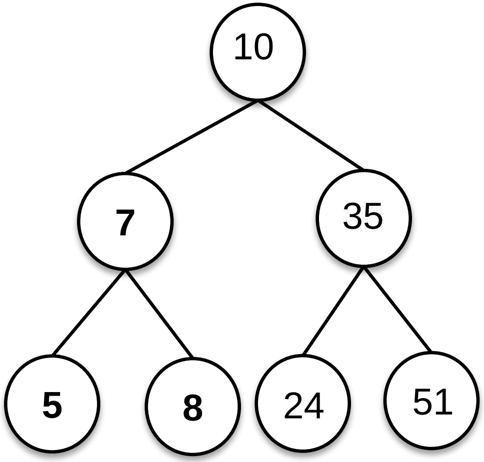
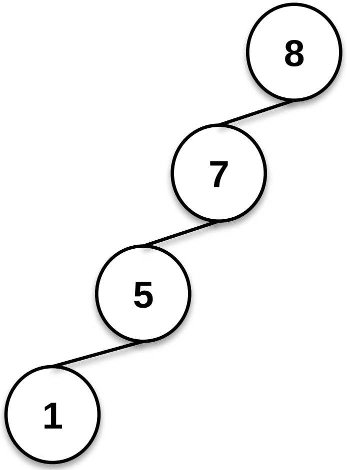
Збалансованість дерев

Складність основних операцій у випадку роботи із бінарним деревом пошуку лінійно залежить від його висоти та дорівнює $O(h)$





$$h = \log_2 N$$

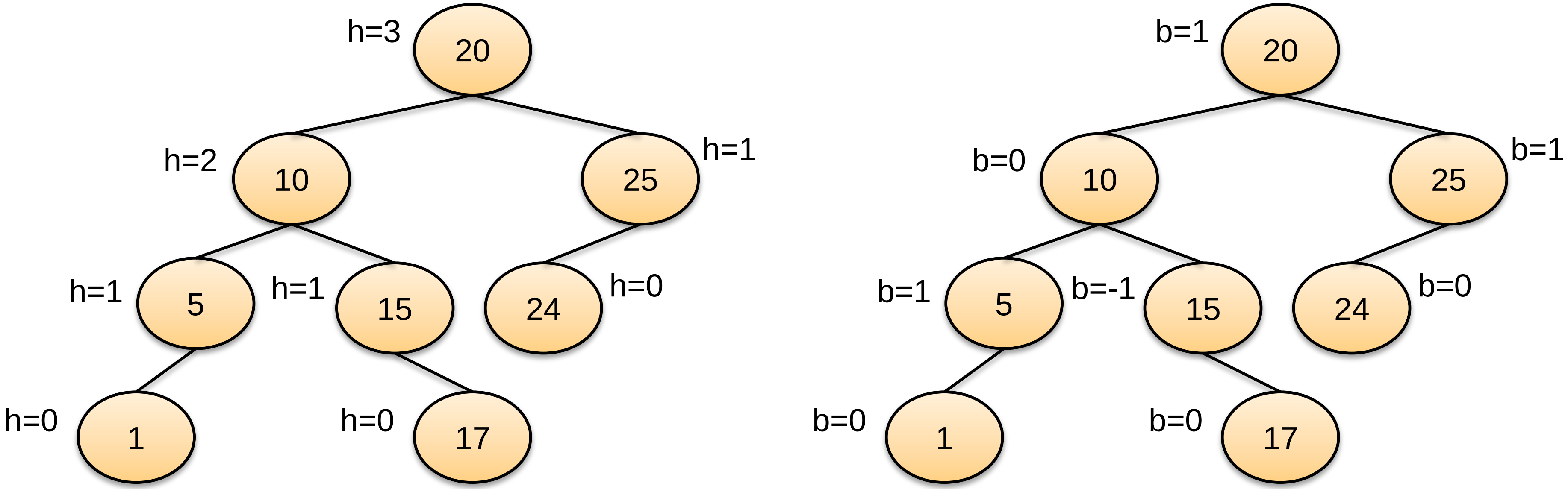
$$h = N$$

Двійкове дерево називають **ідеально збалансованим**, якщо для кожної вершини кількість вершин в лівому і правому її піддеревах відрізняється не більше ніж на одиницю.

Збалансоване дерево пошуку – дерево пошуку, в якому висоти піддерев будь-якого вузла відрізняються не більше ніж на задану константу k.

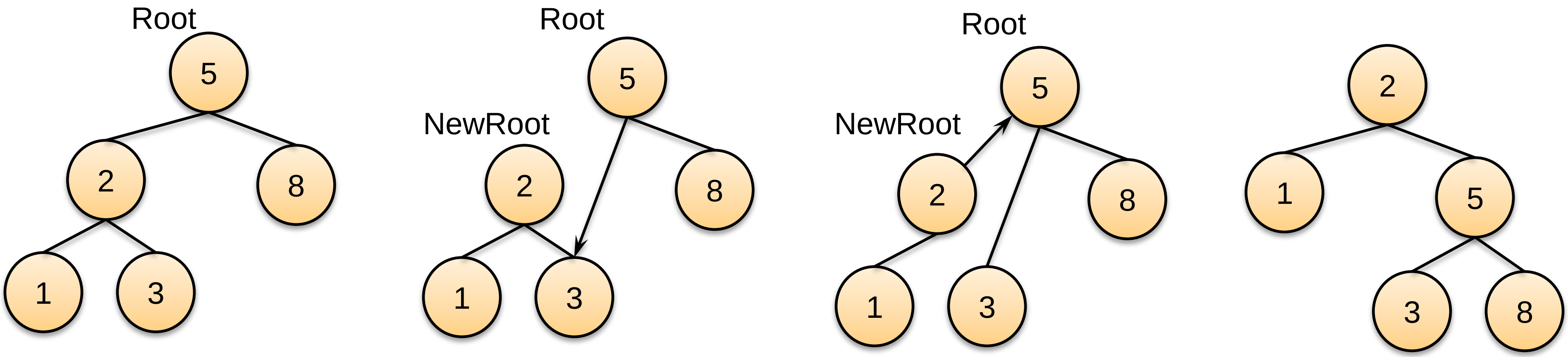
Коефіцієнт збалансованості вузла (balance factor) – це різниця висот його лівого і правого піддерев.

Висоту пустого піддерева вважають такою, що дорівнює -1.



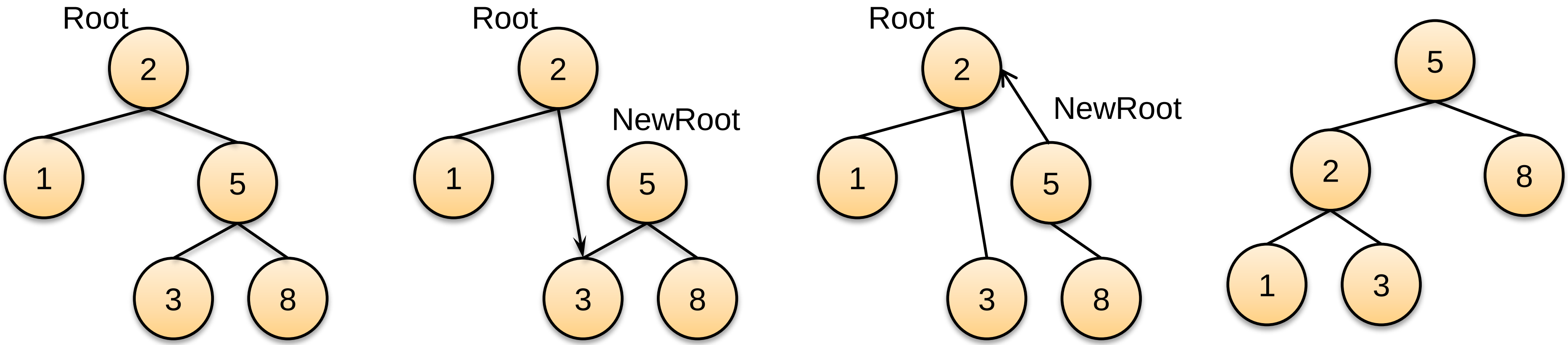
Правий поворот дерева

```
RotateRight (Root)
NewRoot=Root->L
Root->L= NewRoot->R
NewRoot->R=Root
Return NewRoot
```



Лівий поворот дерева

```
RotateLeft (Root)
NewRoot=Root->R
Root->R= NewRoot->L
NewRoot->L=Root
Return NewRoot
```



АВЛ-дерево (AVL-tree)

АВЛ-дерево (AVL-tree) – збалансоване за висотою двійкове дерево пошуку: для кожної його вершини висота її двох піддерев відрізняється не більше ніж на одиницю.

АВЛ-дерева названі за першими літерами прізвищ їх винахідників: **А**дельсона-**В**ельського та **Л**андіса, які вперше в 1962 р. запропонували застосовувати АВЛ-дерева.

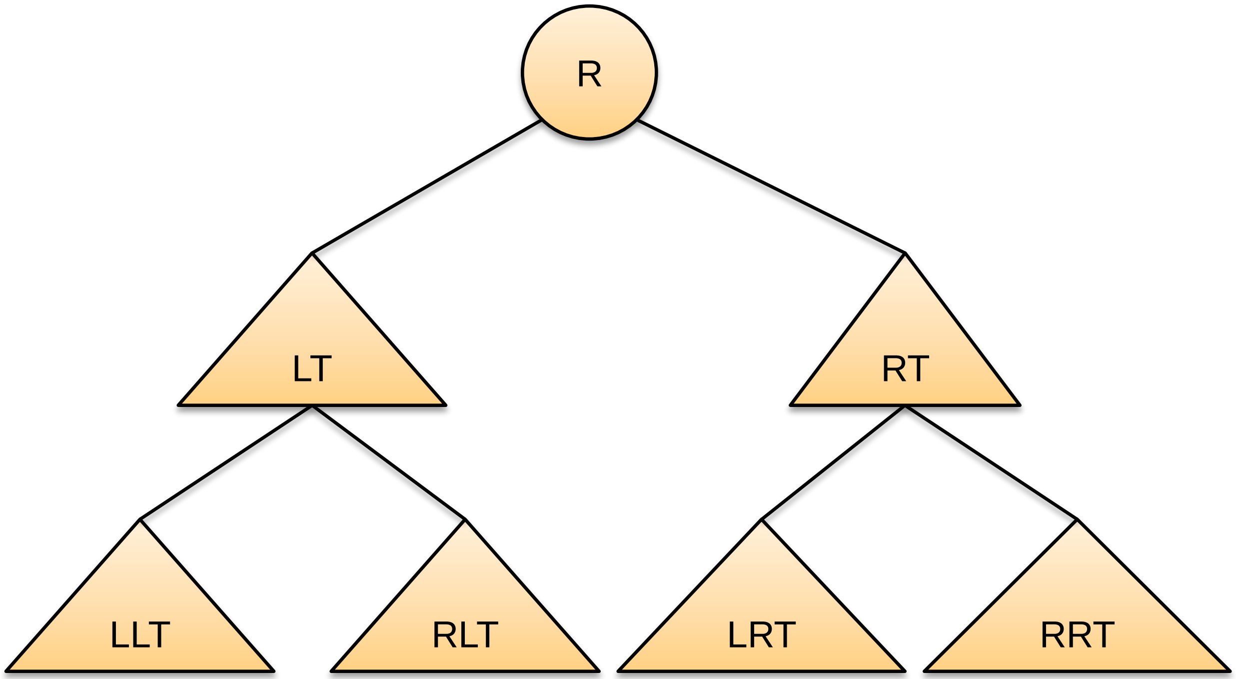
Основна суть даної структури: якщо вставлення або видалення елемента призводить до порушення збалансованості дерева, то виконують його балансування.

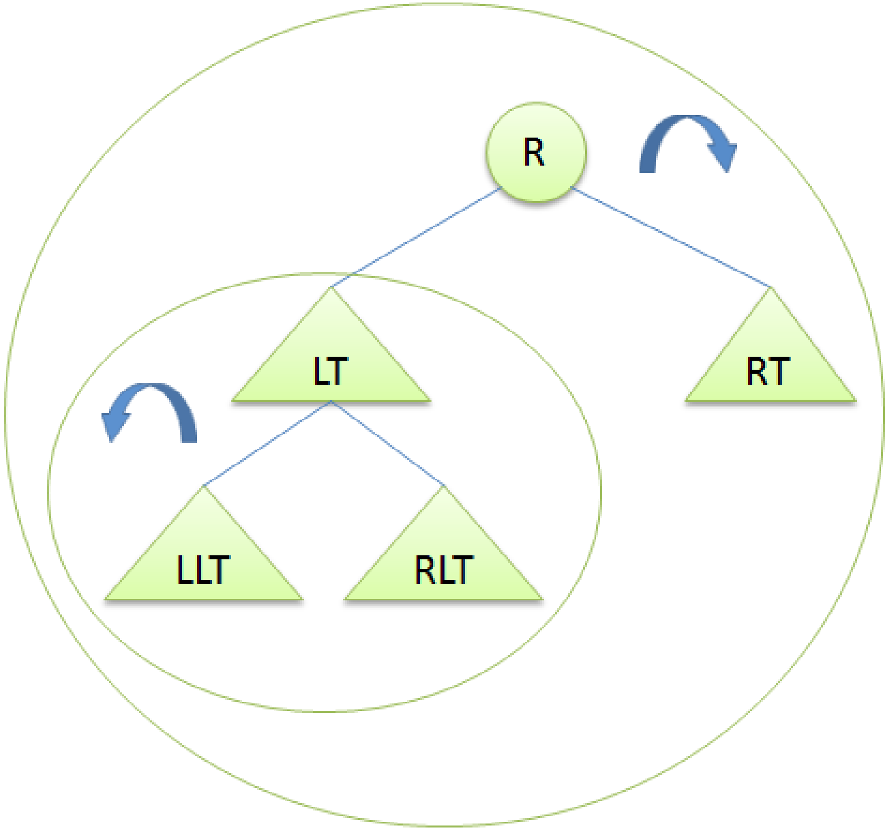
При **додаванні вузла** у АВЛ-дерево можливі 3 ситуації (розглянемо на прикладі додавання в ліве піддерево):

1) якщо висота лівого піддерева менша висоти правого $h(L) < h(R)$, то додавання нової вершини зрівняє висоти піддерев $h(L) = h(R)$, а балансування навіть покращиться;

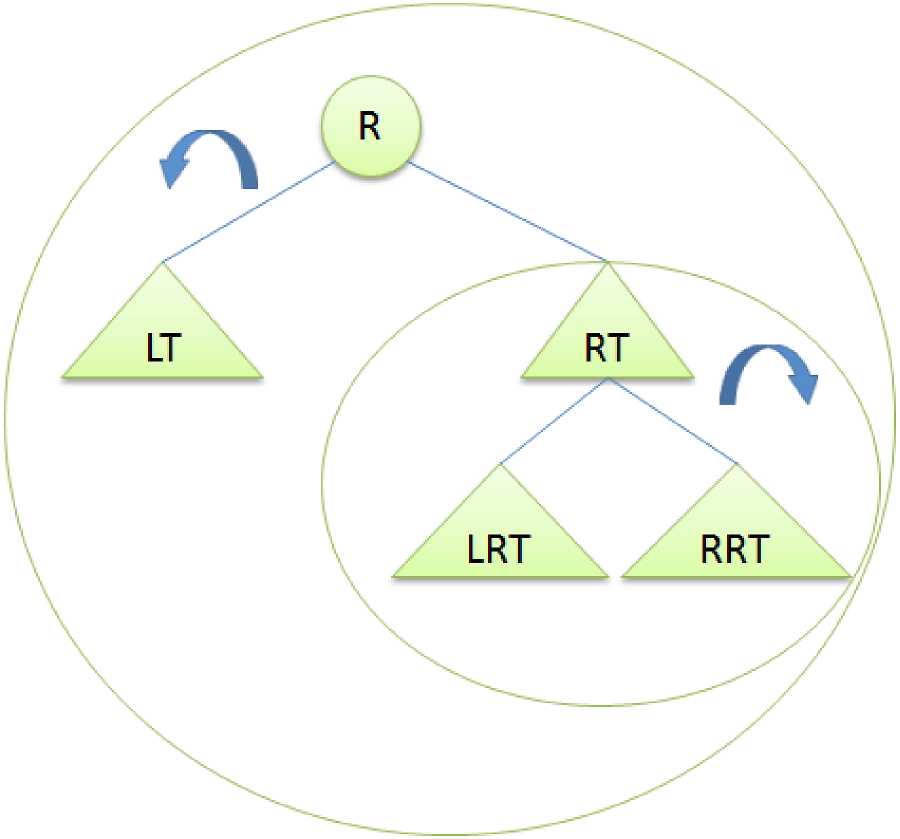
2) якщо $h(L) = h(R)$, то в разі додавання нової вершини висота лівого піддерева стане більша на одиницю, але критерій збалансованості не буде порушений;

3) якщо $h(L) > h(R)$, то додавання нової вершини призведе до такої ситуації: $h(L) – h(R) = 2$. У такому випадку потрібно здійснити балансування дерева.


R
LT
RT
LLT
RLT
LRT
RRT



Великий правий поворот дерева



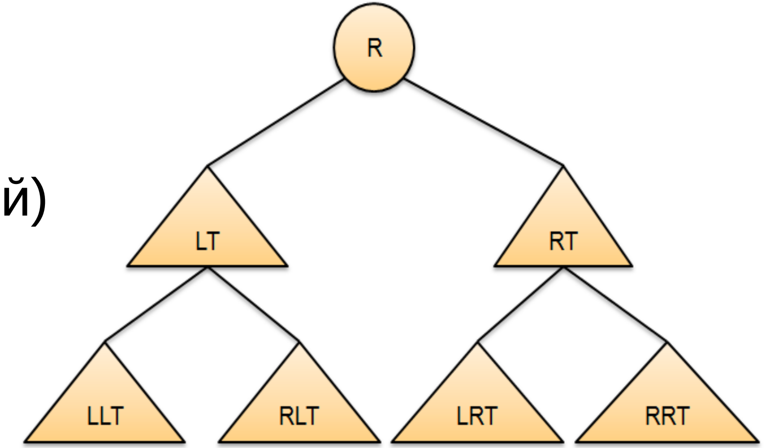
Великий лівий поворот дерева

Алгоритм балансування АВЛ-дерева

1) $h(LT) - h(RT) = 2$, то:

якщо $h(LLT) \geq h(RLT)$ – звичайний правий поворот відносно кореня;

якщо $h(LLT) < h(RLT)$ – великий (подвійний) правий поворот відносно кореня.



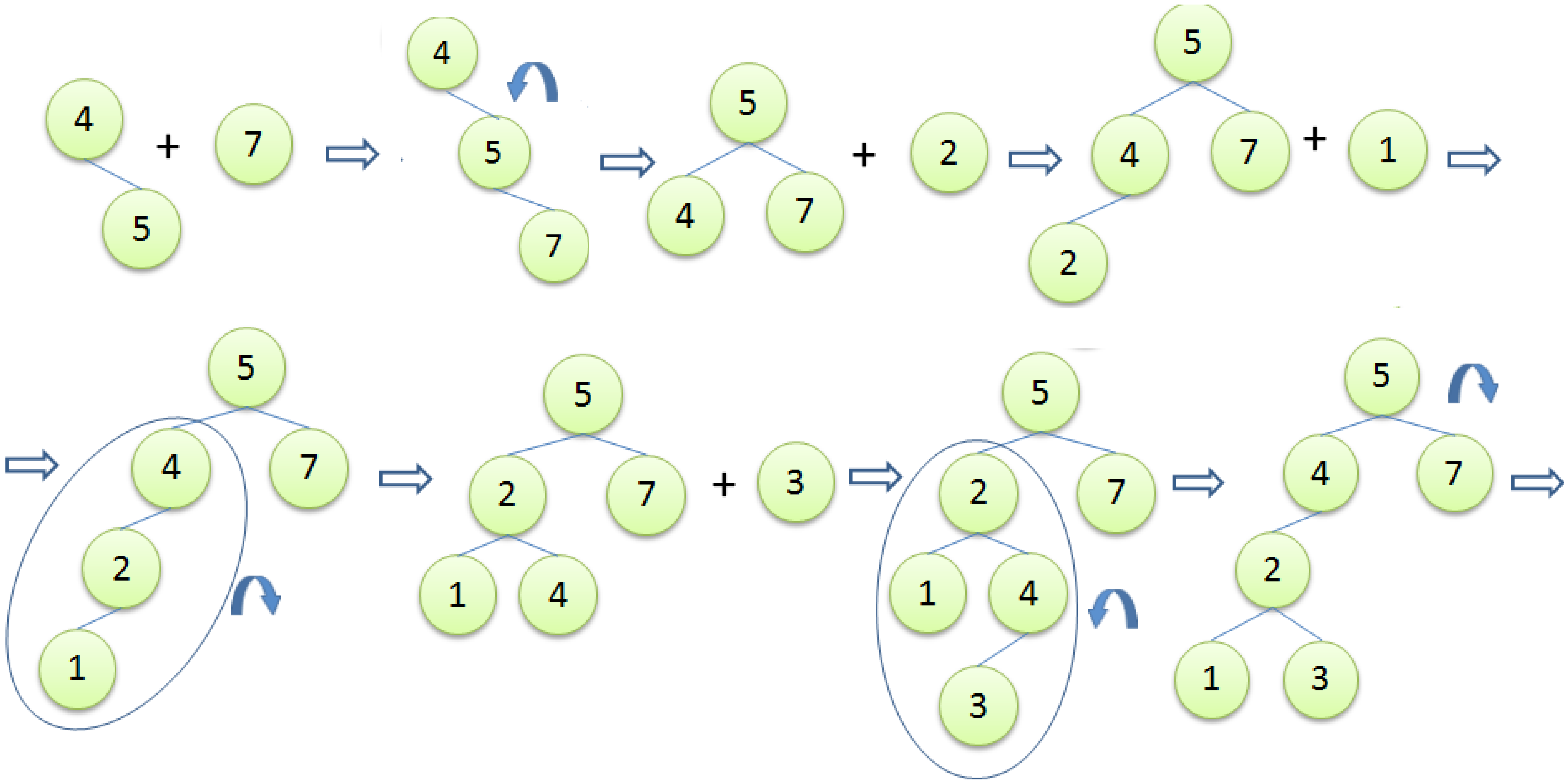
2) $h(LT) - h(RT) = -2$

якщо $h(LRT) \leq h(RRT)$ – звичайний лівий поворот відносно кореня;

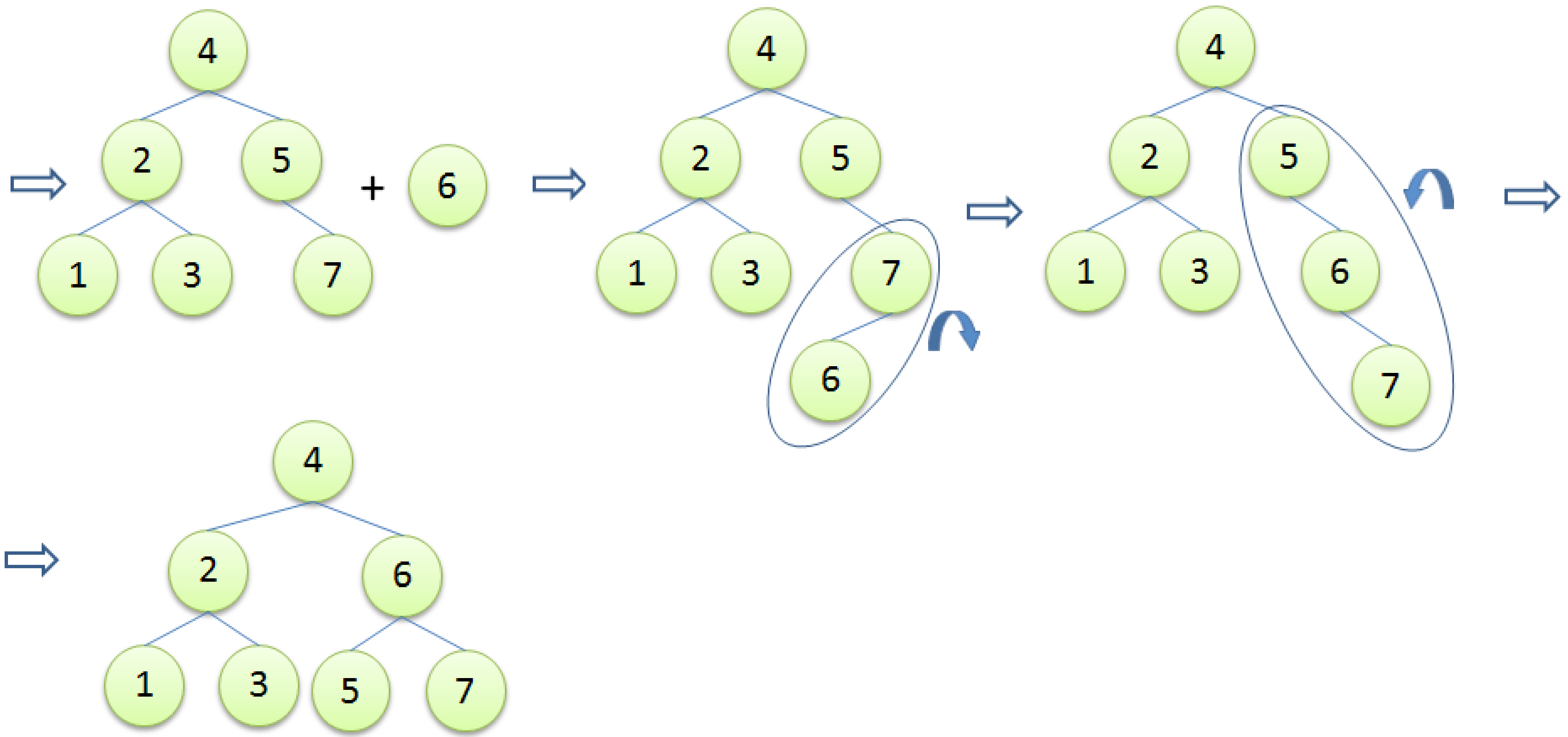
якщо $h(LRT) > h(RRT)$ – великий (подвійний) лівий поворот відносно кореня.

! Після додавання нового елемента слід оновити коефіцієнти збалансованості батьківських вузлів

Приклад побудови АВЛ-дерева із послідовності чисел 4, 5, 7, 2, 1, 3, 6



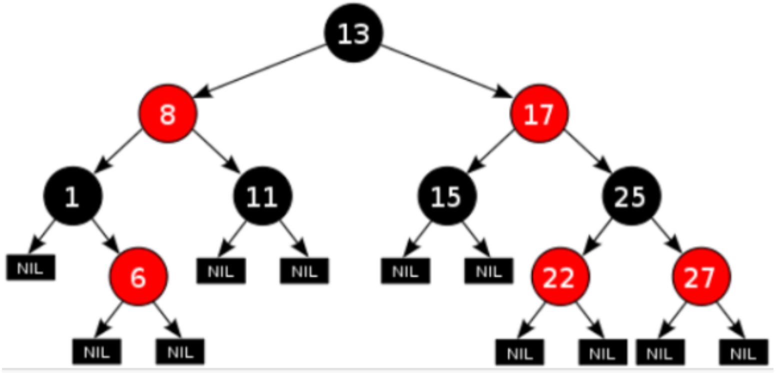
Приклад побудови АВЛ-дерева із послідовності чисел 4, 5, 7, 2, 1, 3, 6



Червоно-чорне дерево

Червоно-чорне дерево – бінарне дерево пошуку, для якого виконуються властивості:

1) кожен вузол або червоний, або чорний;
2) корінь завжди чорний;
3) усі листки дерева фіктивні (нульові) і не містять даних, але належать до дерева і є чорні;
4) якщо вершина червона, то обидва її нащадки чорні;
5) усі шляхи від кореня до листків містять однакову кількість чорних вершин.



Чорна висота (англ. black height) вершини *x* - кількість чорних вершин на шляху із *x* у листок, не враховуючи саму вершину *x*.

Теорема. Червоно-чорне дерево із N ключами має висоту $h = O(\log N)$

Лема. У червоно-чорному дереві з чорною висотою *bh* кількість внутрішніх вершин не менша $2^{bh} - 1$

Доведення (методом індукції).
Якщо *h*(*x*) = 0, то x – листок, тому b*h*(*x*) = 0, $2^{bh(x)} - 1 = 2^0 - 1 = 0$, лема правильна.
Розглянемо внутрішню вершину *x*. Якщо її нащадок *p* чорний, то висота *bh*(*p*) = *bh*(*x*)–1, а якщо червоний, то *bh*(*p*) = *bh*(*x*). Таким чином, за припущенням індукції, у піддеревах дерева з коренем *x* міститься не менше $2^{bh(x)-1} - 1$ вузлів. Отже, піддерево з коренем x містить принаймні $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 =$
$= 2^{bh(x)} - 1$ внутрішніх вузлів, що і треба було довести.

Теорема. Червоно-чорне дерево із N ключами має висоту $h = O(\log N)$

Доведення:

Оскільки тільки чорні вузли можуть мати червоних нащадків (властивість 4), то в найдовшому шляху від кореня до листка червоних вершин буде точно не більше половини, тому якщо звичайна висота дерева дорівнює *h*, то чорна висота дерева буде не менша *h*/2 – 1 і, згідно з лемою, кількість внутрішніх вершин у дереві $N \geq 2^{h/2} - 1$. Прологарифмувавши нерівність, матимемо:

$$\log(N+1) \geq h/2$$

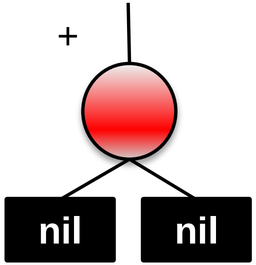
$$2\log(N+1) \geq h$$

$$h \leq 2\log(N+1) = O(\log N)$$

що і слід було довести.

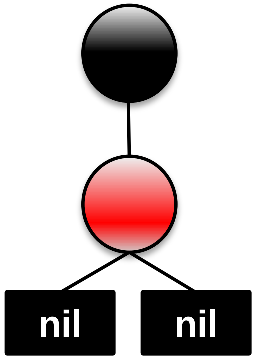
Алгоритм додавання вузла до червоно-чорного дерева:

1) згідно з алгоритмом додавання елемента до бінарного дерева пошуку знаходять відповідний листок і замінюють його на новий вузол червоного кольору із nil-нащадками.



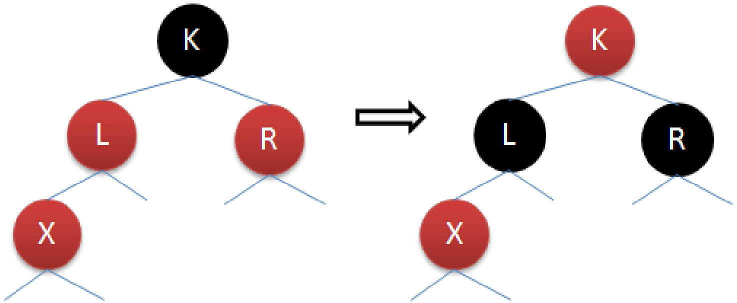
Далі перевіряють балансування;

2) якщо предок поточного вузла чорний, то властивість 4 не порушується. Властивість 5 не порушується, тому що поточний вузол має два чорні листкові нащадки, але оскільки він є червоний, шлях до кожного з цих нащадків містить таку ж кількість чорних вузлів, що і шлях до чорного листка, який було замінено поточним вузлом, так що властивість залишається правильною;

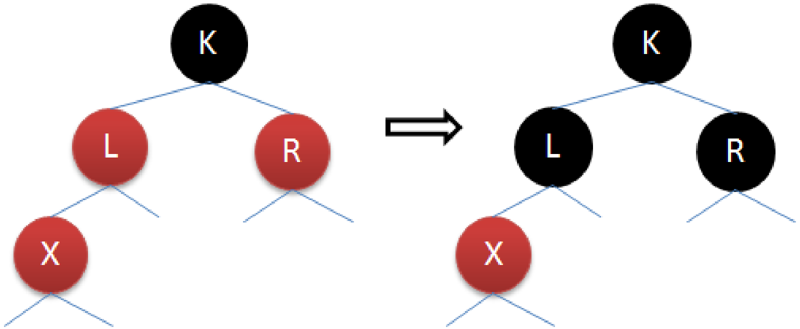


3) якщо предок нового елемента червоний, то слід розглянути три випадки (на прикладі додавання у ліве піддерево, додавання у праве піддерево виконують симетрично):

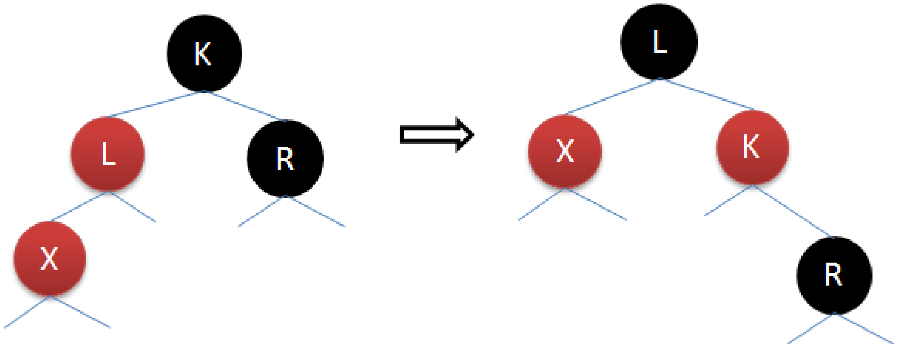
а) сусідній із предком елемент також червоний. У цьому разі просто перефарбовують предка і сусідній із ним елемент у чорний колір, а їх предка – у червоний. Перевіряють, чи не порушує він тепер балансування.



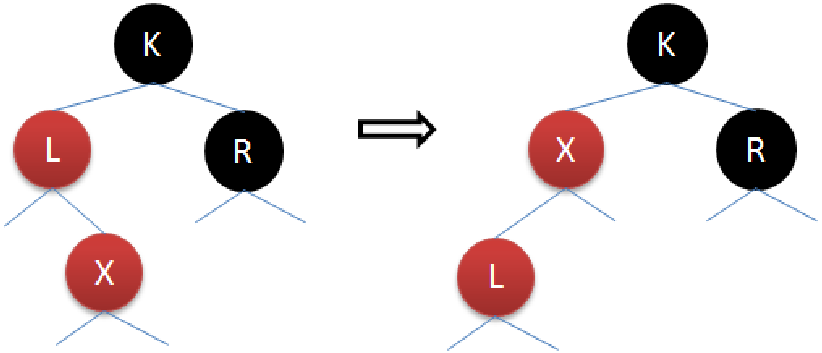
Якщо в результаті цих перефарбовувань можна дійти до кореня, то в ньому в будь-якому випадку задають чорний колір;



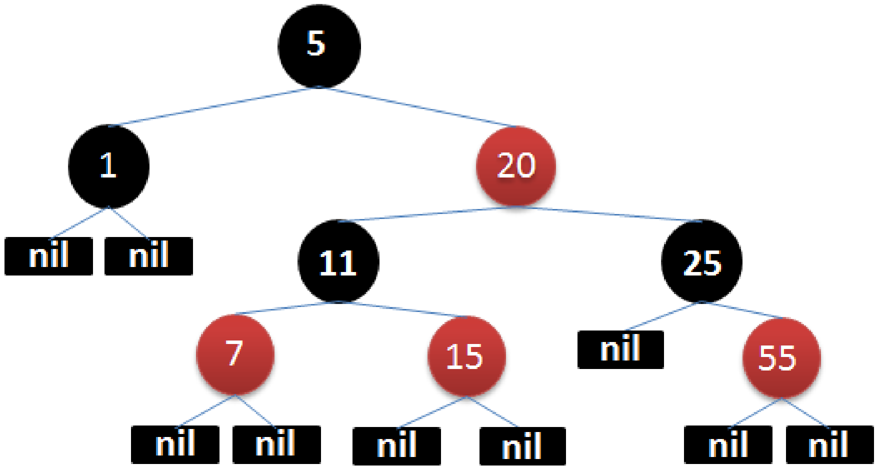
б) сусідній із предком елемент чорний або його немає, а доданий елемент є лівий нащадок свого предка. Перефарбовують предка у чорний, а його предка – у червоний. Якщо виконати тільки перефарбування, то може порушитися сталість чорної висоти дерева за всіма гілками. Тому виконують правий поворот;

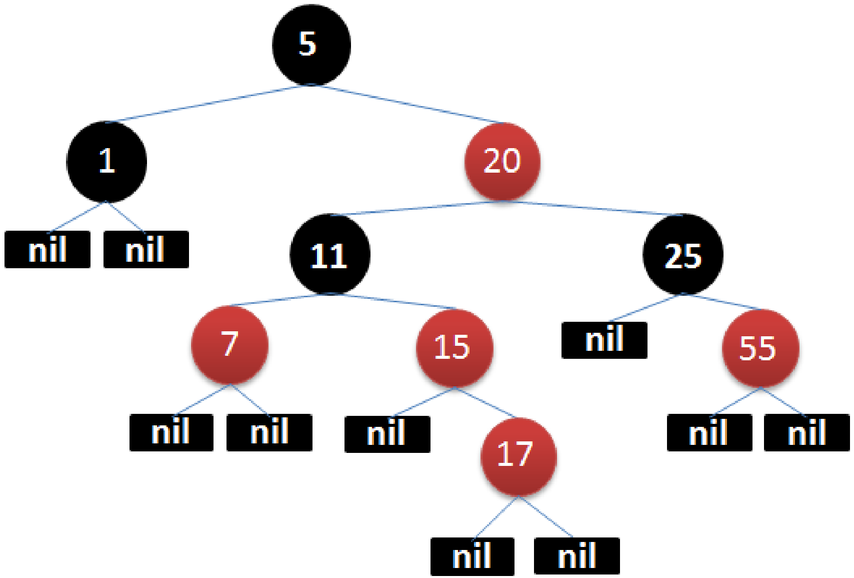


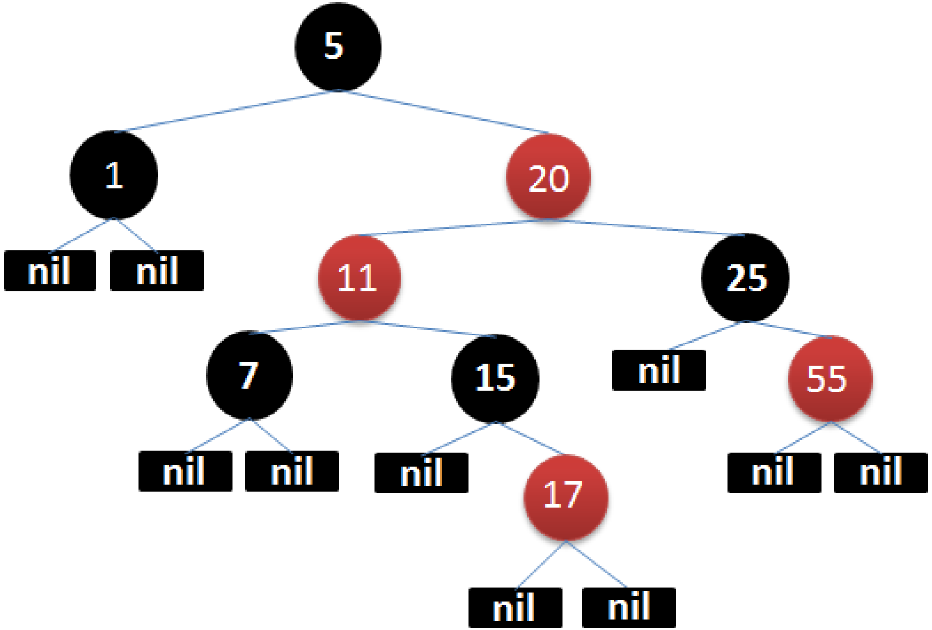
в) сусідній із предком елемент чорний або відсутній, а доданий елемент є правий нащадок свого предка. Виконують лівий поворот, який зробить доданий елемент лівим нащадком свого предка і переходять до ситуації 2.

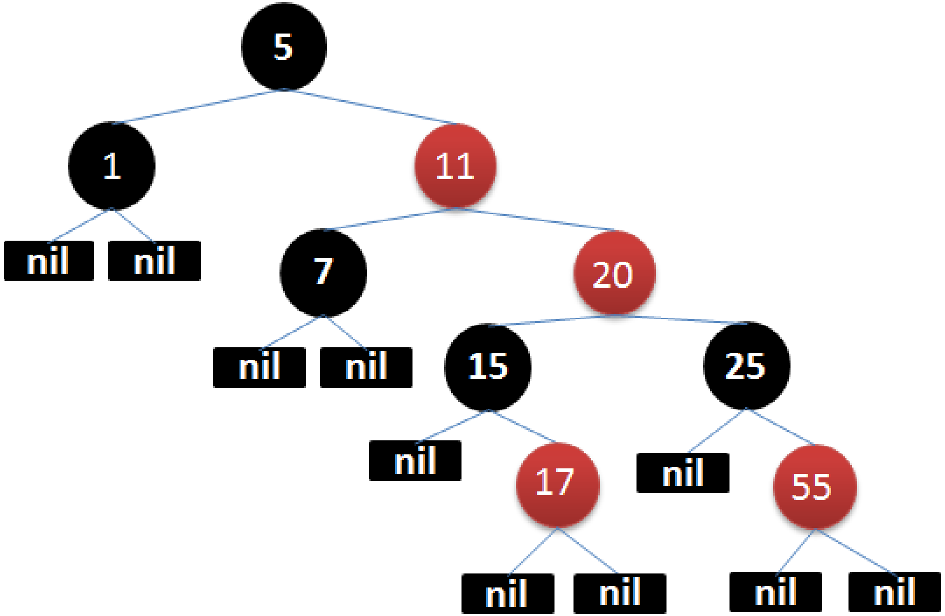


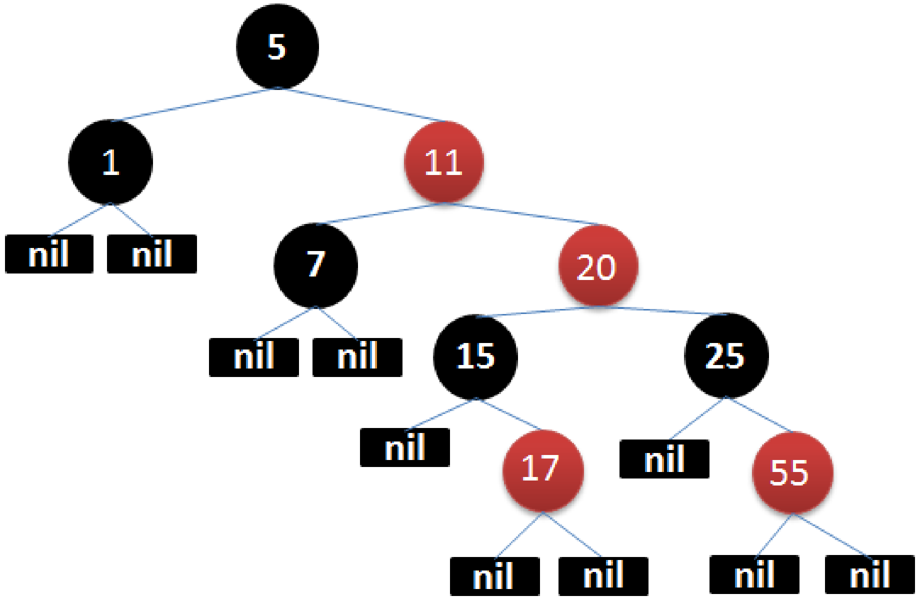
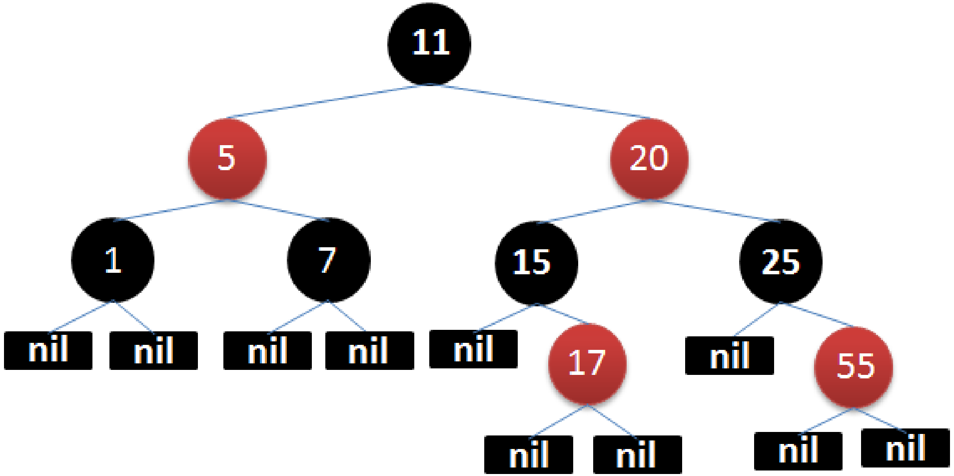
Приклад додавання вузла до червоно-чорного дерева






5
1
20
nil
nil
11
25
7
15
nil
55
nil
nil
nil
17
nil
nil
nil
nil

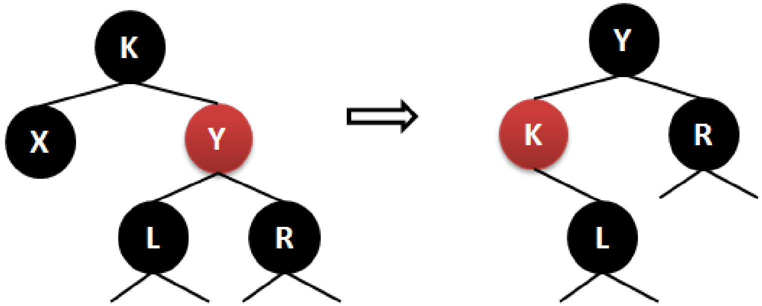

5
1
11
nil
nil
7
20
nil
nil
15
25
nil
17
nil
55
nil
nil
nil
nil


5
1
11
nil
nil
7
20
nil
nil
15
25
nil
17
nil
55
nil
nil
nil
nil

11
5
20
1
7
15
25
nil
nil
nil
nil
nil
17
nil
55
nil
nil
nil
nil

Алгоритм видалення вузла з червоно-чорного дерева

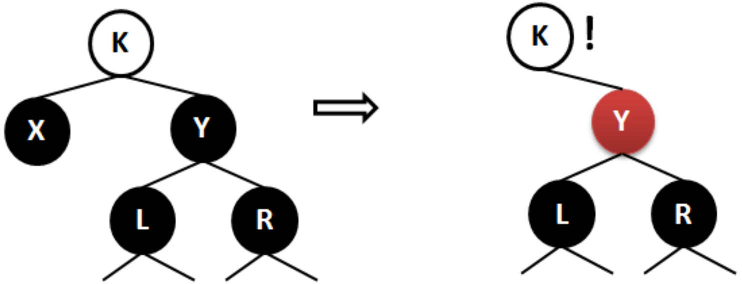
Оскільки в разі видалення червоної вершини властивості дерева не порушуються, то відновлення балансування потрібне тільки у випадку видалення чорної. Розглянемо можливі випадки (на прикладі видалення вершини X із лівого піддерева, видалення із правого піддерева виконують симетрично):

1) якщо сусідня з видаленою вершина (позначимо її Y) червона, то здійснимо обертання навколо ребра між цією вершиною і предком. Зафарбуємо її в чорний, а предка – у червоний кольори

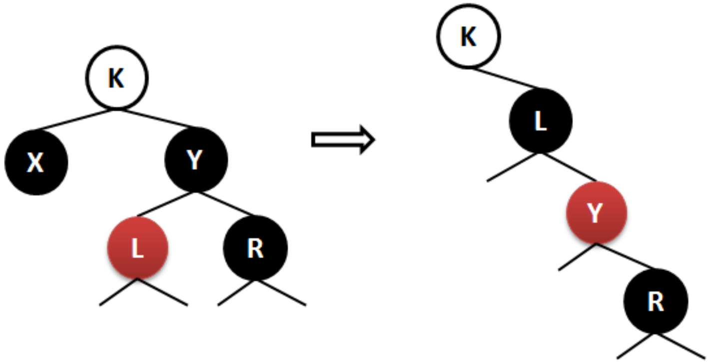


2) якщо вершина Y чорна, розглянемо її нащадків. Матимемо три випадки:

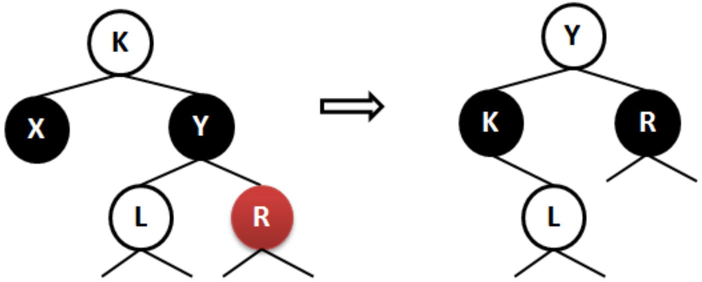
а) обидва нащадки чорні. Зафарбуємо вершину Y у червоний колір і розглянемо далі її предка



б) правий нащадок чорний, а лівий червоний. Перефарбуємо вершину Y у червоний колір, її лівого нащадка – у чорний і виконуємо правий поворот піддерева із коренем у вершині Y



в) правий нащадок червоний. Перефарбуємо вершину Y у колір її предка, а правий нащадок і предка – у чорний, здійснимо лівий поворот



https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

Data Structure Visualizations

About
Algorithms
F.A.Q
Known Bugs / Feature Requests
Java Version
Flash Version
Create Your Own / Source Code
Contact

David Galles
Computer Science
University of San Francisco

Currently, we have visualizations for the following data structures and algorithms:

- Basics
  - Stack: Array Implementation
  - Stack: Linked List Implementation
  - Queues: Array Implementation
  - Queues: Linked List Implementation
  - Lists: Array Implementation (available in java version)
  - Lists: Linked List Implementation (available in java version)
- Recursion
  - Factorial
  - Reversing a String
  - N-Queens Problem
- Indexing
  - Binary and Linear Search (of sorted list)
  - Binary Search Trees
  - AVL Trees (Balanced binary search trees)
  - Red-Black Trees
  - Splay Trees
  - Open Hash Tables (Closed Addressing)
  - Closed Hash Tables (Open Addressing)
  - Closed Hash Tables, using buckets
  - Trie (Prefix Tree, 26-ary Tree)
  - Radix Tree (Compact Trie)
  - Ternary Search Tree (Trie with BST of children)
  - B Trees
  - B+ Trees
- Sorting
  - Comparison Sorting
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
    - Shell Sort