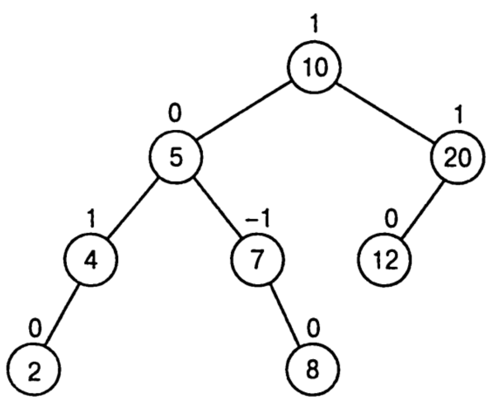
Алгоритми та складність

II семестр

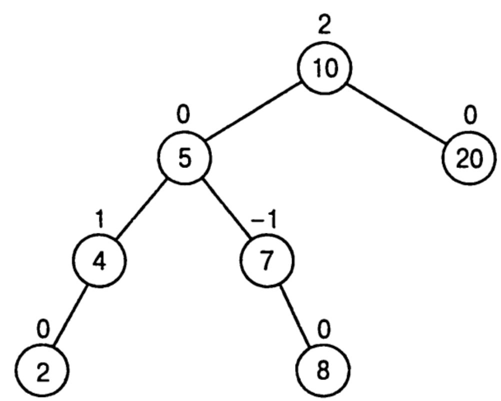
Лекція 3

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).



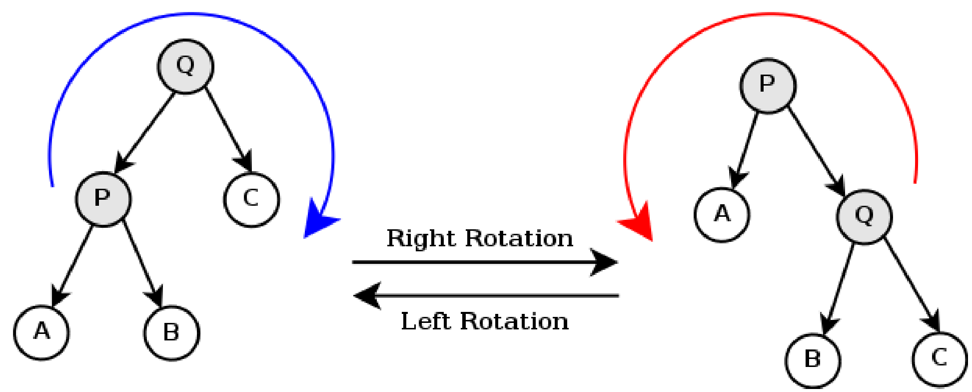
а)



б)

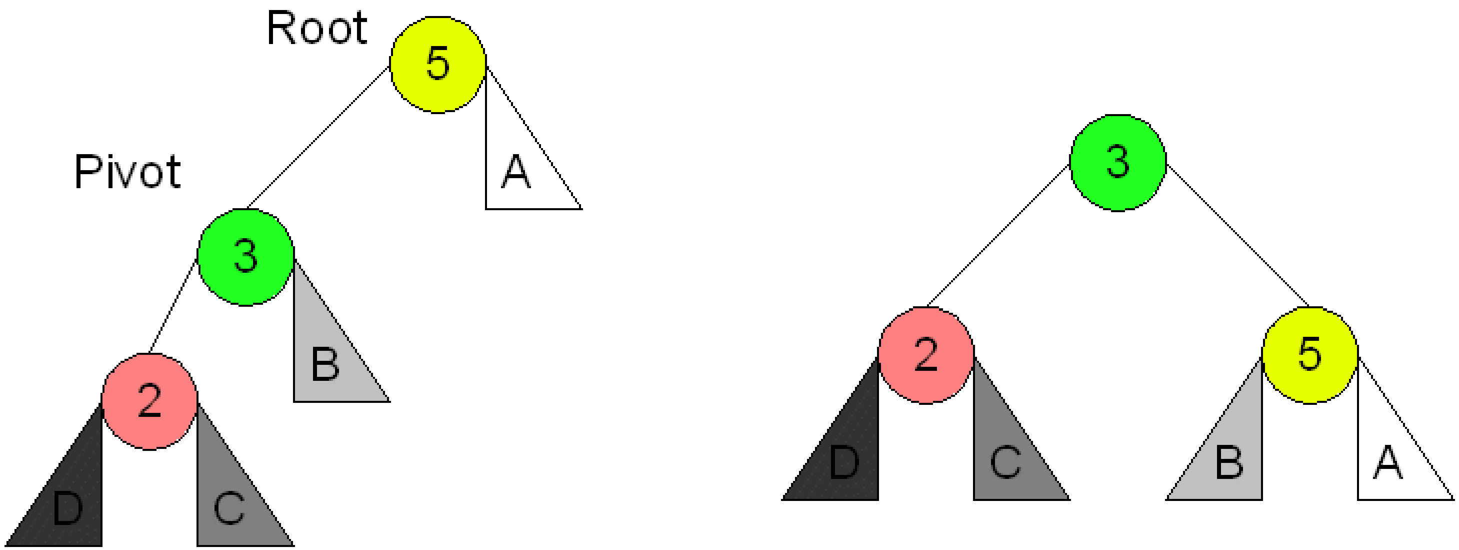
АВЛ-дерева

- Висота АВЛ-дерева з *n* вузлів обмежена знизу $\lfloor \log_2 n \rfloor$ і не перевищує $\log_\varphi(\sqrt{5}(n+2)) - 2 \approx 1.44 \log_2(n+2) - 0.328$, де φ – золотий перетин.
- Експериментально показано, що середня висота АВЛ-дерева 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.



АВЛ-дерева

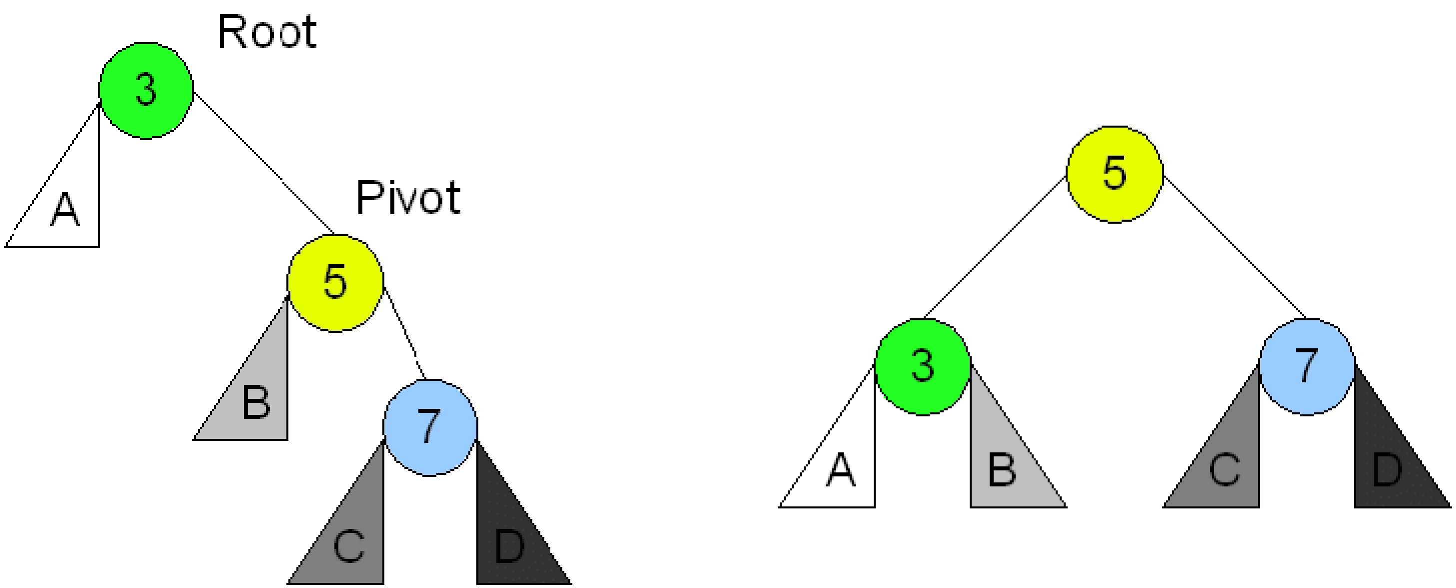
Left Left Case



Right Rotation

АВЛ-дерева

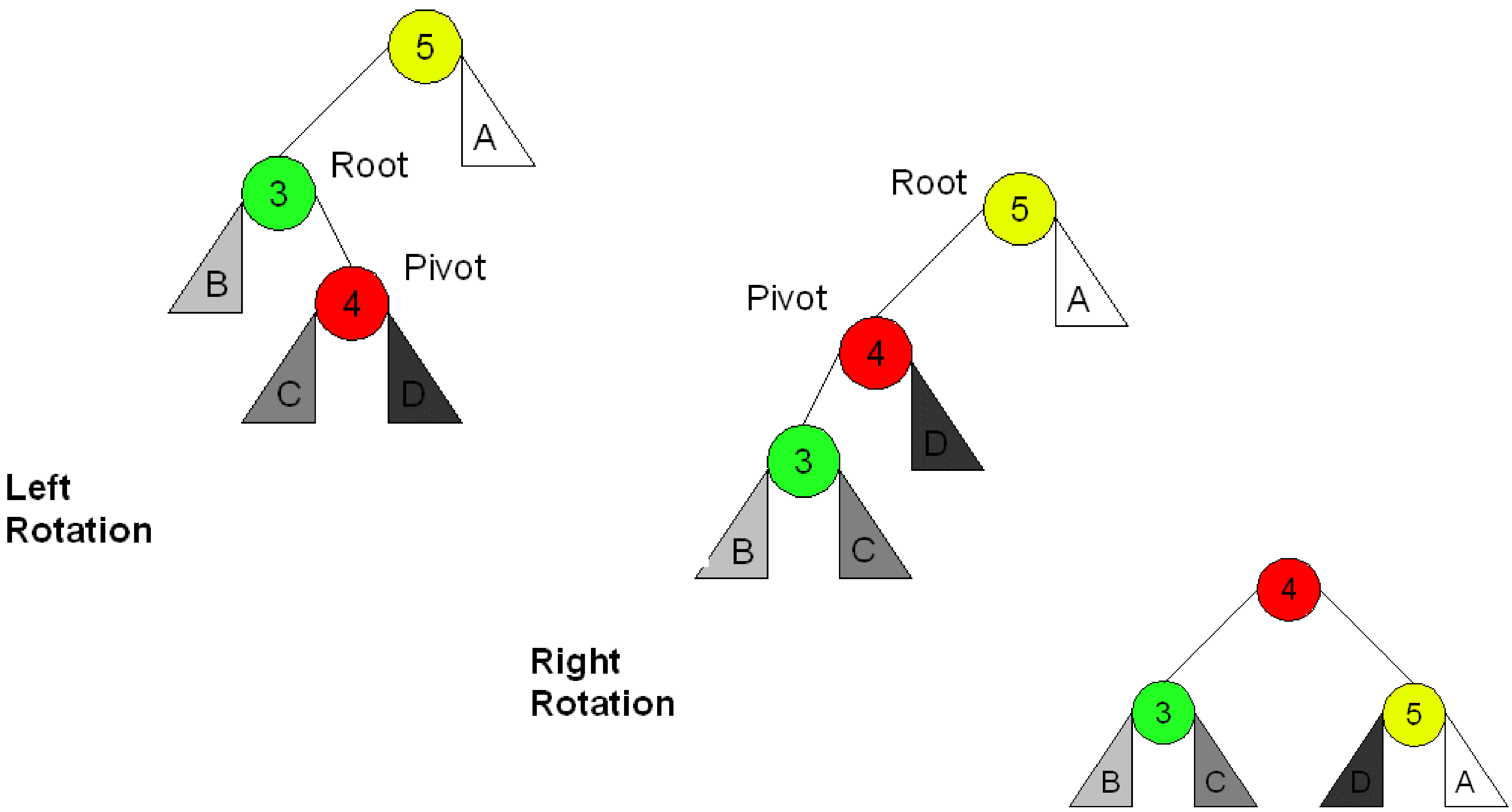
Right Right Case



Left Rotation

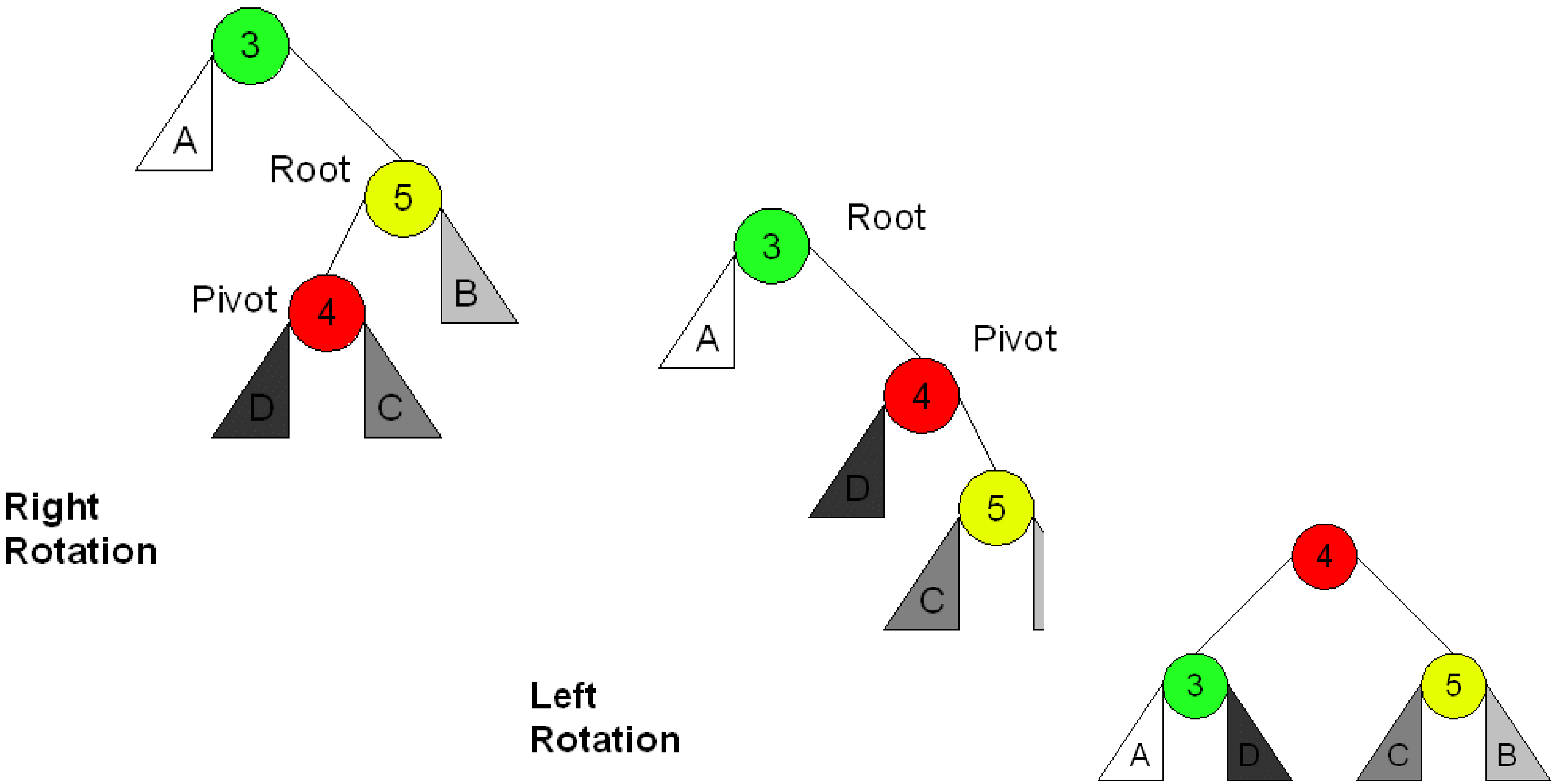
АВЛ-дерева

Left Right Case



АВЛ-дерева

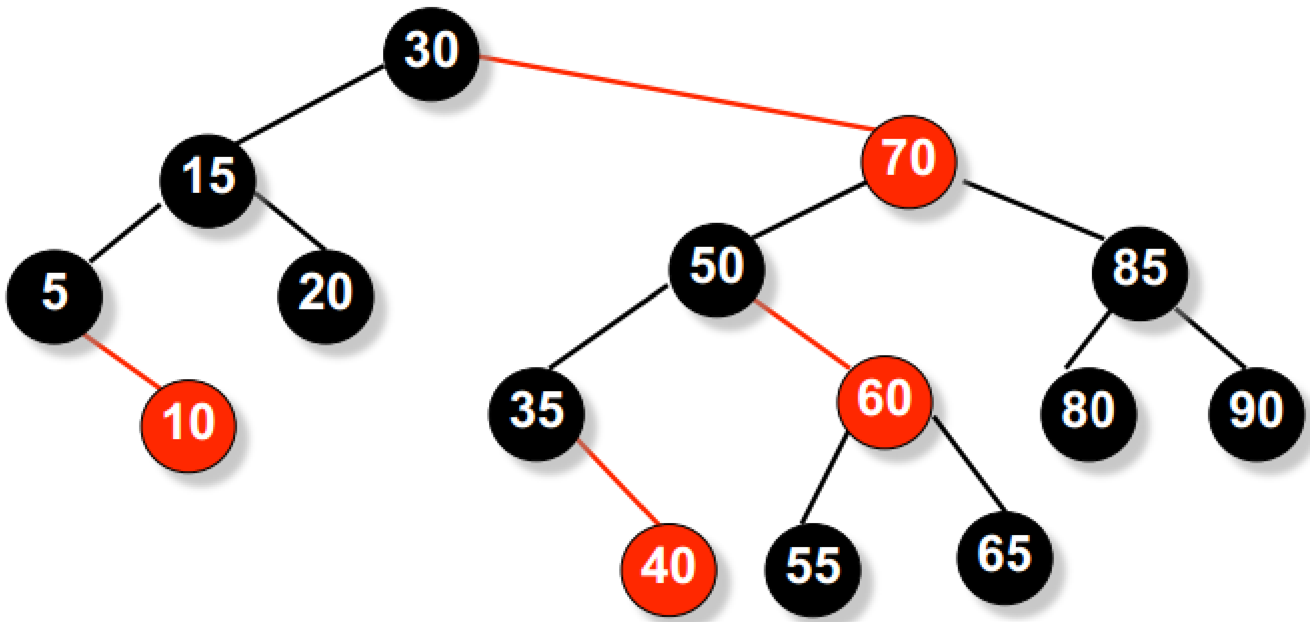
Right Left Case



АА-дерева

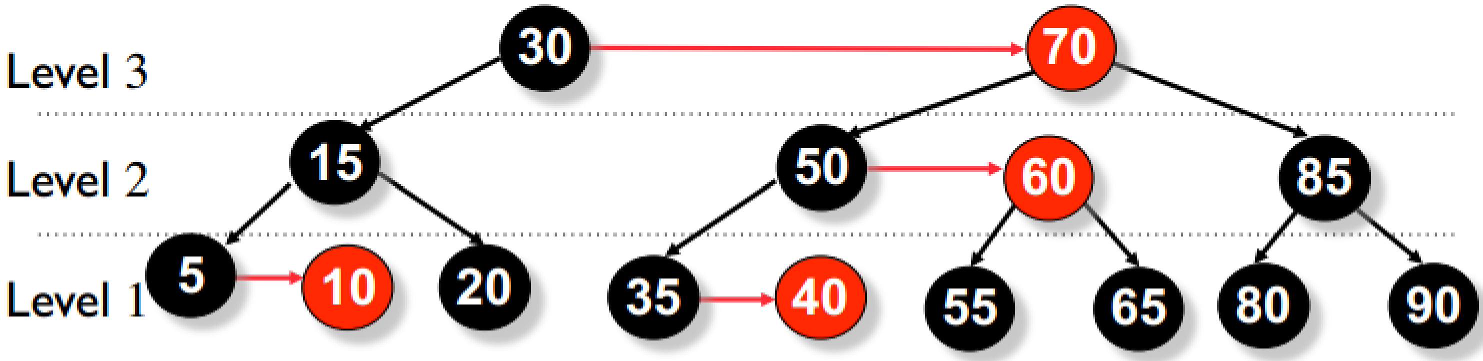
- АА = Arne Andersson (1993).
- Варіація червоно-чорного дерева: червоні вершини можуть бути лише правими потомками.
- *Рівень* вершини: збільшується від листів до кореня; рівень листа 1.
- *Коротке правило АА-дерева*: до однієї вершини можна приєднати іншу вершину того ж рівня лише одну і тільки справа.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

АА-дерева



Приклад АА-дерева

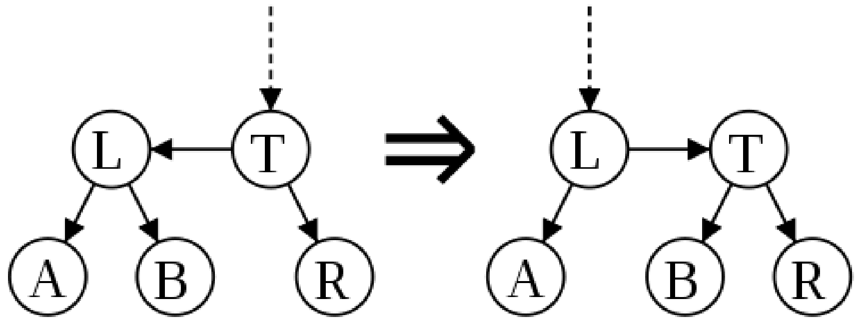
АА-дерева



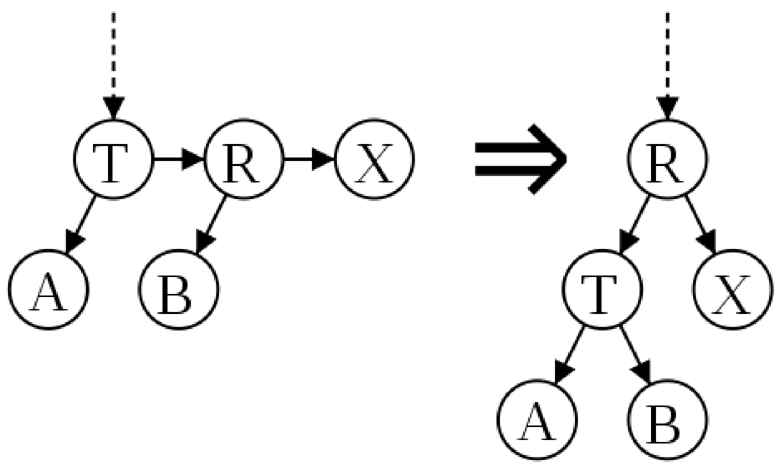
- Кольори в реальності не потрібні.
- З’являється поняття рівня і напрямку зв’язку (правий-лівий).
- Зв’язки можуть бути вертикальні і горизонтальні.

АА-дерева

- SKEW (усунення лівого зв’язку на одному рівні)



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



- Якщо властивість не порушена, процедура не змінює дерево.

АА-дерева

Додавання вузла

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

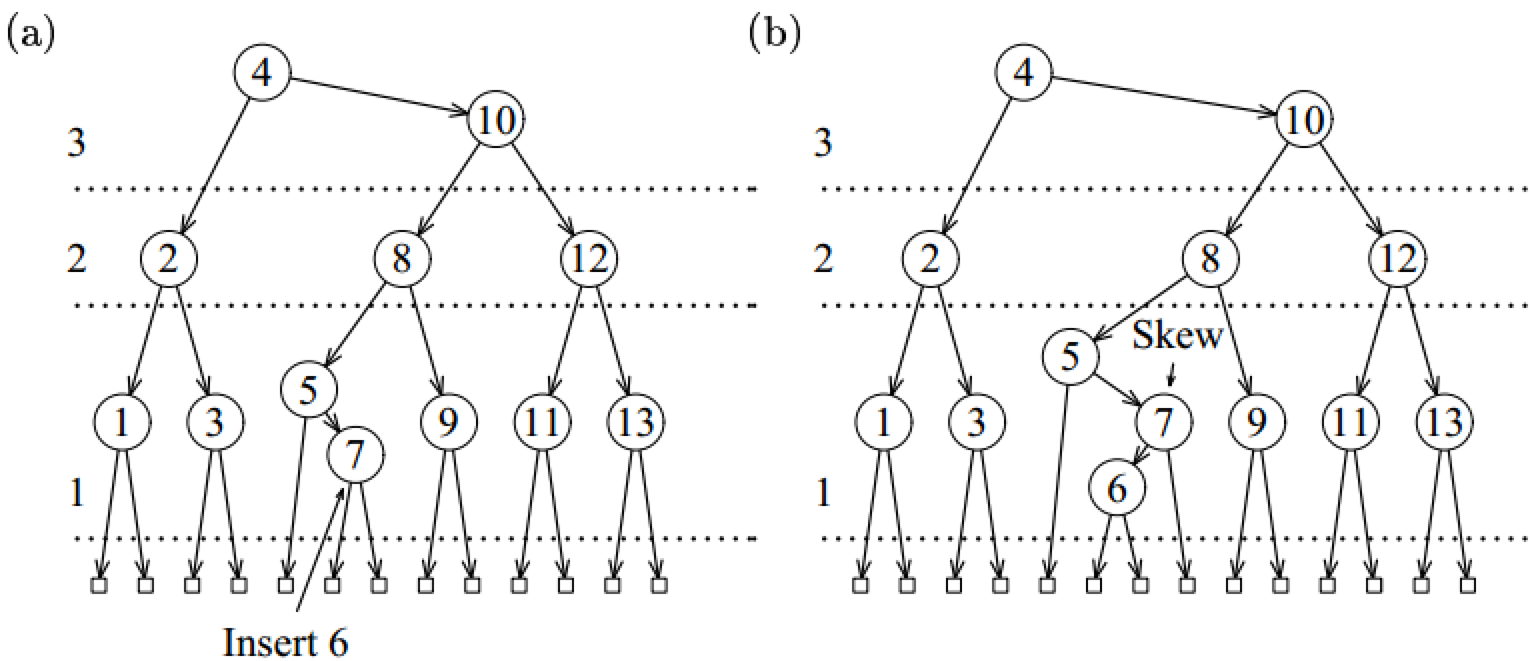
Видалення вузла

- Після видалення листа при русі вгору спочатку за необхідності відбувається пониження рівня (рівень вузла має бути на 1 більшим мінімального рівня його синів), потім тричі викликається SKEW і двічі SPLIT:

```
T := decrease_level(T)
T := skew(T)
right(T) := skew(right(T))
right(right(T)) := skew(right(right(T)))
T := split(T)
right(T) := split(right(T))
```

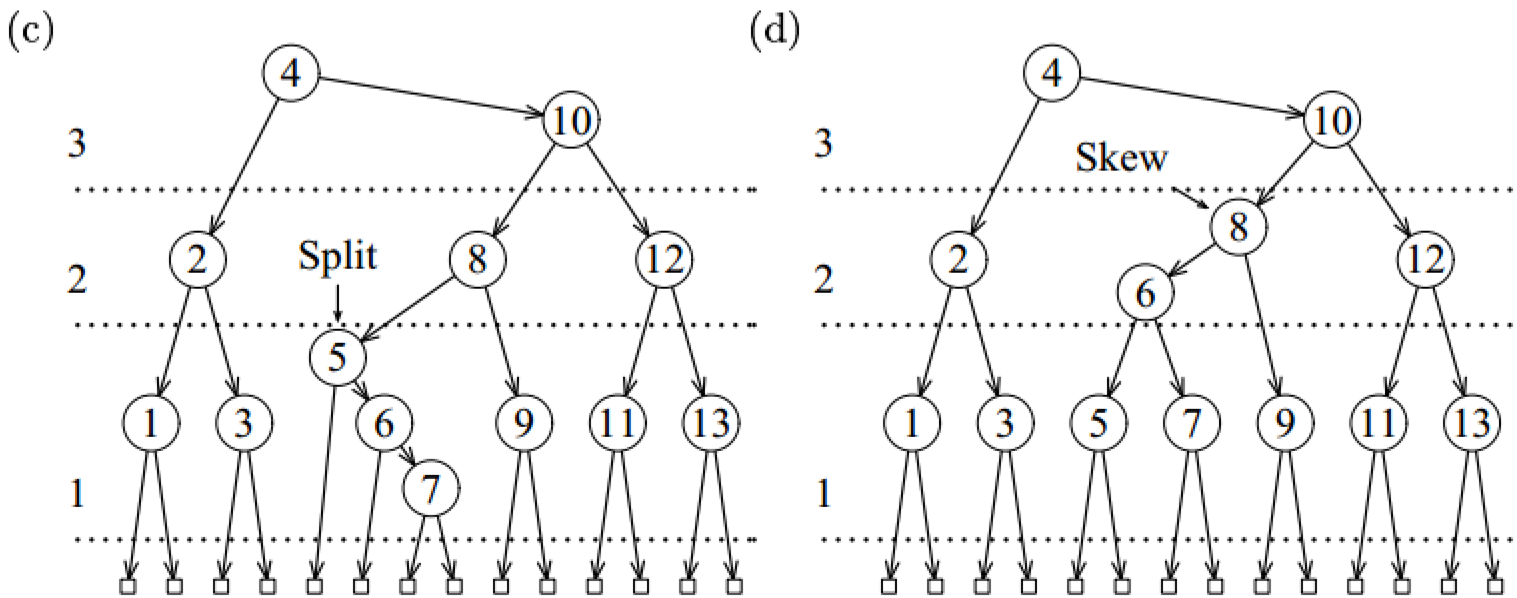
АА-дерева

Додавання вузла



АА-дерева

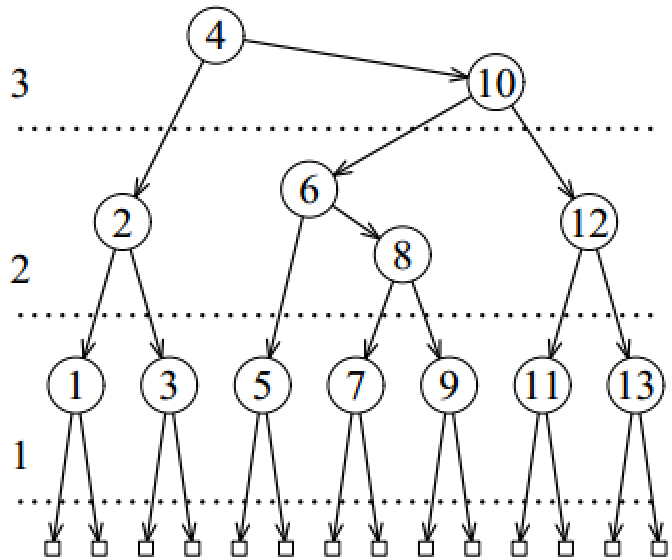
<u>Додавання вузла</u>



АА-дерева

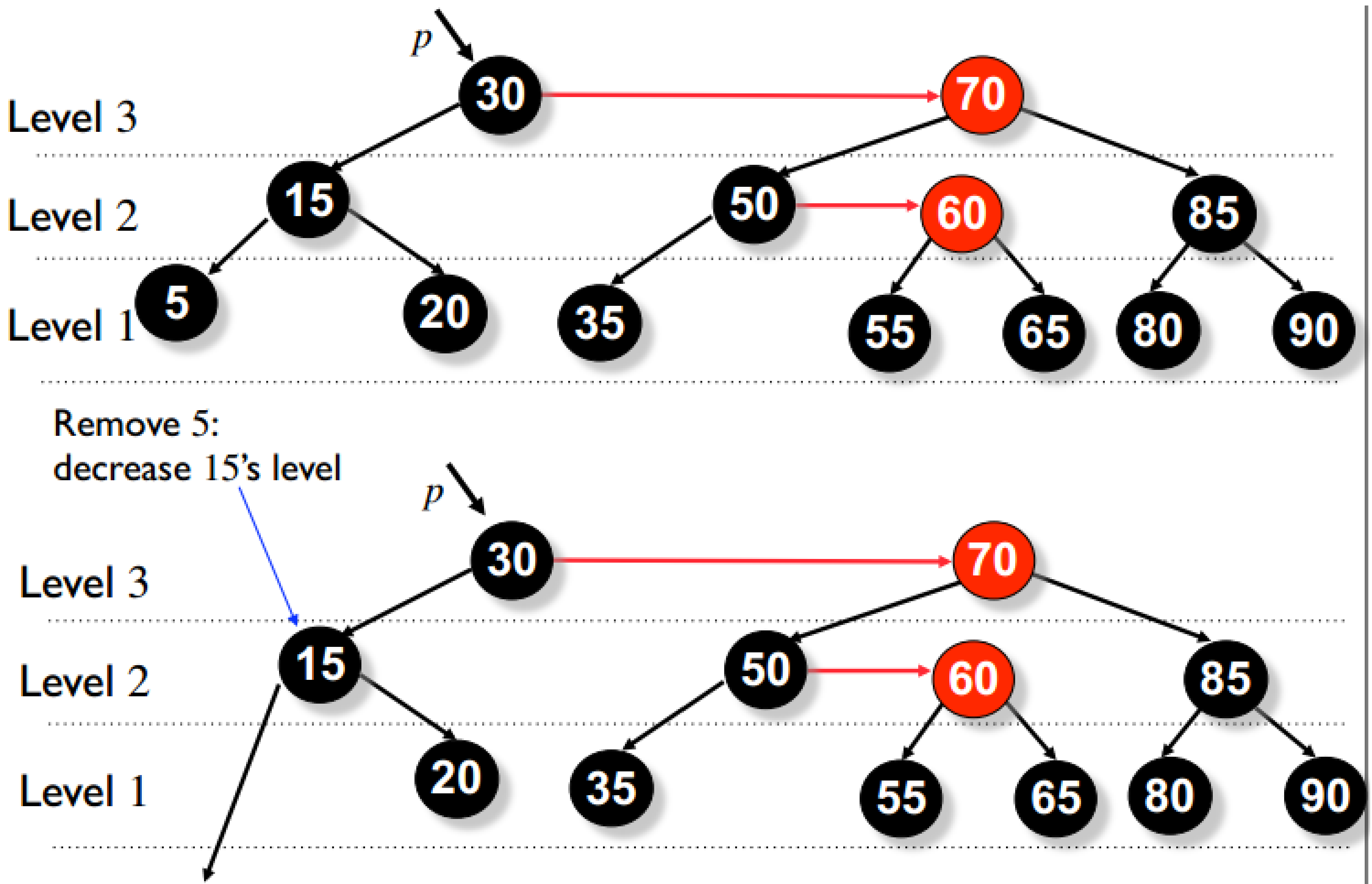
Додавання вузла

(e)



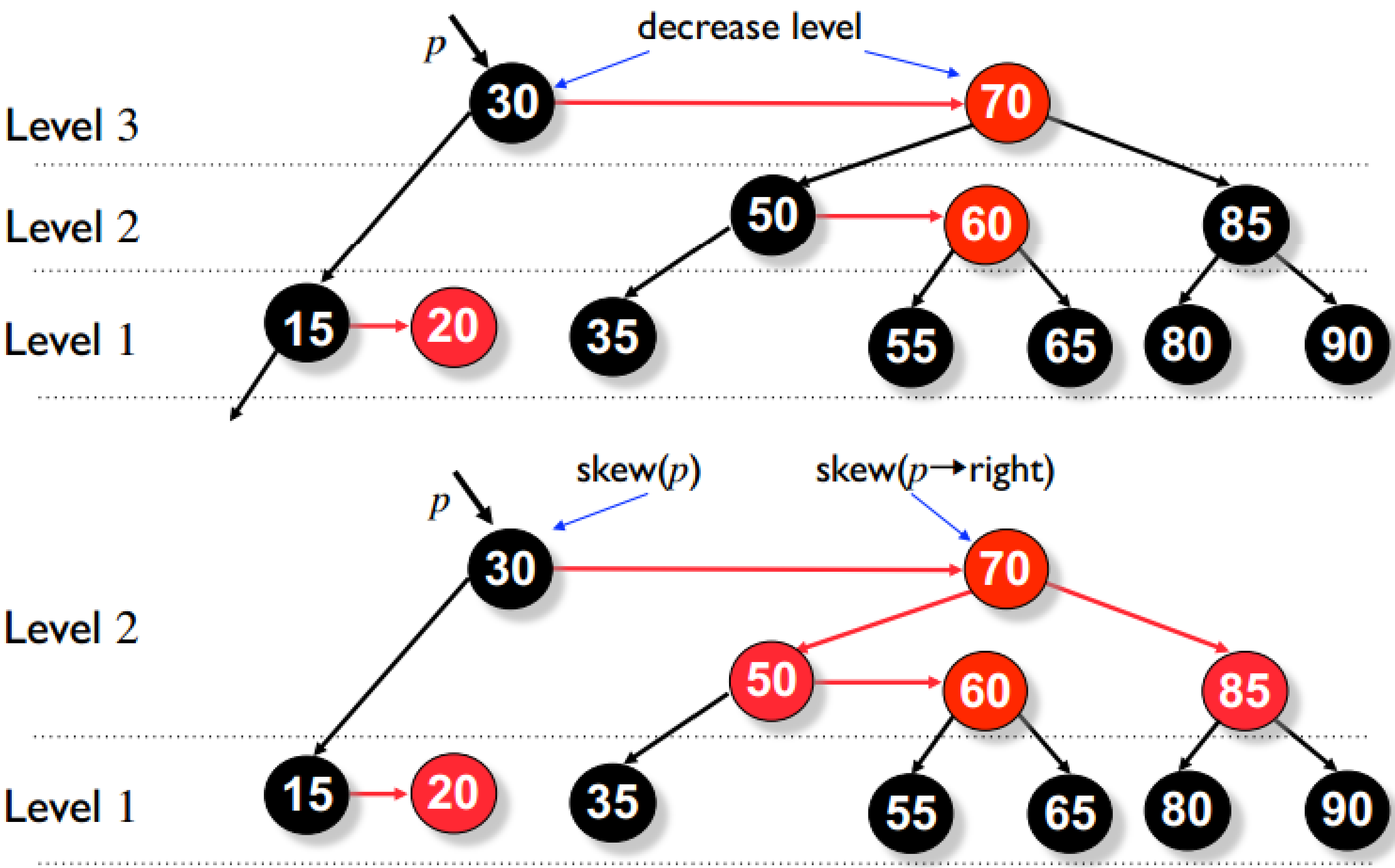
АА-дерева

Видалення вузла



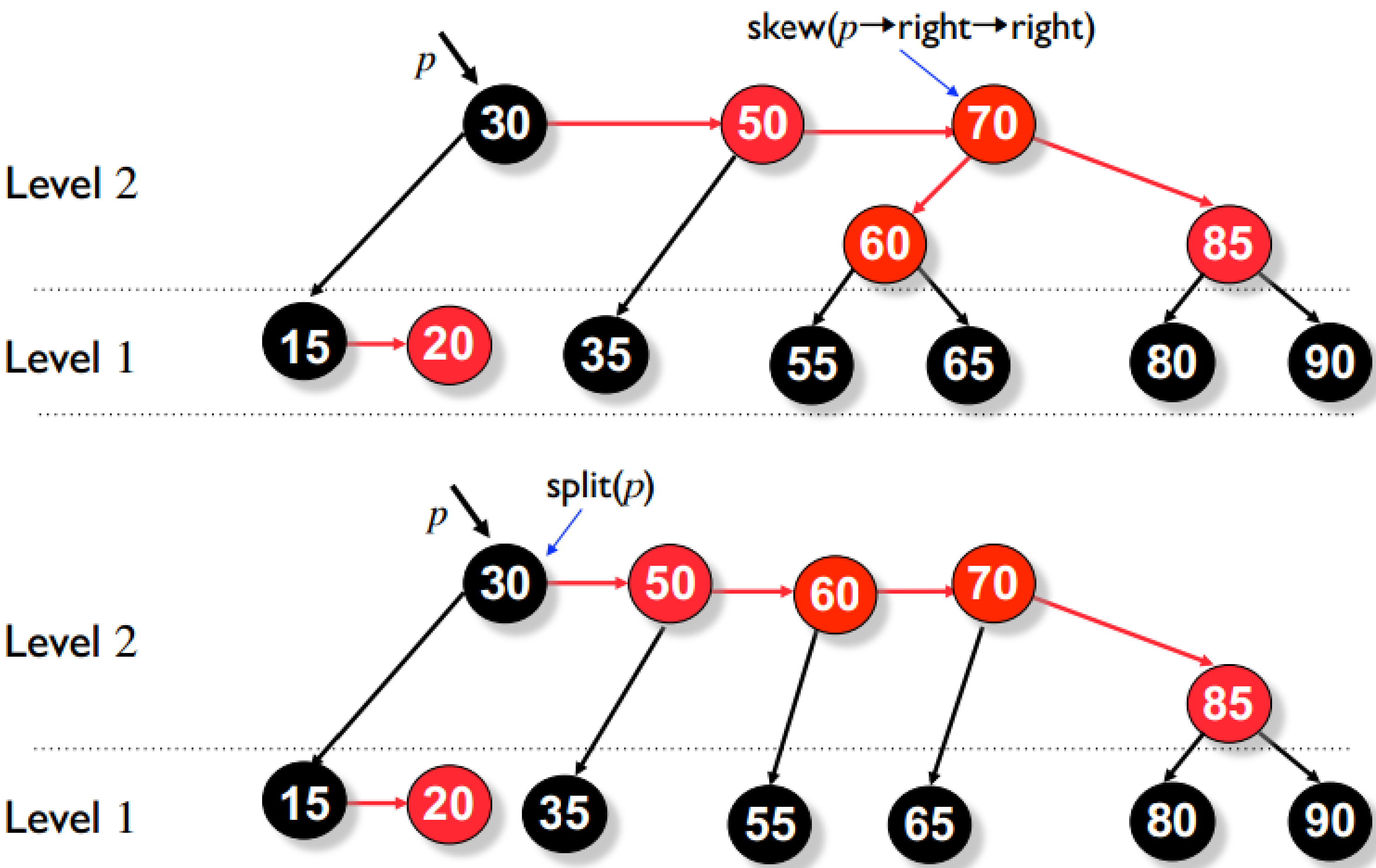
АА-дерева

Видалення вузла



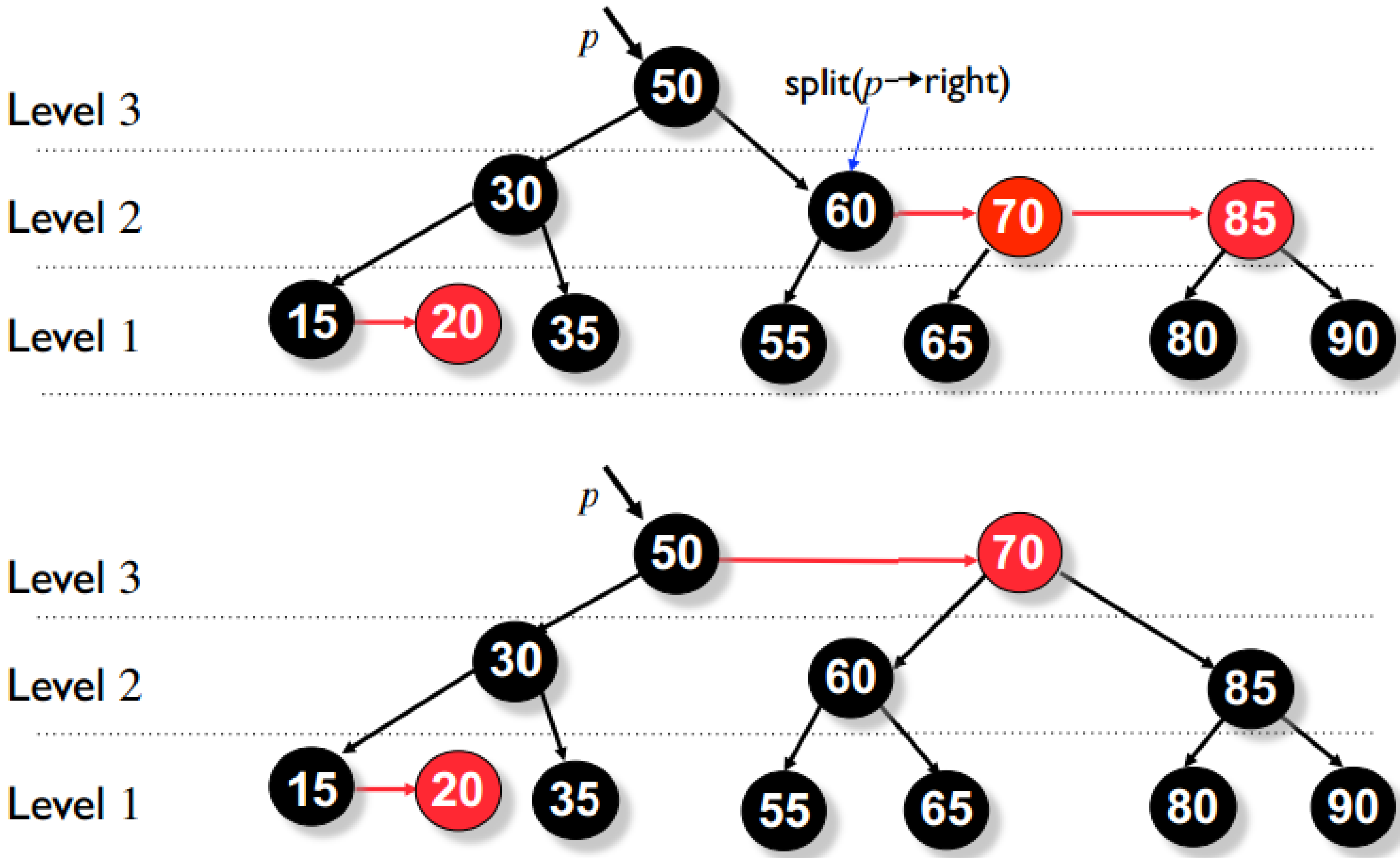
AA-дерева

Видалення вузла



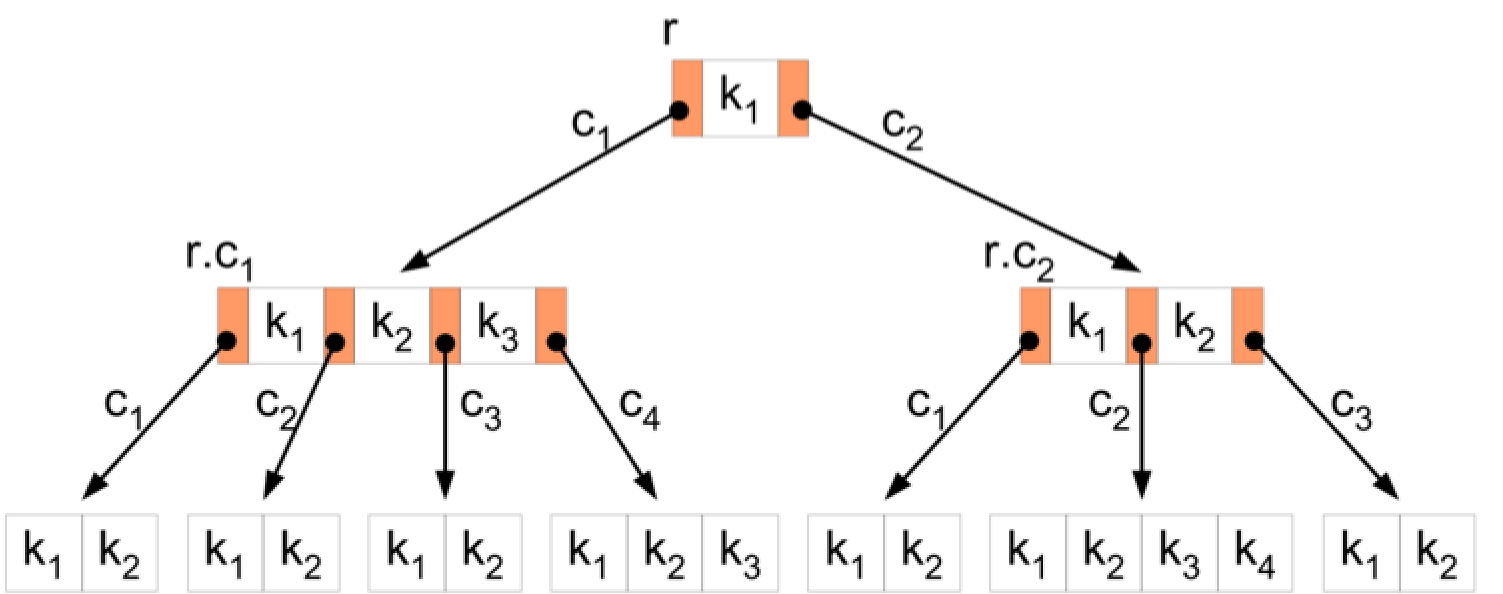
AA-дерева

Видалення вузла



B-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 B-дерев.



Приклад B-дерева

Розширювані дерева (splay trees)

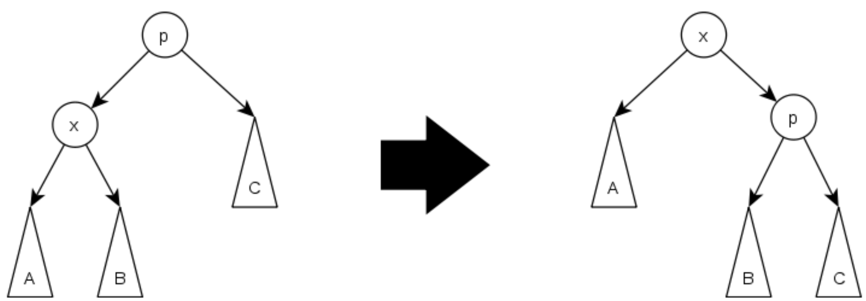
- Двійкове дерево пошуку з підтримкою збалансованості.
- Не потребує додаткових полів у вузлі.
- Явні функції балансування відсутні.
- При кожному звертанні до дерева виконується «операція розширення» (splay operation).
- В результаті вузли, до яких звертаються частіше, зберігаються ближче до кореня,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

Розширювані дерева (splay trees)

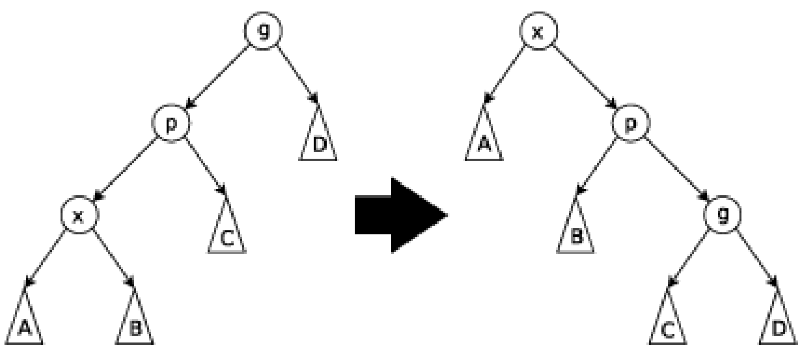
Операція SPLAY

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

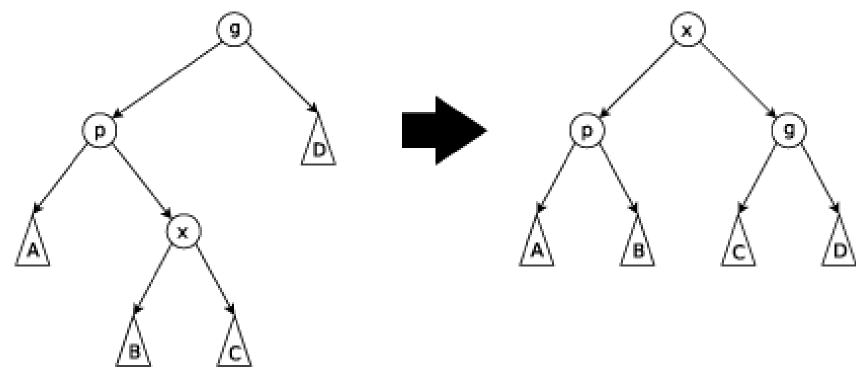
- *Zig*



- *Zig-Zig*



- *Zig-Zag*



Розширювані дерева (splay trees)

- *Merge* (об'єднання двох дерев). Для злиття дерев T1 і T2, в яких всі ключі T1 менше ключів в T2, робимо Splay для максимального елемента T1, тоді біля кореня T1 не буде правого дочірнього елемента. Після цього робимо T2 правим дочірнім елементом T1.
- *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.

Розширювані дерева (splay trees)

- *Search* (пошук елемента). Спочатку звичайний пошук. При знаходженні елементу запускаємо Splay для нього. Елемент відсутній? Виконується Splay для його наступника (чи попередника).
- *Insert* (додавання елемента). Запускаємо Split від елементу, що додається, і підвішуємо дерева, що вийшли, за нього.
- *Delete* (видалення елемента). Знаходимо елемент в дереві, робимо Splay для нього, робимо поточним деревом Merge його дітей.

Декартові дерева (treaps)

- Tree + Heap = Treap

Декартові дерева (treaps)

- Tree + Heap = Treap
- Дерево + Піраміда = Дераміда

Декартові дерева (treaps)

- Tree + Heap = Treap
- Дерево + Піраміда = Дераміда
- Купа + Дерево = Курево

Декартові дерева (treaps)

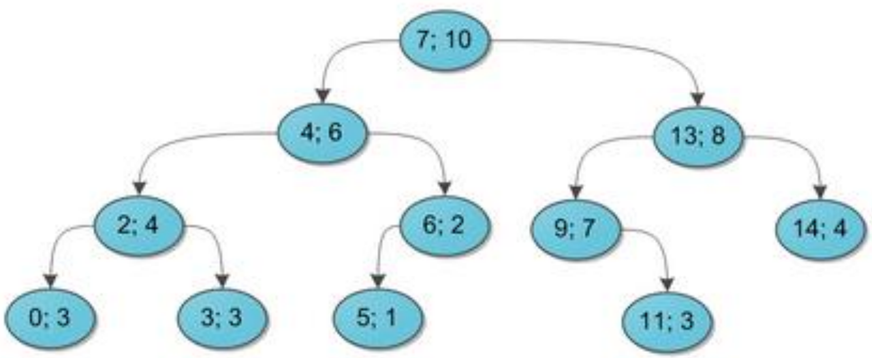
- Tree + Heap = Treap
- Дерево + Піраміда = Дераміда
- Купа + Дерево = Курево
- Дерево + Купа = ?

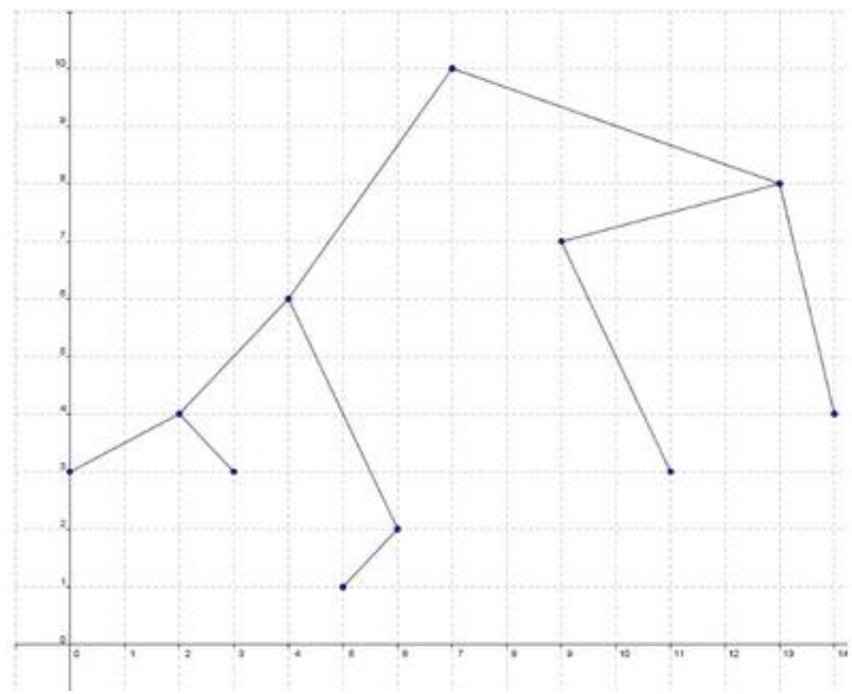
Декартові дерева (treaps)

- 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 $\leq 4 \log_2 n$.

Декартові дерева (treaps)

Чому дерево «декартове»

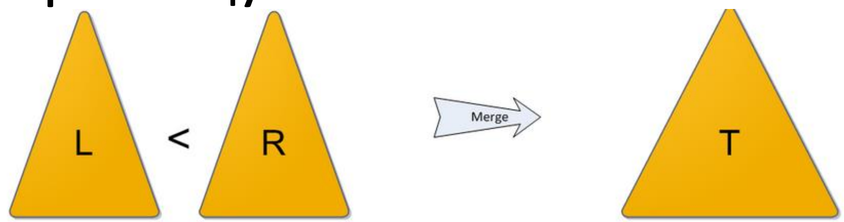




Декартові дерева (treaps)

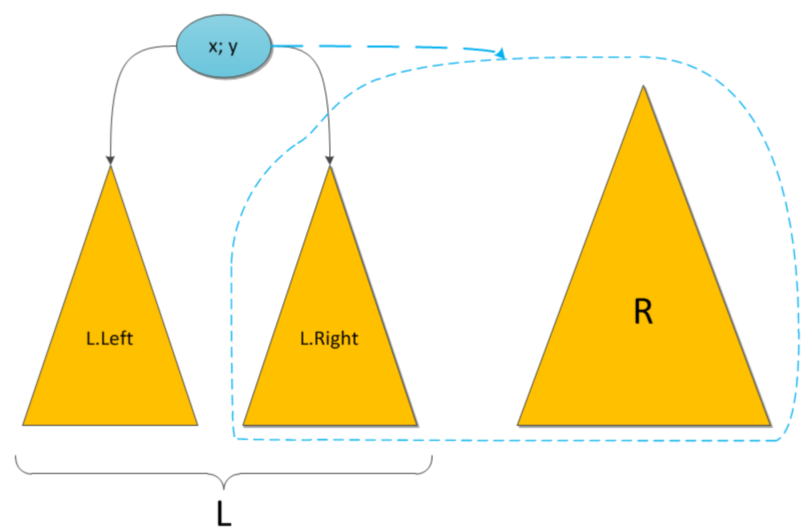
Можна визначити дві базові операції: *Merge* та *Split*.

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



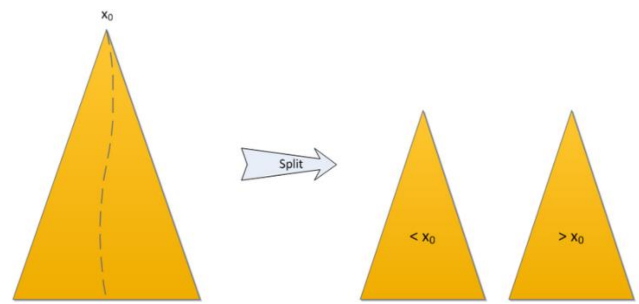
В якості нового кореня береться корінь дерева з більшим пріоритетом. Одне з піддерев зрозуміле, друге отримується рекурсивно.

Пріоритет лівого кореня більший



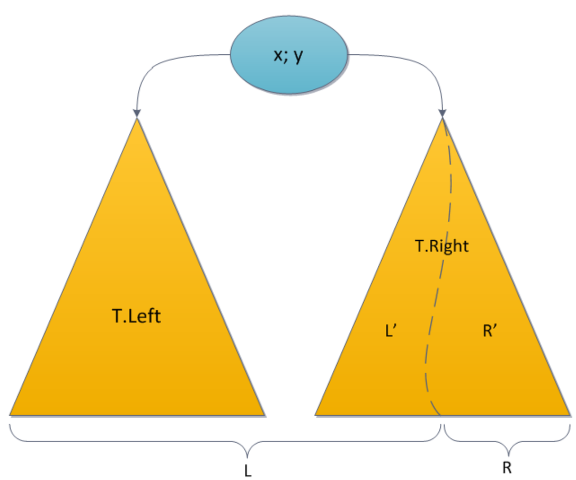
Декартові дерева (treaps)

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



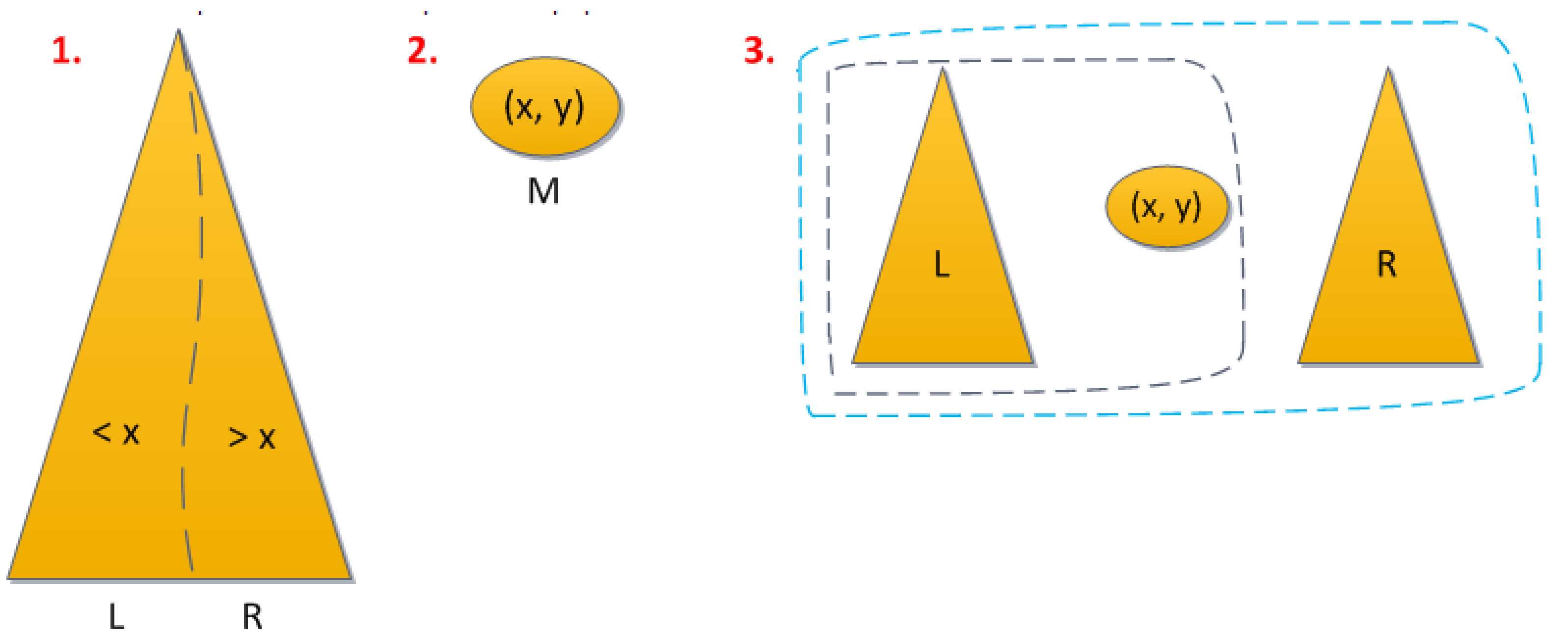
Звіряємо корінь з ключем. В залежності від результату бачимо, до якого з нових дерев належатиме корінь з одним із піддерев. Друге дерево рекурсивно виділяється з іншого піддерева.

Ключ кореня менший за x_0



Декартові дерева (treaps)

- Вставка елемента

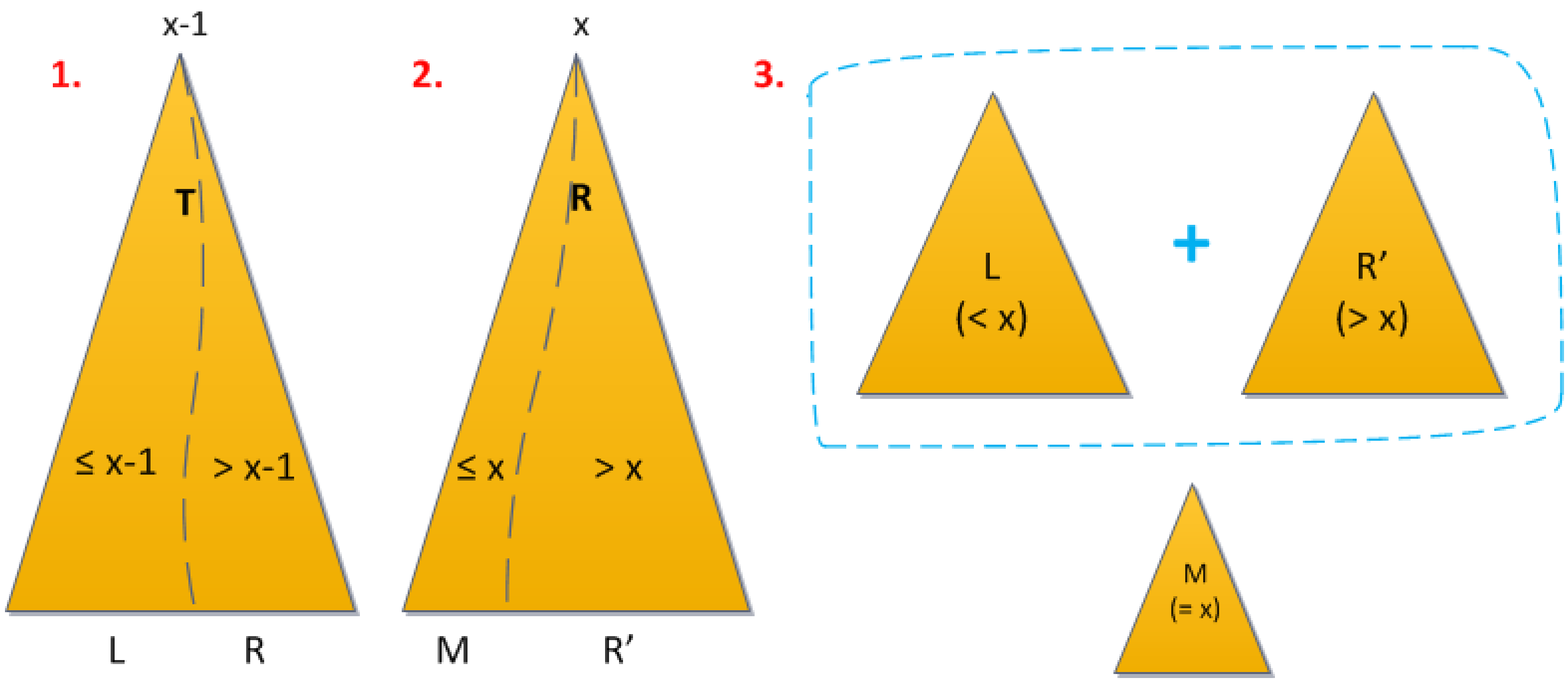


Час виконання операцій *Merge* та *Split* складає O(lg *n*).

Такий же буде час при виконанні операцій, що складаються зі скінченної кількості їх викликів.

Декартові дерева (treaps)

- Видалення елемента



Декартові дерева (treaps)

<u>Інший спосіб вставки та видалення елемента (*x, y*)</u>

- Вставка

1. Спуск як по дереву пошуку, поки не знайдемо перший елемент (*x1, y1)* з пріоритетом y1 < y.

2. Розбиття піддерева з коренем (*x1, y1)* 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)*.

Scapegoat-дерева

- Scapegoat = Цап-відбувайло.
- Arne Andersson (1989), I.Galperin та R.L.Rivest (1993) – незалежно прийшли до ідеї.
- Бінарне дерево пошуку, що не зберігає додаткової інформації у вузлі (як і розширюване дерево).
- В процесі відновлення балансу обертання *не використовуються*, а відбувається *перебудова* розбалансованого піддерева.
- Існують дерева, які не можуть балансуватися через обертання (наприклад, представляють багатовимірні структури в обчислювальній геометрії).
- Висота дерева з n вузлів завжди $O(\lg n)$.

Scapegoat-дерева

- Вводиться *коефіцієнт балансування* $\alpha \in [0{,}5;\ 1)$, що регулює ступінь балансування.
- Вершина x називається *α-збалансованою за вагою*, якщо розмір кожного з її двох піддерев $\leq \alpha \cdot \mathrm{size}(x)$.
- α = 0,5 дає умову ідеально збалансованого дерева, α = 1 задовольняє навіть вироджене дерево – список.
- α = 2/3 одначає, що одне з піддерев може містити до 2/3 всіх вершин (і тоді в іншому залишиться 1/3).
- Значення коефіцієнта α можна вибирати, пришвидшуючи час пошуку за рахунок сповільнення модифікуючої операції чи навпаки, залежно від конкретної ситуації.

Scapegoat-дерева

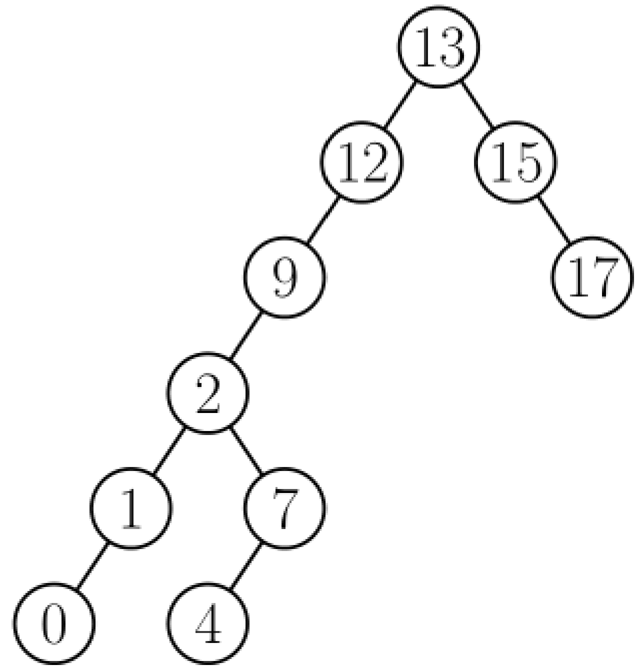
- Висота scapegoat-дерева з n вузлів обмежена згори значенням $\log_{1/\alpha}\mathrm{n}$.

Приклад дерева при α = 2/3.

Воно збалансоване!

В нього 10 вузлів та висота = 5

і не перевищує $\log_{3/2}10 \approx 5{,}68$.



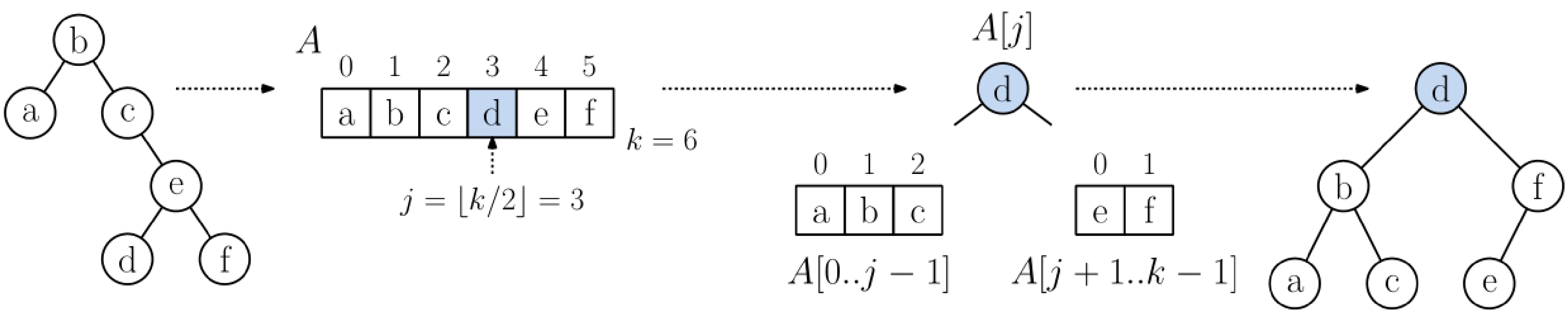
- *Пошук* в scapegoat-дереві відбувається стандартним чином за O(lg *n*).

Scapegoat-дерева

- *Видалення*. Вузол видаляється як зі звичайного дерева пошуку. Далі робиться перевірка, чи не порушився баланс. Порушився => перебудова дерева
- Зберігається параметр maxSize – максимальний розмір дерева з часу останньої перебудови. При додаванні вузла maxSize збільшується, при видаленні не змінюється. Після перебудови встановлюється рівним поточному розміру дерева.
- Порушення балансу: поточний розмір < $\alpha \cdot$maxSize(x).
- Інший варіант умови: поточний розмір < maxSize(x)/2.
- Дерево перебудовується ідеально збалансованим.
- Найгірша робота: O(n). Амортизований час O(lg n).

Scapegoat-дерева

- *Як перебудовується дерево*.
- Симетричний обхід дерева з отриманням упорядкованої послідовності вершин.
- Знаходження медіани знайденої відсортованої послідовності – наш корінь.
- Ліве та праве піддерева знаходяться рекурсивно з відповідних підпослідовностей.
- Час на перебудову – O(*n*) для (під)дерева з n вузлів.



Scapegoat-дерева

- *Вставка*. Вузол вставляється як в звичайне дерево пошуку. Паралельно обчислюється глибина нової вершини.
- В результаті баланс може порушитися, якщо глибина перевищить гранично допустиму висоту $\log_{1/\alpha} n$.
- Тоді з підйомом до кореня шукається *цап-відбувайло* – вершина, в якій порушується α-баланс.
- Піддерево з коренем в цій вершині перебудовується ідеально збалансованим.
- Можна показати, що цап-відбувайло завжди існуватиме в такій ситуації.
- Якщо кандидатів «в цапи» декілька: достатньо взяти першого виявленого «порушника».

Scapegoat-дерева

- Для перевірки α-балансів треба знати розміри піддерева і його нащадків.
- Піднімаючись, можна обчислити розмір піддерева для батьківської вершини x за вже відомим розміром одного нащадка і обійшовши другого:

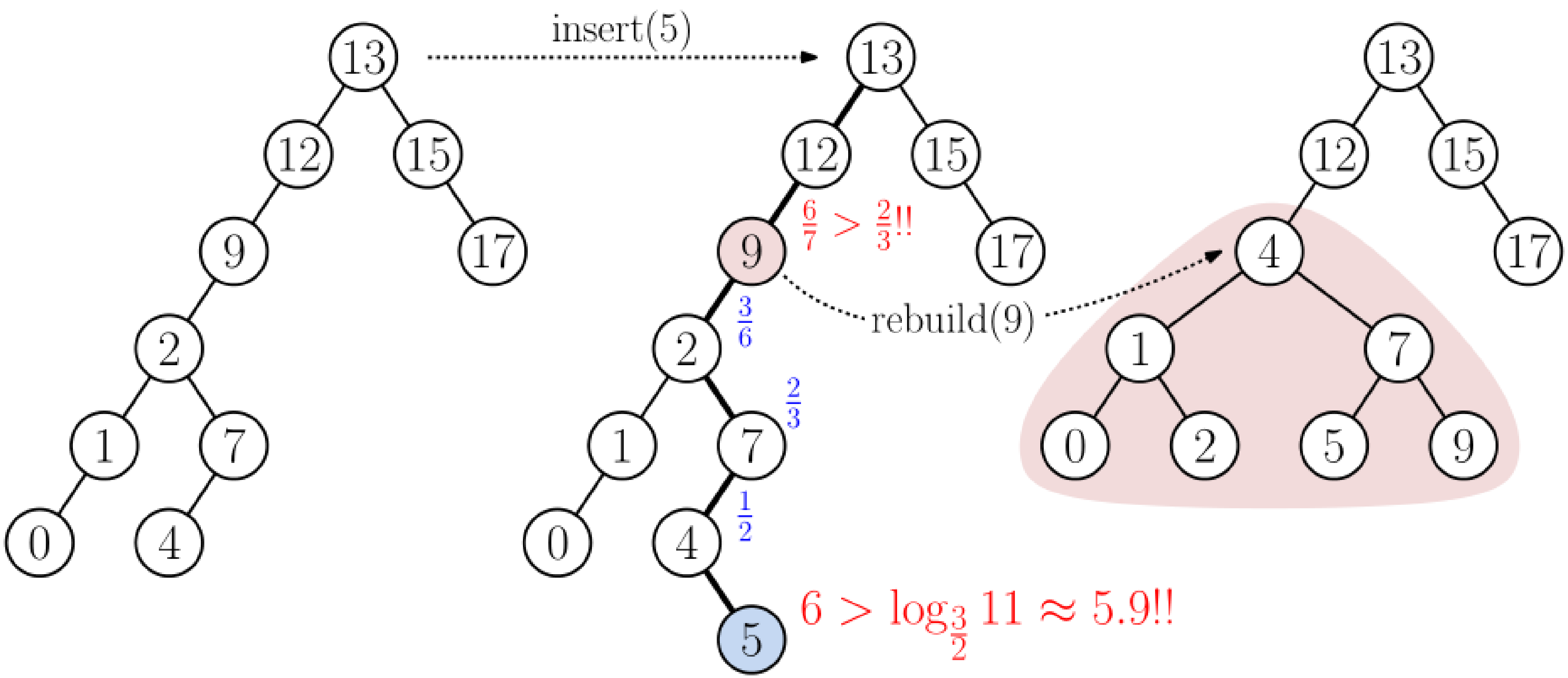
size(x) = size(x.left) + 1 + size(x.right).

- Обхід піддерева відбувається за лінійний час, але це не погіршить оцінку O(*n*) перебудови дерева, яка обов'язково має далі відбутися.
- Якщо $\frac{\mathrm{size(x.child)}}{\mathrm{size(x)}} > \alpha$, то вузол x буде цапом-відбувайлом.
- Найгірший час вставки O(*n*), а амортизований O(lg *n*).

Scapegoat-дерева

Вставка вузла №11 в scapegoat-дерево при α = 2/3.

Біля вершин показано обчислення балансів в процесі пошуку цапа-відбувайла (вузол з ключем 9).



Розширення структур даних

Розширення структур даних можна розбити на чотири кроки.

1. Вибір базової структури даних.

2. Визначення необхідної додаткової інформації, яку слід зберігати в базовій структурі даних і підтримувати її актуальність.

3. Перевірка того, що додаткова інформація може підтримуватися основними модифікуючими операціями над базовою структурою даних.

4. Розробка нових операцій.

Розширення структур даних

<u>Теорема (розширення червоно-чорних дерев).</u>

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*).

Розширення структур даних

<u>Доведення.</u>

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.

Розширення структур даних

Доведення (далі).

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 *nil*[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*).

Розширення структур даних

<u>Доведення (завершення).</u>

Видалення вузла.

Перша фаза. Після власне видалення вузла відбувається оновлення значень поля *f* по ланцюжку вгору, що забере час O(lg *n*).

Друга фаза. Поворотів відбудеться не більше трьох, кожен з яких вимагає часу на оновлення всіх полів *f* на шляху до кореня не більше O(lg *n*).

Таким чином, для видалення потрібен час також O(lg *n*).

Зауваження. У багатьох випадках для оновлення полів при повороті досить часу O(1).

Динамічні порядкові статистики

i-та порядкова статистика множини з *n* елементів – елемент з *i*-м в порядку зростання ключем.

Ранг елемента – його порядковий номер в лінійно впорядкованій множині.

Модифіковане червоно-чорне дерево дозволяє знайти ранг і порядкову статистику за час $O(\lg n)$.

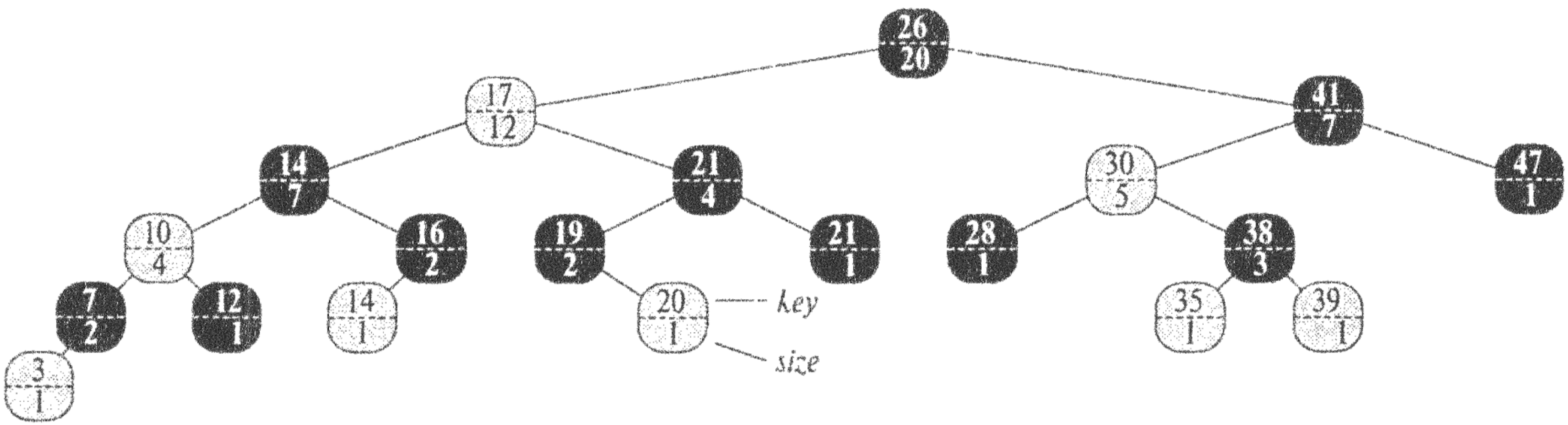
Дерево порядкової статистики T (order-statistic tree) – червоно-чорне дерево з додатковим інформаційним полем $size[x]$ (розмір піддерева з коренем x).

Поклавши $size[nil[T]]=0$, отримуємо тотожність

$$size[x] = size[left[x]] + size[right[x]] + 1.$$

Оскільки ключі потенційно не унікальні, під рангом розумітимемо позицію елемента в дереві при симетричному обході.

Дерево порядкової статистики



Ключі зі значеннями 14 і 21 повторюються.

Які їх ранги?

Дерево порядкової статистики

<u>Пошук елемента з заданим рангом</u> *i* в піддереві з коренем *x*:

```
OS_SELECT(x, i)
r ← size[left[x]]+1
if i = r
    then return x
elseif i < r
    then return OS_SELECT(left[x], i)
else return OS_SELECT(right[x], i − r)
```

(*size*[*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 червоно-чорного дерева з *n* елементів.

(Знайдемо в дереві-прикладі елемент з рангом 17.)

Дерево порядкової статистики

<u>Пошук рангу</u> елемента *x*:

```
OS_RANK(T, x)
r ← size[left[x]] + 1
y ← x
while y ≠ root[T]
      do if y = right[p[y]]
            then r ← r + size[left[p[y]]] + 1
         y ← p[y]
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 чиїмось 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 *key*[*x*] в піддереві з коренем у вузлі *y*.

Ініціалізація. Перед першою ітерацією *r* присвоюється ранг *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 *key*[*x*] в піддереві з коренем *y* на початку виконання тіла циклу, то в кінці виконання *r* є рангом *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 (*size*[*left*[*p*[*y*]]+1).

Завершення. Цикл завершується, коли *y*=*root*[T], тому піддерево з коренем *y* і є цілим деревом. Таким чином *r* є рангом *key*[*x*] в усьому дереві.

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

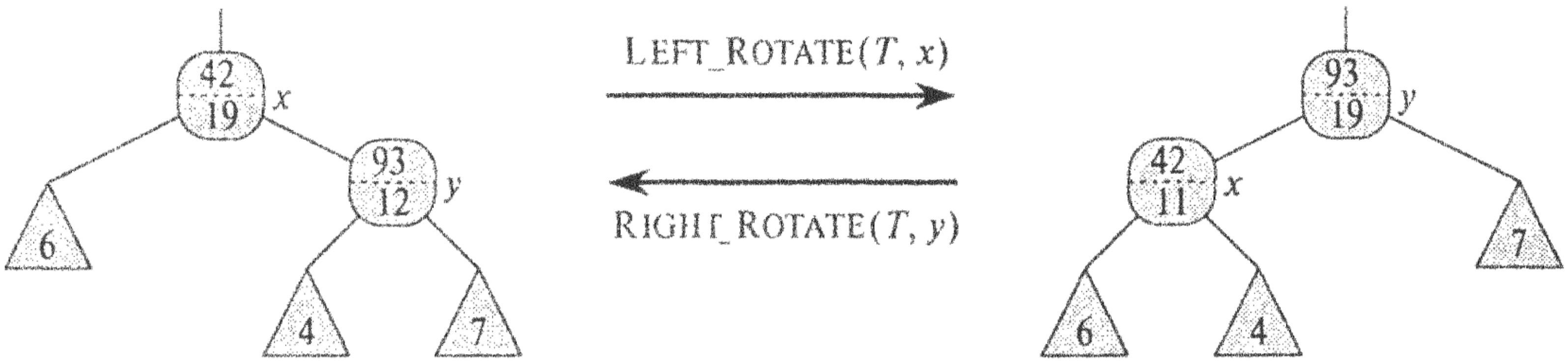
Приклад: послідовність значень *key*[*y*] та *r* на початку кожної ітерації циклу while при пошуку рангу вузла з ключем 38:

Итерация	$key[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17

Дерево порядкової статистики

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).

Дерево порядкової статистики

Підтримка розміру піддерев

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

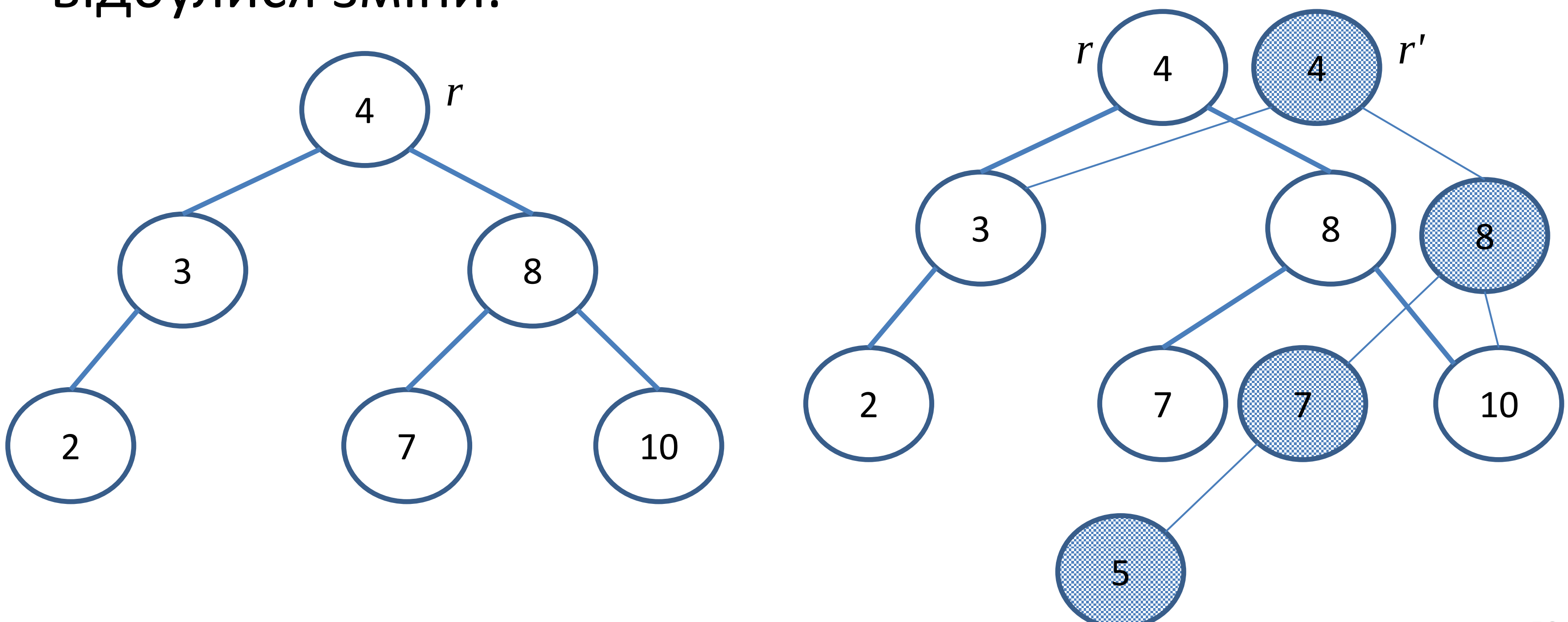
На першому етапі проходимо шлях від позиції видаленого вузла вгору до кореня, зменшуючи значення поля *size* для кожного вузла. В другій фазі може відбутися до трьох поворотів. Отже, загальний час операції O(lg *n*).

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.

Персистентні динамічні множини

- 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.



Запитання і завдання

Нехай в реалізації персистентної множини на основі бінарного дерева пошуку вузол не містить посилання на батька.

- Визначте, які вузли мають змінитися при вставці ключа *k* та видаленні вузла *y* в загальному випадку.
- Напишіть процедуру PERSISTENT_TREE_INSERT, що повертає нове персистентне дерево T' – результат вставки ключа *k* у дерево T. Яким буде її час роботи, якщо висота дерева *h*?
- Використайте для реалізації червоно-чорне дерево. Час роботи вставки і видалення в найгіршому випадку і об'єм необхідної пам'яті мають бути O(lg *n*).