відбувається за час $O(\lg n)$.

Дерево порядкової статистики

Пошук елемента з заданим рангом i в піддереві з коренем x :

```

OS_SELECT( $x, i$ )
1  $r \leftarrow \text{size}[\text{left}[x]] + 1$ 
2 if  $i = r$ 
3   then return  $x$ 
4 elseif  $i < r$ 
5   then return OS_SELECT( $\text{left}[x], i$ )
6 else return OS_SELECT( $\text{right}[x], i - r$ )

```

($\text{size}[\text{left}[x]] + 1$) – ранг вузла x в піддереві з коренем x . Кожен рекуривний виклик опускає нас на один рівень дерева нижче, тому час роботи в найгіршому випадку пропорційний висоті дерева, тобто $O(\lg n)$ для червоночорного дерева з n елементів. (Знайдемо в дереві-прикладі елемент з рангом 17.)

Пошук рангу елемента x :

```

OS_RANK( $T, x$ )
1  $r \leftarrow \text{size}[\text{left}[x]] + 1$ 
2  $y \leftarrow x$ 
3 while  $y \neq \text{root}[T]$ 
4   do if  $y = \text{right}[p[y]]$ 
5     then  $r \leftarrow r + \text{size}[\text{left}[p[y]]] + 1$ 
6      $y \leftarrow p[y]$ 
7 return  $r$ 

```

Ранг вузла x – кількість вершин, обійдених при симетричному обході до x , плюс 1 для самого вузла x . В циклі рухаємось від x до кореня; якщо по дорозі проходимо елемент, що є чиємось правим сином, додаємо кількість елементів, які треба обійти перед ним, плюс 1 для нього.

Покажемо коректність процедури OS_RANK за допомогою наступного інваріанту циклу:

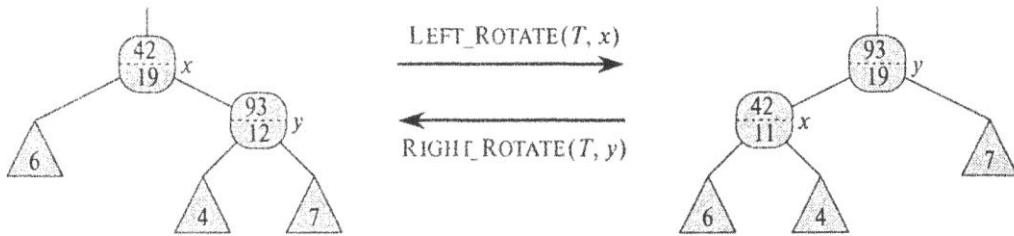
На початку кожної ітерації циклу **while** значення r є рангом $\text{key}[x]$ в піддереві з коренем y вузлі y .

- **Ініціалізація.** Перед першою ітерацією r присвоюється ранг $\text{key}[x]$ в піддереві з коренем x . Присвоєння в рядку 2 робить інваріант циклу істинним перед першою ітерацією.
- **Збереження.** Треба показати, що якщо r – ранг $\text{key}[x]$ в піддереві з коренем y на початку виконання тіла циклу, то в кінці виконання r є рангом $\text{key}[x]$ в піддереві з коренем $p[y]$. Якщо y є лівим потомком, то ні $p[y]$, ні жоден вузол з піддерева правого потомка $p[y]$ не може передувати x при симетричному обході, тому значення r не зміниться. Інакше y є правим потомком, і всі вузли в піддереві лівого потомка $p[y]$, а також $p[y]$, передують x , тому значення r має збільшитися на $(\text{size}[\text{left}[p[y]]] + 1)$.
- **Завершення.** Цикл завершується, коли $y = \text{root}[T]$, тому піддерево з коренем y є цілим деревом. Таким чином r є рангом $\text{key}[x]$ в усьому дереві. Кожна ітерація

циклу має час $O(1)$, а у при кожній ітерації піднімається на один рівень вгору, тому час роботи процедури OS_RANK в найгіршому випадку буде $O(\lg n)$.

Підтримка розміру піддерев

Оновлення розмірів піддерев при поворотах:



Некоректними стають тільки лише два значення поля $size$ вузлів, навколо зв'язку яких відбувається поворот. Досить до псевдокоду LEFT_ROTATE додати

13 $size[y] \leftarrow size[x]$
 14 $size[x] \leftarrow size[left[x]] + size[right[x]] + 1$

На кожен поворот витрачається час $O(1)$.

Додавання вузла

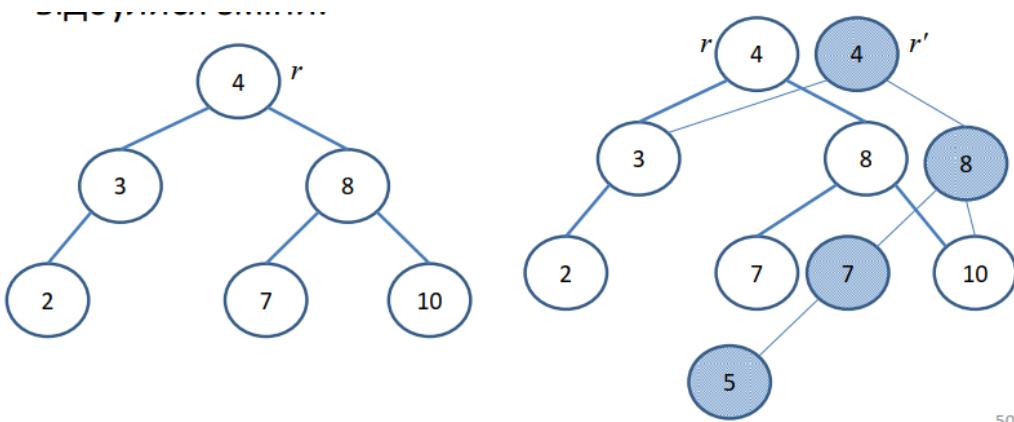
В першій фазі потрібно збільшити значення $size[x]$ для кожного вузла x на шляху від нового вузла до кореня. Новий вузол отримує значення $size$ 1. Друга фаза складатиметься максимум з двох поворотів. Тому загальний час операції $O(\lg n)$

Видалення вузла

На першому етапі проходимо шлях від позиції видаленого вузла вгору до кореня, зменшуючи значення поля $size$ для кожного вузла. В другій фазі може відбутися до трьох поворотів. Отже, загальний час операції $O(\lg n)$.

Персистентні динамічні множини

- Зберігають свої попередні версії (і доступ до них) в процесі внесення змін.
- Може зберігатися тільки остання версія або всі існуючі попередні.
- Персистентними можна зробити різні структури даних.
- Для ефективної реалізації просте копіювання не підходить.
- Розглянемо реалізацію персистентної множини з операціями пошуку, видалення та вставки на основі бінарного дерева пошуку.
- Для кожної версії множини зберігається свій корінь.
- Фактично будується копія лише тієї вітки (шляху), де відбулися зміни.

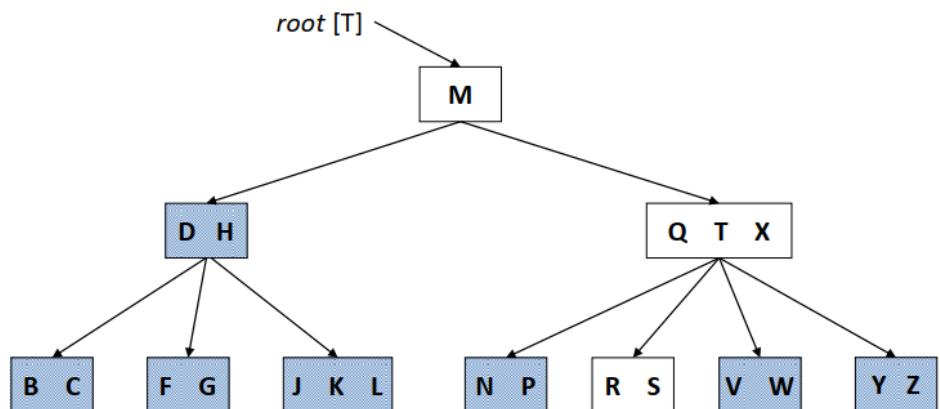


50

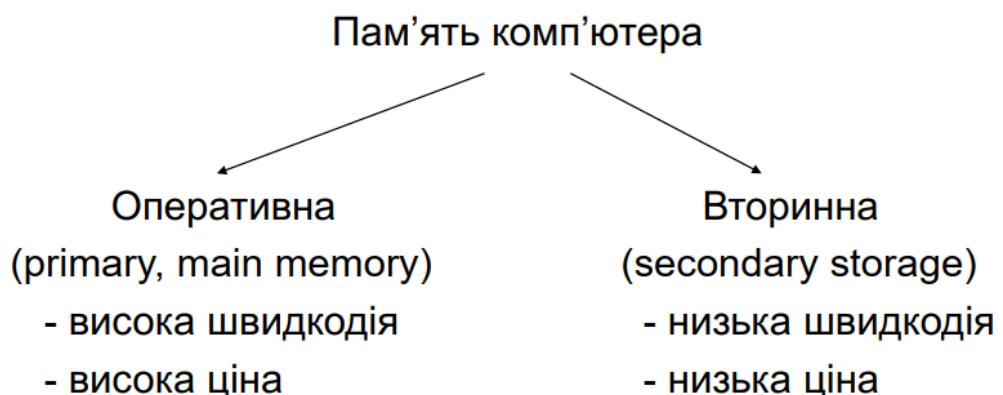
ЛЕКЦІЯ 4

В-дерева

- Робота з великими об'ємами даних, які не поміщаються в оперативну пам'ять і зберігаються на диску (наприклад, СУБД, файлові системи).
- В-дерева – узагальнення бінарних дерев пошуку.
- Висока ступінь розгалуження – вузли можуть мати до тисяч потомків.
- Якщо внутрішній вузол містить $n[x]$ ключів, то він має $(n[x]+1)$ синів.
- Ключі у вузлі x є роздільниками діапазону ключів на $(n[x]+1)$ піддіапазонів.
- При пошуку переходимо до сина з потрібним діапазоном.



Структури даних у вторинній пам'яті



- Аналізуючи алгоритми, що враховують роботу над даними у зовнішній пам'яті, доцільно розглядати
 - кількість звернень до диску,
 - час обчислень (процесорний час).
- Кількість звернень до диску вимірюється кількістю зчитаних або записаних сторінок інформації.
- Алгоритми роботи над В-деревами копіюють до оперативної пам'яті тільки необхідні для роботи сторінки і записують назад лише змінені сторінки.
- В довільний момент часу алгоритм працює над деякою постійною кількістю сторінок в оперативній пам'яті – а розмір самого В-дерева при цьому не обмежений.
- Вважаємо, що система сама видаляє з основної пам'яті сторінки, що більше не використовуються.

- Типовий алгоритм роботи з об'єктом x :

$x \leq$ вказівник на деякий об'єкт

`DISK_READ(x)`

Операції, що звертаються та/або змінюють поля x

`DISK_WRITE(x)` // не потрібно, якщо поля x не змінювалися

Інші операції, що не змінюють полів x

- Бажано мінімізувати кількість звернень до диску.
- Розмір вузла В-дерева звичайно відповідає розміру сторінки на накопичувачі.
- Таким чином кількість потомків вузла обмежена розміром сторінки.
- Зазвичай для великих В-дерев ступінь розгалуження становить від 50 до 2000.
- Чим більше розгалуження, тим менша висота дерева, а отже і кількість звернень до диску.

Означення В-дерева Дерево T з коренем $root[T]$ та властивостями:

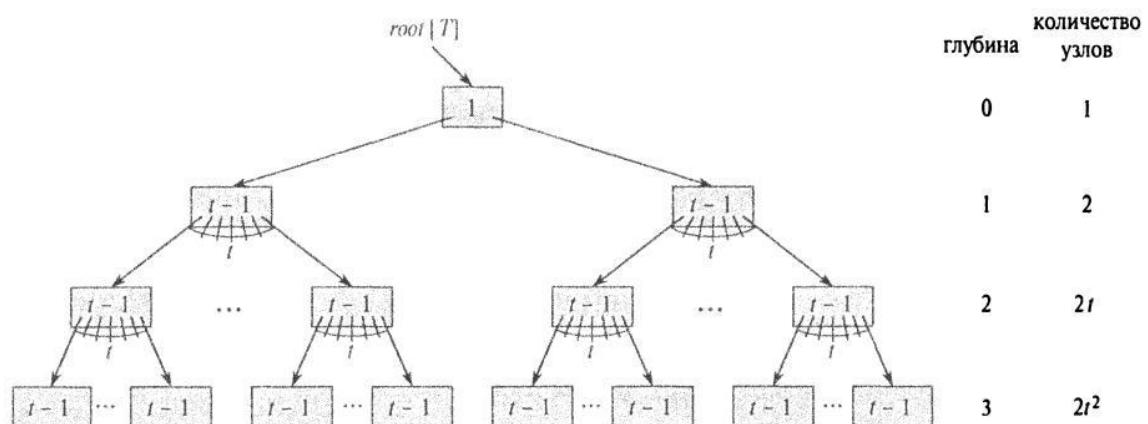
1. Кожен вузол x містить поля:

- $n[x]$ – поточна кількість ключів вузла x ;
- впорядковано збережені ключі, так що $key_1[x] \leq key_2[x] \leq \dots \leq key_{n[x]}[x]$;
- логічне значення $leaf[x]$, істинне, якщо x – лист.

2. Кожен внутрішній вузол x містить $(n[x]+1)$ вказівник $c_1[x], \dots, c_{n[x]+1}[x]$ на дочірні вузли.

3. Ключі $key_i[x]$ розділяють піддіапазони ключів піддерев: якщо k_i – ключ, що зберігається у піддереві з коренем $c_i[x]$, то $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \dots \leq key_{n[x]}[x] \leq k_{n[x]+1}$.

4. Всі листи розташовані на одній глибині h , що дорівнює висоті дерева. (Тобто В-дерево ідеально збалансоване за висотою.)
5. Мінімальна і максимальна кількість ключів у вузлі регламентовані фіксованим цілим $t \geq 2$ (мінімальна степінь, *minimum degree*):
 - кожен вузол крім кореня містить як мінімум $(t-1)$ ключ, тобто матиме принаймні t синів; непорожнє дерево має в корені хоча б один ключ;
 - кожен вузол містить не більше $(2t-1)$ ключів, тобто матиме максимум $2t$ синів; вузол вважається *повним*, якщо має рівно $(2t-1)$ ключ.



Висота В-дерева

Теорема. Висота В-дерева T з $n \geq 1$ вузлами та мінімальною степінню $t \geq 2$ не перевищує $\log_t(n+1)/2$.

Доведення. Нехай В-дерево має висоту h .

Корінь містить мінімум 1 ключ, решта вузлів – не менше $(t-1)$ ключа кожен.

Отже на глибині 1 маємо мінімум 2 вузли.

На глибині 2 мінімум $2t$ вузлів.

На глибині 3 мінімум $2t^2$ вузлів.

На глибині h мінімум $2th-1$ вузол.

Кількість ключів задовольняє нерівність

$$n \geq 1 + (t-1) \sum_{i=1}^h 2t^{i-1} = 1 + 2(t-1) \left(\frac{t^h - 1}{t-1} \right) = 2t^h - 1.$$

Тобто $t^h \leq (n+1)/2$.

Прологарифмуємо по t : $h \leq \log_t(n+1)/2$.

Основні операції над В-деревами

- Пошук
- Створення нового дерева
- Додавання вузла
- Видалення вузла

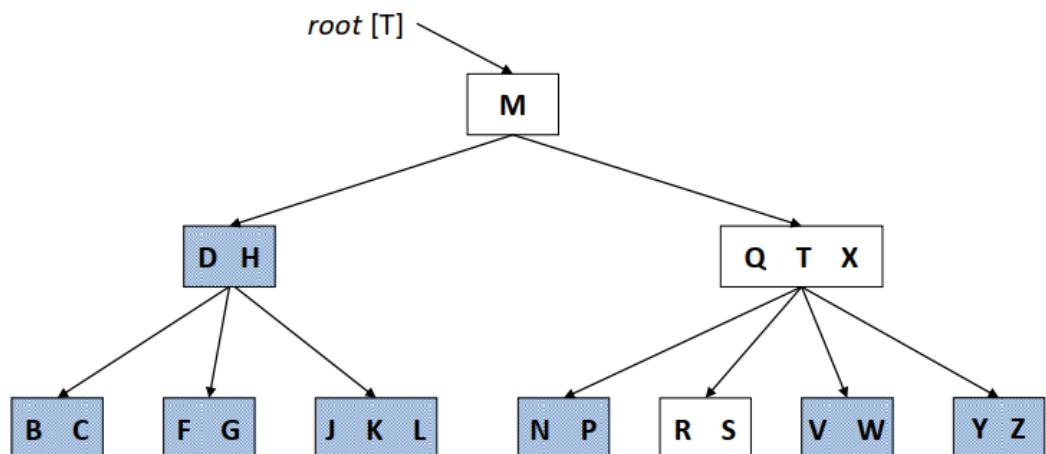
АЛГОРИТМ $B_TREE_SEARCH(x, k)$

```

 $i \leq 1$ 
while  $i \leq n[x]$  та  $k > key_i[x]$ 
  do  $i \leftarrow i + 1$ 
if  $i \leq n[x]$  та  $k = key_i[x]$ 
  then return  $(x, i)$ 
if  $leaf[x]$ 
  then return NIL
else DISK_READ( $c_i[x]$ )
  return  $B\_TREE\_SEARCH(c_i[x], k)$ 

```

- Повертає пару (x, i) : $key_i[x] = k$ або NIL.
- Серед ключів вузла виконується лінійний пошук.
- Рекурсивно спускаємось від кореня до листа.
- Пройдемо дискових сторінок $O(h) = O(\log t n)$, де h – висота В-дерева, n – кількість його вузлів.
- Перегляд ключів займе час $O(t)$, оскільки $n[x] < 2t$.
- Загальний час обчислень $O(t \cdot h) = O(t \log t n)$.



АЛГОРИТМ $B_TREE_CREATE(T)$

$x \leq \text{ALLOCATE_NODE}()$

$leaf[x] \leq \text{true}$

$n[x] \leq 0$

DISK_WRITE(x)

$root[T] \leq x$

- При створенні дерева спочатку створюється порожній кореневий вузол, а потім вносяться нові ключі шляхом вставки вершин.
- Використовується стандартна процедура виділення нової дискової сторінки для створеного вузла ALLOCATE_NODE, яка виконується за константний час.
- Кількість дискових операцій $O(1)$.
- Загальний час виконання $O(1)$.

Вставка ключа у В-дерево

- Замість вставки в потрібне місце листа, як у бінарному дереві пошуку, вставляємо *новий ключ* в існуючий лист.
- Якщо лист вже повний, він розбивається на два, а ключ, по якому відбувається розбиття, вставляється в батьківський вузол.
- Цей вузол також може виявитися повним, тому процес розбиття може “підніматися” до кореня.
- Вставку ключа можливо здійснити за один прохід від кореня до листа, якщо по ходу робити розбиття відвіданих заповнених вузлів, включаючи лист.

АЛГОРИТМ $B_TREE_INSERT(T, k)$

- 1 $r \leq root[T]$
- 2 **if** $n[r] = 2t - 1$
- 3 **then** $s \leq \text{ALLOCATE_NODE}()$
- 4 $root[T] \leq s$
- 5 $leaf[s] \leq \text{false}$
- 6 $n[s] \leq 0$
- 7 $c_1[s] \leq r$
- 8 $B_TREE_SPLIT_CHILD(s, 1, r)$
- 9 $B_TREE_INSERT_NONFULL(s, k)$
- 10 **else** $B_TREE_INSERT_NONFULL(r, k)$

- Рядки 3-7: випадок заповненого кореня.
- Процедура $B_TREE_INSERT_NONFULL$ робить вставку ключа k в дерево з незаповненим коренем.
- Необхідний процесорний час $O(t \cdot h) = O(t \log t)$.

Видалення ключа з В-дерева

- Видалення ключа з В-дерева є складнішим за вставку.
- Ключ може видалятися з довільного вузла, а не тільки з листа. Це призведе до перебудови дочірніх вузлів.
- Тепер слід перевіряти, чи достатньо заповнені вузли (крім кореня).

- Щоб виконувати видалення ключа за один прохід, для роботи процедури потрібна сильніша умова: при виклику для вузла x там міститься не менше t ключів. Тому перед рекурсивним викликом можливе переміщення ключа в синівський вузол.
- В результаті видалення ключа висота дерева може зменшитися на 1.

Процедура B TREE DELETE(k, x).

1. Якщо ключ k знаходиться у вузлі x та останній є листом – видаляємо k з x .
2. Якщо ключ k знаходиться у вузлі x та x є внутрішнім вузлом:
 - а) якщо дочірній для x вузол y , що є попередником ключа k у вузлі x , містить не менше t ключів, то знаходимо k_1 – попередника k в піддереві з коренем y ; рекурсивно видаляємо k_1 і замінюємо k в x ключем k_1 (при цьому пошук і видалення k_1 виконується за один прохід вниз);
 - б) симетрична ситуація: дочірній вузол z – наступний за ключем k і містить не менше ключів; тоді шукаємо наступника k ;
 - в) інакше, якщо y та z містять по $(t-1)$ ключів, вносимо k та всі ключі z в y (при цьому з x видаляється ключ k та вказівник на z , а у після цього міститиме $(2t-1)$ ключ); звільняємо пам'ять від z та рекурсивно видаляємо k з y .
3. Якщо ключ k відсутній у внутрішньому вузлі x , знаходимо корінь $c_i[x]$ піддерева, що має містити k (якщо такий ключ існує). Якщо $c_i[x]$ містить лише $(t-1)$ ключ, виконуємо крок 3а) або 3б) для гарантії переходу у вузол з мінімум t ключами. Потім рекурсивно видаляємо k з піддерева $c_i[x]$
 - а) якщо $c_i[x]$ містить тільки $(t-1)$ ключ, але один з його безпосередніх сусідів (брати справа і зліва, відділені єдиним ключем-роздільником) має хоча б t ключів, передамо $c_i[x]$ відповідний ключ-роздільник, на його місце поставимо крайній ключ сусіда і перенесемо відповідний вказівник від сусіда до $c_i[x]$.
 - б) якщо $c_i[x]$ з обома прямыми сусідами містять по $(t-1)$ ключу, об'єднаємо $c_i[x]$ з одним з цих сусідів (при цьому відповідний ключ-роздільник стане медіаною нового вузла).
- Більшість ключів В-дерева знаходяться в листках, тому на практиці видалення найчастіше з листів і відбуватимуться. В цьому випадку B_TREE_DELETE виконується в один прохід униз.
- Повернення може відбутися при видаленні ключа з внутрішнього вузла (випадки 2а) та 2б)).

- Для В-дерева висоти h потрібно $O(h)$ дискових операцій.
- Загальний процесорний час $O(t \cdot h) = O(t \log t)$.

B+-дерева

- Істинні значення ключів містяться тільки в листах, внутрішні вузли містять лише ключі-роздільники діапазонів піддерев.
- Листки додатково зв'язані у список. Це дозволяє швидкий доступ до ключів в порядку зростання.
- Легко реалізується незалежність програми від структури інформаційної запису.
- Пошук обов'язково закінчується в листі. Видалення ключа завжди з листа.
- Вимагають більше пам'яті для представлення, порівняно з В-деревами.
- Різновид В-дерева, що вимагає заповненості кожного вузла мінімум на $2/3$ (а не наполовину).
- Компактніші за звичайні В-дерева.
- Просте розділення вузлів при розбитті вже не працює.
- Замість цього – “переливання” до сусідського вузла.
- Якщо сусід також повний, відбувається поділ ключів приблизно порівну між трьома новими вузлами.
- B^* -дерево, що задовольняє умовам B^+ дерева, називають B^{*+} -деревом.

2-3-4-дерева

- Різновид В-дерева: кожен проміжний вузол має або двох нащадків і одне поле (2-вузол), або трьох нащадків і два поля (3-вузол).
- Всі листи знаходяться на одному рівні і містять 1 або 2 поля (власне, вони й містять всю інформацію).
- Значення поля 2-вузла строго більше найбільшого значення в лівому піддереві і не менше найменшого значення в правому піддереві.
- Значення первого поля 3-вузла строго більше найбільшого значення в лівому піддереві і не менше найменшого значення в центральному піддереві. Значення другого поля 3-вузла строго більше найбільшого значення в центральному піддереві і не менше найменшого значення в правому піддереві.
- 2-3 дерева – ізометрія АА-дерев.
- Різновид В-дерева: мінімальна степінь $t=2$.
 - 2-вузол містить 1 поле і (якщо не лист) 2 нащадки.
 - 3-вузол містить 2 поля і (якщо не лист) 3 нащадки.
 - 4-вузол містить 3 поля і (якщо не лист) 4 нащадки.
- 2-3-4-дерева – ізометрія червоно-чорних дерев: це еквівалентні структури даних. Кожен чорний вузол можна об'єднати з його червоними потомками, при цьому результиуючий вузол матиме ≤ 3 ключів та ≤ 4 нащадків.
- Операції вставки і видалення ключів спричиняють розширення вузлів, розбиття і злиття, що будуть еквівалентними перефарбуванням і обертанням червоно-чорних дерев.
- Ідейно 2-3-4-дерева простіші для розуміння, але червоно-чорні дерева легші в реалізації, тому саме вони і отримали використання.

ЛЕКЦІЯ 5

Піраміди злиття (mergeable heaps)

Піраміди, що підтримують операцію злиття. За замовчуванням вважаємо їх неспадаючими (вузол з мінімальним ключем у вершині).

- **MAKE_HEAP()**: створення нової порожньої піраміди.
- **INSERT(H,x)**: вставка готового вузла x в піраміду H .
- **MINIMUM(H)**: повертає вказівник на вузол піраміди H з найменшим ключем.
- **EXTRACT_MIN(H)**: видаляє вузол піраміди H з найменшим ключем і повертає вказівник на нього.
- **UNION(H1,H2)**: повертає нову піраміду – результат злиття $H1, H2$ (вони не зберігаються).
- **DECREASE_KEY(H,x,k)**: присвоєння вузлу x піраміди H значення ключа k , що є меншим за поточне.
- **DELETE(H,x)**: видалення вузла x з піраміди H .

Процедура	Бинарная пирамида (наихудший случай)	Биномиальная пирамида (наихудший случай)	Пирамида Фibonacci (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	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

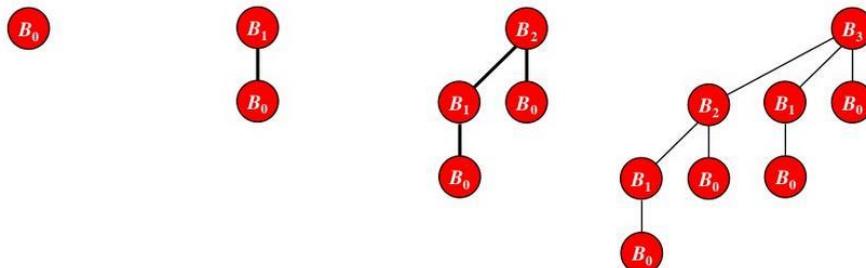
- При необхідності злиття бінарна піраміда виявляється не найкращим варіантом.
- Жодна з реалізацій не надає ефективної реалізації пошуку за ключем.
- Впорядковане дерево, що визначається рекурсивно.
- Біноміальне дерево B_0 порядку 0 складається з єдиної вершини.
- Біноміальне дерево B_k порядку k складається з двох зв'язаних біноміальних дерев B_{k-1} : корінь одного є крайнім лівим сином іншого.

$$(n = 2^0 = 1)$$

$$(n = 2^1 = 2)$$

$$(n = 2^2 = 4)$$

$$(n = 2^3 = 8)$$



Лема (властивості біноміальних дерев) Біноміальне дерево B_k

1. має 2^k вузлів;
2. має висоту k ;
3. має рівно $\binom{k}{i}$ вузлів на глибині $i = 0, 1, \dots, k$;
4. має корінь степені k , а степінь синів менша степеня кореня; при цьому якщо синів пронумерувати зліва направо числами $(k-1), (k-2), \dots, 0$, то i -й син є коренем біноміального дерева B_i .

Доведення. Індукція по k .

Для бази індукції B_0 перевірка тривіальна.

Нехай всі властивості виконуються для B_{k-1} .

1. Біноміальне дерево B_k складається з двох копій B_{k-1} , тому містить $2^{k-1} + 2^{k-1} = 2^k$ вузлів.
2. Виходячи зі способу зв'язування двох копій B_{k-1} в дерево B_k , глибина останнього на одиницю перевищує глибину B_{k-1} . Враховуючи припущення індукції, висота B_k дорівнює $(k-1)+1=k$.

Доведення (далі)

3. Нехай $D(k, i)$ – кількість вузлів на глибині i біноміального дерева B_k . Виходячи зі способу зв'язування двох копій B_{k-1} в дерево B_k , кількість вузлів біноміального дерева B_k на глибині i дорівнює кількості вузлів біноміального дерева B_{k-1} на глибині i плюс кількість вузлів у B_{k-1} на глибині $(i-1)$:

$$D(k, i) = D(k-1, i) + D(k-1, i-1) =$$

$$= \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}.$$

Доведення (далі)

4. Корінь біноміального дерева B_k є єдиним вузлом, що перевищує степінь B_{k-1} : він має на одного сина більше. Оскільки (припущення індукції) корінь B_{k-1} має степінь $(k-1)$, то степінь кореня дерева B_k буде k . Якщо за припущенням синами біноміального дерева B_{k-1} будуть відповідно зліва направо біноміальні дерева $B_{k-2}, B_{k-3}, \dots, B_0$, то після прив'язки зліва ще одного дерева B_{k-1} синами новоствореного B_k стають $B_{k-1}, B_{k-2}, \dots, B_0$.

Наслідок. Максимальна степінь вузла біноміального дерева з n вершинами складає $\lg n$.

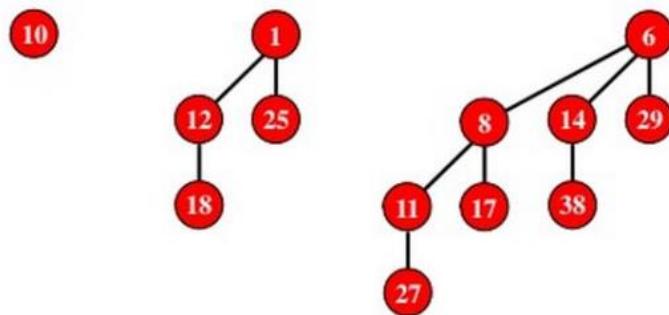
Біноміальна піраміда (binomial heap)

Біноміальна піраміда (біноміальна купа) H – множина біноміальних дерев, що задовольняють властивостям біноміальних пірамід:

1. кожне біноміальне дерево в H є неспадаючою пірамідою (мінімальний елемент на вершині);
2. для довільного невід'ємного k в H існує не більше одного біноміального дерева відповідного порядку.

Біноміальна піраміда з n вузлів складається не більше ніж з $(\lg n + 1)$ біноміальних дерев.

- Нехай біноміальна купа має n вузлів.
- Одиниці в двійковому записі n відповідають порядкам біноміальних дерев, що входять до цієї купи. Наприклад, для 13 вузлів: $13 = 11012$. Біноміальна піраміда складатиметься з біноміальних дерев B_0 , B_2 та B_3 з 1, 4 і 8 вузлів відповідно.

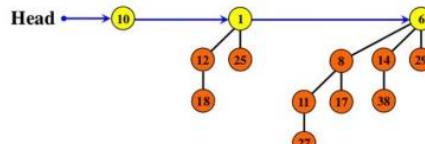


Операції над біноміальними пірамідами

- Створення порожньої біноміальної піраміди. Час роботи $\Theta(1)$.
- Пошук мінімального ключа (вважаємо, що відсутні ключі зі значенням ∞):

`BINOMIAL_HEAP_MINIMUM(H)`

```
1   $y \leftarrow \text{NIL}$ 
2   $x \leftarrow \text{head}[H]$ 
3   $min \leftarrow \infty$ 
4  while  $x \neq \text{NIL}$ 
5    do if  $\text{key}[x] < min$ 
6      then  $min \leftarrow \text{key}[x]$ 
7       $y \leftarrow x$ 
8       $x \leftarrow \text{ sibling}[x]$ 
9  return  $y$ 
```



Час роботи $O(\lg n)$, бо перевіряємо не більше $(\lfloor \lg n \rfloor + 1)$ коренів.

Операції над біноміальними пірамідами

- Злиття двох біноміальних пірамід.
 1. Злити списки коренів H_1 та H_2 в упорядкований список (BINOMIAL_HEAP_MERGE).
 2. Відновити властивості біноміальної піраміди H .

Процедура BINOMIAL_HEAP_MERGE діє аналогічно етапу злиття в сортуванні злиттям, на кожному кроці переміщаючи в результатуючий список дерево меншого порядку. Час її роботи $O(\lg n)$, де n – сумарна кількість вершин в двох пірамідах.

Після злиття списків коренів відомо, що купа H містить не більше двох коренів однакової степені, і вони стоять підряд. Тому будемо зв'язувати корені однієї степені поки всі корені не отримають різні степені.

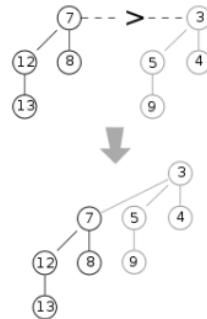
16

Операції над біноміальними пірамідами

Допоміжна операція зв'язування двох біноміальних дерев одного порядку B_{k-1} в біноміальне дерево B_k (дерево з коренем у "підчеплюється" до дерева z):

BINOMIAL_LINK(y, z)

- 1 $p[y] \leftarrow z$
- 2 $sibling[y] \leftarrow child[z]$
- 3 $child[z] \leftarrow y$
- 4 $degree[z] \leftarrow degree[z] + 1$



Час виконання процедури $O(1)$.

Операції над біноміальними пірамідами

```
BINOMIAL_HEAP_UNION( $H_1, H_2$ )
1  $H \leftarrow \text{MAKE\_BINOMIAL\_HEAP}()$ 
2  $head[H] \leftarrow \text{BINOMIAL\_HEAP\_MERGE}(H_1, H_2)$ 
3 Освобождение объектов  $H_1$  и  $H_2$ , но не
   списков, на которые они указывают
4 if  $head[H] = \text{NIL}$ 
5   then return  $H$ 
6  $prev-x \leftarrow \text{NIL}$ 
7  $x \leftarrow head[H]$ 
8  $next-x \leftarrow sibling[x]$ 
9 while  $next-x \neq \text{NIL}$ 
10   do if ( $degree[x] \neq degree[next-x]$ ) или
      ( $sibling[next-x] \neq \text{NIL}$  и  $degree[sibling[next-x]] = degree[x]$ )
```

Операції над біноміальними пірамідами

```
11      then  $prev-x \leftarrow x$                                  $\triangleright$  Случай 1 и 2
12           $x \leftarrow next-x$                                  $\triangleright$  Случай 1 и 2
13      else if  $key[x] \leqslant key[next-x]$ 
14          then  $sibling[x] \leftarrow sibling[next-x]$            $\triangleright$  Случай 3
15               $\text{BINOMIAL\_LINK}(next-x, x)$                    $\triangleright$  Случай 3
16          else if  $prev-x = \text{NIL}$                        $\triangleright$  Случай 4
17              then  $head[H] \leftarrow next-x$                    $\triangleright$  Случай 4
18          else  $sibling[prev-x] \leftarrow next-x$            $\triangleright$  Случай 4
19               $\text{BINOMIAL\_LINK}(x, next-x)$                    $\triangleright$  Случай 4
20           $x \leftarrow next-x$                                  $\triangleright$  Случай 4
21       $next-x \leftarrow sibling[x]$ 
22 return  $H$ 
```

Порахуємо час роботи BINOMIAL_HEAP_UNION.

Нехай біноміальна купа H_1 містить n_1 вузлів, а H_2 – n_2 вузлів та $n_1 + n_2 = n$.

Тоді H_1 має максимум $\log n_1 + 1$ корінь, а H_2 – $\log n_2 + 1$ корінь, тому в H буде не більше $\log n_1 + \log n_2 + 2 \leq 2 \log n + 2 = O(\log n)$ коренів – час роботи

BINOMIAL_HEAP_MERGE. Кожна ітерація циклу виконується за константний час.

Оскільки ми проходимо весь список, то ітерацій буде не більше $\log n_1 + \log n_2 + 2$. Отже, загальний час виконання BINOMIAL_HEAP_UNION складе $O(\log n)$.

- Вставка вузла.

Створюється біноміальна піраміда з одним вузлом (час $O(1)$) та зливається з початковою пірамідою (загальний час $O(\lg n)$).

```
BINOMIAL_HEAP_INSERT( $H, x$ )
1  $H' \leftarrow \text{MAKE\_BINOMIAL\_HEAP}()$ 
2  $p[x] \leftarrow \text{NIL}$ 
3  $child[x] \leftarrow \text{NIL}$ 
4  $sibling[x] \leftarrow \text{NIL}$ 
5  $degree[x] \leftarrow 0$ 
6  $head[H'] \leftarrow x$ 
7  $H \leftarrow \text{BINOMIAL\_HEAP\_UNION}(H, H')$ 
```

- Вилучення мінімального вузла.

BINOMIAL_HEAP_EXTRACT_MIN(H)

- 1 Поиск корня x с минимальным значением ключа в списке корней H , и удаление x из списка корней H
- 2 $H' \leftarrow \text{MAKE_BINOMIAL_HEAP}()$
- 3 Обращение порядка связанных списков дочерних узлов x , установка поля p каждого дочернего узла равным NIL и присвоение указателю $head[H']$ адреса заголовка получающегося списка
- 4 $H \leftarrow \text{BINOMIAL_HEAP_UNION}(H, H')$
- 5 **return** x

Видаляємо мінімальний корінь. Утворюємо з його синів нову біноміальну піраміду (перестановкою у зворотному порядку). Зливаємо новоутворену піраміду з вихідною. Час роботи процедури $O(\lg n)$.

- Зменшення ключа.

Значення ключа замінюється на менше. Після цього рухаємось у напрямку кореня, обмінюючи значення, якщо порушенна умова неспадаючої піраміди.

BINOMIAL_HEAP_DECREASE_KEY(H, x, k)

- 1 **if** $k > key[x]$
- 2 **then error** “Новый ключ больше текущего”
- 3 $key[x] \leftarrow k$
- 4 $y \leftarrow x$
- 5 $z \leftarrow p[y]$
- 6 **while** $z \neq \text{NIL}$ и $key[y] < key[z]$
- 7 **do** Обменять $key[y] \leftrightarrow key[z]$
8 ▷ Если y и z содержат сопутствующую
9 ▷ информацию, обменять также и ее
- 10 $y \leftarrow z$
- 11 $z \leftarrow p[y]$

Процедура виконується максимум за час $O(\lg n)$.

- Видалення ключа.

BINOMIAL_HEAP_DELETE(H, x)

- 1 **BINOMIAL_HEAP_DECREASE_KEY($H, x, -\infty$)**
- 2 **BINOMIAL_HEAP_EXTRACT_MIN(H)**

Вважаємо, що жоден ключ не може містити ключ $-\infty$. Ключ у вузлі для видалення робиться $-\infty$. При цьому розглянута вершина стає одним з коренів, мінімальним, і може бути вилучена за допомогою процедури **BINOMIAL_HEAP_EXTRACT_MIN**.

Сумарний час виконання складе $O(\lg n)$

Амортизаційний аналіз (amortized analysis)

- Амортизаційний аналіз – метод підрахунку часу, необхідного для виконання послідовності операцій над структурою даних.
- Час усереднюється по всіх виконуваних операціях, і аналізується середня продуктивність операцій в гіршому випадку.
- Використовується, щоб показати, що навіть якщо деякі з операцій послідовності є дорогими, то при усередненні по всіх операціях середня їх вартість буде невеликою за рахунок того, що ці операції нечасто зустрічаються.
- Отримана оцінка не є ймовірнісною: це оцінка середнього часу виконання операцій для найгіршого випадку.

Основні методи амортизаційного аналізу:

- метод усереднення (метод групового аналізу, *aggregate analysis*);
- метод передплати (метод бухгалтерського обліку, *accounting method*);
- метод потенціалів (*potential method*)

Груповий аналіз

- Якщо в найгіршому випадку загальний час виконання послідовності всіх n операцій сумарно дорівнює $T(n)$, то в найгіршому випадку **середня** (або **амортизована**) ціна однієї операції визначається співвідношенням $T(n)/n$.
- Знайдена амортизована вартість може бути застосована до всіх операцій послідовності, навіть якщо вони різноманітні.

Якою буде вартість послідовності з n операцій PUSH, POP та MULTIPOP за умови порожнього спочатку стека?

В найгіршому випадку вартість операції MULTIPOP $O(n)$, бо в стеку не більше n об'єктів. Це є верхньою границею часу роботи довільної стекової операції. Тоді в найгіршому випадку вартість послідовності з n операцій складе $O(n^2)$. Знайдена таким чином оцінка занадто груба. Розглядалася найгірша вартість кожної операції **окремо**. Проаналізуємо границю часу виконання послідовності з n операцій в найгіршому випадку.

Якою буде вартість послідовності з n операцій PUSH, POP та MULTIPOP за умови порожнього спочатку стека? Перед тим, як щось дістати зі стека, його треба туди помістити. Сумарна кількість операцій POP (включаючи виклики з MULTIPOP) для непорожнього стека не може перевищити кількість операцій PUSH, яких буде не більше n . Для будь-якого n послідовність з n операцій PUSH, POP та MULTIPOP займе час $O(n)$. Тому середня вартість кожної з операцій буде $O(n) / n = O(1)$.

Метод бухгалтерського обліку

- Кожна операція має свою нараховану амортизовану вартість та фактичну вартість.
- *Кредит* – додатна різниця між амортизованою та фактичною вартістю операції.
- Наявний кредит можна використати на покриття від'ємної різниці між

амортизованою та фактичною вартістю інших операцій.

- Для всіх послідовностей операцій їх повна амортизована вартість має бути верхньою границею їх повної фактичної вартості, а повний кредит завжди невід'ємним.

Фактичні вартості операцій для розглянутого стеку:

PUSH: 1

POP: 1

MULTIPOP: $\min(k, s)$, де k – аргумент процедури, s – поточна кількість елементів у стеку.

Візьмемо наступні амортизовані вартості:

PUSH: 2

POP: 0

MULTIPOP: 0.

В нашому випадку всі амортизовані вартості дорівнюють $O(1)$.

При цьому фактична вартість MULTIPOP – величина змінна.

Покажемо, що довільну послідовність стекових операцій можна оплатити через нарахування амортизованих вартостей. Уявімо стек як стос тарілок у кафе. При додаванні тарілки 1 грошова одиниця витрачається на оплату самої операції PUSH, а ще 1 грошова одиниця (з нарахованих 2-х) залишається в запасі, ніби на тарілці. В довільний момент часу кожній тарілці стека відповідає 1 грошова одиниця кредиту. Запасна одиниця кредиту на тарілці піде на оплату її діставання зі стеку. На операцію POP плата не нараховується, а її фактична вартість оплачується кредитом. Завдяки невеликій переоцінці операції PUSH зникає необхідність нарахувань на операцію POP.

Покажемо, що довільну послідовність стекових операцій можна оплатити через нарахування амортизованих вартостей.

Уявімо стек як стос тарілок у кафе. При додаванні тарілки 1 грошова одиниця витрачається на оплату самої операції PUSH, а ще 1 грошова одиниця (з нарахованих 2-х) залишається в запасі, ніби на тарілці. В довільний момент часу кожній тарілці стека відповідає 1 грошова одиниця кредиту. Запасна одиниця кредиту на тарілці піде на оплату її діставання зі стеку. На операцію POP плата не нараховується, а її фактична вартість оплачується кредитом. Завдяки невеликій переоцінці операції PUSH зникає необхідність нарахувань на операцію POP.

Метод потенціалів

- Введемо для кожного стану структури даних D_i величину $\Phi(D_i)$ – *потенціал*. Спочатку потенціал дорівнює $\Phi(D_0)$, а після i -ї операції $\Phi(D_i)$.
- Амортизована вартість \hat{c}_i кожної операції c_i – її фактична вартість плюс приріст потенціалу в результаті її виконання.
- Тоді повна амортизована вартість n операцій

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0).$$

- Якщо Φ можна визначити так, що $\Phi(D_n) \geq \Phi(D_0)$, то повна амортизована вартість буде верхньою границею повної фактичної вартості. Якщо n невідоме: умова $\Phi(D_i) \geq \Phi(D_0)$ для всіх i (напр. $\Phi(D_0) = 0$)

Для розглянутого стеку визначимо функцію потенціалу Φ як кількість елементів в цьому стеку.

Для порожнього стека $\Phi(D_0) = 0$. Оскільки кількість об'єктів у стеку невід'ємна, то $\Phi(D_i) \geq 0 = \Phi(D_0)$ для всіх i .

Нехай на i -му кроці стек містить s елементів.

PUSH: різниця потенціалів $\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$;

амортизована вартість $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+1 = 2$.

POP: різниця потенціалів $\Phi(D_i) - \Phi(D_{i-1}) = -1$;

амортизована вартість $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1-1 = 0$.

MULTIPOP: видалили $r = \min(k, s)$ елементів;

різниця потенціалів $\Phi(D_i) - \Phi(D_{i-1}) = -r$;

амортизована вартість $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = r-r = 0$.

Для всіх операцій амортизовані вартості дорівнюють $O(1)$.

Тому повна амортизована вартість для n операцій буде $O(n)$. 47

ЛЕКЦІЯ 6

Піраміди Фібоначчі

- Слабша структура, ніж у біноміальної піраміди.
- Амортизований час операцій, що не використовують видалення, дорівнює $O(1)$.
- Можуть виявлятися корисними в алгоритмах, де частка операцій EXTRACT_MIN та DELETE відносно мала (ряд алгоритмів на графах: наприклад, пошук мінімального кістякового дерева, найкоротший шлях з однієї вершини, алгоритми, що використовують DECREASE_KEY для кожного ребра в майже повних графах).
- Складність реалізації, порівняно з бінарними (та k -арними) пірамідами, вища.
- Значення констант у формулах часу виконання також високі.

- Невпорядкований набір дерев з коренем, кожне з яких є незростаючою пірамідою.
- Кожен вузол x містить вказівник на батька $p[x]$ і вказівник на одного з синів $child[x]$.
- Дочірні вузли вершини об'єднані в двозв'язний циклічний список (*child list*). (Операції видалення елемента та об'єднання двох таких списків займають константний час.)
- Кожен дочірній вузол у має вказівники на лівого та правого своїх братів: $left[y]$, $right[y]$. Порядок братських вузлів довільний.
- Кожен вузол містить поле кількості синів $degree[x]$ та логічне поле $mark[x]$ – ознаку наявності втрат дочірніх вузлів з моменту, коли x сам став дочірнім вузлом.
- Значення $mark[x]$ для новостворених вузлів FALSE; наявна мітка знімається, коли вузол стає дочірнім.
- Всі корені дерев також зв'язані в двозв'язний циклічний список (*root list*). Звернення до піраміди H іде через корінь дерева з мінімальним ключем $min[H]$ (мінімальний вузол).
- Кількість вузлів піраміди зберігається в $n[H]$.

Піраміди Фібоначчі: функція потенціалу

- Для піраміди Фібоначчі H позначимо

$$t(H) \text{ -- кількість дерев у списку коренів } H,$$

$$m(H) \text{ -- кількість вузлів з мітками в піраміді } H.$$

Тоді потенціал піраміди визначається як

$$\Phi(H) = t(H) + 2m(H).$$

- Потенціал множини пірамід – сума потенціалів пірамід-складників.
- Вважаємо, що одиниці потенціалу достатньо для покриття вартості довільної операції часу $O(1)$.
 - На початку роботи піраміда порожня, тобто початковий потенціал дорівнює 0. Тоді в подальшому потенціал завжди буде залишатися невід'ємним.
 - Верхня границя загальної амортизованої вартості є верхньою границею загальної фактичної вартості послідовності операцій.

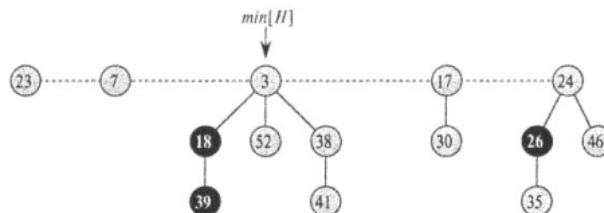
Верхня границя $D(n)$ максимальної степені вузла піраміди Фібоначчі з n вузлів:

- піраміди без підтримки DECREASE_KEY та DELETE:

$$D(n) \leq \lfloor \lg n \rfloor$$

- з підтримкою DECREASE_KEY та DELETE:

$$D(n) = O(\lg n)$$



Потенціал піраміди: $5 + 2 \cdot 3 = 11$.

9

Операції над пірамідами злиття

- Якщо підтримуються лише операції MAKE_HEAP, INSERT, MINIMUM, EXTRACT_MIN та UNION, кожна піраміда Фібоначчі буде набором *невпорядкованих біноміальних дерев* (unordered binomial tree):

- невпорядковане біноміальне дерево U_0 складається з єдиного вузла;
- невпорядковане біноміальне дерево U_k складається з двох невпорядкованих біноміальних дерев U_{k-1} , причому корінь одного з них є довільним дочірнім вузлом іншого.

- Виконується лема про властивості біноміальних дерев з заміною пункту 4 на 4^* (буде далі).

- Особливість пірамід Фібоначчі: об'єднання дерев у піраміді буде відбуватися лише під час операції EXTRACT_MIN.

Лема

(властивості невпорядкованих біноміальних дерев)

Невпорядковане біноміальне дерево U_k

1. має 2^k вузлів;

2. має висоту k ;

3. має $\binom{k}{i}$ вузлів на глибині $i = 0, 1, \dots, k$;

4* має корінь степеня k , а степінь інших вузлів буде менша; при цьому синами кореня є корені піддерев U_0, U_1, \dots, U_{k-1} в деякому порядку.

- Для піраміди Фібоначчі з n вузлами максимальна степінь вузла $D(n) = \lfloor \lg n \rfloor$.

- Створення піраміди Фібоначчі MAKE_FIB_HEAP

Повертає нову порожню піраміду Фібоначчі H з $n[H]=0$ та $min[H]=NIL$.

Маємо $t(H)=0$ та $m(H)=0$, тому потенціал $\Phi(H)=0$.

Амортизована вартість процедури MAKE_FIB_HEAP дорівнює її фактичній вартості $O(1)$.

- Вставка вузла FIB_HEAP_INSERT

Вузол x вже готовий і має заповнене поле $key[x]$.

FIB_HEAP_INSERT(H, x)

```

1   $degree[x] \leftarrow 0$ 
2   $p[x] \leftarrow NIL$       6   $mark[x] \leftarrow \text{FALSE}$ 
3   $child[x] \leftarrow NIL$  7  Присоединение списка корней, содержащего  $x$ , к списку корней  $H$ 
4   $left[x] \leftarrow x$      8  if  $min[H] = NIL$  или  $key[x] < key[min[H]]$ 
5   $right[x] \leftarrow x$    9    then  $min[H] \leftarrow x$ 
10  $n[H] \leftarrow n[H] + 1$ 
```

- Вставка вузла FIB_HEAP_INSERT (далі)

Процедура не намагається об'єднати дерева в піраміді. Послідовне виконання FIB_HEAP_INSERT k разів призведе до додавання до списку коренів k дерев з одного вузла.

Додавання вузла відбувається за постійний час $O(1)$.

Амортизована вартість процедури.

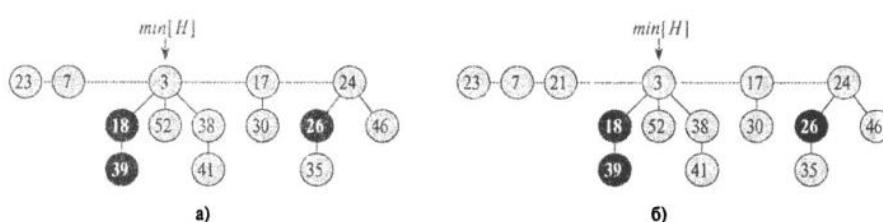
Нехай отримуємо з піраміди H піраміду H^* .

$t(H^*) = t(H) + 1$, $m(H^*) = m(H)$.

Збільшення потенціалу

$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$.

Фактична вартість дорівнює $O(1)$, тому амортизована вартість буде $O(1) + 1 = O(1)$.



Вставка вузла з ключем 21 в піраміду Фібоначчі

- Пошук мінімального вузла

На нього вказує $min[H]$, тому пошук відбувається за час $O(1)$. Потенціал H не змінюється, тому амортизована вартість рівна фактичній $O(1)$.

- Об'єднання двох пірамід FIB HEAP UNION

Списки коренів пірамід H_1 та H_2 просто об'єднуються і шукається новий мінімальний вузол.

```
FIB_HEAP_UNION( $H_1, H_2$ )
1  $H \leftarrow \text{MAKE\_FIB\_HEAP}()$ 
2  $min[H] \leftarrow min[H_1]$ 
3 Добавление списка корней  $H_2$  к списку корней  $H$ 
4 if ( $min[H_1] = \text{NIL}$ ) или ( $min[H_2] \neq \text{NIL}$  и  $key[min[H_2]] < key[min[H_1]]$ )
5   then  $min[H] \leftarrow min[H_2]$ 
6  $n[H] \leftarrow n[H_1] + n[H_2]$ 
7 Освобождение объектов  $H_1$  и  $H_2$ 
8 return  $H$ 
```

- Об'єднання двох пірамід FIB HEAP UNION (далі)

Знову ж таки, об'єднання дерев відсутнє.

Амортизована вартість процедурі.

$$t(H) = t(H_1) + t(H_2),$$
$$m(H) = m(H_1) + m(H_2).$$

Зміна потенціалу дорівнює

$$\begin{aligned} \Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= \\ &= (t(H) + 2m(H)) - \\ &\quad - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) = \\ &= 0. \end{aligned}$$

Отже, амортизована вартість рівна фактичній $O(1)$.

- Видалення мінімального вузла

```
FIB_HEAP_EXTRACT_MIN( $H$ )
1  $z \leftarrow min[H]$ 
2 if  $z \neq \text{NIL}$ 
3   then for (для) кожного дочернього по отношению к  $z$  узла  $x$ 
4     do Добавить  $x$  в список корней  $H$ 
5      $p[x] \leftarrow \text{NIL}$ 
6   Удалить  $z$  из списка корней  $H$ 
7   if  $z = right[z]$ 
8     then  $min[H] \leftarrow \text{NIL}$ 
9     else  $min[H] \leftarrow right[z]$ 
10    CONSOLIDATE( $H$ )
11     $n[H] \leftarrow n[H] - 1$ 
12 return  $z$ 
```

Спочатку всі дочірні вузли мінімального вузла переміщуються в список коренів піраміди, який потім ущільнюється процедурою CONSOLIDATE, щоб не було коренів однакової ступені.

• Видалення мінімального вузла (далі)

Ущільнення полягає у повторному виконанні наступних кроків, поки всі корені в списку не матимуть різні значення поля $degree$.

1. Знайти два корені x та y у однакової степені, причому $key[x] \leq key[y]$.
2. Прив'язати (link) y до x : видалити y зі списку коренів та зробити сином x . Поле $degree[x]$ при цьому збільшується, а мітка $mark[y]$, якщо вона була, знімається.

```
FIB_HEAP_LINK( $H, y, x$ )
1 Удалить  $y$  из списка корней  $H$ 
2 Сделать  $y$  дочерним узлом  $x$ , увеличить  $degree[x]$ 
3  $mark[y] \leftarrow \text{FALSE}$ 
```

• Видалення мінімального вузла (далі)

Процедура CONSOLIDATE використовує допоміжний масив $A[0..D(n(H))]$. Якщо $A[i]=y$, то у в даний момент є коренем степені $degree[y]=i$.

По завершенні циклу **for** в списку коренів залишиться не більше одного кореня кожної степені, і елементи масиву A вказуватимуть на ці корені.

Якщо перед викликом FIB_HEAP_EXTRACT_MIN всі дерева піраміди були невпорядкованими біноміальними деревами, то після виконання процедури вони такими і залишаться. Шляхи отримання нових дерев:

- дочірні дерева вузла, що видаляється (вони за умовою є невпорядкованими біноміальними деревами);
- дерево, отримане об'єднанням двох дерев однакової степені (якщо вони мають структуру U_k , то їх об'єднання процедурою FIB_HEAP_LINK дає дерево типу U_{k+1}).

19

• Видалення мінімального вузла (далі)

```
CONSOLIDATE( $H$ )
1 for  $i \leftarrow 0$  to  $D(n[H])$ 
2   do  $A[i] \leftarrow \text{NIL}$ 
3   for (для) кожного узла  $w$  в списке корней  $H$ 
4     do  $x \leftarrow w$ 
5      $d \leftarrow degree[x]$ 
6     while  $A[d] \neq \text{NIL}$ 
7       do  $y \leftarrow A[d]$             $\triangleright$  Узел с той же степенью, что и у  $x$ .
8         if  $key[x] > key[y]$ 
9           then обменять  $x \leftrightarrow y$ 
10          FIB_HEAP_LINK( $H, y, x$ )
11           $A[d] \leftarrow \text{NIL}$ 
12           $d \leftarrow d + 1$ 
13           $A[d] \leftarrow x$ 
14  $min[H] \leftarrow \text{NIL}$ 
15 for  $i \leftarrow 0$  to  $D(n[H])$ 
16   do if  $A[i] \neq \text{NIL}$ 
17     then Добавить  $A[i]$  в список корней  $H$ 
18     .   if  $min[H] = \text{NIL}$  или  $key[A[i]] < key[min[H]]$ 
19       then  $min[H] \leftarrow A[i]$ 
```

- Видалення мінімального вузла (далі)

Розглянемо цикл **while** (рядки 6-12).

```
while A[d] ≠ NIL
  do y ← A[d]           ▷ Узел с той же степенью, что и у x.
    if key[x] > key[y]
      then обменять x ↔ y
    Fib_HEAP_LINK(H, y, x)
    A[d] ← NIL
    d ← d + 1
```

Він зв'язує корінь x дерева, яке містить вузол y , з іншим деревом степені, що співпадає зі степінню x . Це робиться, доки жоден інший корінь не матиме степінь, як у кореня x .

Інваріант циклу **while**:

На початку кожної ітерації $d = \text{degree}[x]$.

Скориставшись ним, доведемо коректність алгоритму.

- Видалення мінімального вузла (далі)

Ініціалізація. Рядок 5 гарантує виконання інваріанту.

Збереження. В кожній ітерації $A[d]$ вказує на деякий корінь y . Оскільки $d = \text{degree}[x] = \text{degree}[y]$, треба зв'язати x та y . Батьком стає вузол з меншим ключем (за необхідності відбувається обмін значень x та y в рядках 8-9).

Потім y прив'язується до x при виклику `FIB_HEAP_LINK(H, y, x)`. Цей виклик збільшує $\text{degree}[x]$.

З масиву A видаляється посилання на y .

В рядку 12 здійснюється відновлення інваріанту $d = \text{degree}[x]$.

Завершення. Цикл виконується, поки не отримується $A[d]=\text{NIL}$, так що не буде коренів такої ж степені, як в x .

- Видалення мінімального вузла (далі)

Амортизована вартість процедури

Нехай працюємо над пірамідою H з n вузлами.

Підрахуємо фактичну вартість.

Вклад $O(D(n))$ дає обробка максимум $D(n)$ дочірніх вузлів мінімального вузла в `FIB_HEAP_EXTRACT_MIN` та рядки 1-2 і 14-19 в `CONSOLIDATE`.

Розмір списку при виклику `CONSOLIDATE` не перевищує $(t(H)+D(n)-1)$. При кожному виконанні циклу **while** один з коренів зв'язується з іншим, тому загальний час роботи циклу **for** буде обмеженим згори $(t(H)+D(n))$.

Отже, загальний фактичний час дорівнює $O(t(H)+D(n))$.

• Видалення мінімального вузла (далі)

Потенціал до вилучення мінімального вузла $t(H) + 2m(H)$, а після – не перевищує $(D(n)+1) + 2m(H)$, бо залишається не більше $(D(n)+1)$ коренів і нових міток не утворюється.

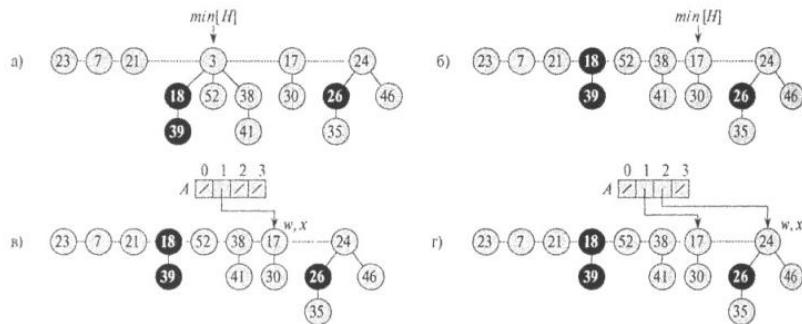
Верхня оцінка амортизованої вартості:

$$\begin{aligned} O(t(H) + D(n)) + ((D(n)+1) + 2m(H)) - (t(H) + 2m(H)) = \\ = O(D(n)) + O(t(H)) - t(H) = O(D(n)). \end{aligned}$$

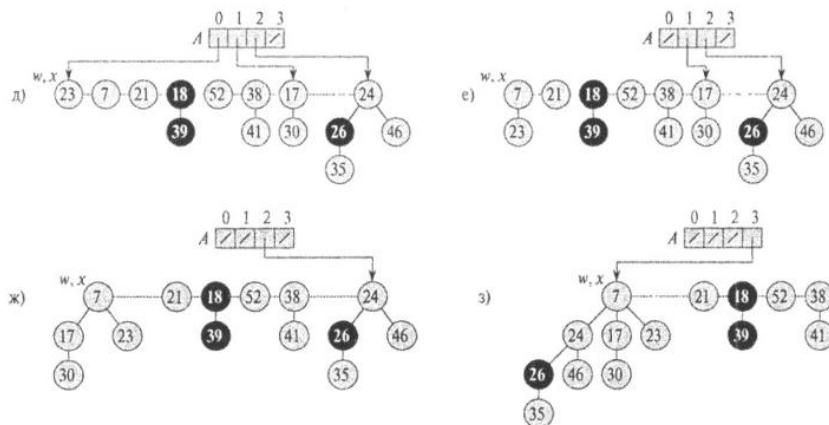
Остання рівність справедлива, оскільки можна масштабувати одиниці потенціалу так, щоб захтувати константою, прихованою в $O(t(H))$.

Далі покажемо, що $D(n) = O(\lg n)$, тому амортизована вартість процедури складе $O(\lg n)$.

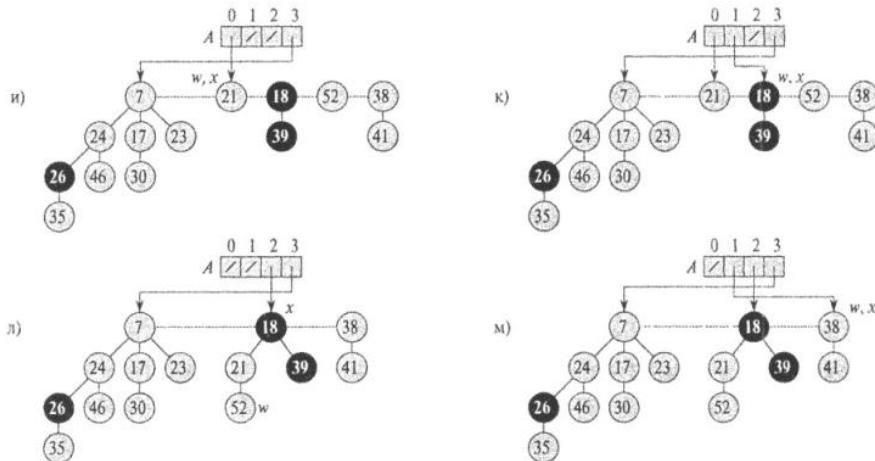
Приклад видалення мінімального вузла



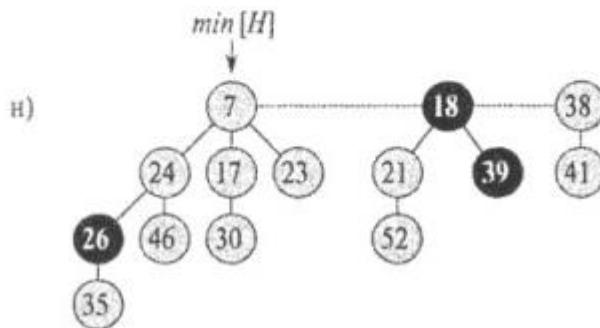
Приклад видалення мінімального вузла (далі)



Приклад видалення мінімального вузла (продовження)



Приклад видалення мінімального вузла (завершальний етап)



Додаткові операції над пірамідами злиття

• Зменшення ключа

Операція порушує властивість того, що піраміда Фібоначчі складається з невпорядкованих біноміальних дерев (втім, вони близькі до них).

Максимальна степінь $D(n)$ матиме порядок $O(\lg n)$.

```

FIB_HEAP_DECREASE_KEY( $H, x, k$ )
1  if  $k > key[x]$ 
2    then error "Новый ключ больше текущего"
3   $key[x] \leftarrow k$ 
4   $y \leftarrow p[x]$ 
5  if  $y \neq \text{NIL}$  и  $key[x] < key[y]$ 
6    then CUT( $H, x, y$ )
7      CASCADING-CUT( $H, y$ )
8  if  $key[x] < key[min[H]]$ 
9    then  $min[H] \leftarrow x$ 

```

- Зменшення ключа (далі)

Якщо властивість піраміди в дереві порушена, відбувається операція вирізання і, можливо, каскадного вирізання.

CUT(H, x, y)

- 1 Удаление x из списка дочерних узлов y , уменьшение $degree[y]$
- 2 Добавление x в список корней H
- 3 $p[x] \leftarrow \text{NIL}$
- 4 $mark[x] \leftarrow \text{FALSE}$

CASCADING_CUT(H, y)

- 1 $z \leftarrow p[y]$
- 2 **if** $z \neq \text{NIL}$
 - 3 **then if** $mark[y] = \text{FALSE}$
 - 4 **then** $mark[y] \leftarrow \text{TRUE}$
 - 5 **else** CUT(H, y, z)
 - 6 CASCADING_CUT(H, z)

- Зменшення ключа (далі)

Вирізання означає, що утворюється нове дерево з коренем x .

Каскадне вирізання робить вершину новим коренем, якщо вона була помічена, і рекурсивно піднімається вище, або помічає її і зупиняється. Процес також зупиняється при досягненні кореня.

Тобто при каскадному видаленні вершина вирізатиметься, якщо вона перед цим втратила другого сина.

- Зменшення ключа (далі)

Таким чином, поле мітки $mark[x]$ буде змінюватися в таких випадках:

1. x став коренем (скидання мітки);
2. x прив'язали до іншого вузла (скидання мітки);
3. вирізали сина x (поява мітки).

В кінці процедури FIB_HEAP_DECREASE_KEY може відбутися оновлення мінімального вузла піраміди. Єдиний інший кандидат – перший вирізаний вузол.

- Зменшення ключа (далі)

Амортизована вартість процедури

Фактична вартість.

FIB_HEAP_DECREASE_KEY вимагає $O(1)$ часу плюс час на каскадне вирізання. Нехай CASCADING_CUT викликалося c разів, кожен з яких без врахування рекурсії вимагає час $O(1)$. Тобто вартість процедури зменшення ключа складе $O(c)$.

- Зменшення ключа (далі)

Амортизована вартість процедури

Зміна потенціалу.

Кожен рекурсивний виклик CASCADING_CUT, крім останнього, здійснює вирізання та скидає мітку. Після цього отримуємо $(t(H)+c)$ дерев (початкові $t(H)$, $(c-1)$ вирізані та одне з коренем x) та не більше $(t(H)-c+2)$ помічених вузла (каскадом зняли мітку в $(c-1)$ вузла та один можливо помітили в кінці).

- Зменшення ключа (далі)

Амортизована вартість процедури

Зміна потенціалу складе

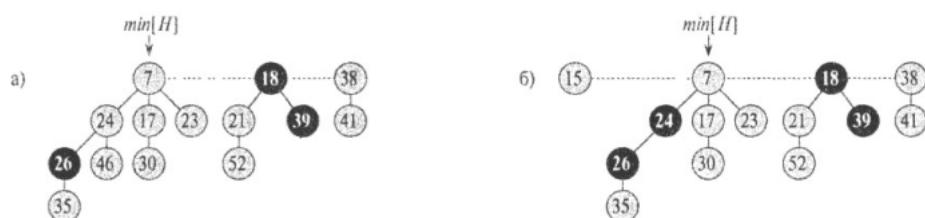
$$((t(H)+c)+2(t(H)-c+2)) - (t(H)+2t(H)) = 4-c.$$

Амортизована вартість не перевищить $O(c) + 4 - c = O(1)$.

(Можна відповідно масштабувати одиниці потенціалу для домінування над константою в $O(c)$.)

Коли мічений вузол видаляється в процесі каскадного вирізання, скидання мітки призводить до зменшення потенціалу на 2: одна одиниця оплачує вирізання і скидання мітки, друга компенсує збільшення потенціалу через появу нового кореня. Тому функція потенціалу містить як член подвоєну кількість помічених вузлів.

Приклад зменшення ключа 46 до 15



• Видалення вузла

```

FIB_HEAP_DELETE( $H, x$ )
1 FIB_HEAP_DECREASE_KEY( $H, x, -\infty$ )
2 FIB_HEAP_EXTRACT_MIN( $H$ )

```

Вважається, що піраміда не містить ключів зі значенням $-\infty$.

Процедура `FIB_HEAP_DELETE` цілком аналогічна `BINOMIAL_HEAP_DELETE`: робить x мінімальним значенням піраміди і видає його.

Амортизований час – сума амортизованого часу процедури зменшення ключа $O(1)$ та амортизованого часу роботи процедури видалення мінімуму $O(D(n))$, тобто він складе $O(\lg n)$.

Оцінка максимальної степені вузла

Покажемо, що максимальна степінь $D(n)$ матиме порядок $O(\lg n)$ за умови вирізання вузла, як тільки він втратив двох синів.

Для кожного вузла x визначимо $\text{size}(x)$ – кількість вузлів у піддереві з коренем x , включно з ним.

Лема 1. Нехай x – довільний вузол піраміди Фібоначчі та y_1, \dots, y_k – дочірні вузли x в порядку їх зв'язування з x від ранніх до пізніх. Тоді $\text{degree}[y_1] \geq 0$ та $\text{degree}[y_i] \geq i - 2$ при $i=2,3,\dots,k$.

Доведення. Очевидно $\text{degree}[y_1] \geq 0$.

Для $i \geq 2$ при зв'язуванні y_i з x всі вузли y_1, y_2, \dots, y_{i-1} вже є дочірніми для x , тому $\text{degree}[x] \geq i - 1$. Зв'язування y_i з x можливе лише за умови $\text{degree}[x] = \text{degree}[y_i]$, тому на момент зв'язування $\text{degree}[y_i] \geq i - 1$. Після цього y_i міг втратити не більше одного сина (інакше його вже би вирізали). Тому $\text{degree}[y_i] \geq i - 2$.

к-те число Фібоначчі: $F_k = \begin{cases} 0 & \text{при } k = 0, \\ 1 & \text{при } k = 1, \\ F_{k-1} + F_{k-2} & \text{при } k \geq 2. \end{cases}$

Лема 2. Для всіх цілих $k \geq 0$: $F_{k+2} = 1 + \sum_{i=0}^k F_i$.

Доведення. За математичною індукцією по k .

При $k = 0$:

$$1 + \sum_{i=0}^0 F_i = 1 + F_0 = 1 + 0 = 1 = F_2.$$

Припустимо $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$. Тоді

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^k F_i.$$

Лема 3. Нехай x – довільний вузол піраміди Фібоначчі, а $k = \text{degree}[x]$ – його степінь.

Тоді $\text{size}(x) \geq F_{k+2} \geq \varphi^k$, де $\varphi = (1 + \sqrt{5})/2$.

Наслідок. Максимальна степінь $D(n)$ довільного вузла в піраміді Фібоначчі з n вузлами дорівнює $O(\lg n)$.

Доведення. Нехай x – довільний вузол в піраміді Фібоначчі з n вузлами і нехай $k = \text{degree}[x]$. За лемою 3, $n \geq \text{size}(x) \geq \varphi^k$. Прологарифмуємо за основою φ :

$$k \leq \log_{\varphi} n.$$

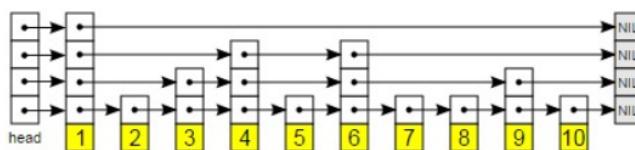
Тому максимальна степінь $D(n)$ довільного вузла дорівнює $O(\lg n)$.

Зauważення. В дійсності k є цілим числом, тому справедливою буде оцінка

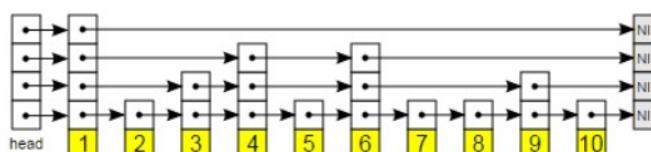
$$k \leq \lfloor \log_{\varphi} n \rfloor.$$

Список з пропусками (Skip List)

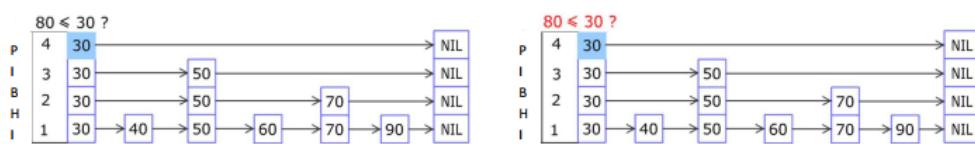
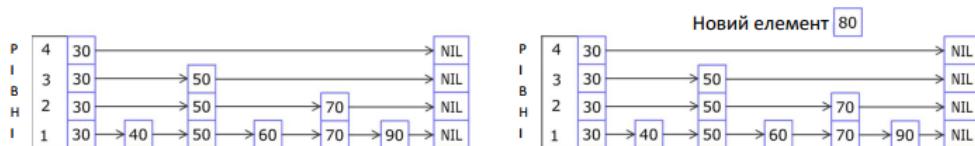
- Ймовірнісна структура даних, в основі якої декілька паралельних відсортованих зв'язаних списки (з пропущеними вузлами).
- Основна ідея – реалізація бінарного пошуку для зв'язаних списків.
- Вставка, пошук і видалення виконуються за логарифмічний випадковий час.
- Елементи, що використовуються в списку з пропусками, можуть мати більше одного вказівника і відповідно належати до більш ніж одного списку.

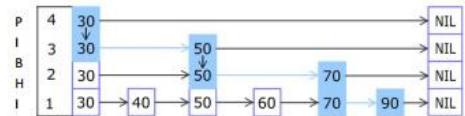
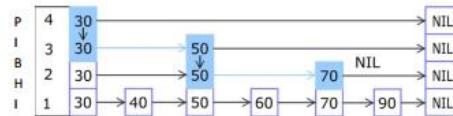
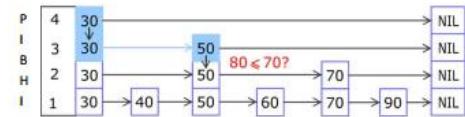
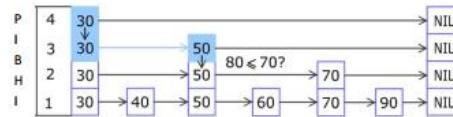
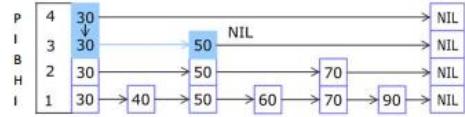
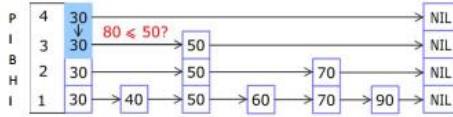
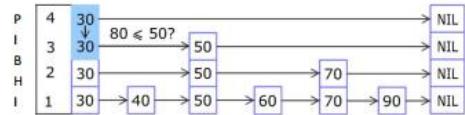
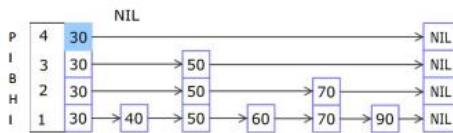


- На нижньому рівні – звичайний впорядкований зв'язний список, що містить всі вузли (з імовірністю 1).
- Вузол з кожного рівня i присутній в шарі $(i+1)$ з фіксованою імовірністю p (найчастіше $1/2$ чи $1/4$).
- В середньому кожен елемент зустрічається в $1/(1-p)$ списках, а верхній елемент (зазвичай особливий елемент на початку списку з пропусками) – в $\log_{1/p} n$ списках.

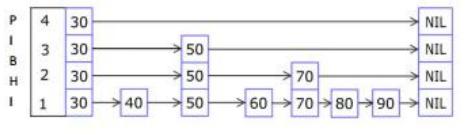
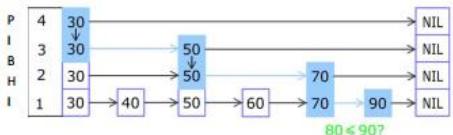


- Пошук починається з головного елемента верхнього списку і продовжується горизонтально, поки поточний елемент \leq цільового.
- Якщо елемент знайшли, пошук завершується; якщо поточний елемент більший ніж цільовий або пошук досяг кінця списку в шарі, процедуру повторюють після повернення до попереднього елемента і спуску на один рівень нижче.
- Очікуване число кроків при пошуку елемента становить $1/p$.
- Вибір різних значень для p дає можливість досягти необхідного балансу між швидкістю пошуку і затратами пам'яті на зберігання списку.
- Вставка* нового елемента робиться так.
 - За допомогою алгоритму пошуку знаходиться позиція вставки елемента в нижньому списку.
 - Елемент вставляється.
 - «Підкидається монетка» і в залежності від результату елемент проштовхується на рівень вище.
 - Попередній крок повторюється, поки «підкидання монетки» дає позитивний результат.
- Видалення* елемента відбувається елементарно: елемент знаходиться і видаляється з усіх рівнів.

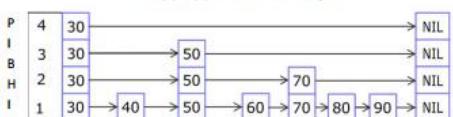




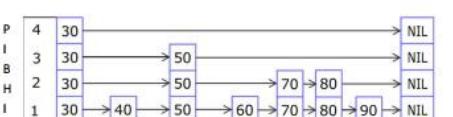
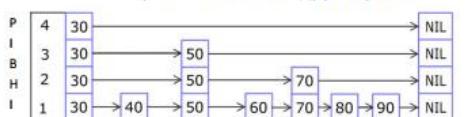
80 <= 90?



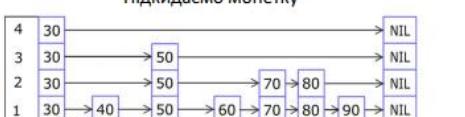
Підкидаємо монетку



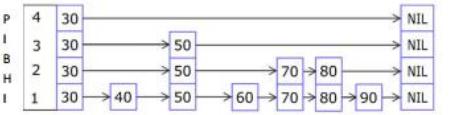
Орел \Rightarrow вставка на другий рівень



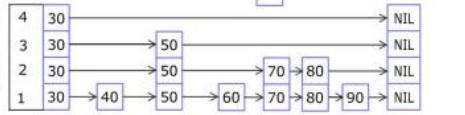
Підкидаємо монетку



Решка \Rightarrow вставка елемента завершена



Новий елемент 45





ЛЕКЦІЯ 7

Реалізації черг 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.
 - 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 – вЕВ-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 $\{\{0,1,\dots,(u-1)\}\}$.
- Тоді час виконання операцій складе $O(\log \log u)$.

4

Попередні підходи

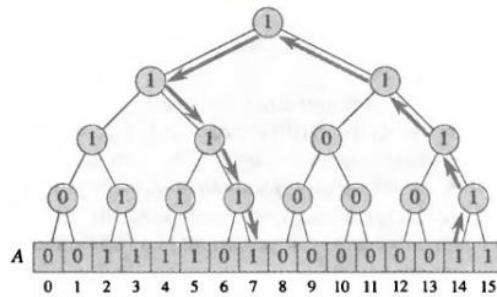
Розглянемо деякі підходи до зберігання динамічної множини, які приведуть до ідей в основі дерев ван Емде Боаса.

Пряма адресація

- Найпростіший варіант – динамічна множина $\{0,1,\dots,(u-1)\}$ 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.

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 $i = 16$.

За стрілками – пошук попередника ключа 14 у множині.

Пошук використовує структуру дерева.

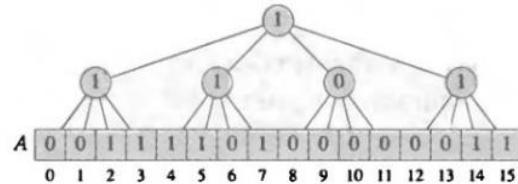
- Мінімум: рух від кореня до листа, завжди вибираючи *найлівіший* вузол, що містить 1.
- Максимум: рух від кореня до листа, завжди вибираючи *найправіший* вузол, що містить 1.
- Наступник x : спочатку рух вгору від листа x , поки не зайдемо зліва у вузол, *правий син* якого з містить 1; потім пошук мінімуму в піддереві z .
- Попередник x : спочатку рух вгору від листа x , поки не зайдемо справа у вузол, *лівий син* якого з містить 1; потім пошук максимуму в піддереві z .
- Вставка: зберігається 1 у кожному вузлі на простому шляху від вставленого у лист значення додори 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 $\log i$, тому їх найгірша часова оцінка $O(\log i)$.

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 i , червоно-чорні дерева працюватимуть швидше на всіх інших операціях.

Накладання дерева константної висоти

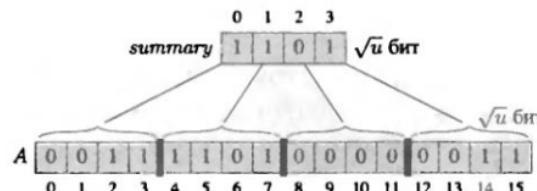
- Припустимо $u = 2^{2k}$ для деякого цілого k , що \sqrt{u} – ціле.
- Накладемо на бітовий вектор дерево степеня \sqrt{u} .
- Висота такого дерева завжди дорівнюватиме 2.



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.

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



- Такий \sqrt{u} -бітовий підмасив A назовемо 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 $summary[\lfloor x/\sqrt{u} \rfloor]$.

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.

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 $u = 2^k$.)
- Спробуємо уявити, звідки можна отримати час роботи $O(\log \log u)$.
- Розглянемо рекурентне спiввiдношення
$$T(u) = T(\sqrt{u}) + O(1).$$
- 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.

- Для полегшення індексації визначимо функції:

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

дає $(\log u)/2$ старших бітів числа x ,

повертаючи номер кластера x ;

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

дає $(\log u)/2$ молодших бітів числа x ,

вказуючи позицію x в його кластері;

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

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 $x = \text{index}(\text{high}(x), \text{low}(y))$.

- За u береться розмір універсуму структури даних на поточному рівні.

- *Протоструктура ван Емде Боаса* (proto van Emde Boas structure, *proto-vEB*-структуря) служитиме основою для структури дерева ван Емде Боаса.

- Зберігає ключі з універсума $\{0, 1, \dots, (u-1)\}$.

- Кожен вузол дерева *proto-vEB* містить розмір універсуму u .

- При $u = 2$:

- базовий розмір, структура містить масив з двох бітів $A[0..1]$.

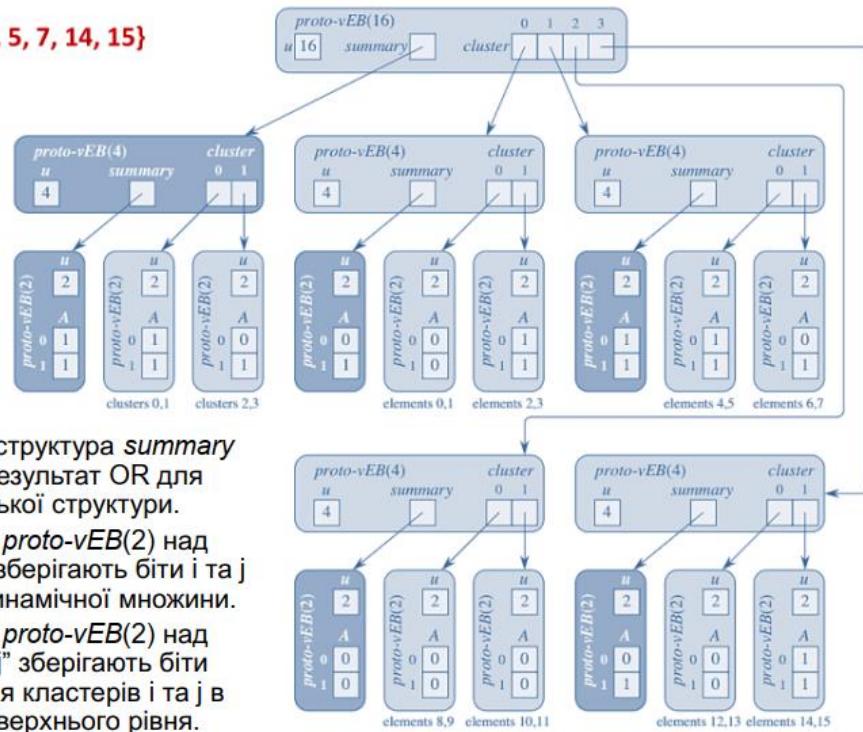
- Інакше:

- 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

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

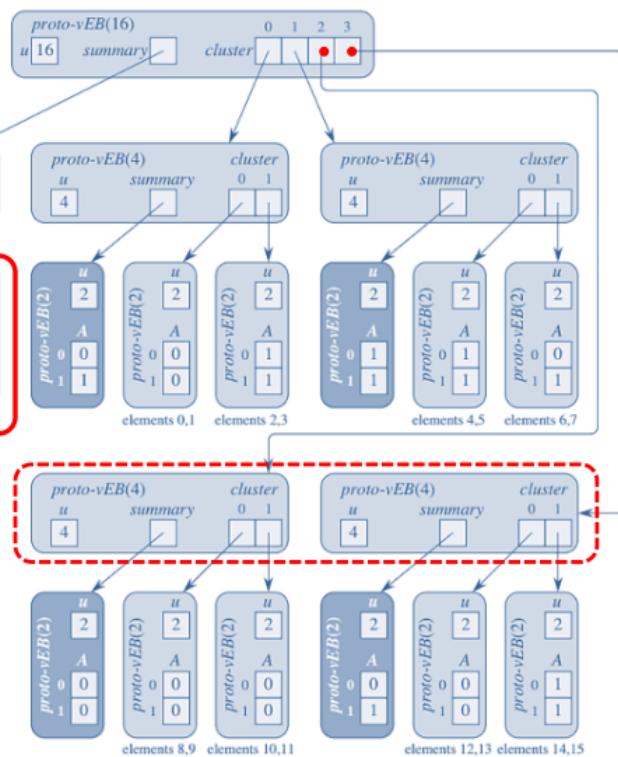


Резюмуюча структура *summary* зберігає результат OR для батьківської структури.

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.

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

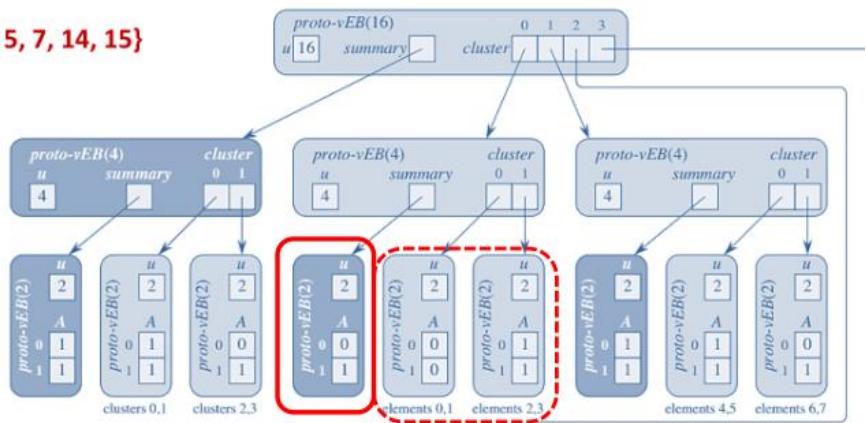


Структура *proto-vEB(2)* з поміткою "Clusters 2,3" має $A[0]=0$, тобто кластер 2 структури *proto-vEB(16)* містить тільки 0 (елементи 8-11).

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

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

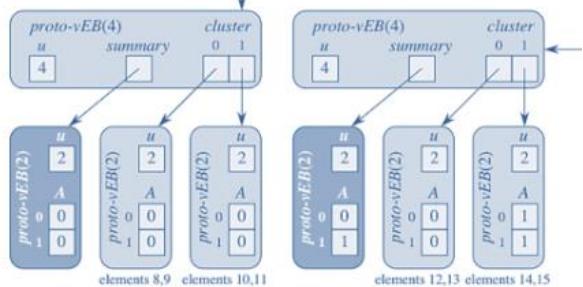
$$u = 16$$



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.



- *Перевірка наявності елемента в множині MEMBER.*

PROTO-vEB-MEMBER(V, x)

```

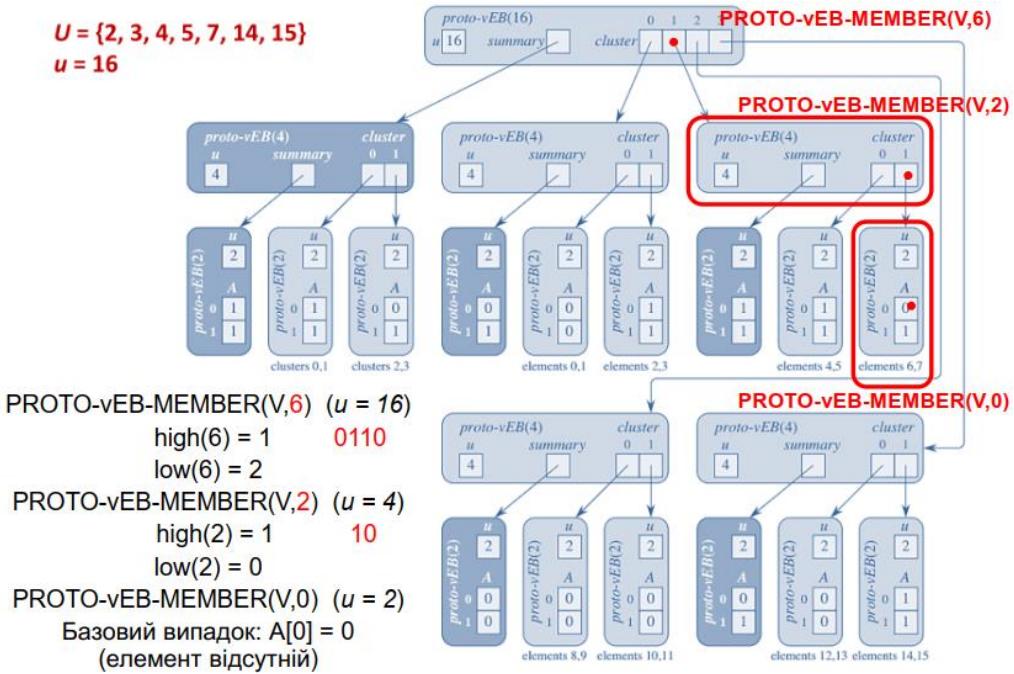
1  if  $V.u == 2$ 
2      return  $V.A[x]$ 
3  else return PROTO-vEB-MEMBER( $V.cluster[\text{high}(x)], \text{low}(x)$ )

```

Рядок 2 – базовий випадок, повертається відповідний біт масиву.

Інакше рекурсивний пошук у кластері номер $\text{high}(x)$ елемента зі значенням $\text{low}(x)$.

- 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 $O(\log \log u)$.



Операції над *proto-vEB*-структурами

- Пошук мінімального елемента **MINIMUM**.

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  $min-cluster = \text{PROTO-vEB-MINIMUM}(V.summary)$ 
8    if  $min-cluster == \text{NIL}$ 
9      return NIL
10   else  $offset = \text{PROTO-vEB-MINIMUM}(V.cluster[min-cluster])$ 
11   return  $\text{index}(min-cluster, offset)$ 

```

Операції над *proto-vEB*-структурами

Пошук мінімального елемента **MINIMUM** (далі)

- Спочатку безпосередньо перевіряється базовий випадок.
- Інакше в рядку 7 рекурсивно знаходитьться 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 $\Theta(\log u)$.

- Пошук наступника **SUCCESSOR**.

```

PROTO-vEB-SUCCESSOR( $V, x$ )
1  if  $V.u == 2$ 
2    if  $x == 0$  и  $V.A[1] == 1$ 
3      return 1
4    else return NIL
5  else  $offset = \text{PROTO-vEB-SUCCESSOR}(V.cluster[\text{high}(x)], \text{low}(x))$ 
6    if  $offset \neq \text{NIL}$ 
7      return  $\text{index}(\text{high}(x), offset)$ 
8    else  $\text{succ-cluster} = \text{PROTO-vEB-SUCCESSOR}(V.summary, \text{high}(x))$ 
9      if  $\text{succ-cluster} == \text{NIL}$ 
10     return NIL
11    else  $offset = \text{PROTO-vEB-MINIMUM}(V.cluster[\text{succ-cluster}])$ 
12    return  $\text{index}(\text{succ-cluster}, offset)$ 

```

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 спiввiдношення для найгiршого випадку:

$$T(u) = 2T(\sqrt{u}) + \Theta(\lg \sqrt{u}) = 2T(\sqrt{u}) + \Theta(\lg u).$$
- Його розв'язок $\Theta(\log u \log \log u)$.

- Вставка елемента **INSERT**.

```

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

```

- В базовому випадку вiдповiдний бiт масиву A стає 1.
- В рекурсивному випадку вставляємо x у вiдповiдний кластер та встановлюємо його резюмуючий бiт = 1.
- Рекурентне спiввiдношення для операцiї

$$T(u) = 2T(\sqrt{u}) + O(1).$$
- Розв'язок $\Theta(\log u)$.

- 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:

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

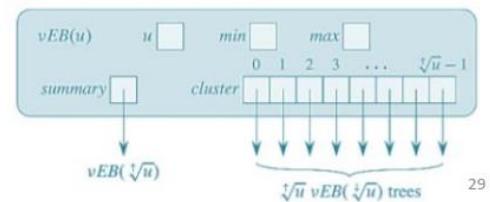
$$\text{low}(x) = x \bmod \downarrow\sqrt{u},$$

$$\text{index}(x, y) = x\downarrow\sqrt{u} + y.$$

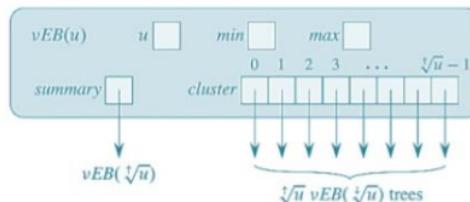
Дерево ван Емде Боаса

- Дерево ван Емде Боаса (van Emde Boas tree, vEB -дерево) матиме наступну структуру.

- Зберігає ключі з універсума $\{0, 1, \dots, (u-1)\}$ // $vEB(u)$
- Кожен вузол дерева vEB містить розмір u , значення мінімального елемента min та максимального елемента max .
- В небазовому випадку ($u > 2$):
 - вказівник *summary* на дерево $vEB(\sqrt{u})$;
 - масив *cluster*[0.. $\sqrt{u} - 1$] з \sqrt{u} вказівників на корені дерев $vEB(\sqrt{u})$.



29



- 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.

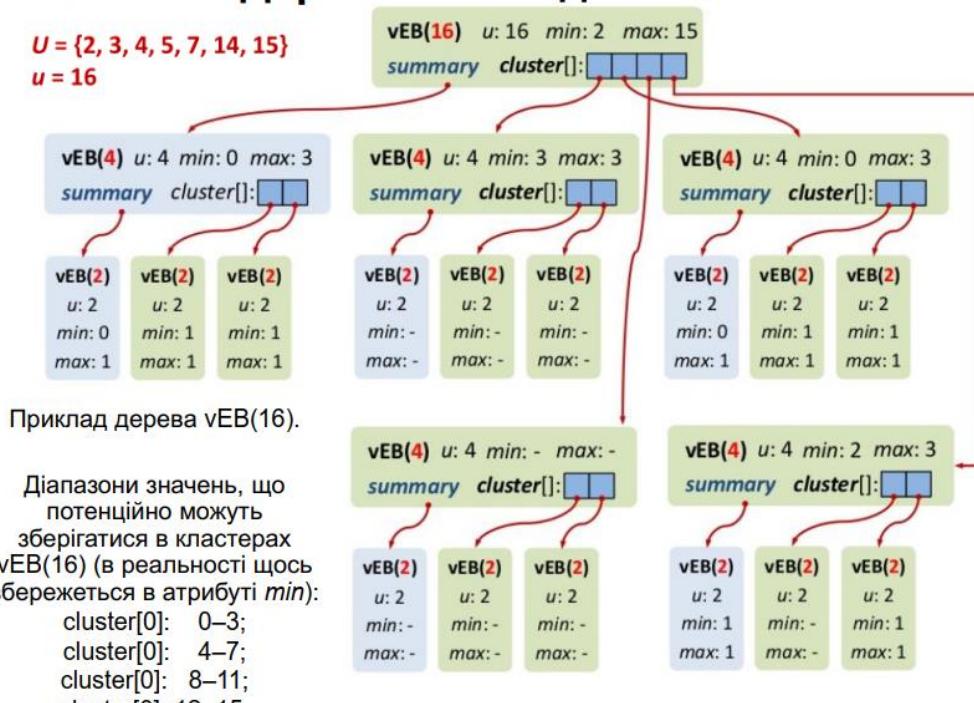
Атрибути *min* та *max* дозволяють зменшити кількість рекурсивних викликів при роботі з vEB-деревом.

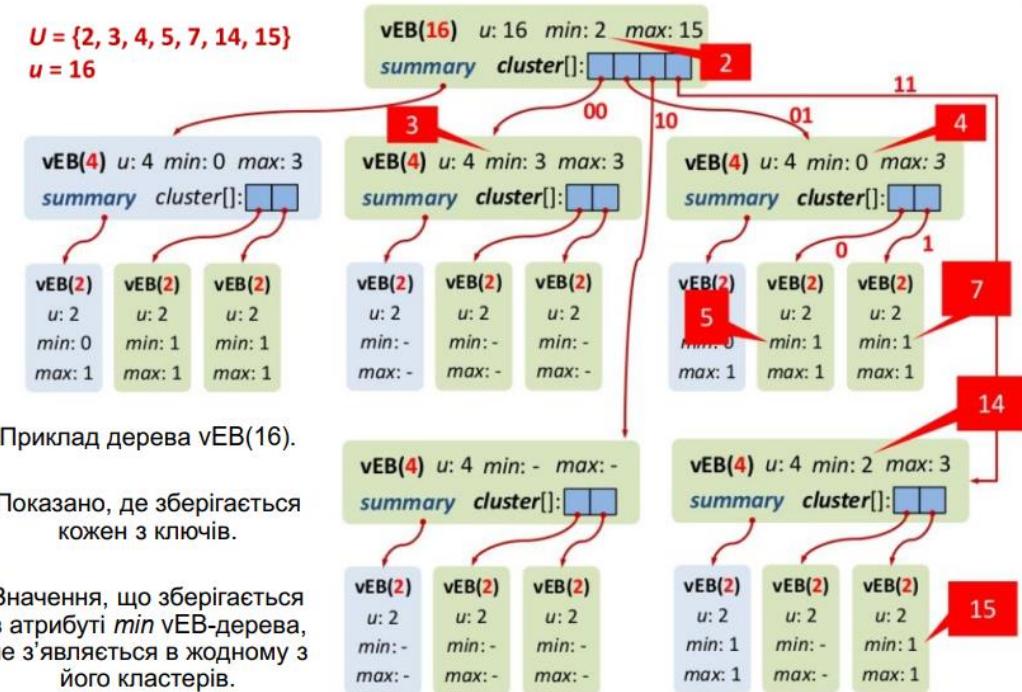
- Операції MINIMUM (MAXIMUM) повертають значення відповідних атрибутів.
- При визначенні наступника операцією SUCCESSOR можна позбутися рекурсії: наступник x буде міститися в цьому ж кластері, тільки якщо $x < \text{max}$ (аналогічно для PREDECESSOR та *min*).
- За значеннями *min* та *max* можна за 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.
- Всі рекурсивні процедури, які реалізують операції над vEB-деревом, описуються рекурентним співвідношенням

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

- 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 $\Theta(u)$.





34

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.

- Перевірка наявності елемента в множині MEMBER.

vEB-TREE-MEMBER(V, x)

1 if $x == V.\min$ или $x == V.\max$

2 **return** TRUE

3 **elseif** $V.u == 2$

4 **return** FALSE

5 **else return** VEB-TREE-MEMBER($V.\text{cluster}[\text{high}(x)]$, $\text{low}(x)$)

- 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.

- Пошук наступника SUCCESSOR.

VEB-TREE-SUCCESSOR(V, x)

```

1  if  $V.u == 2$ 
2    if  $x == 0$  и  $V.max == 1$ 
3      return 1
4    else return NIL
5  elseif  $V.min \neq \text{NIL}$  и  $x < V.min$ 
6    return  $V.min$ 
7  else  $max-low = \text{VEB-TREE-MAXIMUM}(V.cluster[\text{high}(x)])$ 
8    if  $max-low \neq \text{NIL}$  и  $\text{low}(x) < max-low$ 
9       $offset = \text{VEB-TREE-SUCCESSOR}(V.cluster[\text{high}(x)], \text{low}(x))$ 
10     return  $\text{index}(\text{high}(x), offset)$ 
11   else  $succ-cluster = \text{VEB-TREE-SUCCESSOR}(V.summary, \text{high}(x))$ 
12   if  $succ-cluster == \text{NIL}$ 
13     return NIL
14   else  $offset = \text{VEB-TREE-MINIMUM}(V.cluster[succ-cluster])$ 
15   return  $\text{index}(succ-cluster, offset)$ 

```

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.

- *Пошук попередника PREDECESSOR.*

```

vEB-TREE-PREDECESSOR( $V, x$ )
1  if  $V.u == 2$ 
2    if  $x == 1$  и  $V.min == 0$ 
3      return 0
4    else return NIL
5  elseif  $V.max \neq \text{NIL}$  и  $x > V.max$ 
6    return  $V.max$ 
7  else  $min-low = \text{vEB-TREE-MINIMUM}(V.cluster[\text{high}(x)])$ 
8    if  $min-low \neq \text{NIL}$  и  $\text{low}(x) > min-low$ 
9       $offset = \text{vEB-TREE-PREDECESSOR}(V.cluster[\text{high}(x)], \text{low}(x))$ 
10     return  $\text{index}(\text{high}(x), offset)$ 
11  else  $pred-cluster = \text{vEB-TREE-PREDECESSOR}(V.summary, \text{high}(x))$ 
12    if  $pred-cluster == \text{NIL}$ 
13      if  $V.min \neq \text{NIL}$  и  $x > V.min$ 
14        return  $V.min$ 
15      else return NIL
16    else  $offset = \text{vEB-TREE-MAXIMUM}(V.cluster[pred-cluster])$ 
17    return  $\text{index}(pred-cluster, offset)$ 

```

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 $O(\log \log u)$.

- Вставка елемента INSERT.

VEB-EMPTY-TREE-INSERT(V, x)

- 1 $V.\min = x$
- 2 $V.\max = x$

VEB-TREE-INSERT(V, x)

- 1 **if** $V.\min == \text{NIL}$
- 2 VEB-EMPTY-TREE-INSERT(V, x)
- 3 **else if** $x < V.\min$
 Обменять x с $V.\min$
- 4 **if** $V.u > 2$
 if VEB-TREE-MINIMUM($V.\text{cluster}[\text{high}(x)]$) == NIL
 7 VEB-TREE-INSERT($V.\text{summary}, \text{high}(x)$)
 8 VEB-EMPTY-TREE-INSERT($V.\text{cluster}[\text{high}(x)], \text{low}(x)$)
 9 **else** VEB-TREE-INSERT($V.\text{cluster}[\text{high}(x)], \text{low}(x)$)
- 10 **if** $x > V.\max$
 11 $V.\max = x$

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).

- *Видалення елемента DELETE.*

```

VEB-TREE-DELETE( $V, x$ )
1  if  $V.\min == V.\max$ 
2     $V.\min = \text{NIL}$ 
3     $V.\max = \text{NIL}$ 
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     $\text{first-cluster} = \text{VEB-TREE-MINIMUM}(V.\text{summary})$ 
11     $x = \text{index}(\text{first-cluster},$ 
         $\text{VEB-TREE-MINIMUM}(V.\text{cluster}[\text{first-cluster}]))$ 
12     $V.\min = x$ 
13     $\text{VEB-TREE-DELETE}(V.\text{cluster}[\text{high}(x)], \text{low}(x))$ 
14    if  $\text{VEB-TREE-MINIMUM}(V.\text{cluster}[\text{high}(x)]) == \text{NIL}$ 
15       $\text{VEB-TREE-DELETE}(V.\text{summary}, \text{high}(x))$ 
16      if  $x == V.\max$ 
17         $\text{summary-max} = \text{VEB-TREE-MAXIMUM}(V.\text{summary})$ 
18        if  $\text{summary-max} == \text{NIL}$ 
19           $V.\max = V.\min$ 
20        else  $V.\max = \text{index}(\text{summary-max},$ 
             $\text{VEB-TREE-MAXIMUM}(V.\text{cluster}[\text{summary-max}]))$ 
21    elseif  $x == V.\max$ 
22       $V.\max = \text{index}(\text{high}(x),$ 
         $\text{VEB-TREE-MAXIMUM}(V.\text{cluster}[\text{high}(x)]))$ 

```

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.

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.