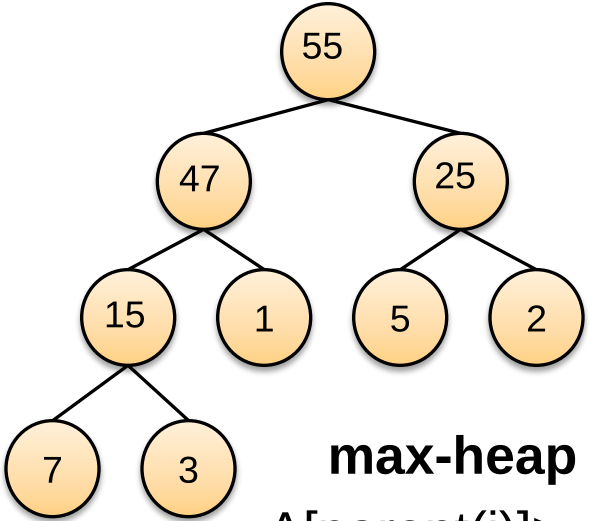
Піраміда.
Двійкова купа.
Binary Heap

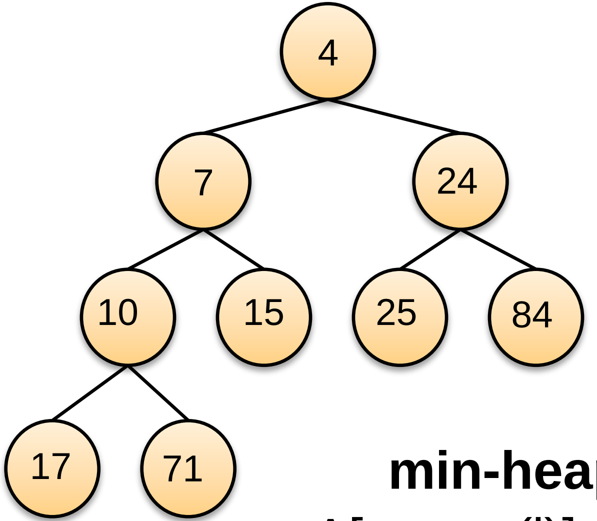
Двійкова купа, або піраміда (англ. binary heap) – бінарне дерево, для якого виконуються такі три умови:

1) значення в будь-якій вершині не менше, ніж значення її нащадків (купа max-heap) або значення в будь-якій вершині не більше, ніж значення її нащадків (купа min-heap);
2) наявність на i-му рівні крім останнього 2^{i-1} вершин;
3) заповнення останнього рівня зліва направо.



max-heap

$A[parent(i)]>=A[i]$



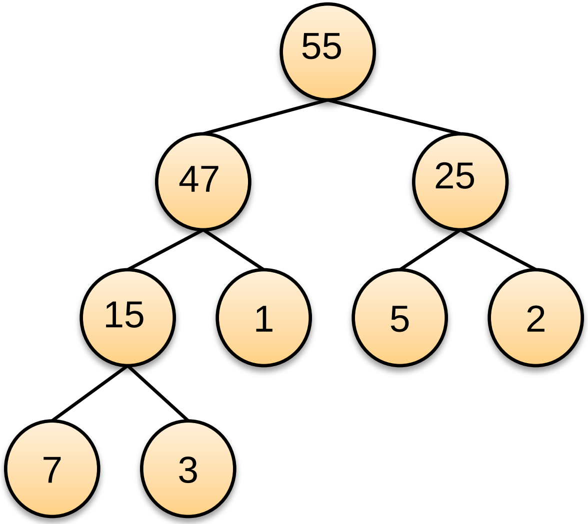
min-heap

$A[parent(i)]<=A[i]$

Реалізація піраміди

Двійкову купу найзручніше зберігати у вигляді масиву A[0 ... N–1]

Адреса будь-якої вершини в масиві : $2^{k-1}+i-2$,



A[55, 47, 25, 15, 1, 5, 2, 7, 3]

parent(i)

$A[\lceil \frac{i}{2} \rceil - 1]$

↑

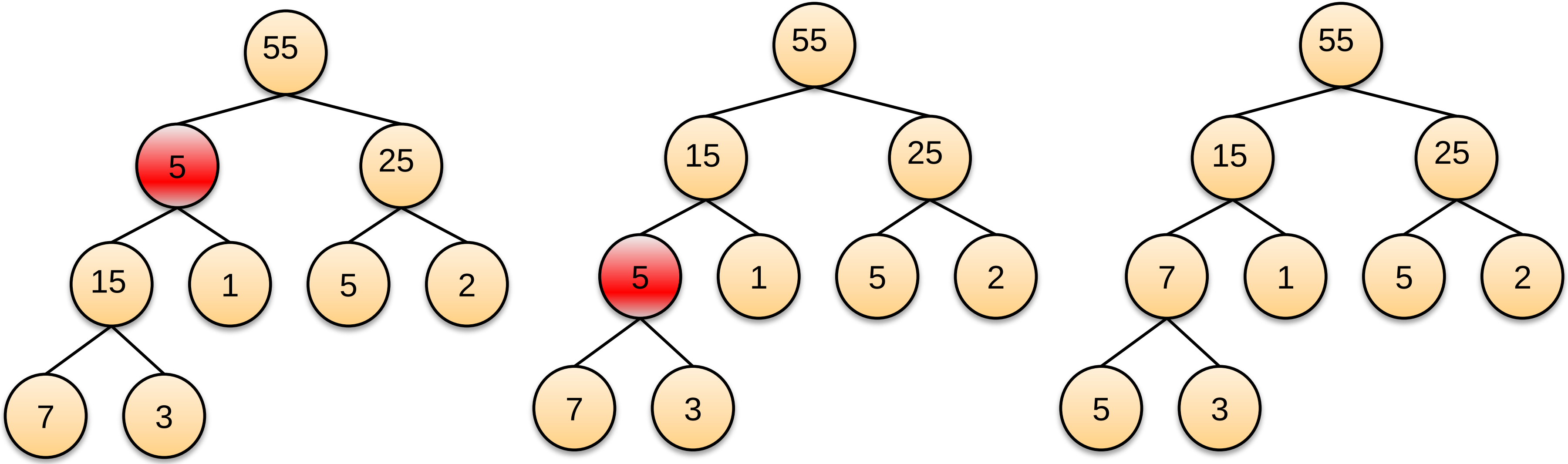
A[i]

↙ ↘

A[2i + 1] *left(i)*

A[2i + 2] *right(i)*

Підтримка властивості піраміди (heapify)

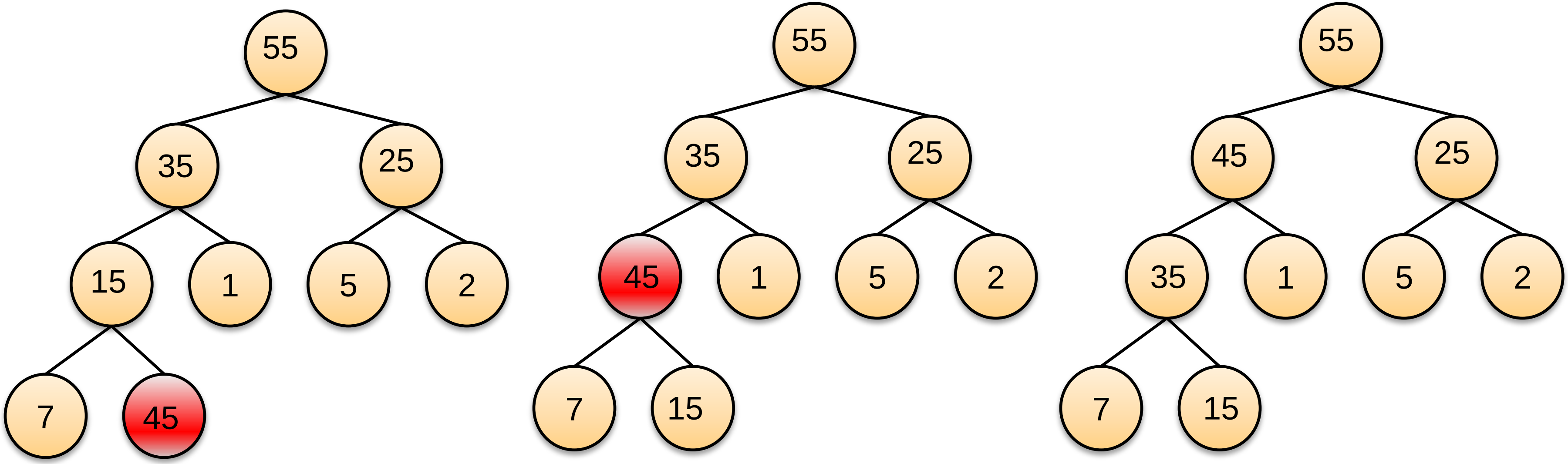


Підтримка властивості піраміди (heapify)

```
MaxHeapify(A, i)
p ← Left(i)
q ← Right(i)
if p ≤ heap_size[A] та A[p] > A[i]
     then largest ← p
     else largest ← i
if q ≤ heap_size[A] та A[q] > A[largest]
     then largest ← q
if largest ≠ i
     then Обміняти A[i] ↔ A[largest]
          MaxHeapify(A, largest)
```

https://edx.prometheus.org.ua/courses/KPI/Algorithms101/2015_Spring/pdfbook/0/

Додавання вузла до піраміди



Побудова піраміди

Варіант 1. Найбільш очевидний спосіб побудови купи – це по черзі додати всі його елементи.
Часова складність такого алгоритму – *O(NlogN).*

Варіант 2. Спочатку слід побудувати дерево з усіх елементів масиву, не дотримуючись основної властивості купи, а потім викликати метод heapify для всіх вершин, у яких є хоча б один нащадок (оскільки піддерева, що складаються з однієї вершини без нащадків, уже впорядковані). Нащадки гарантовано будуть у перших N / 2 вершин.
Часова складність такого алгоритму – *O(N).*

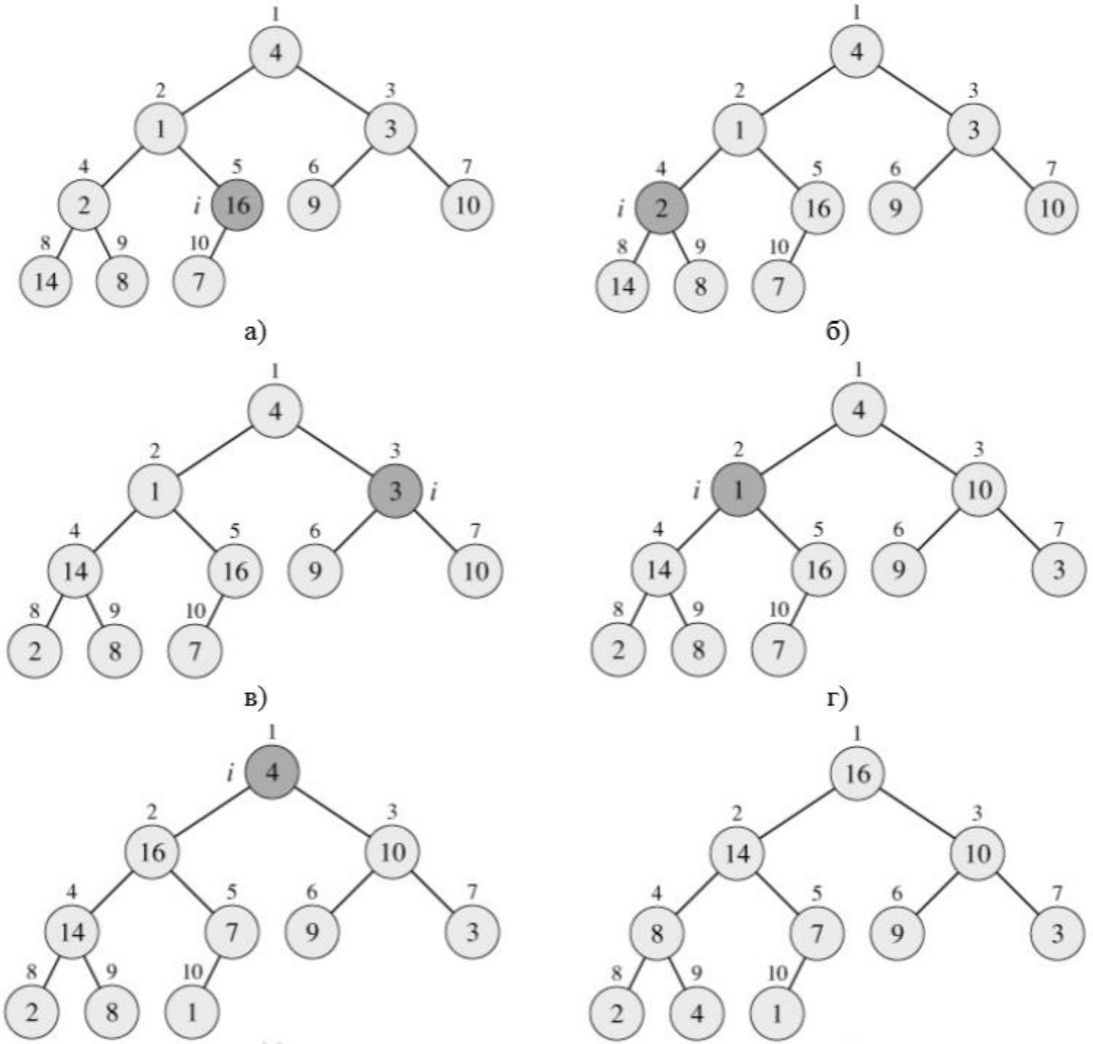
https://edx.prometheus.org.ua/courses/KPI/Algorithms101/2015_Spring/pdfbook/0/

Приклад

Побудова піраміди
з масиву

A	4	1	3	2	16	9	10	14	8	7

```
BuildMaxHeap(A)
heap_size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    do MaxHeapify(A, i)
```



https://edx.prometheus.org.ua/courses/KPI/Algorithms101/2015_Spring/pdfbook/0/

Застосування пірамід

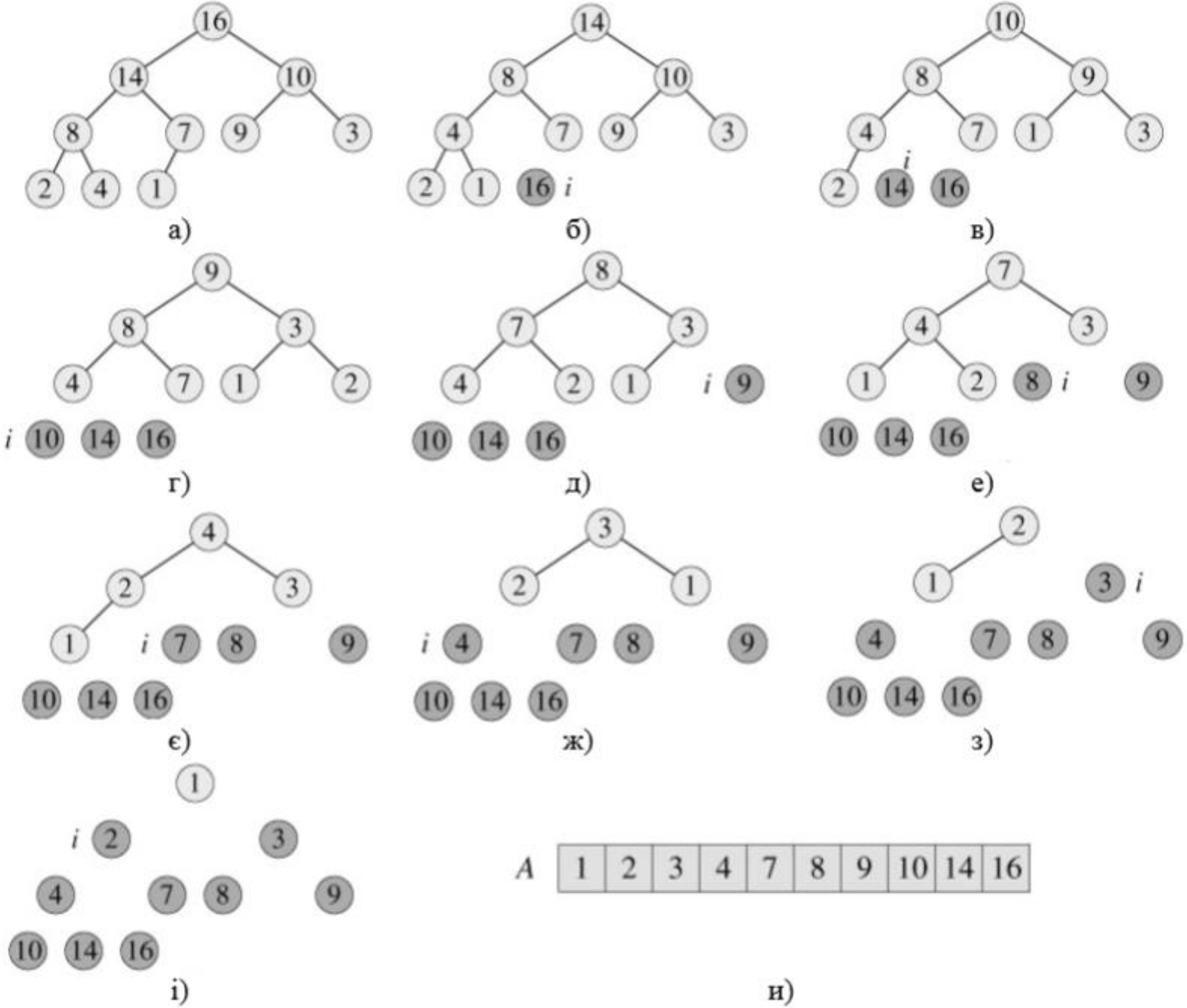
- Пірамідальне сортування
- Черги з пріоритетами

Пірамідальне сортування

```
HeapSort(A)
  BuildMaxHeap(A)
  for i ← length[A] downto 2
      do Обміняти A[1] ↔ A[i]
         heap_size[A] ← heap_size[A] - 1
         MaxHeapify(A, 1)
```

Часова складність алгоритму

O(NlogN)



https://edx.prometheus.org.ua/courses/KPI/Algorithms101/2015_Spring/pdfbook/0/

Черги з пріоритетами

Черга з пріоритетами (priority queue) – це структура даних, призначена для обслуговування множини S, з кожним елементом якої пов’язане певне значення, яке називається ключем.

Основні операції:

- Insert (S,x) – вставка елемента x в множину S;
- Maximum(S) – повертає елемент множини S з найбільшим ключем;
- ExtractMax(S) – повертає елемент з найбільшим ключем, при цьому видаляючи його з множини S;
- IncreaseKey(S,x,k) – збільшує значення ключа, відповідного елементу x, шляхом заміни його значенням k. Передбачається, що величина k не менше поточного ключа елементу x.

```
HeapMaximum(A)
  return A[1]
```

```
HeapExtractMax(A)
  if heap_size[A] < 1
        then error "Черга порожня"
  max ← A[1]
  A[1] ← A[heap_size[A]]
  heap_size[A] ← heap_size[A] - 1
  MaxHeapify(A, 1)
  return max
```

```
HeapIncreaseKey(A, i, key)
  if key < A[i]
        then error "Новий ключ менше поточного"
  A[i] ← key
  while i > 1 та A[Parent(i)] < A[i]
        do Обміняти A[i] ↔ A[Parent(i)]
           i ← Parent(i)
```

```
MaxHeapInsert(A, key)
  heap_size[A] ← heap_size[A] + 1
  A[heap_size[A]] ← -∞
  HeapIncreaseKey(A, heap_size[A], key)
```

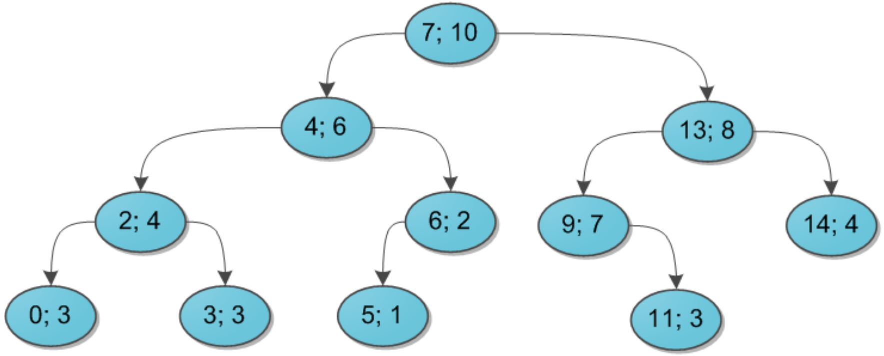
***** Для нумерації з 1, а не з 0

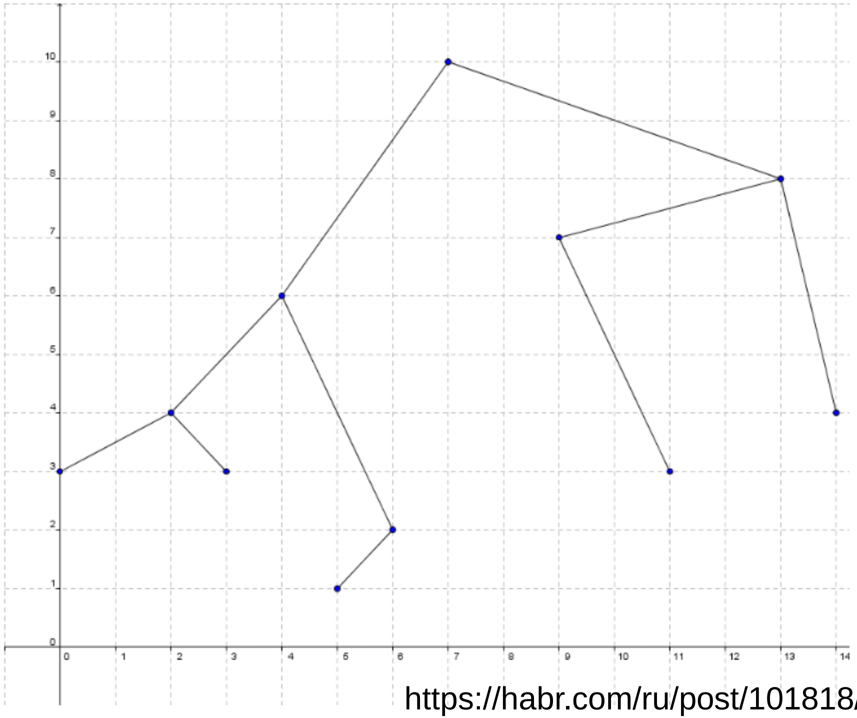
https://edx.prometheus.org.ua/courses/KPI/Algorithms101/2015_Spring/pdfbook/0/

Декартове дерево.

TREAP (TREE+HEAP)

Декартове дерево – це структура даних, яка об'єднує бінарне дерево пошуку і бінарну купу. У кожній вершині декартового дерева зберігаються два параметри: ключ x і пріоритет y. При цьому за ключами виконується властивість дерева пошуку: $key[X.left] < key[X] \leq key[X.right]$, а за пріоритетами – купи (значення в будь-якій вершині не менше, ніж значення її нащадків)





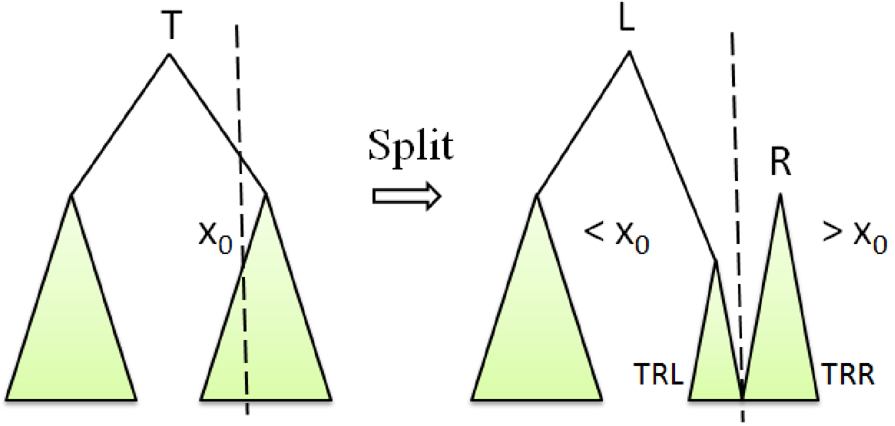
https://habr.com/ru/post/101818/

Допоміжні операції. Split.

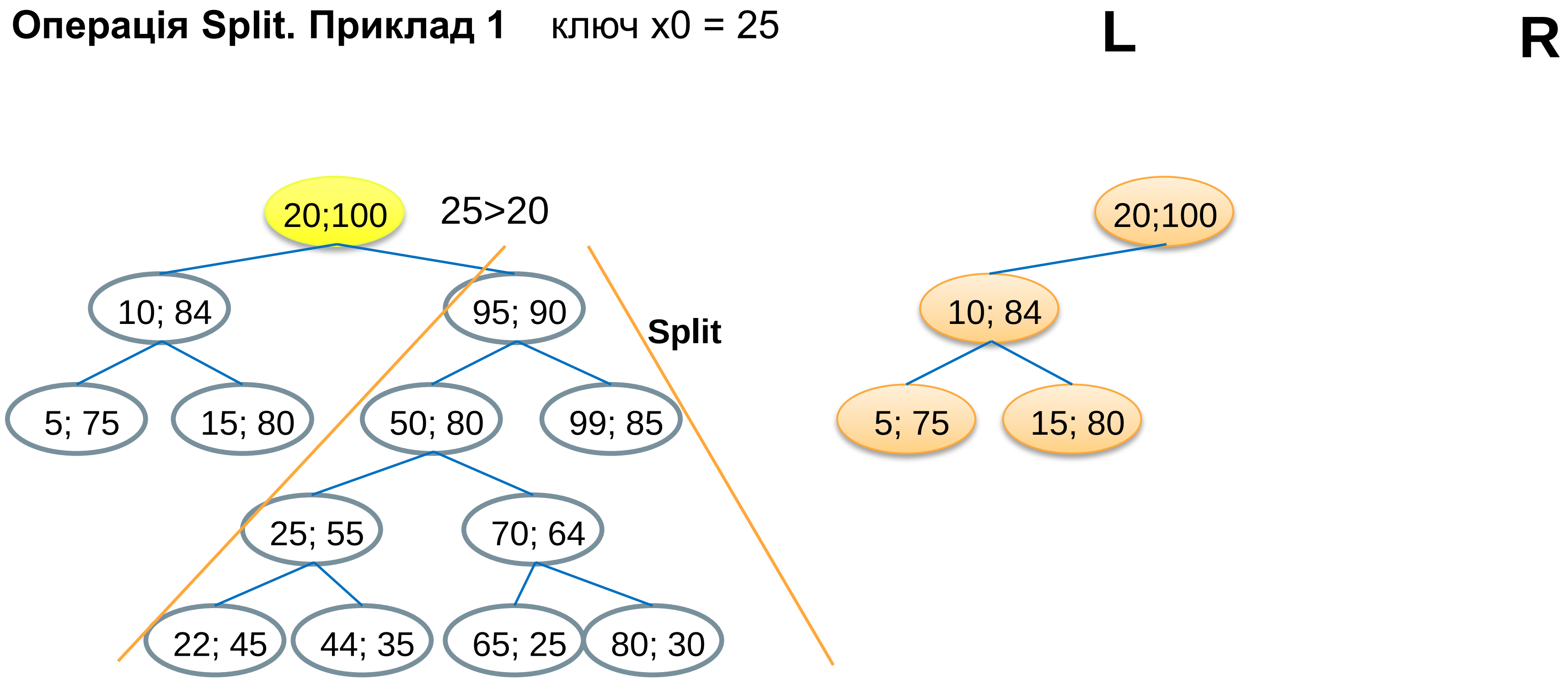
Операція Split. Параметрами є коректне декартове дерево T і ключ x0.

Суть операції: розділити дерево на два дерева так, щоб в одному із них (L) виявилися всі елементи вихідного дерева із ключами, меншими x0, а в іншому (R) – із більшими.

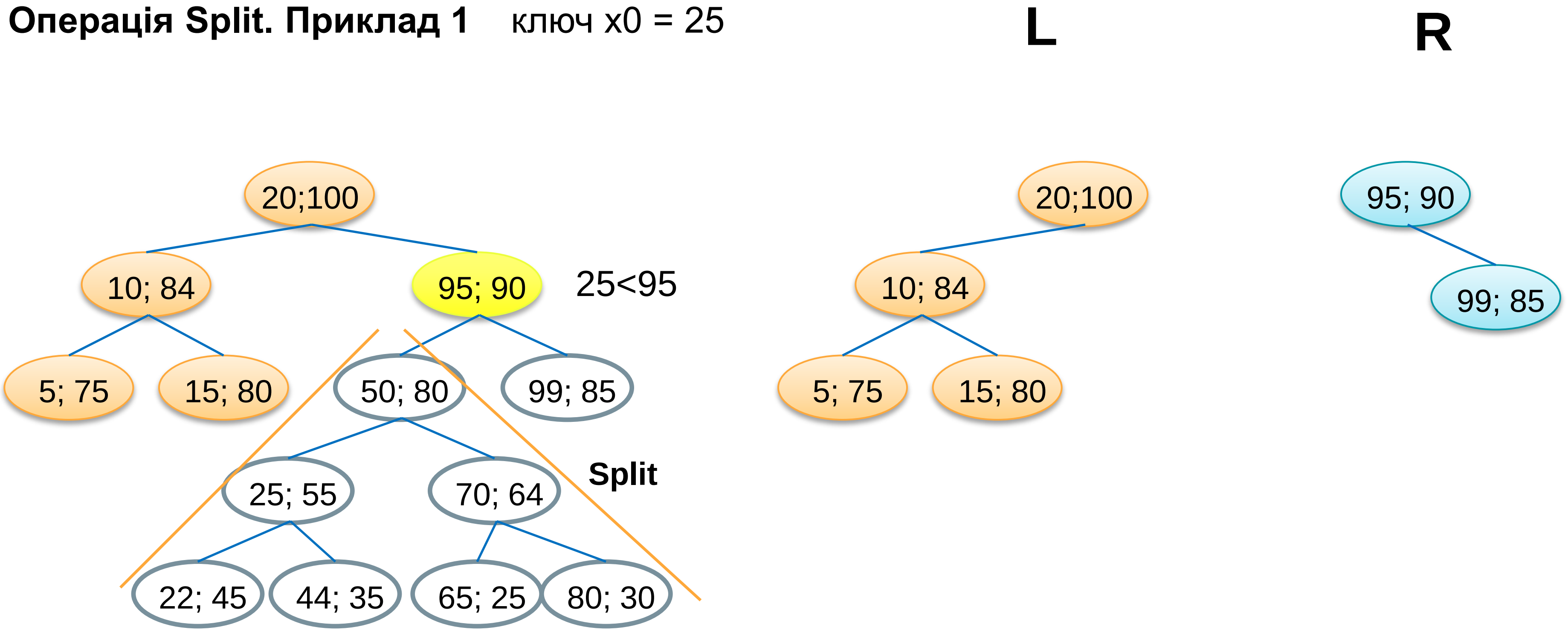
```
split(T: Treap, x0: int):
  if T == ∅
    return (∅, ∅)
  else if x0 > T.x
    (L, R) = split(T.right, x0)
    T.right = L
    return (T, R)
  else
    (L, R) = split(T.left, x0)
    T.left = R
    return (L, T)
```



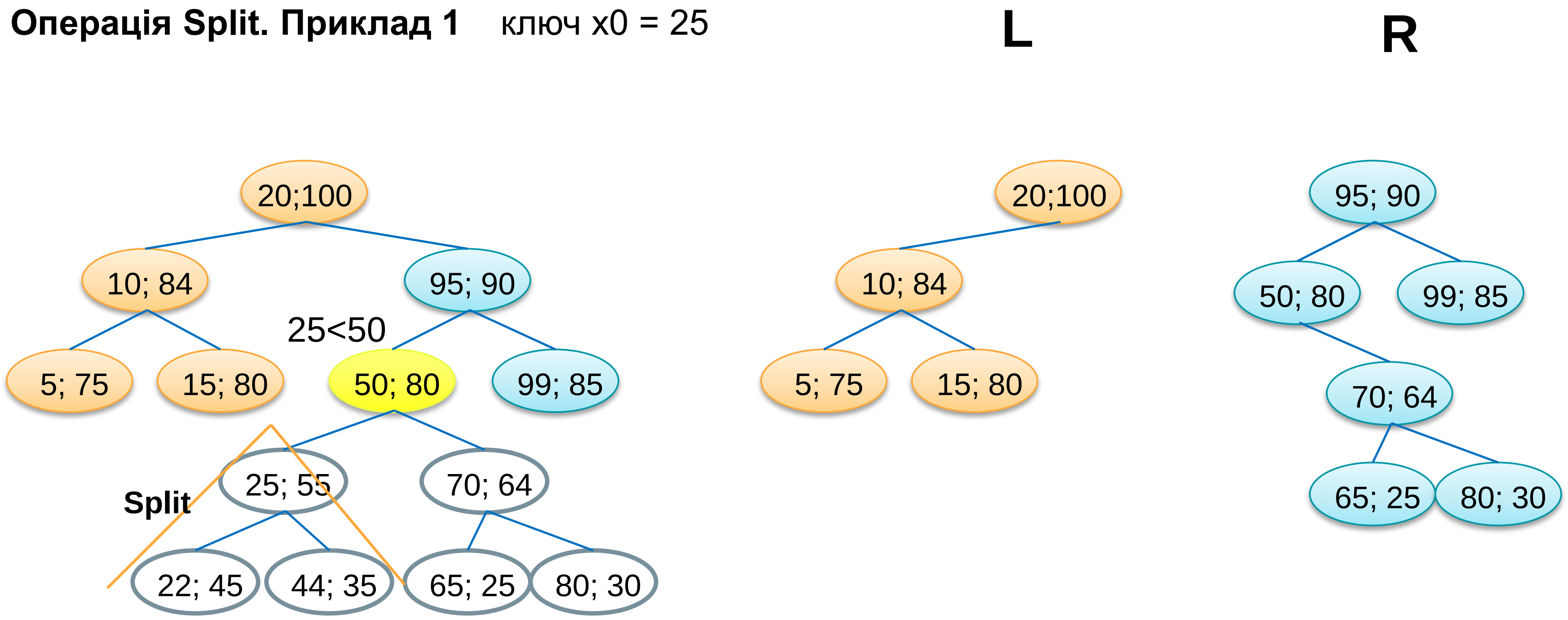
Операція Split. Приклад 1 ключ x0 = 25



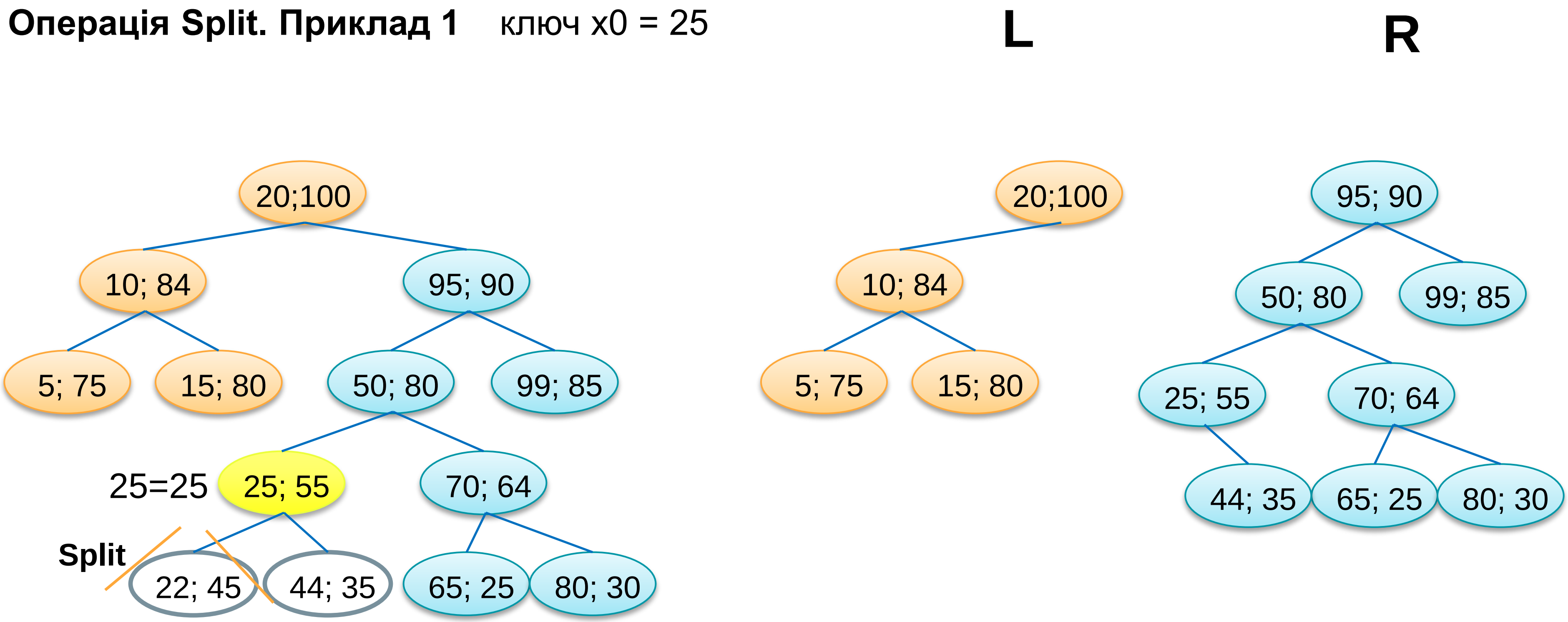
Операція Split. Приклад 1 ключ x0 = 25



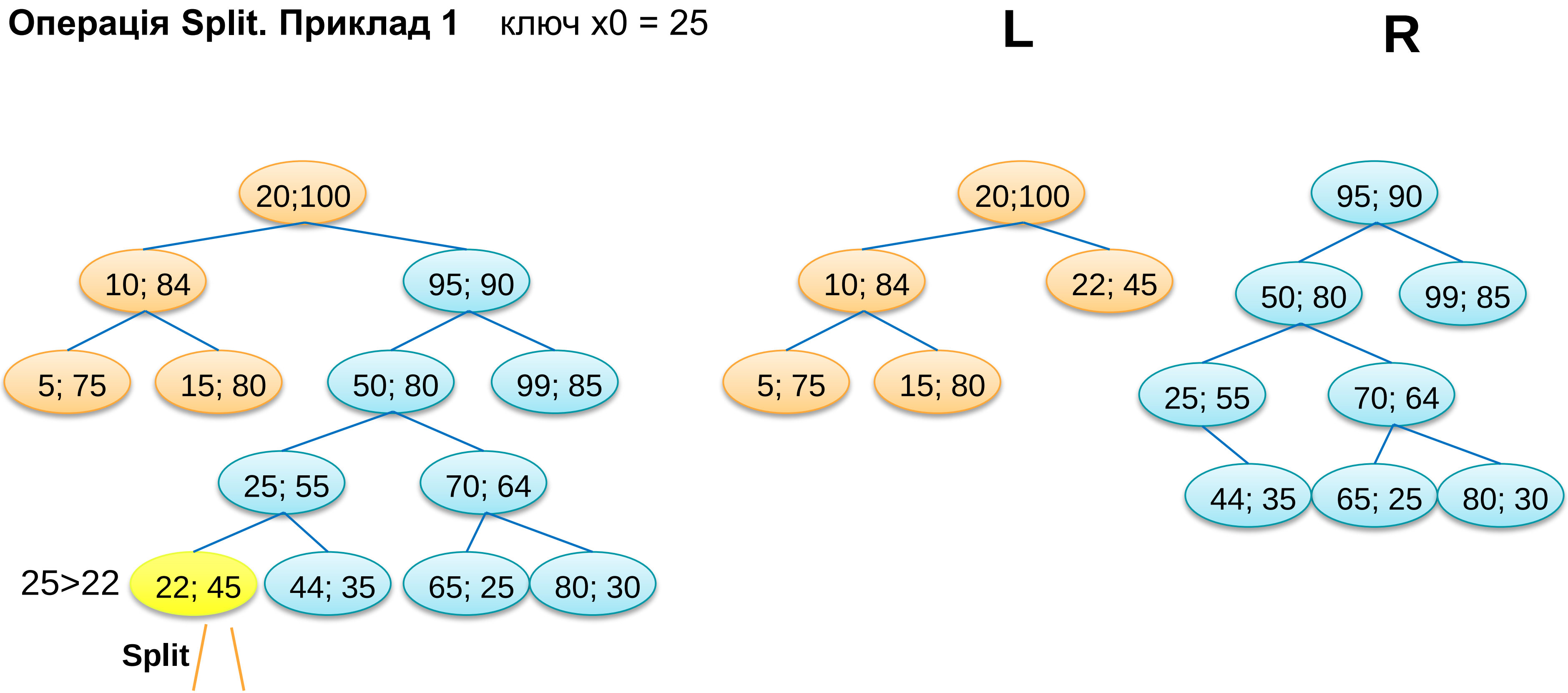
Операція Split. Приклад 1 ключ x0 = 25



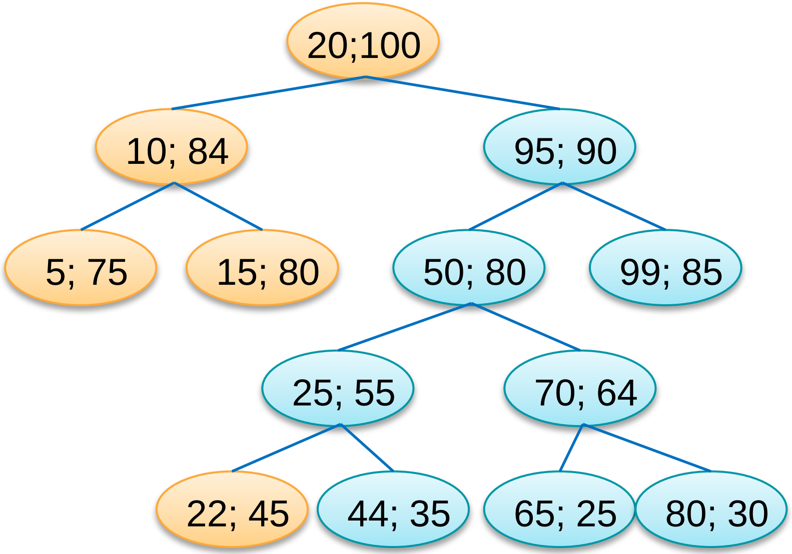
Операція Split. Приклад 1 ключ x0 = 25

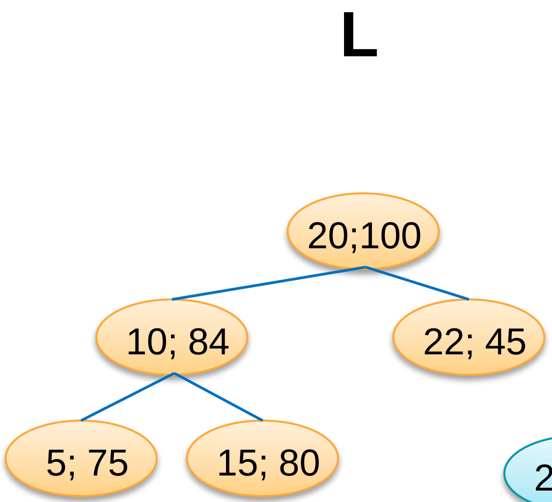


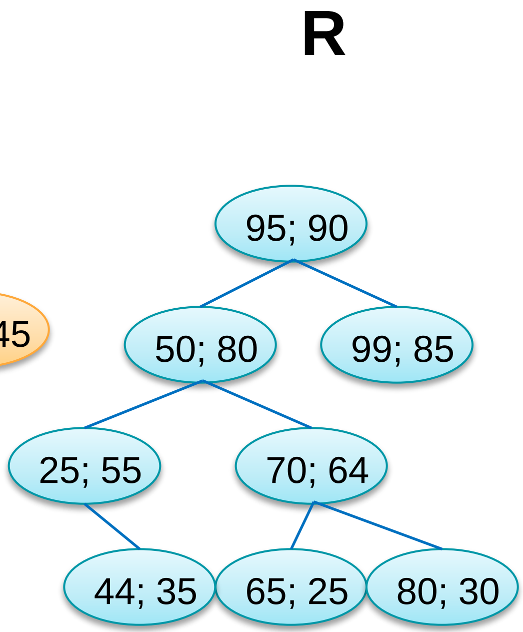
Операція Split. Приклад 1 ключ x0 = 25



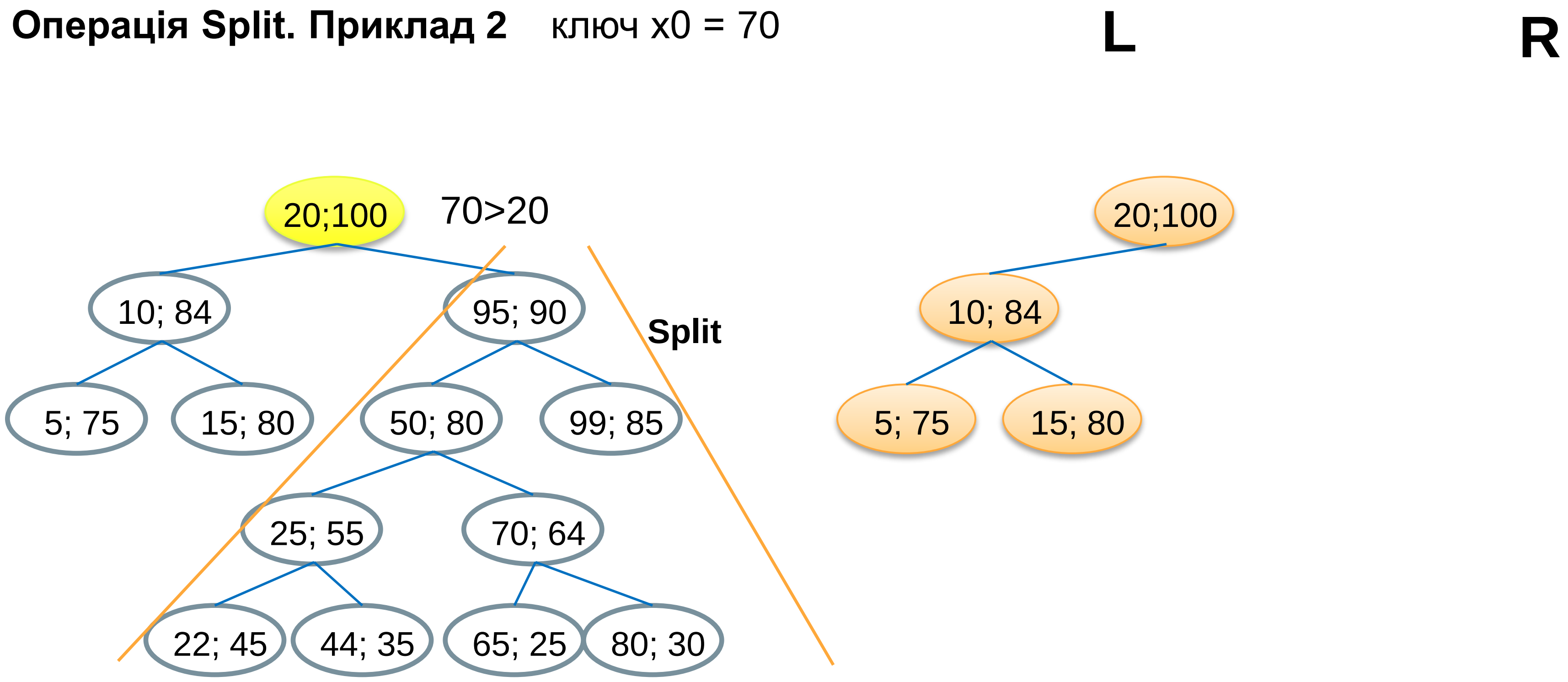
Операція Split. Приклад 1 ключ x0 = 25



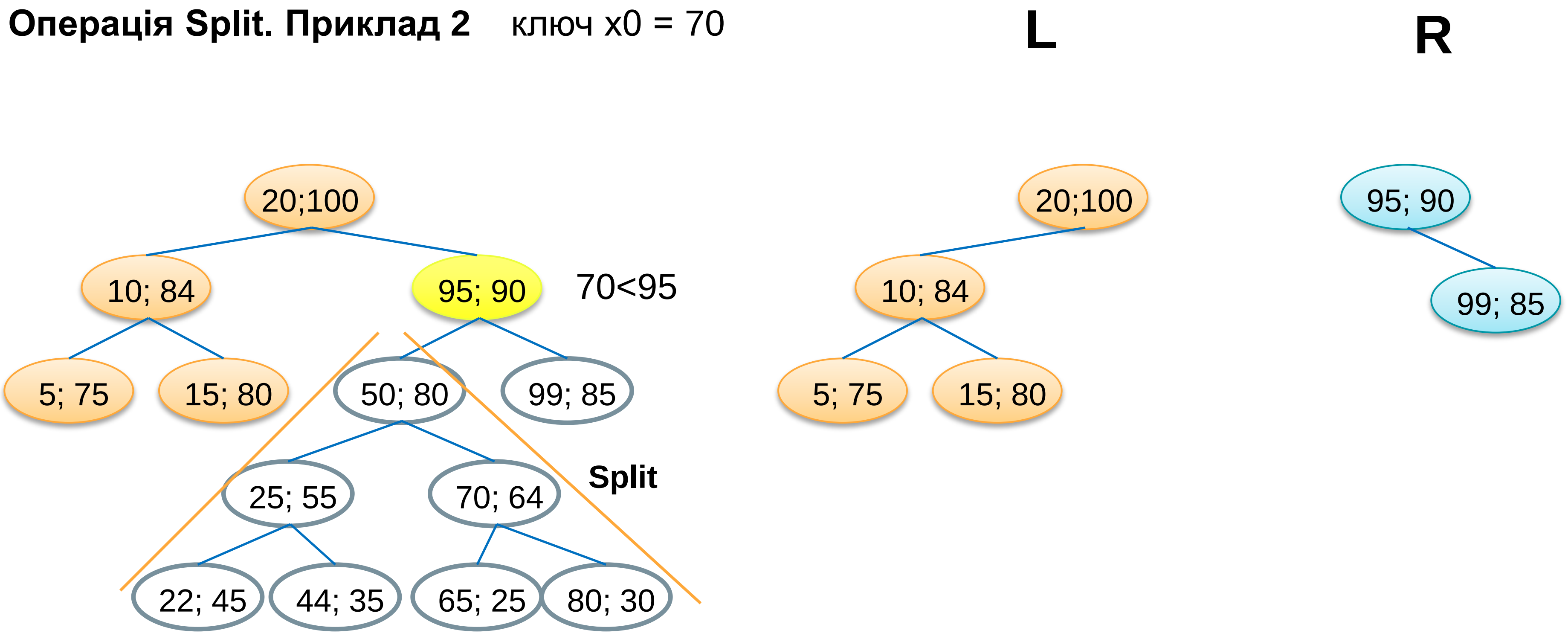




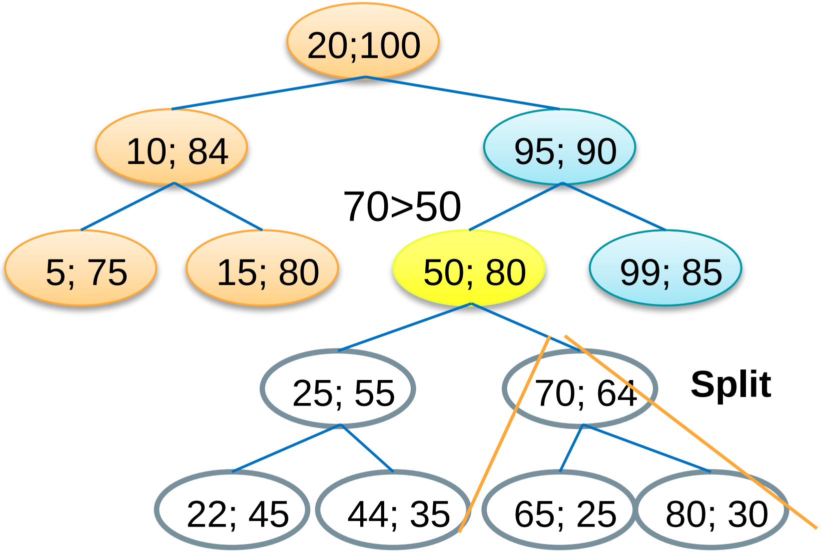
Операція Split. Приклад 2 ключ x0 = 70

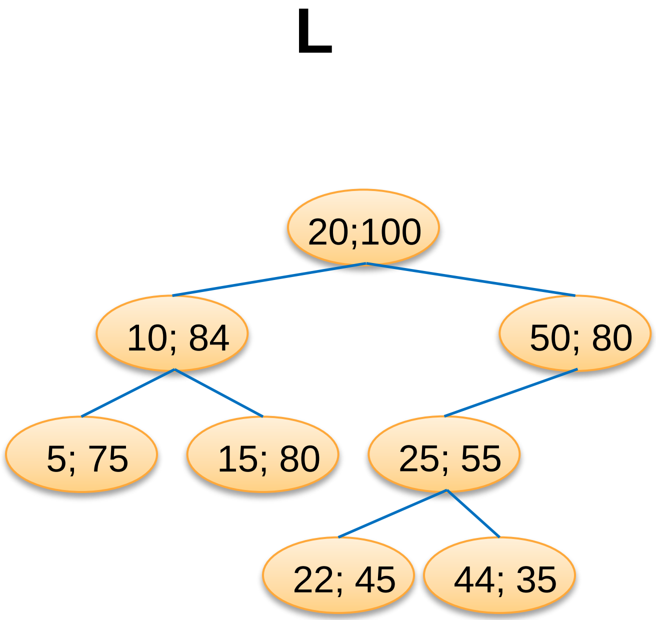


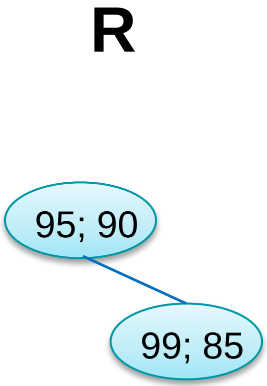
Операція Split. Приклад 2 ключ x0 = 70



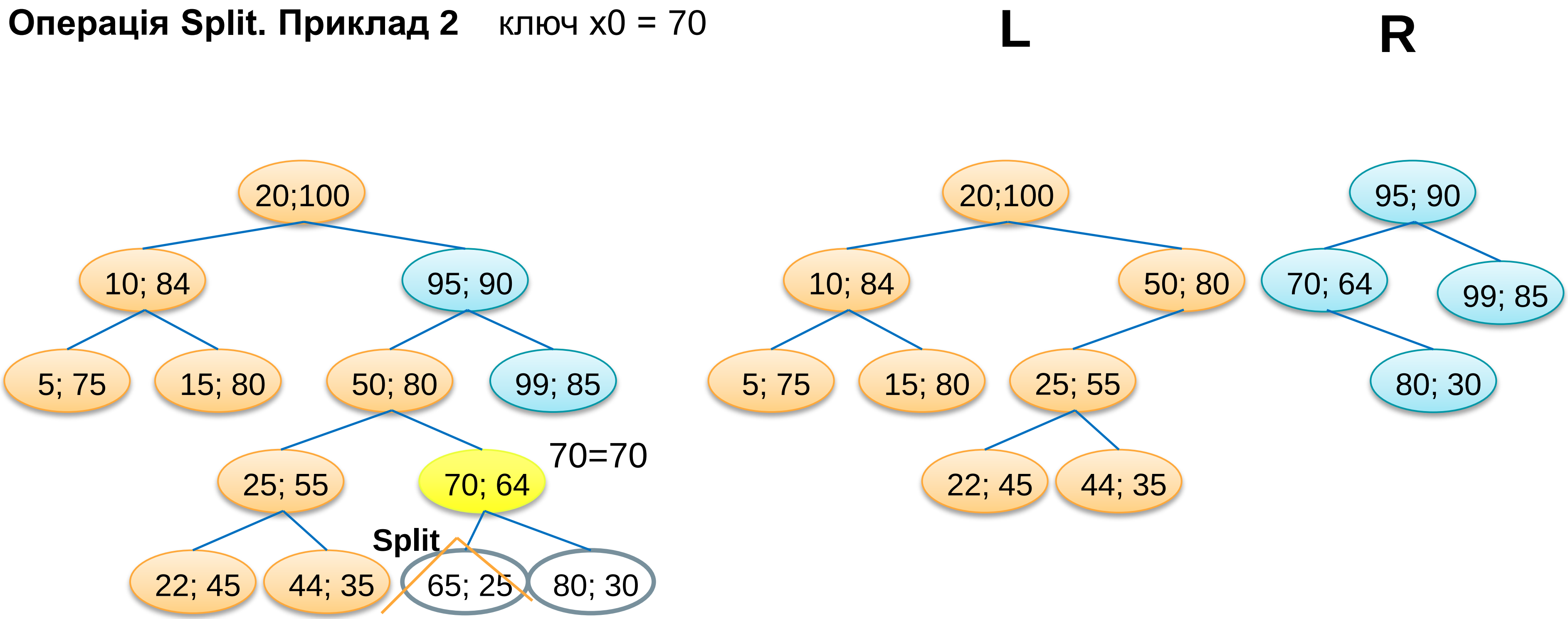
Операція Split. Приклад 2 ключ x0 = 70



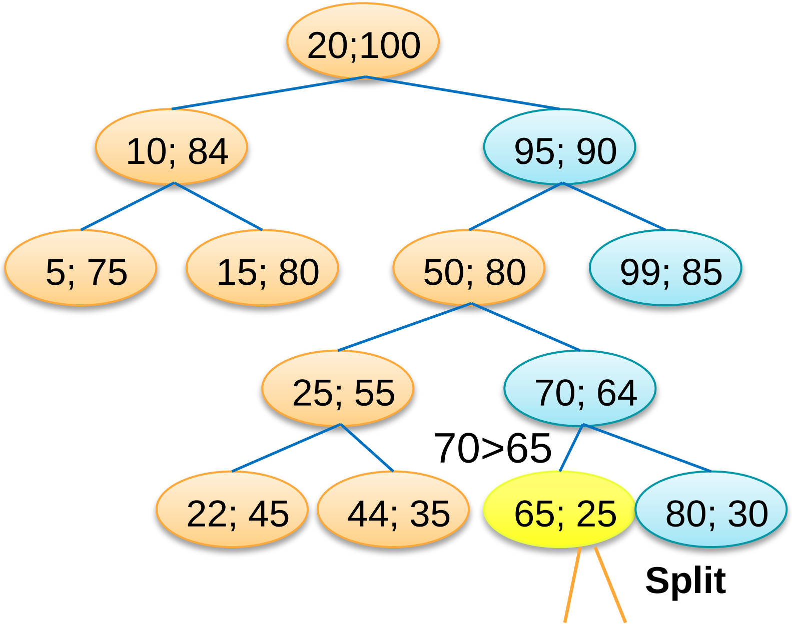


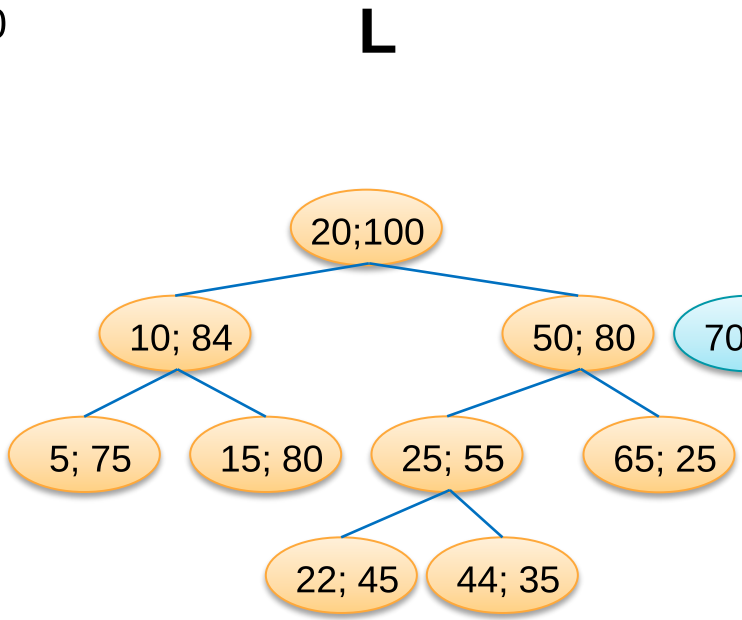


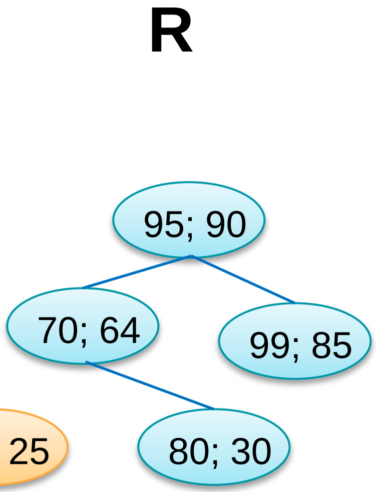
Операція Split. Приклад 2 ключ x0 = 70



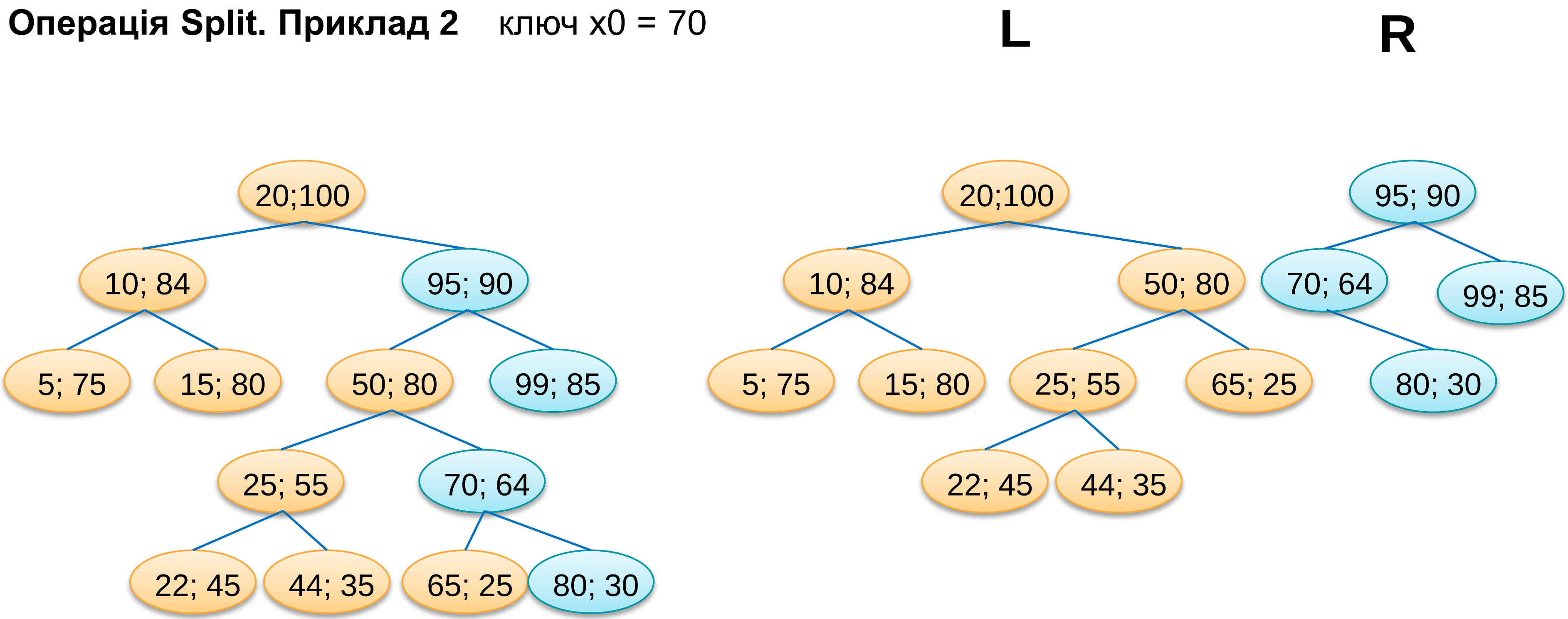
Операція Split. Приклад 2 ключ x0 = 70



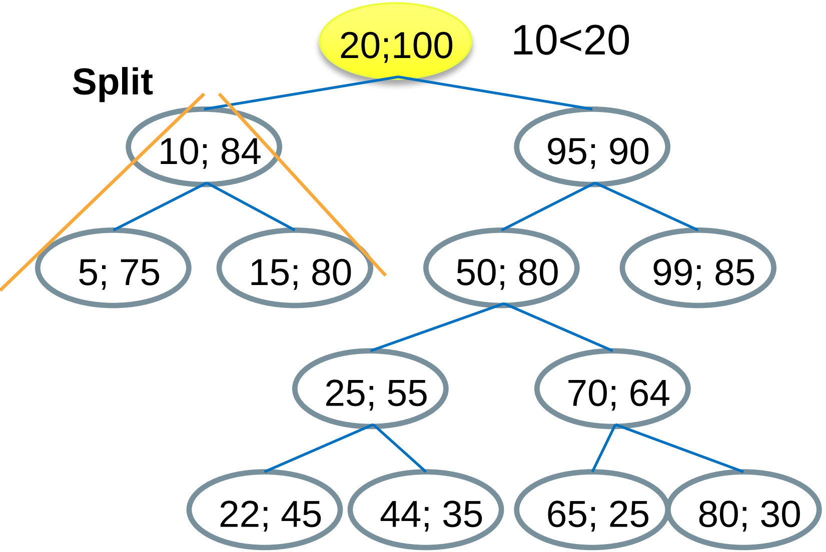




Операція Split. Приклад 2 ключ x0 = 70

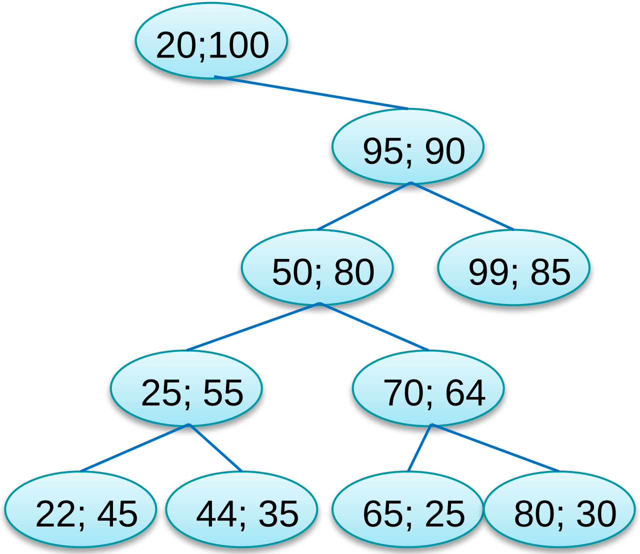


Операція Split. Приклад 3 ключ x0 = 10

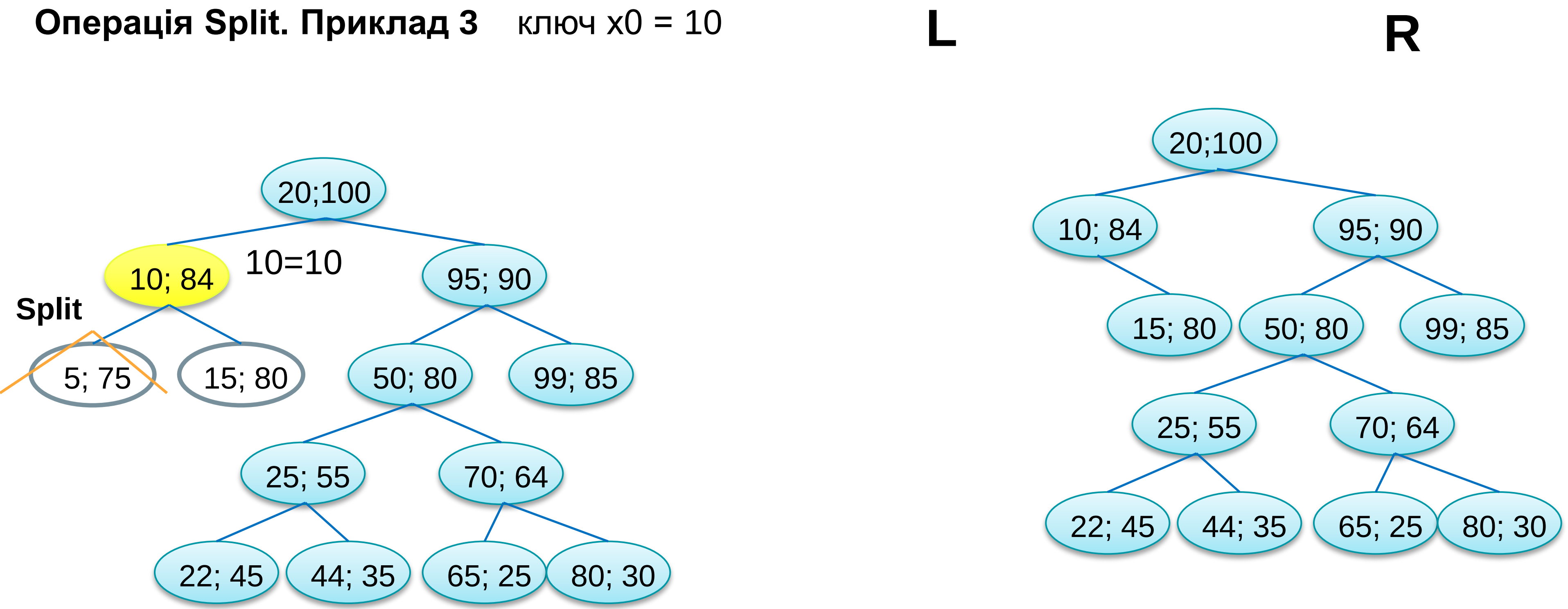


L

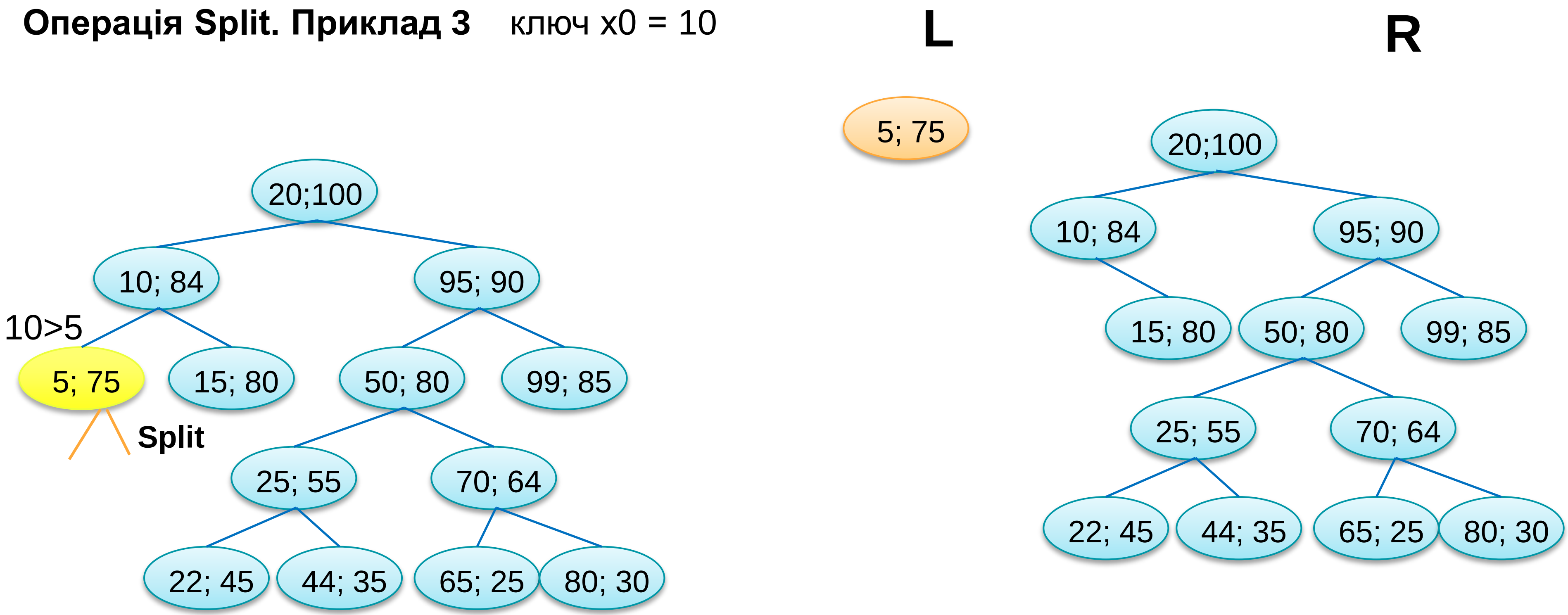
R



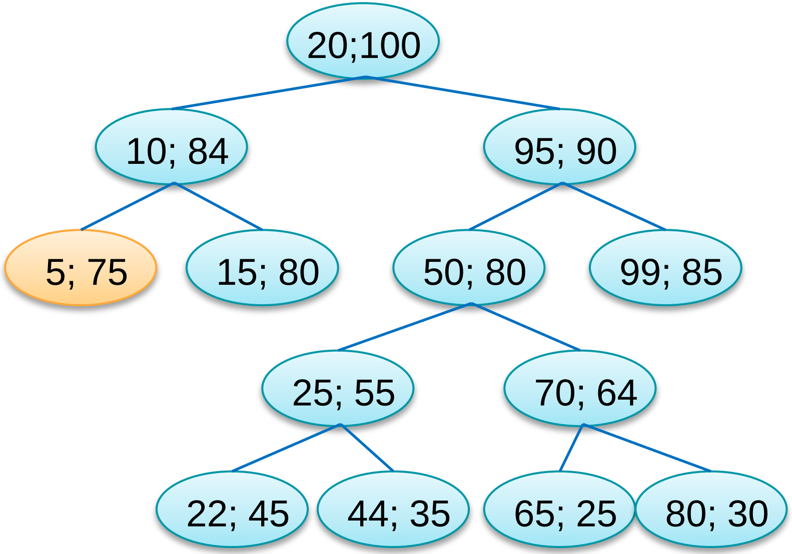
Операція Split. Приклад 3 ключ x0 = 10



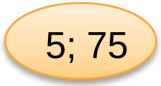
Операція Split. Приклад 3 ключ x0 = 10



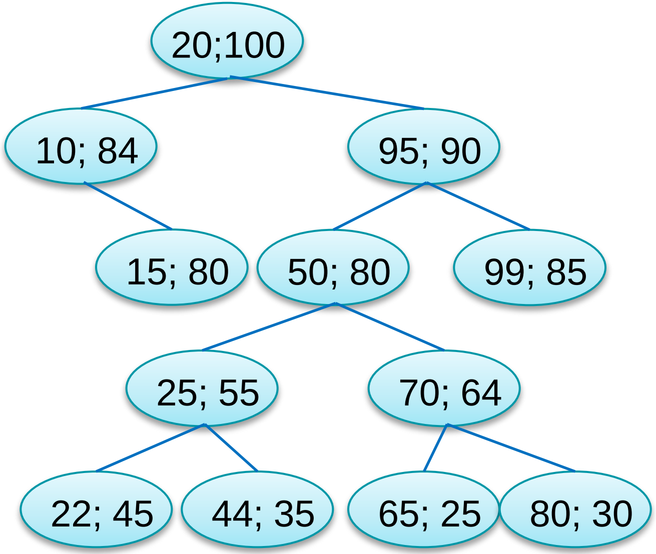
Операція Split. Приклад 3 ключ x0 = 10



L



R

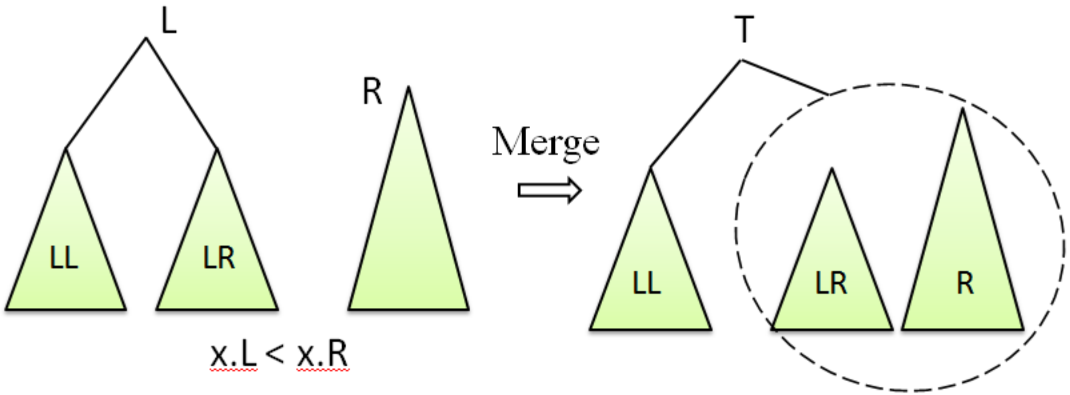


Допоміжні операції. Merge.

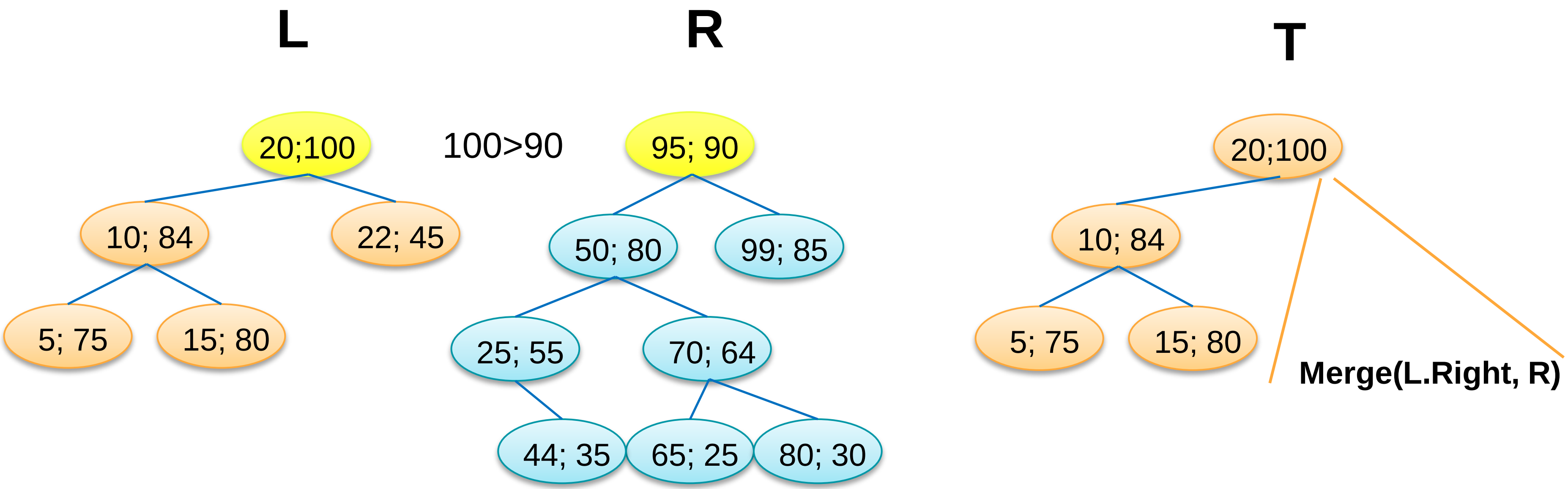
Merge. Параметрами є два декартові дерева L і R. Причому всі ключі дерева L не перевищують ключів дерева R.

Суть операції: об’єднати вхідні дерева L і R в одне декартове дерево T.

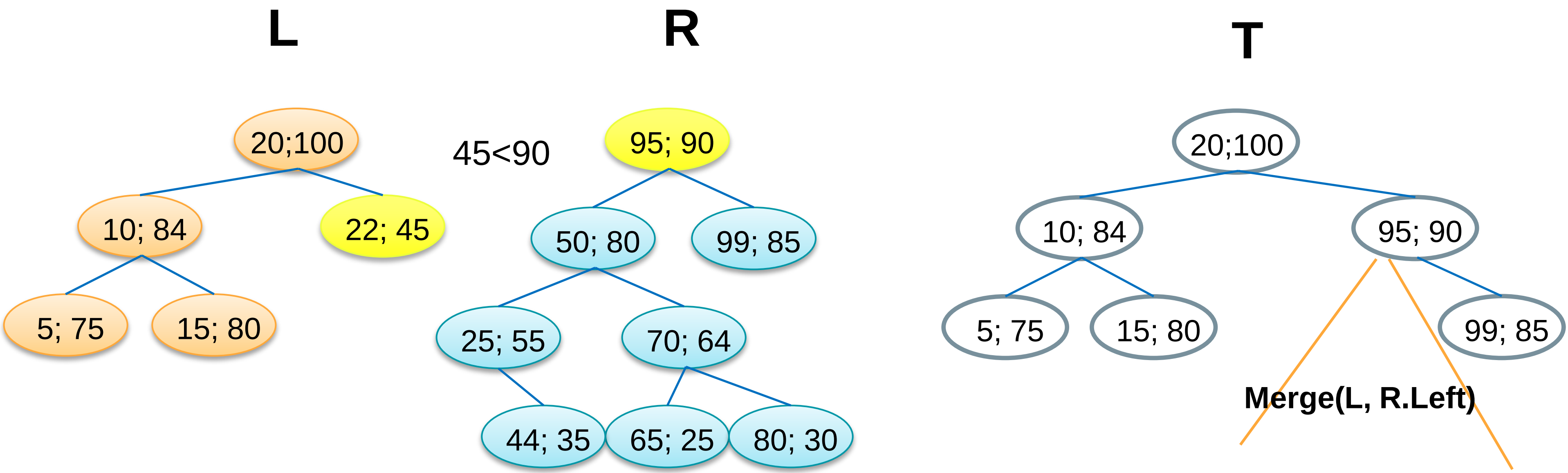
```
Merge(L: Treap, R: Treap):
  if R == ∅
    return L
  if L == ∅
    return R
  else if L.y > R.y
    L.right = merge(L.right, R)
    return L
  else
    R.left = merge(L, R.left)
    return R
```



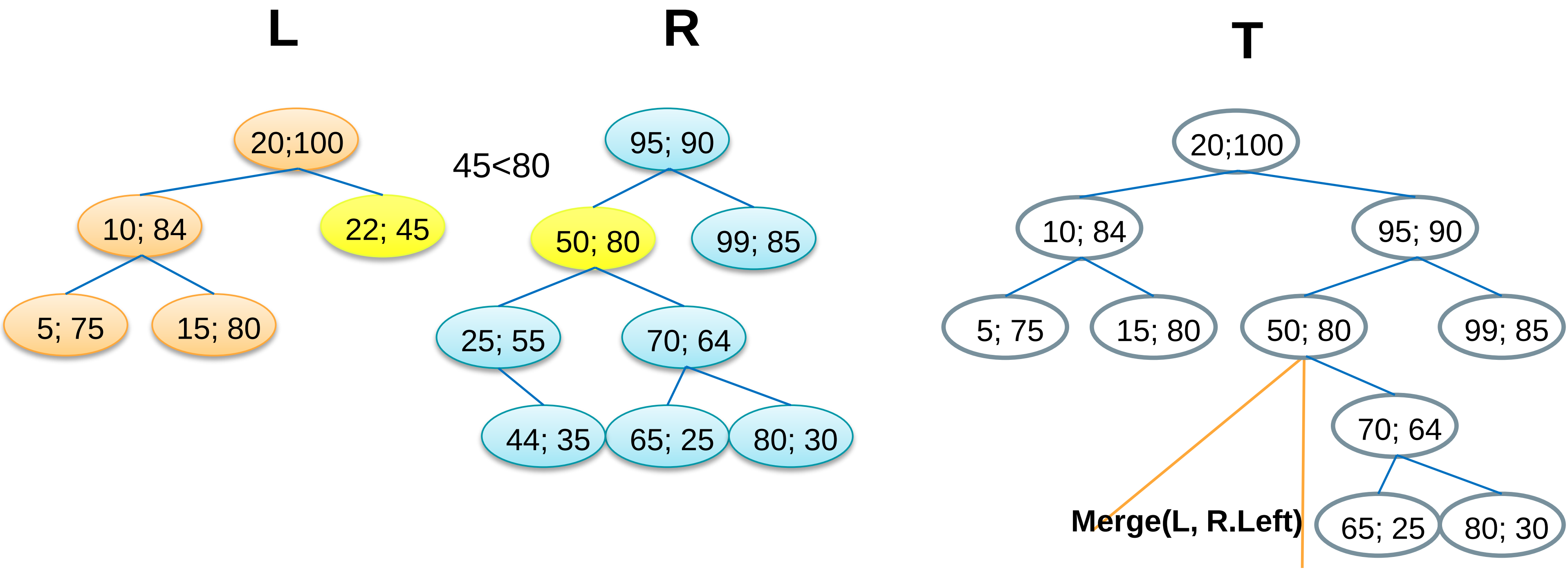
Операція Merge. Приклад 1



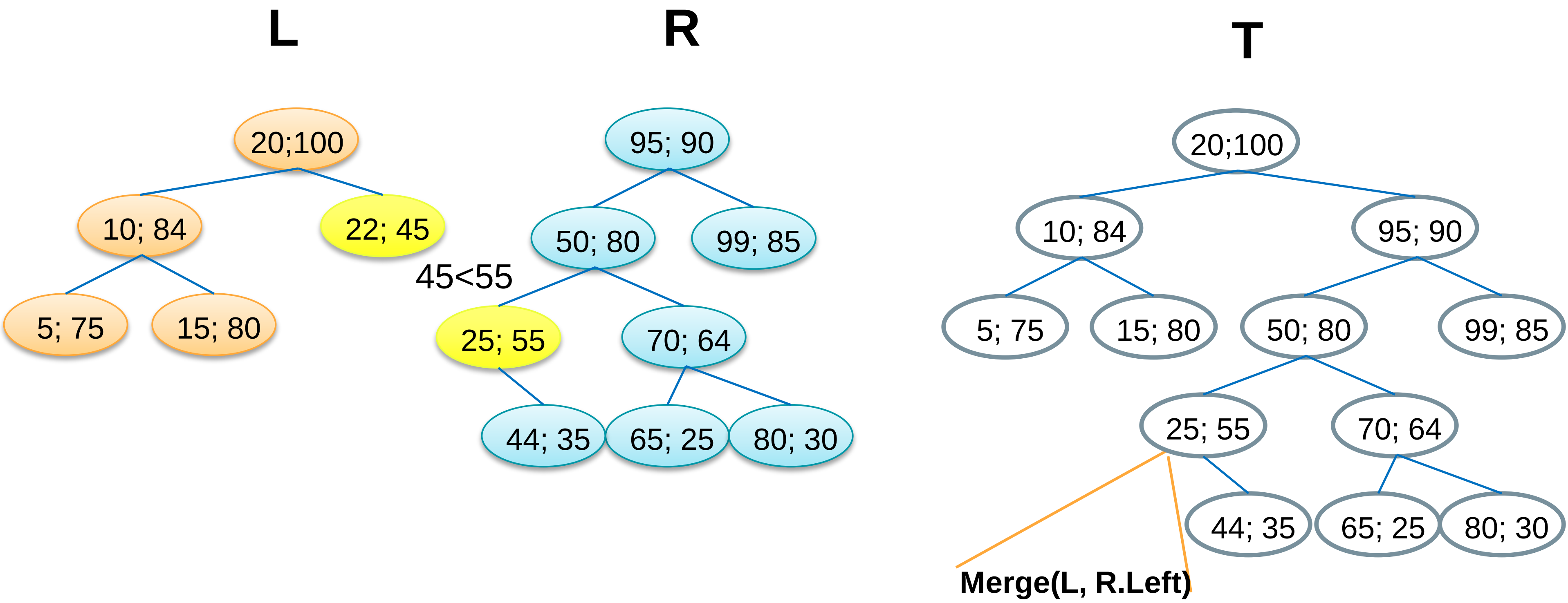
Операція Merge. Приклад 1



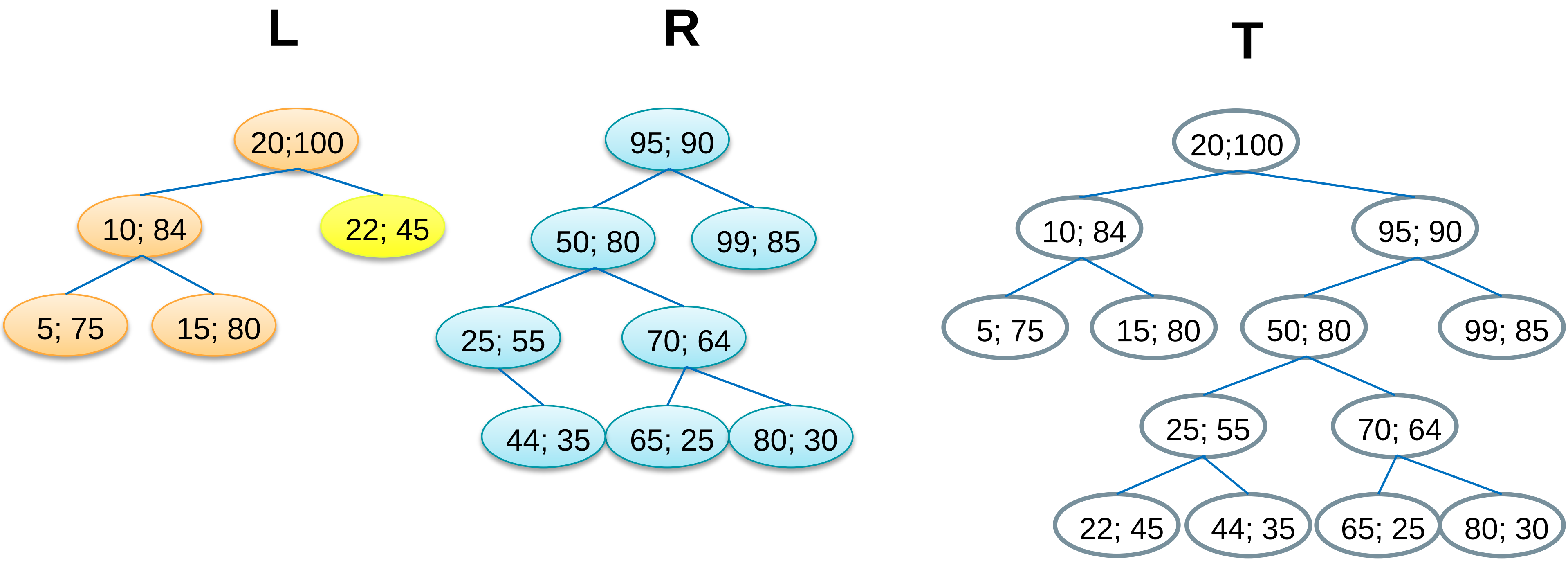
Операція Merge. Приклад 1



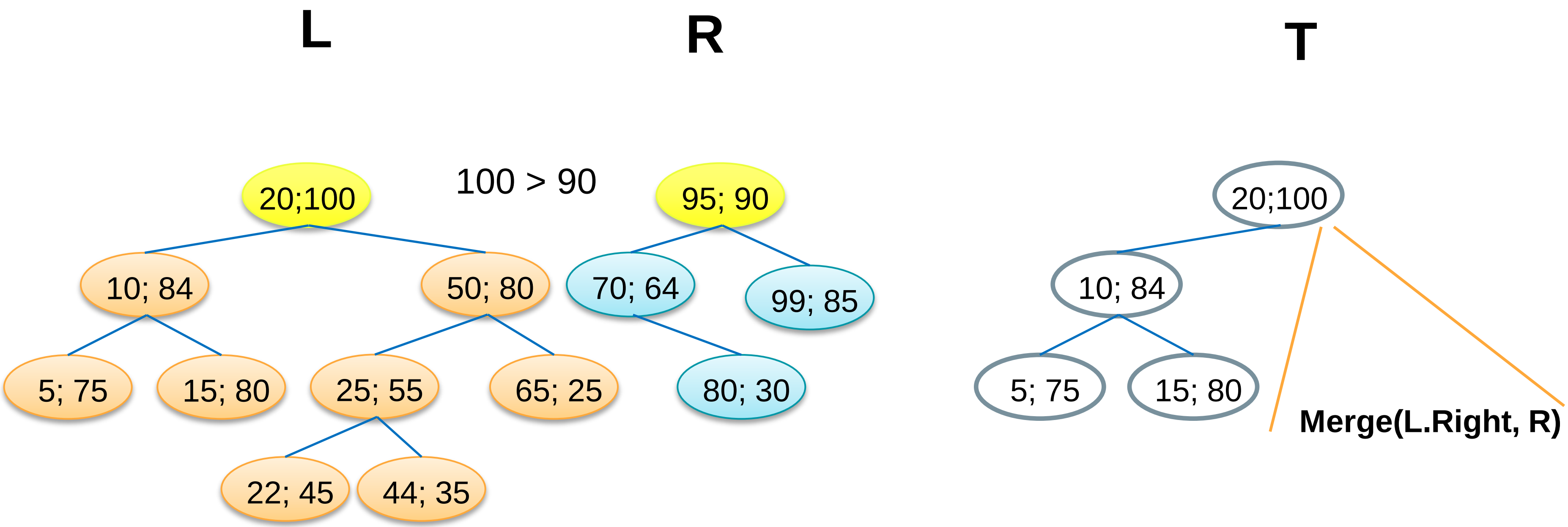
Операція Merge. Приклад 1



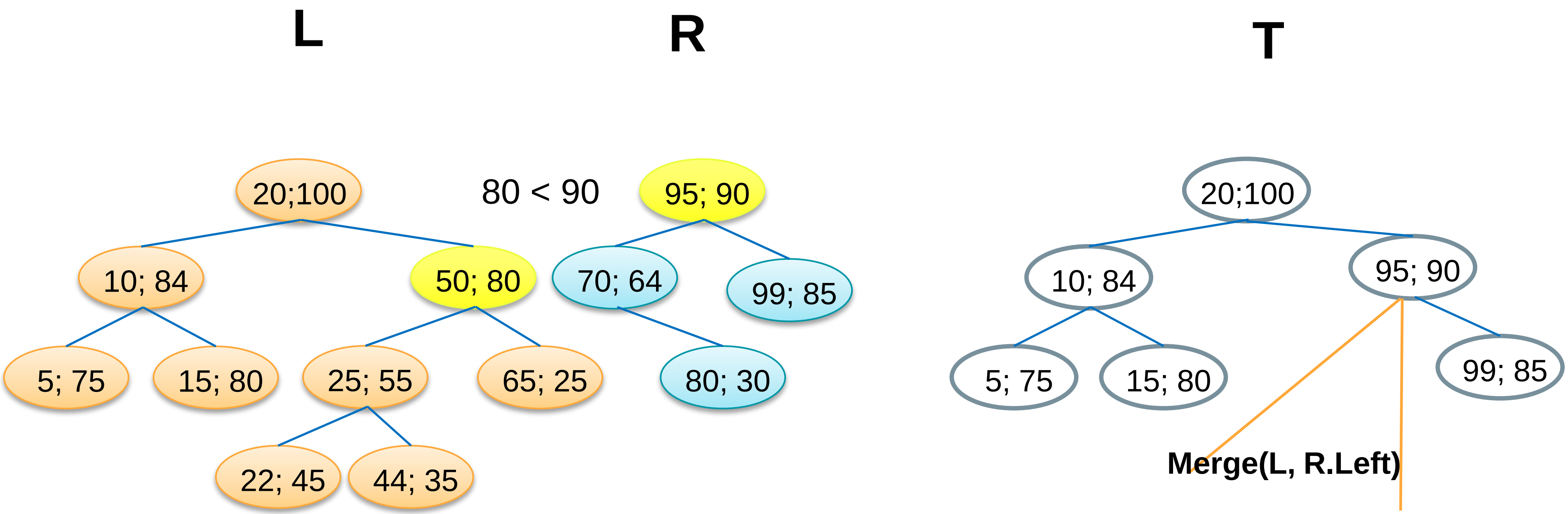
Операція Merge. Приклад 1



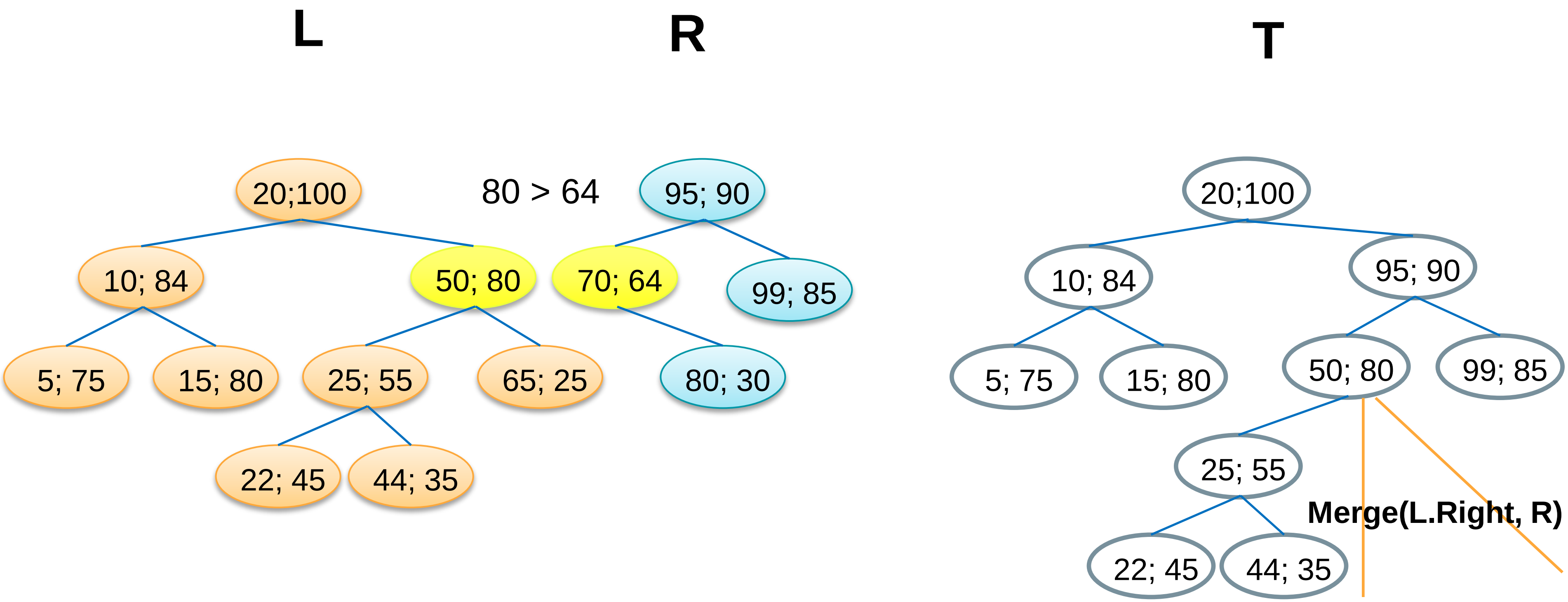
Операція Merge. Приклад 2



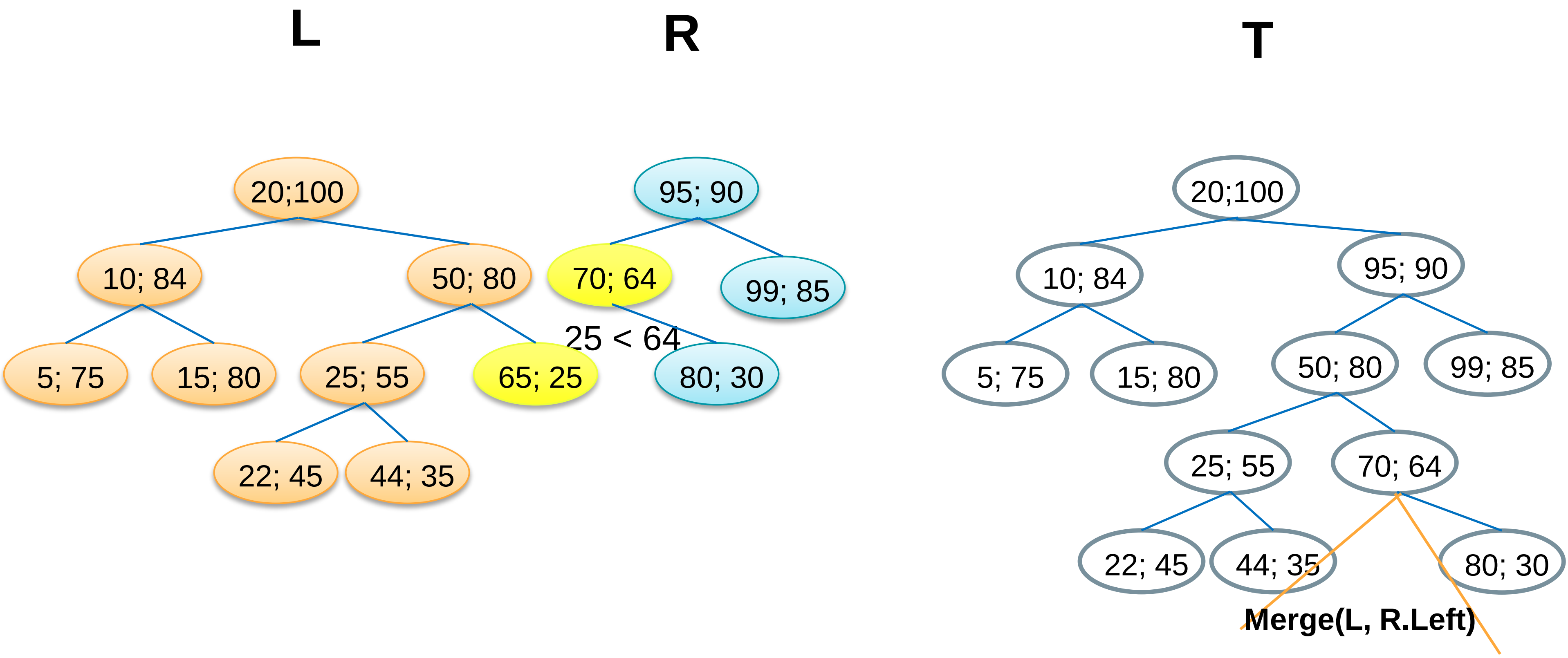
Операція Merge. Приклад 2



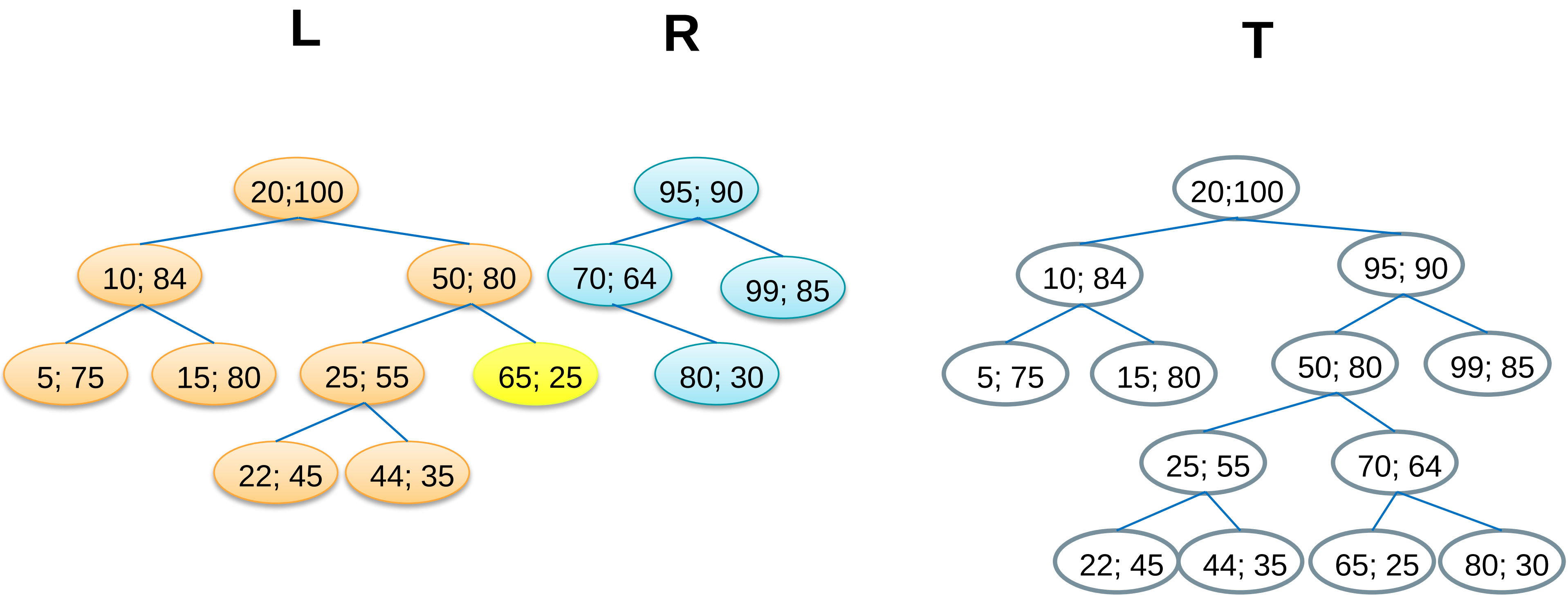
Операція Merge. Приклад 2



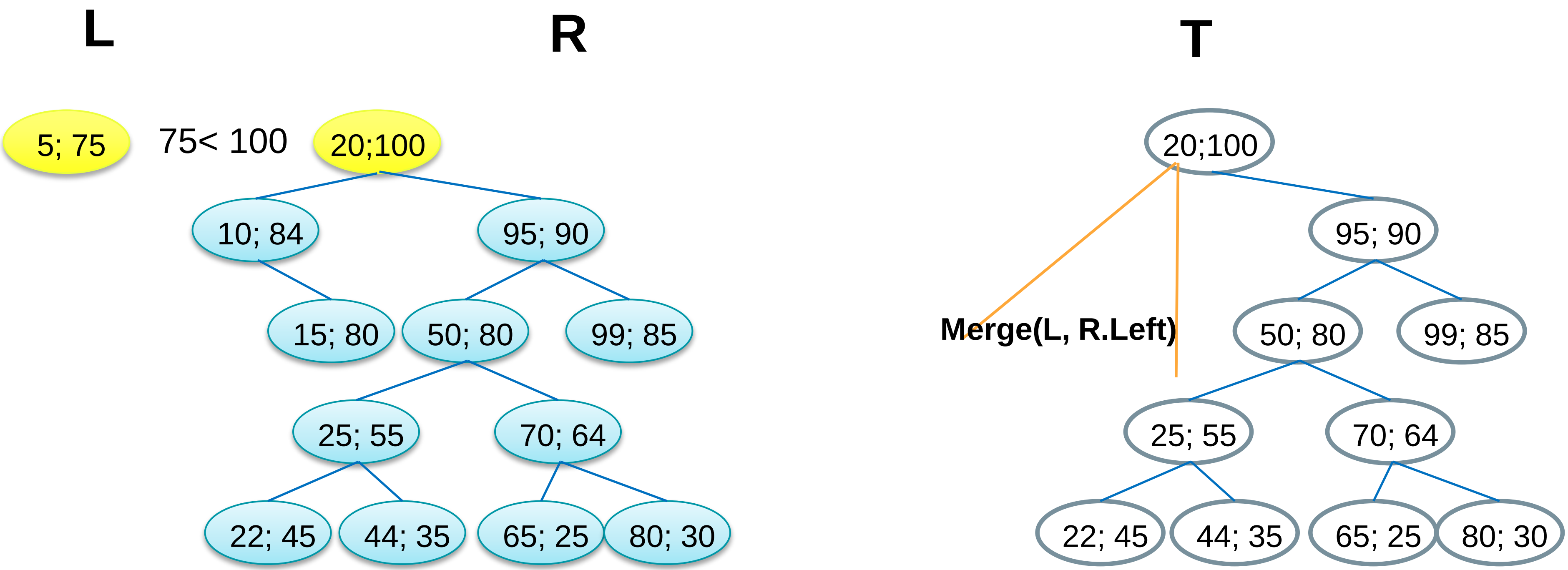
Операція Merge. Приклад 2



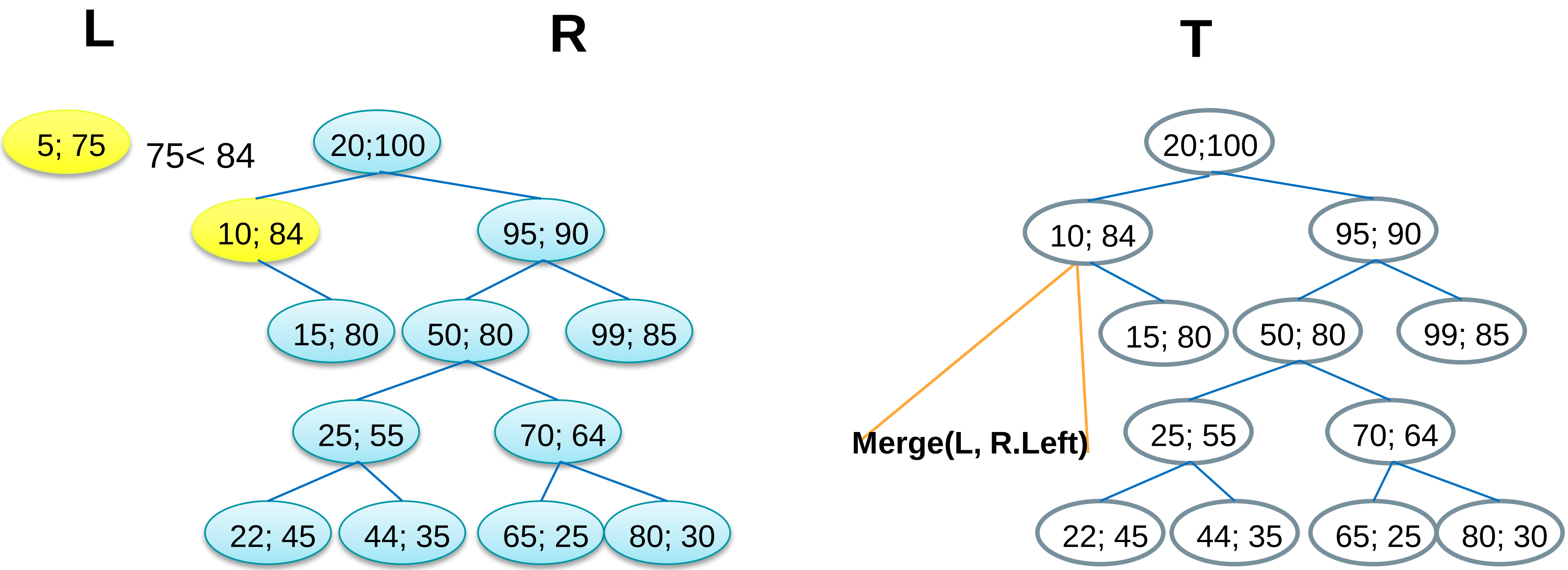
Операція Merge. Приклад 2



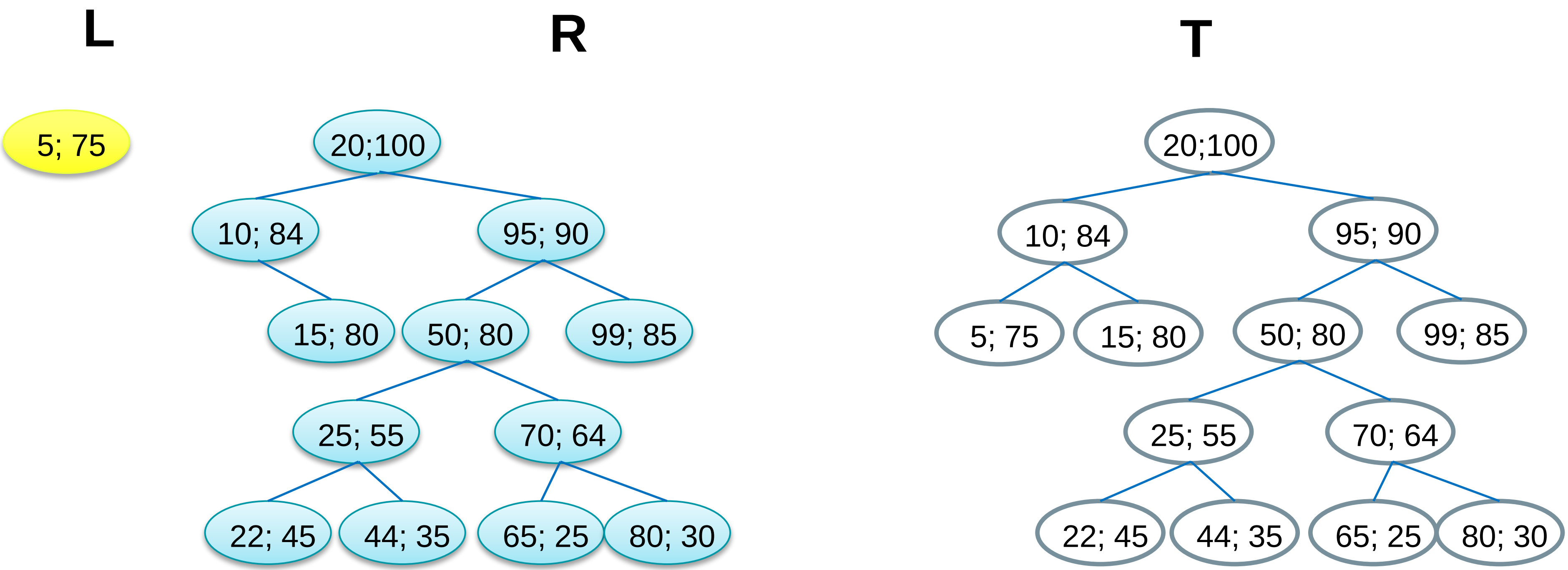
Операція Merge. Приклад 3



Операція Merge. Приклад 3



Операція Merge. Приклад 3



Додавання елемента до декартового дерева

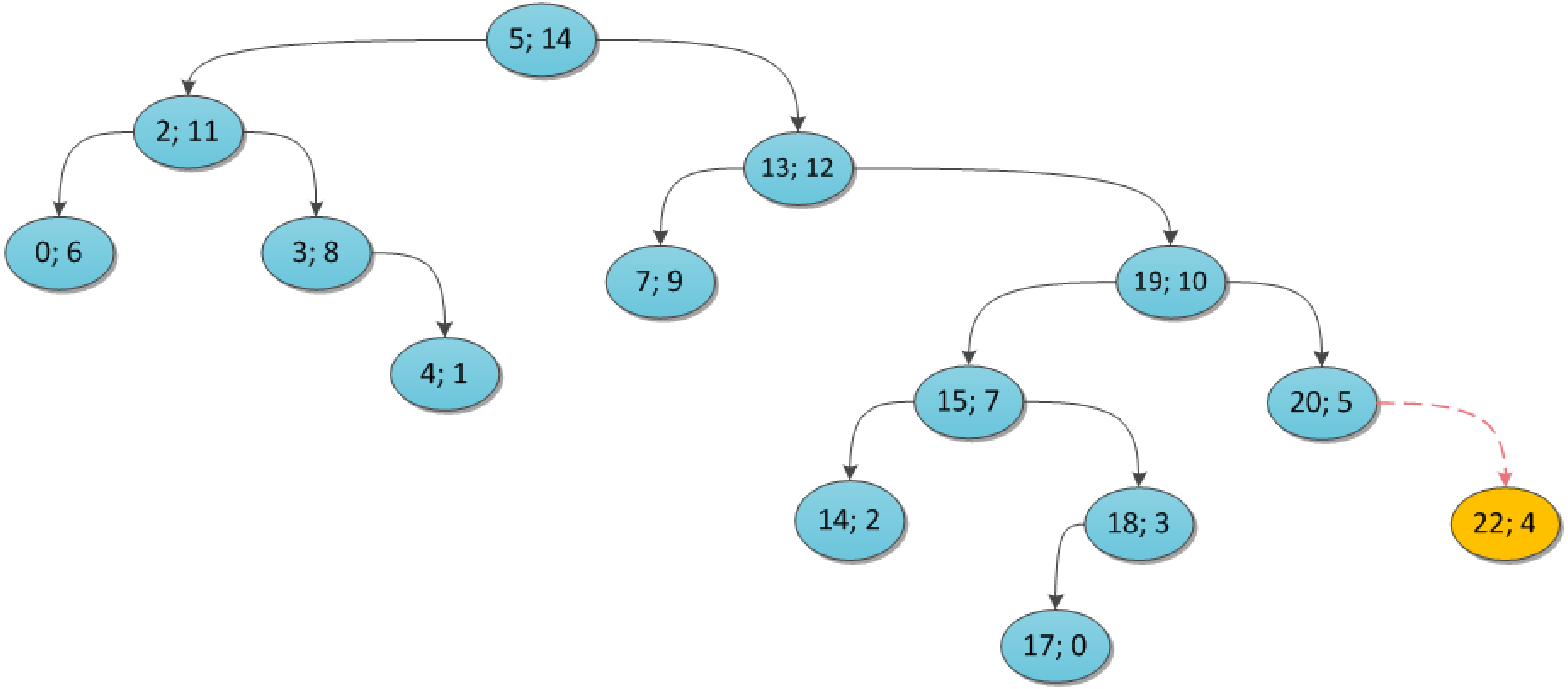
Варіант 1:

1) розіб'ємо дерево T за ключем k.x, тобто split (T, k.x) -> (L, R);
2) об'єднаємо перше дерево із новим елементом, тобто Merge (L, k) -> L;
3) об'єднаємо одержане дерево з іншим, тобто Merge (L, R) -> T.

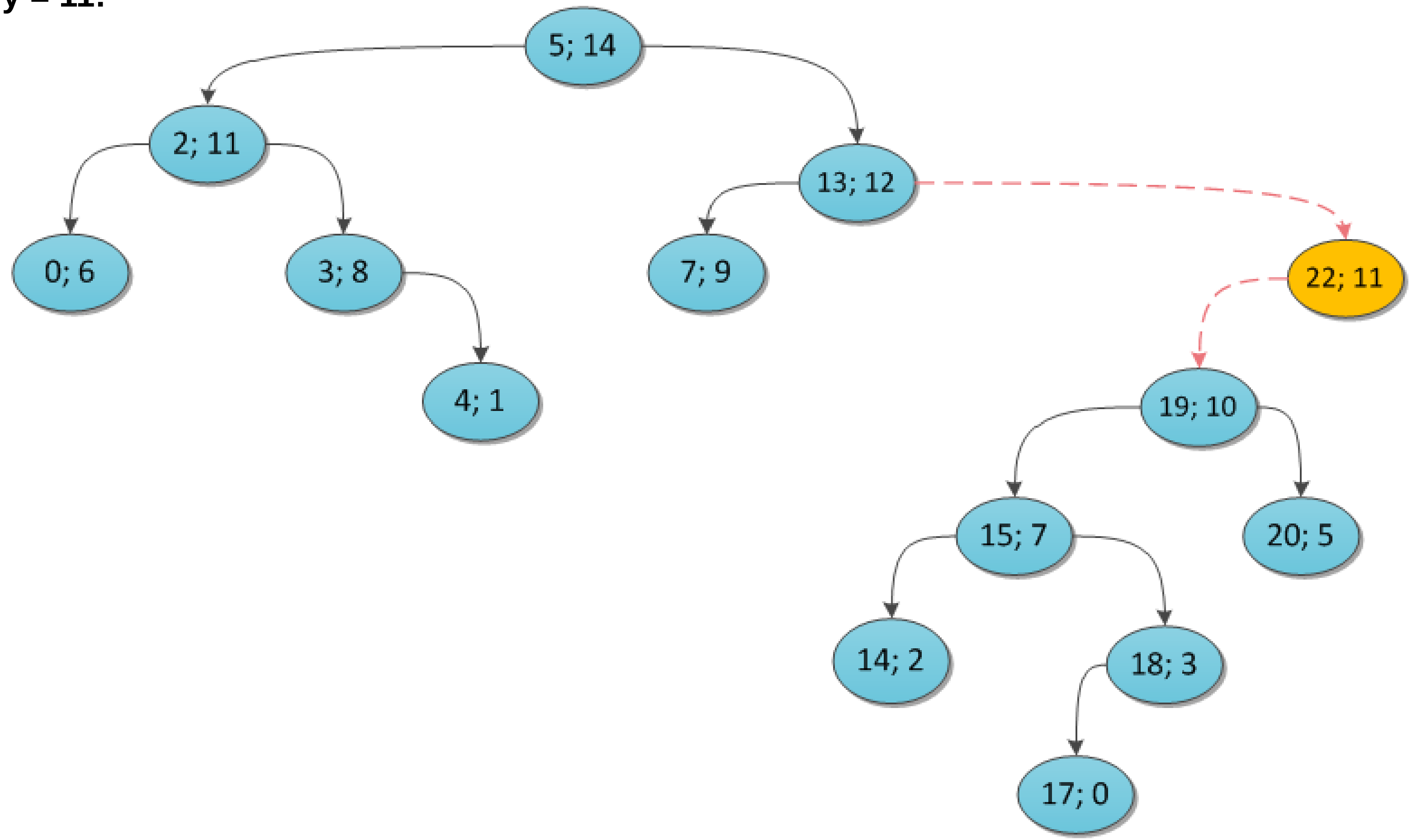
Варіант 2:

1) спустимося по дереву як у звичайному бінарному дереві пошуку за k.x, зупинившись на першому елементі T', у якому значення пріоритету виявилося менше k.y;
2) викличемо split (T', k.x) -> (L, R) від знайденого елемента (від елемента разом із його піддеревом);
3) запишемо одержані L і R як лівий і правий нащадки елемента, що додають;
4) поставимо одержане дерево на місце елемента, знайденого в 1му пункті.

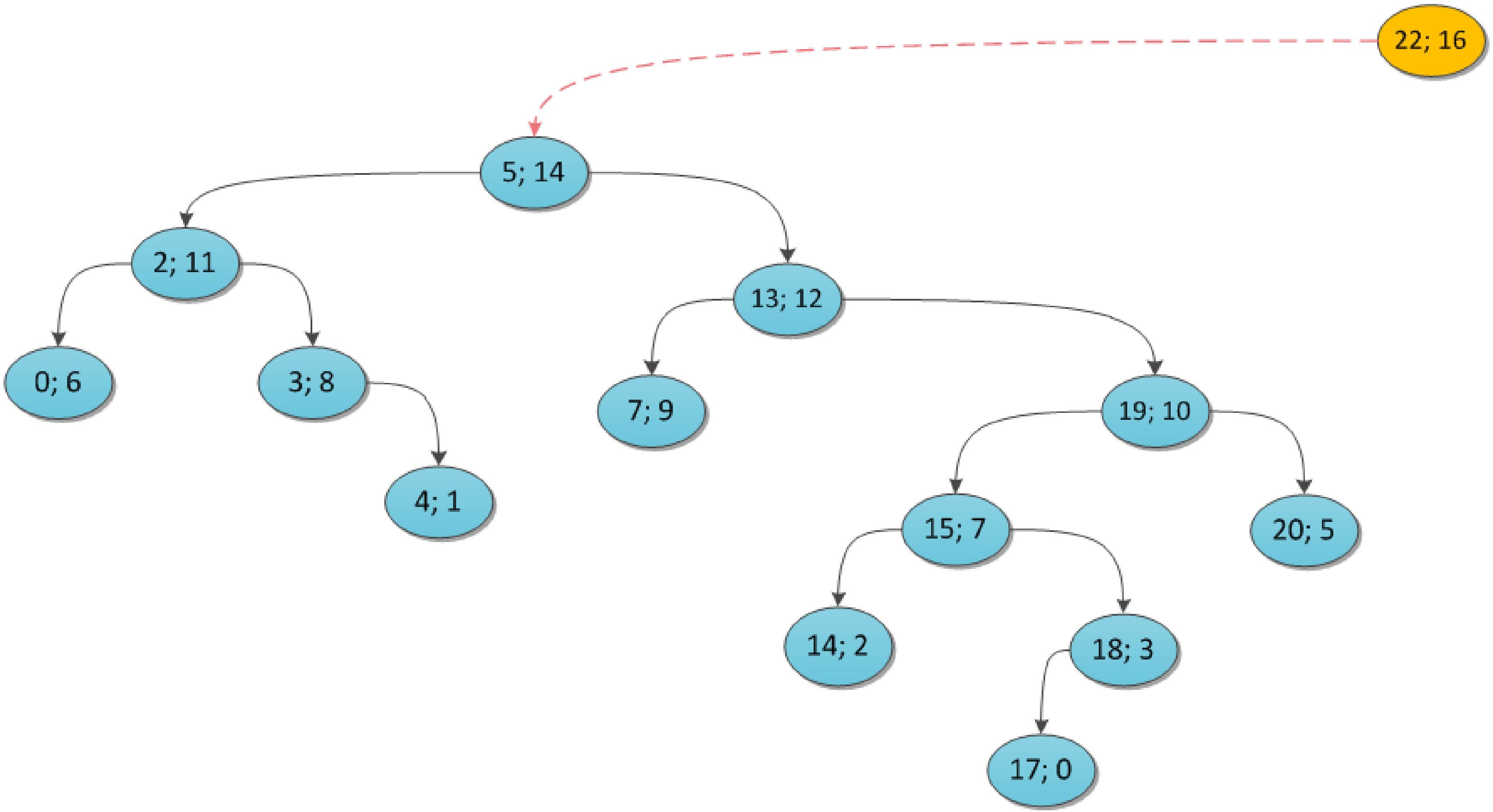
x=22; y = 4:



x=22; y = 11:



x=22; y = 16:



Видалення елемента з декартового дерева

Варіант 1:
1)спустимося по дереву (як у звичайному бінарному дереві пошуку за ключем k.x) і розшукаємо елемент, який необхідно видалити;
2)знайшовши елемент, викличемо Merge для його лівого і правого нащадків;
3)поставимо результат процедури Merge на місце видаленого елемента.

Варіант 2:
1)розіб'ємо дерево T за ключем k.x, тобто split (T, k.x) -> (L, R);
2)видалимо k як найменший за ключем елемент в R;
3)Merge(L, R) ->T.