Splay Tree

Introduction

- Invented by Daniel Sleator and Robert Tarjan in 1985.
- Self-adjusting BST that combines all normal BST operations with one basic operation, i.e. splaying.
  - Placing recently accessed element at the root of the tree.
- Splaying can be done using either of the two algorithms.
  - Top-down or Bottom-up.
- Main property: Recently accessed elements are quick to access again.
- Performs basic operations in $\mathbf{O(\log n)}$ amortized time.
- Particularly useful for implementing caches and garbage collection algorithms.

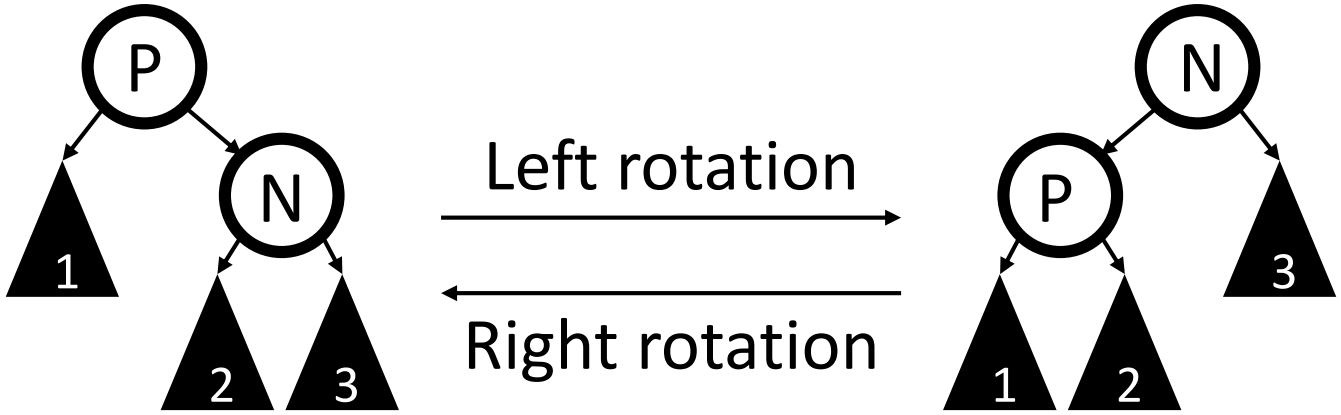
The main idea is....

- Balanced BST
  - Require storage of an extra piece of information per node.
  - Complicated to implement.
  - Have identical worst-case, average-case, and best-case performances.
- Make second access to the same piece of data cheaper than the first.
- The 90-10 rule: Empirical studies suggest that in practice 90% of the accesses are to 10% of the data items.

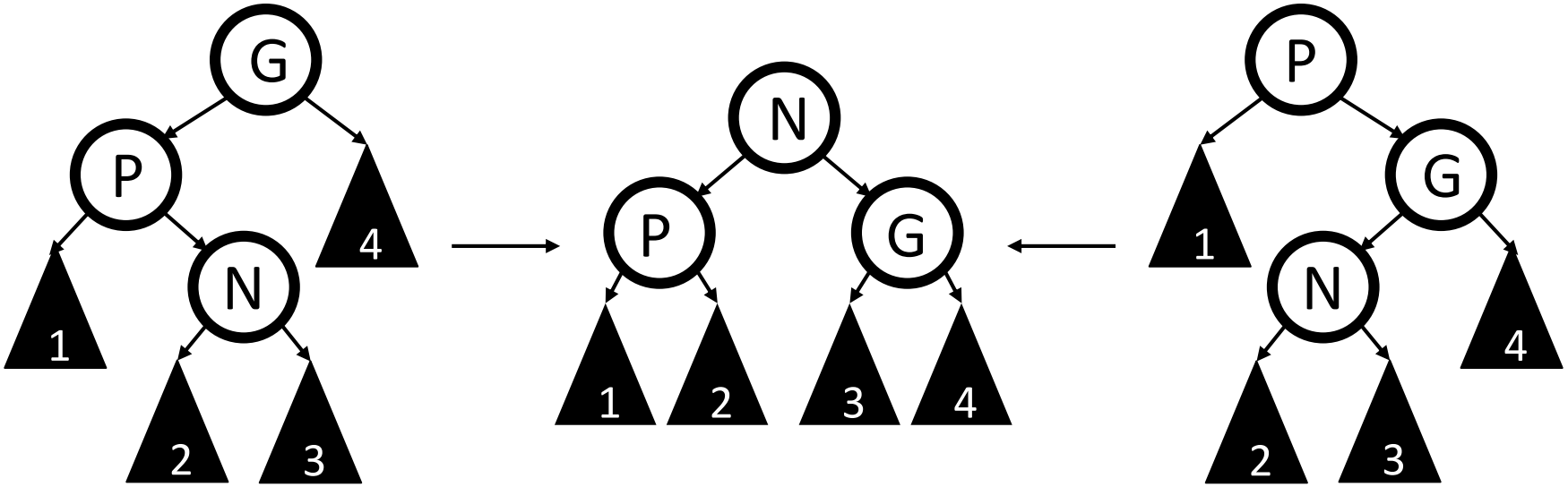
Bottom-up Splaying

Contd…

- Zig (Single rotation)

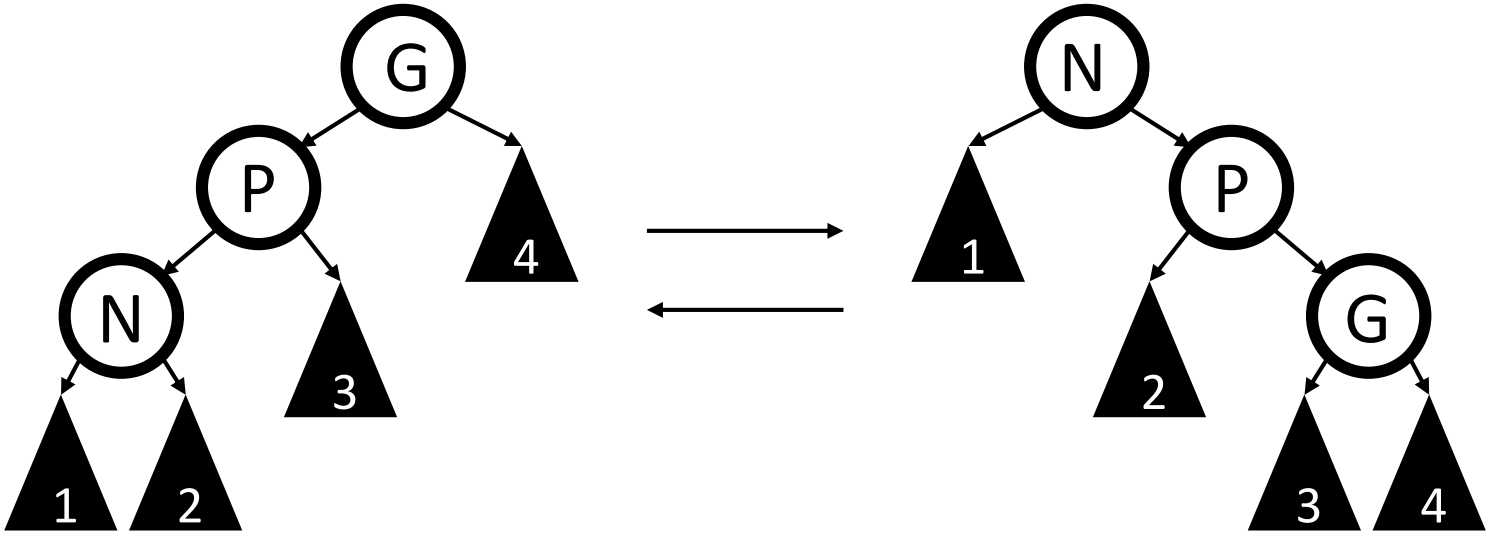


- Zig-Zag (Double rotation)



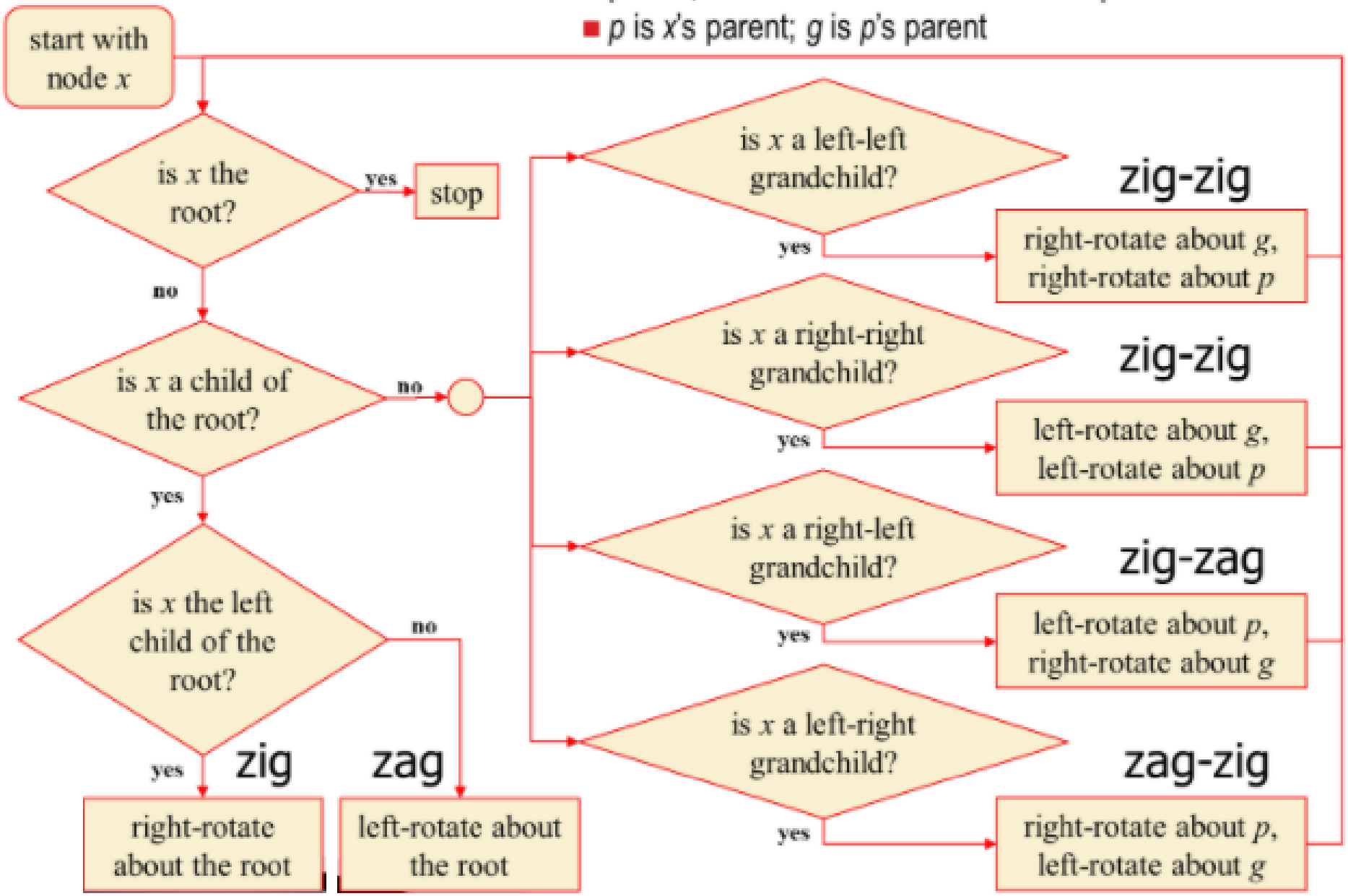
Contd…

- Zig-Zig

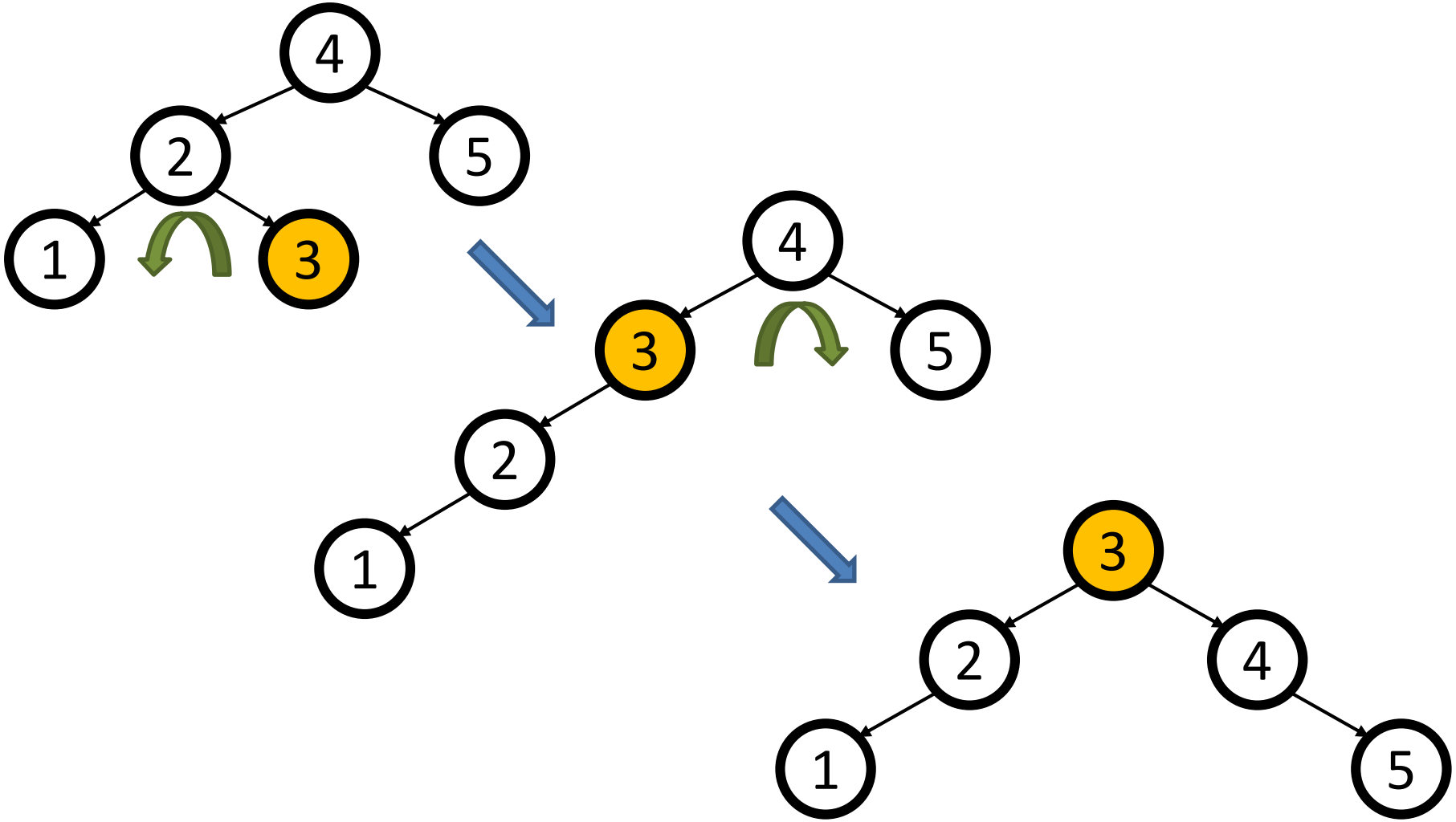


Splaying:

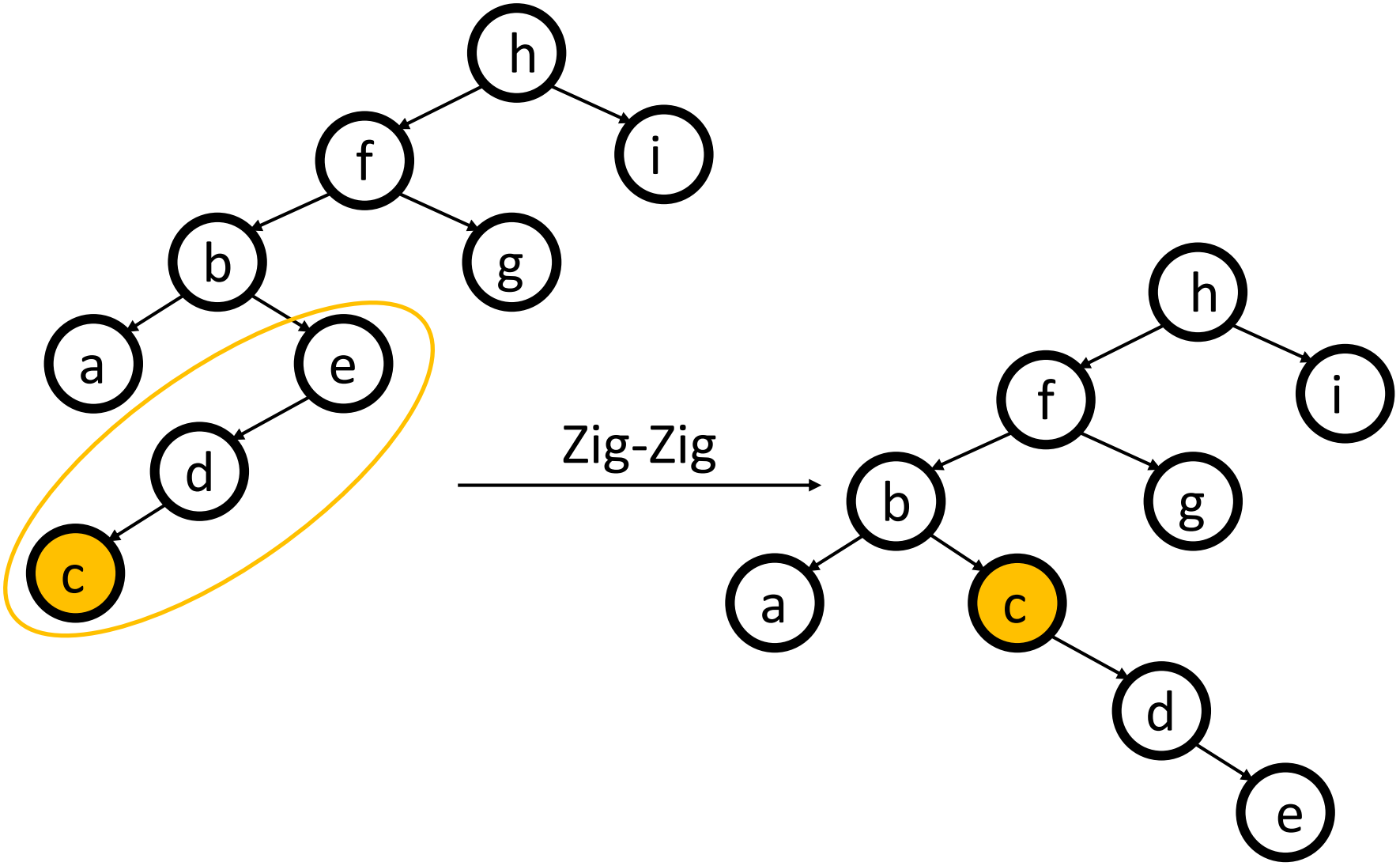
- "*x* is a left-left grandchild" means *x* is a left child of its parent, which is itself a left child of its parent
- *p* is *x*'s parent; *g* is *p*'s parent



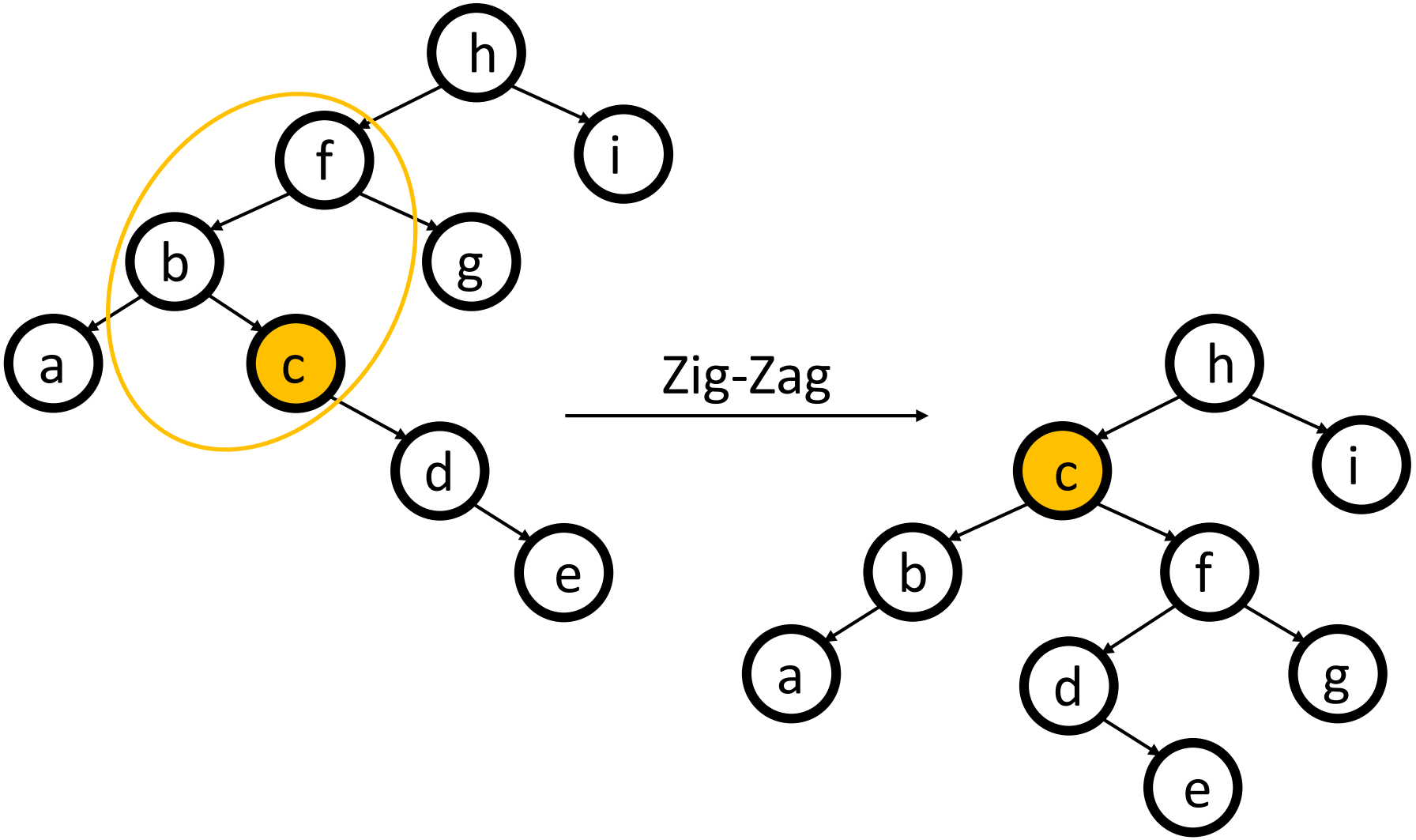
Splaying



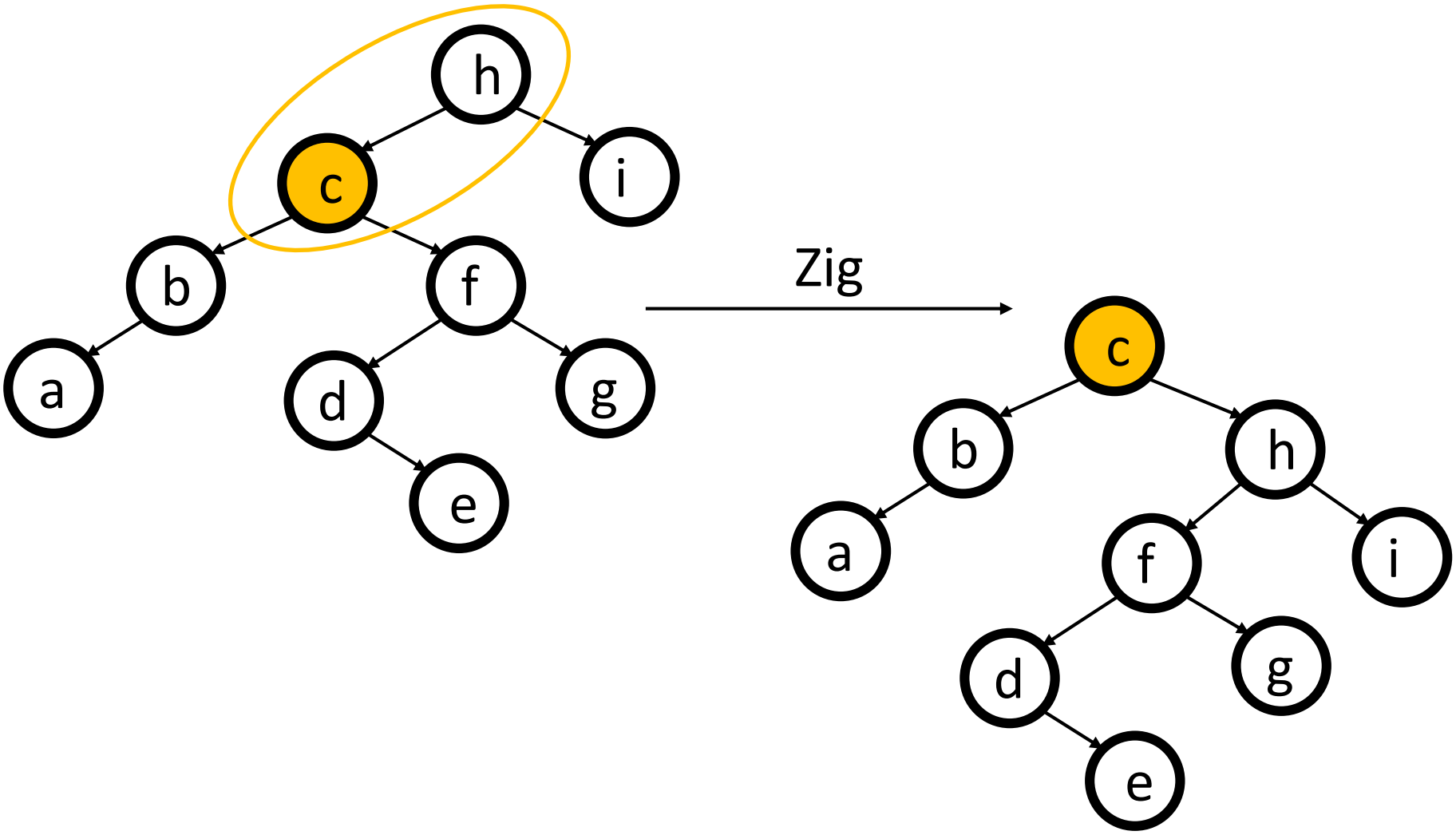
Example – Splay at 'c'



Contd…



Contd…

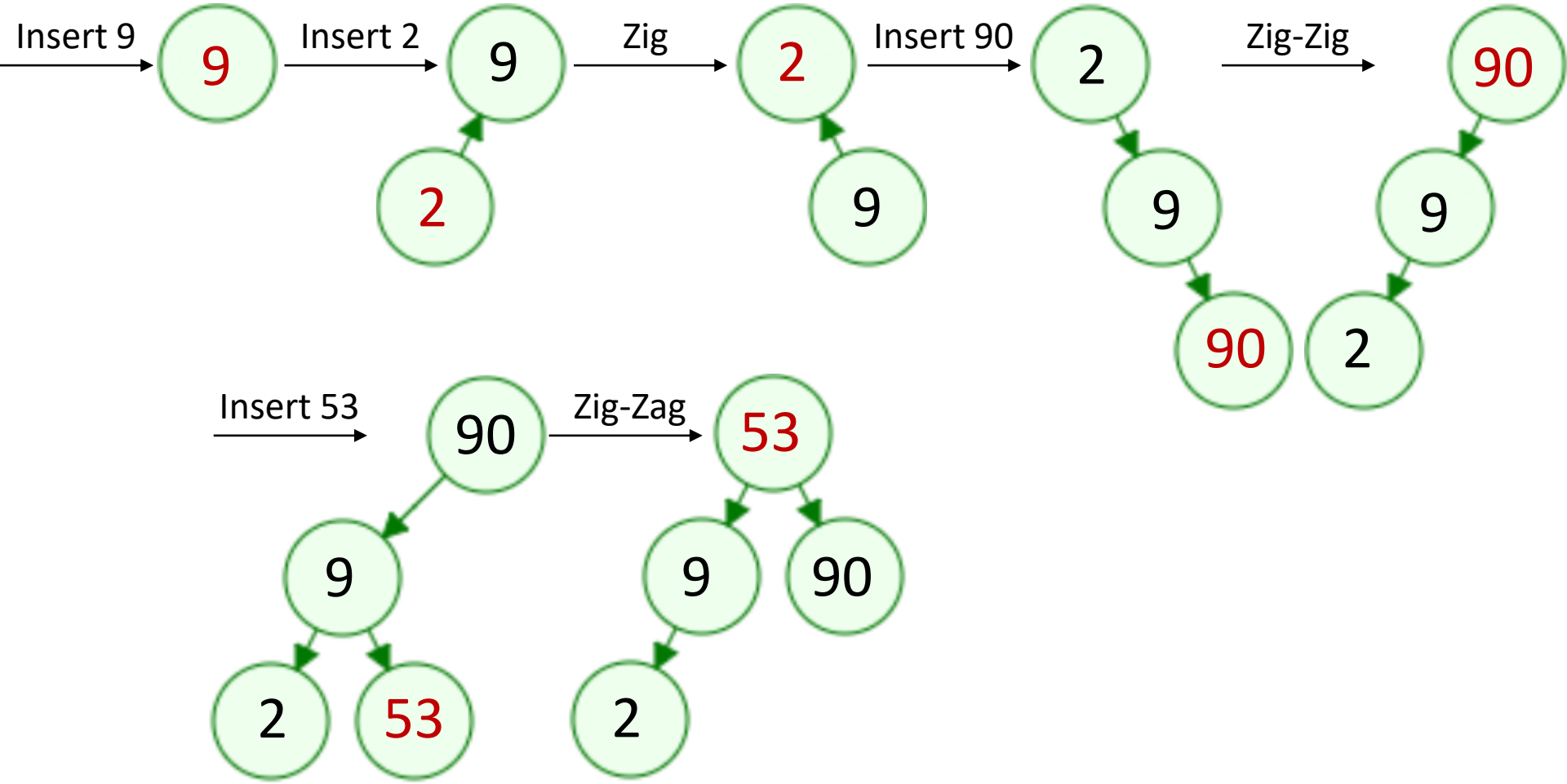


Operations

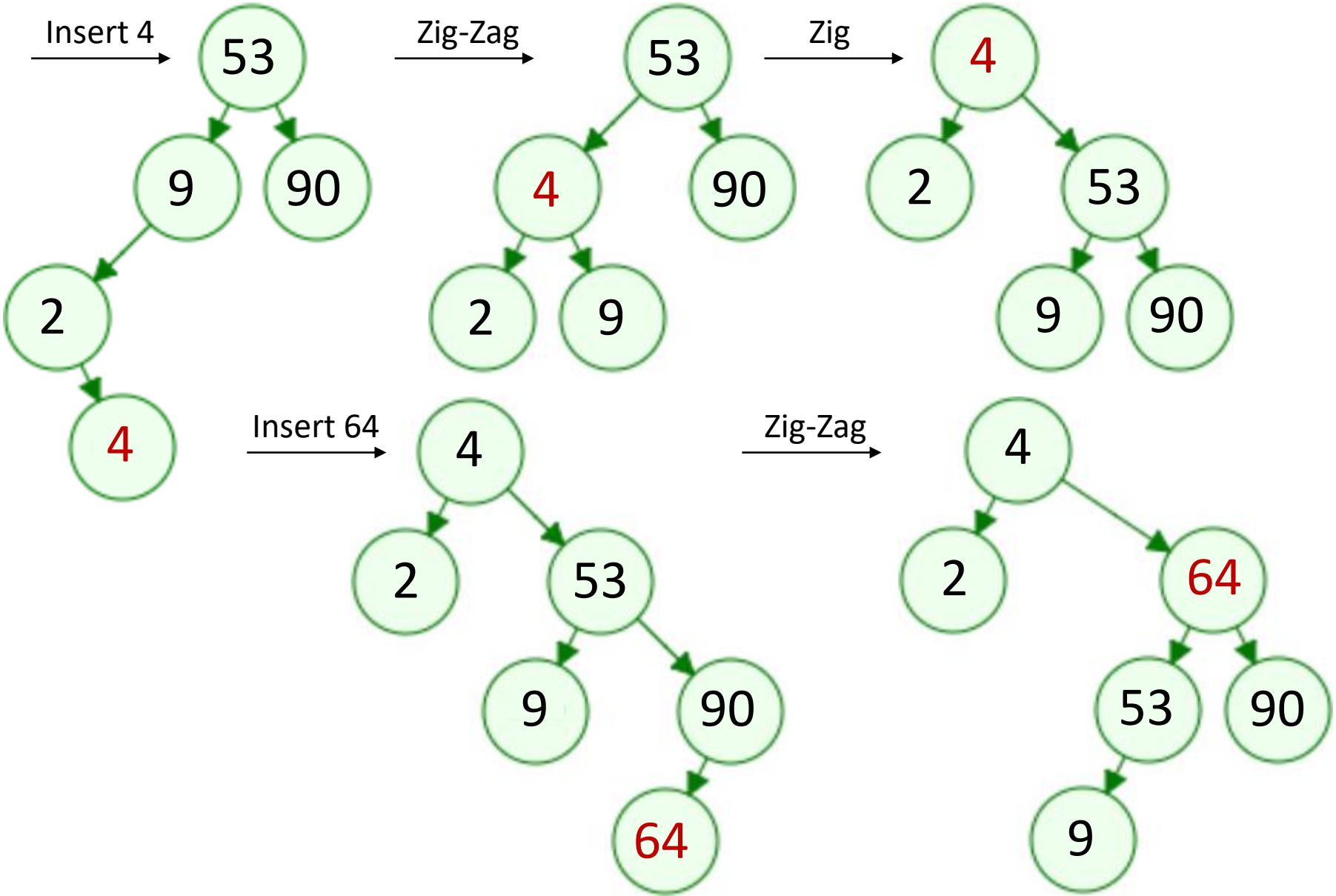
- Insertion
  - Splay at the newly inserted node.
- Searching
  - Successful: Splay at the node being searched.
  - Unsuccessful: Splay at the node accessed just before reaching the NULL pointer.
- FindMin
  - Splay at the minimum node.
- FindMax
  - Splay at the maximum node.

Example – Insertion (Bottom-up)

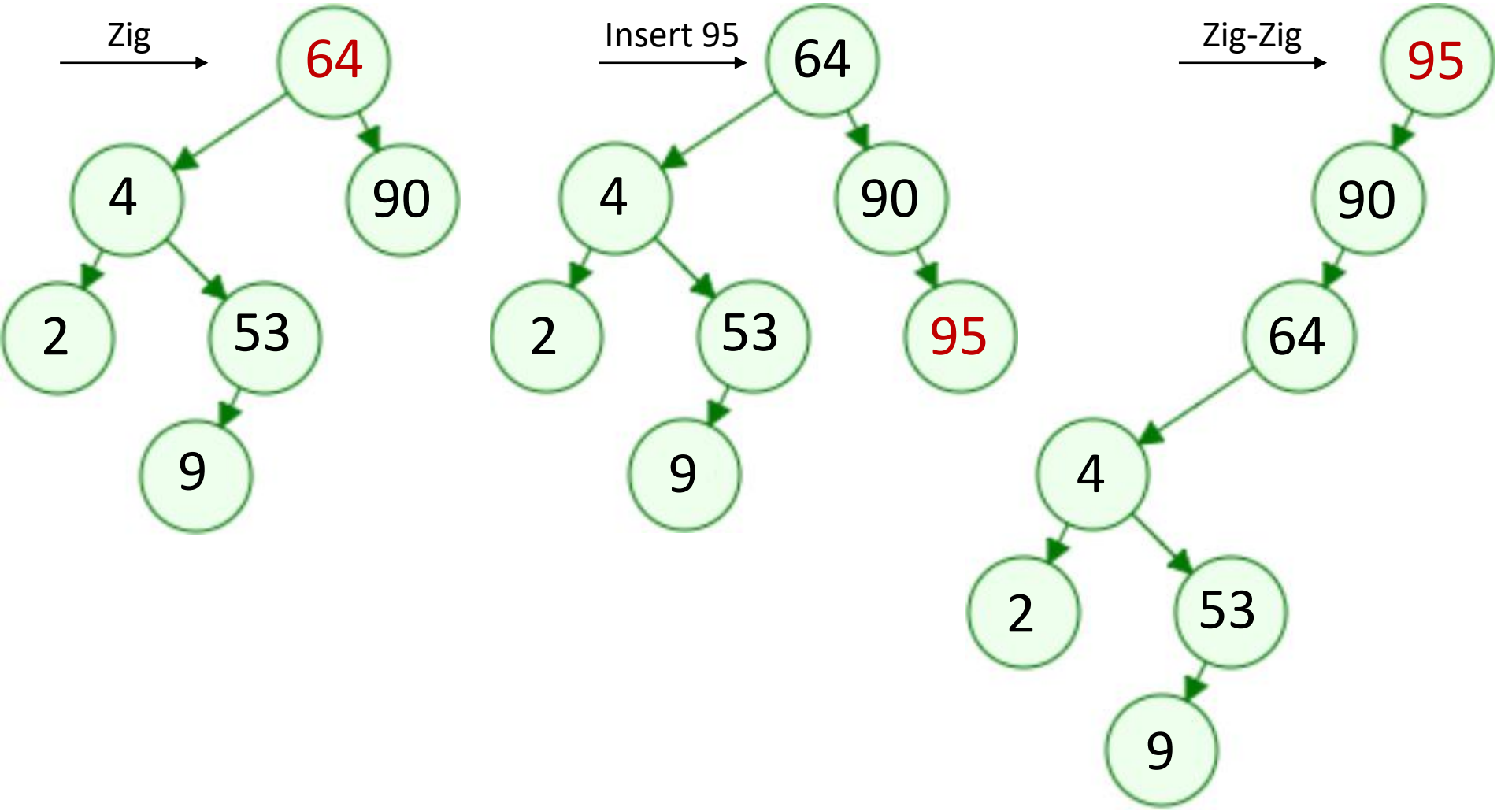
- 9, 2, 90, 53, 4, 64, 95, 59



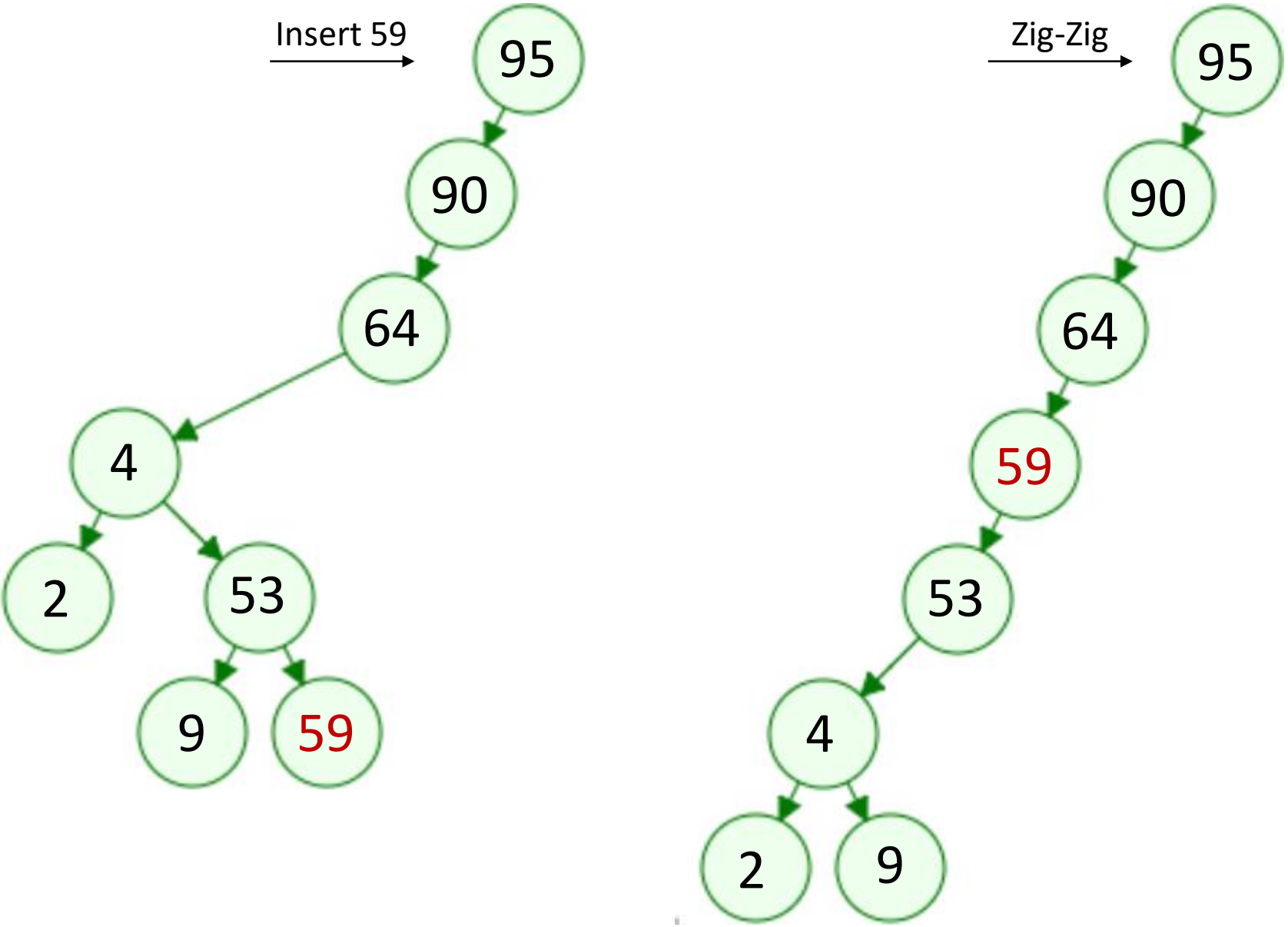
Contd…


Insert 4
53
9
90
2
4
Zig-Zag
53
4
90
2
9
Zig
4
2
53
9
90
Insert 64
4
2
53
9
90
64
Zig-Zag
4
2
64
53
90
9

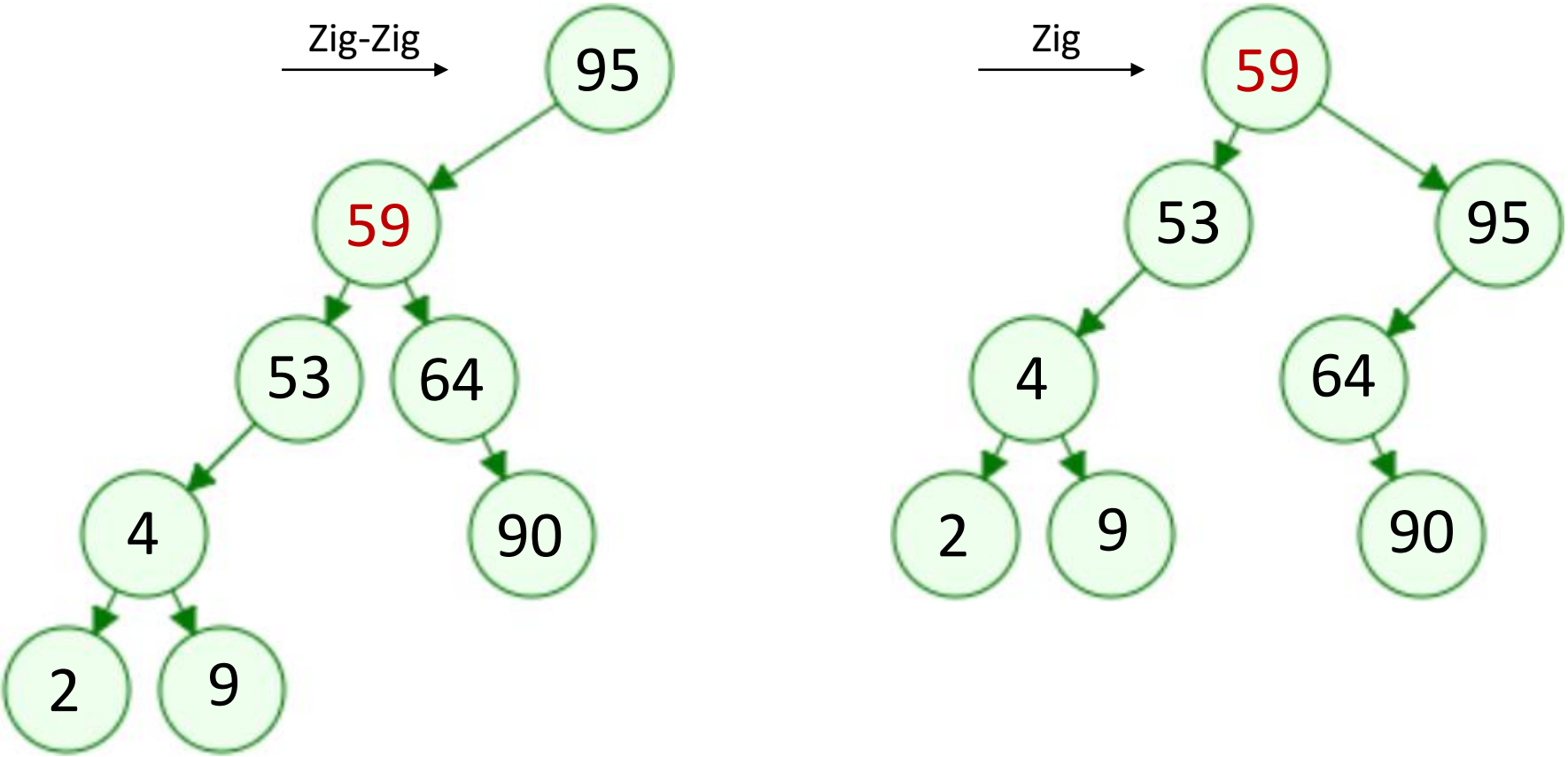
Contd…



Contd…



Contd…



Splay Tree

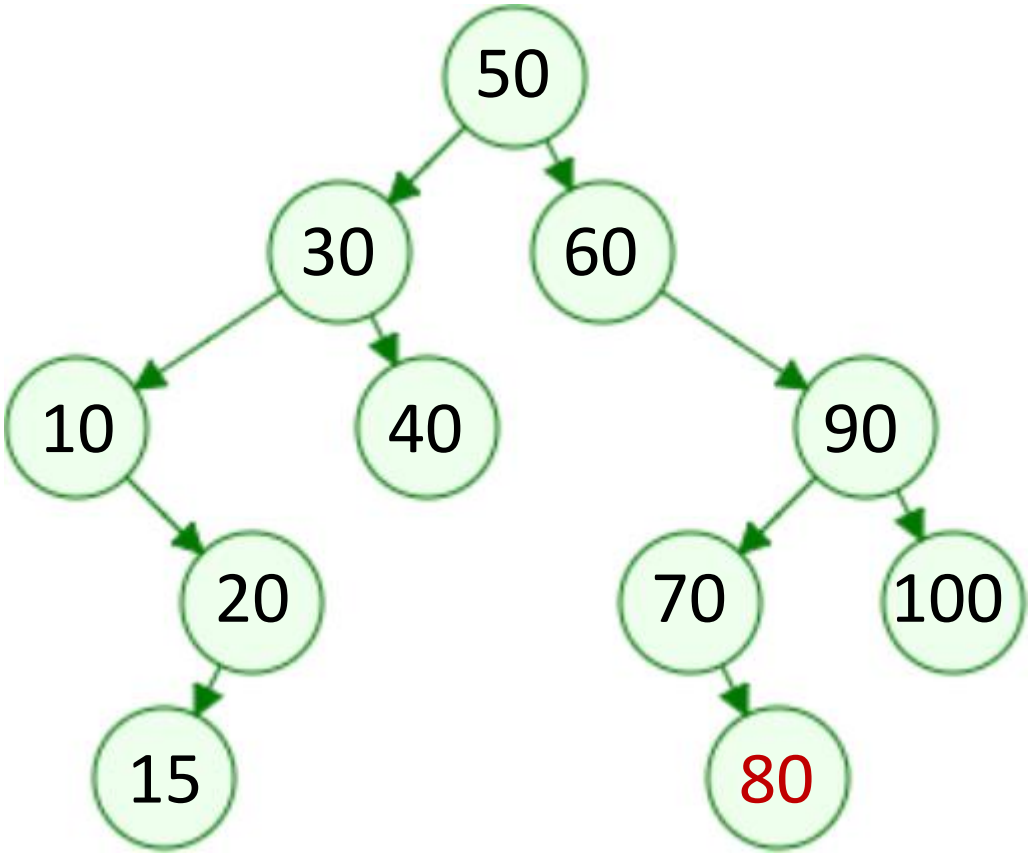
Operations using Bottom-up Approach

Operations

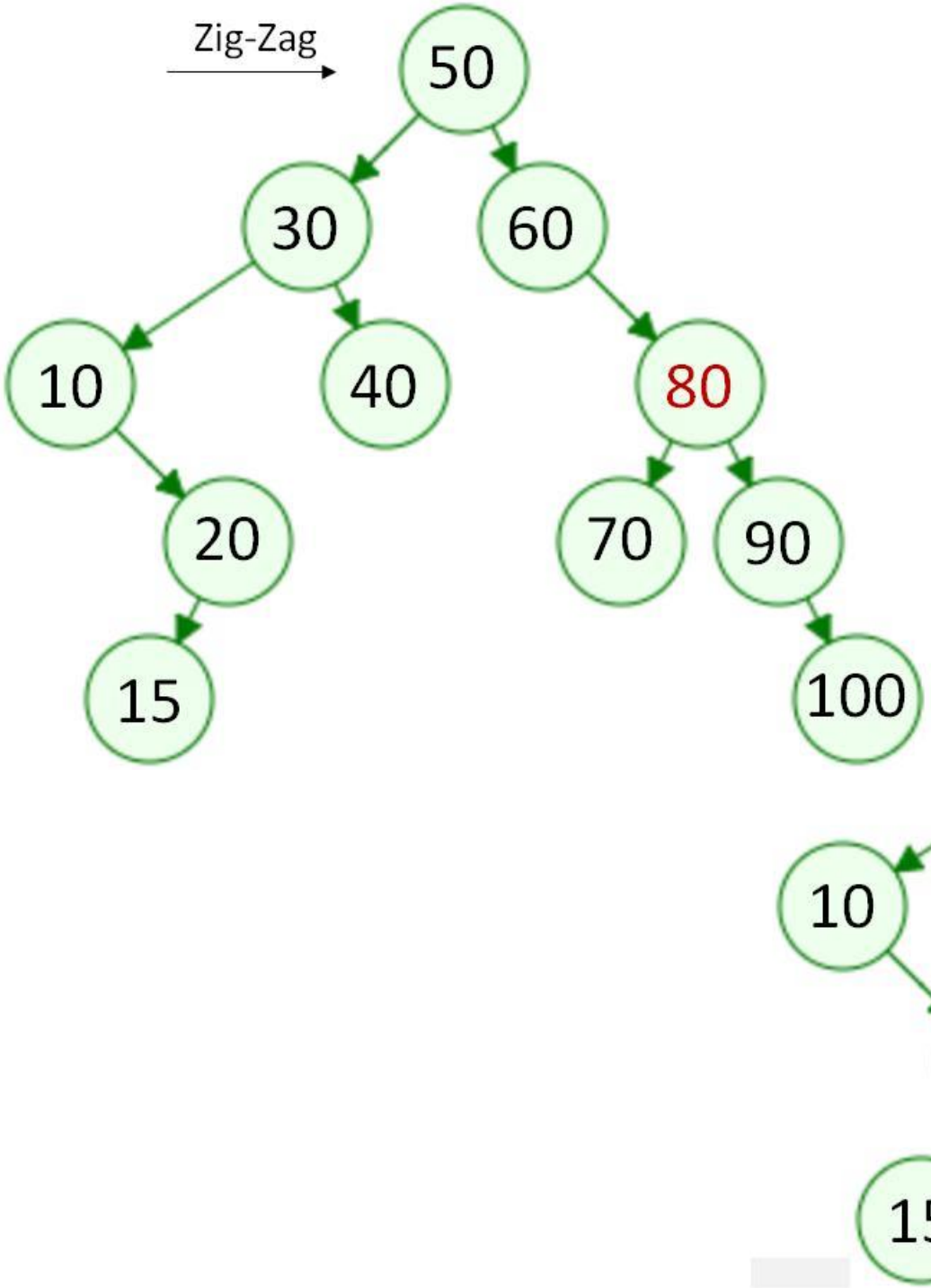
- Searching
  - Successful: Splay at the node being searched.
  - Unsuccessful: Splay at the node accessed just before reaching the NULL pointer.

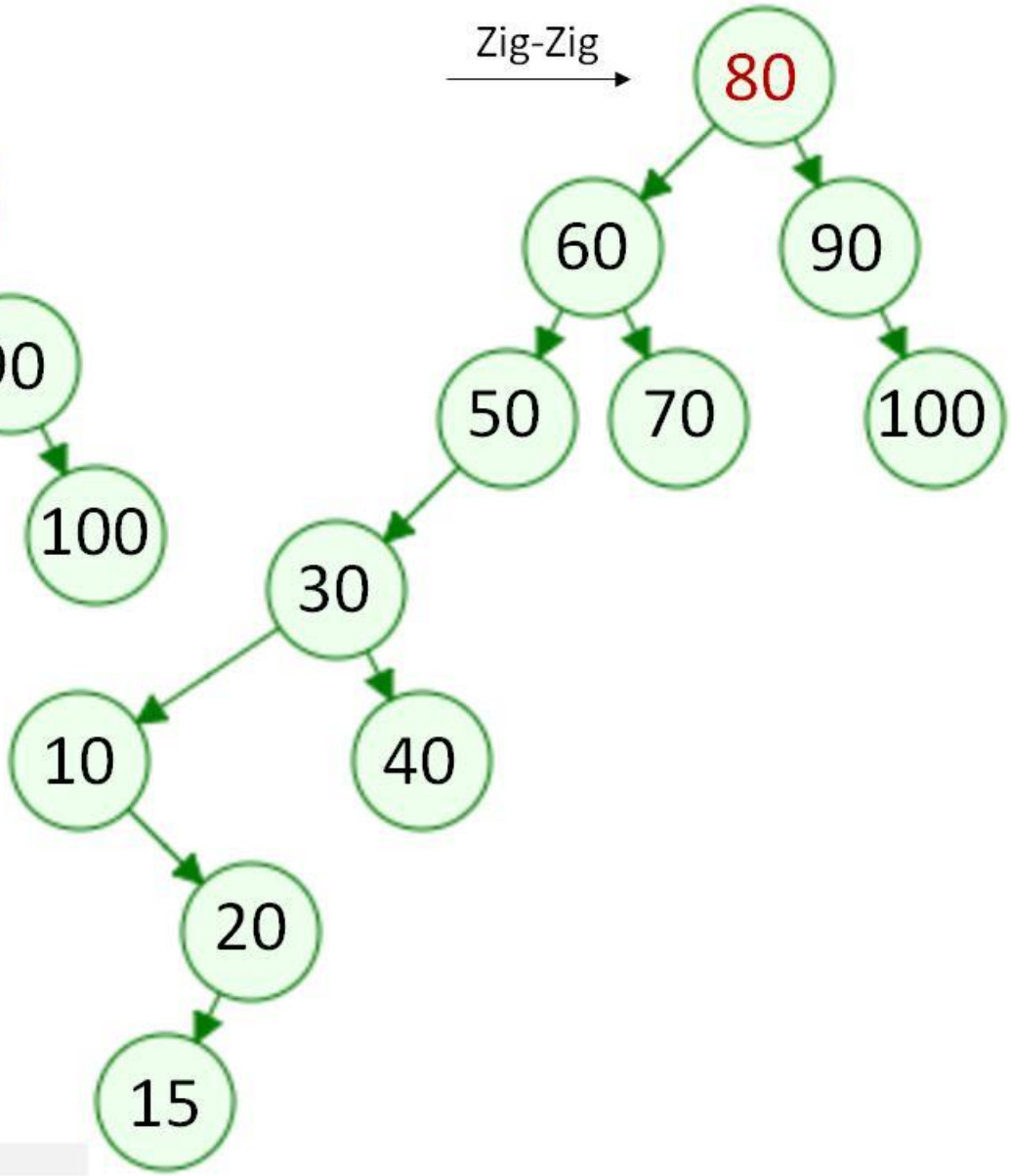
Example – Successful Searching (Bottom-up)

- Search 80.



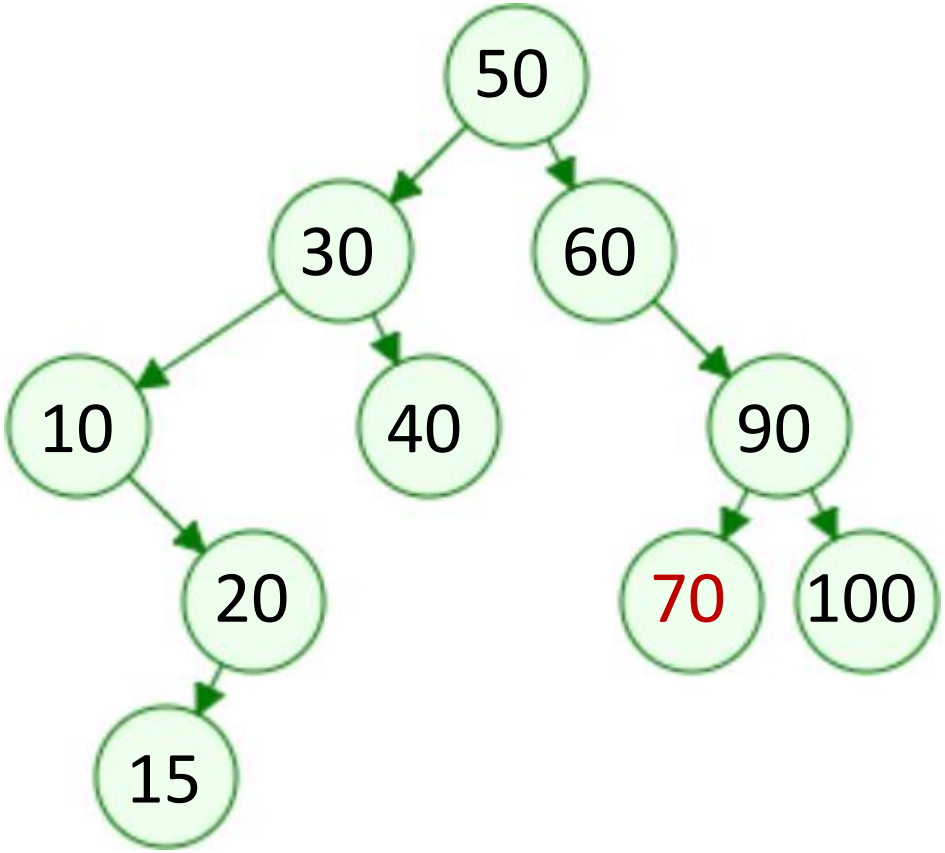
Contd…


Zig-Zag
50
30
60
10
40
80
20
70
90
15
100

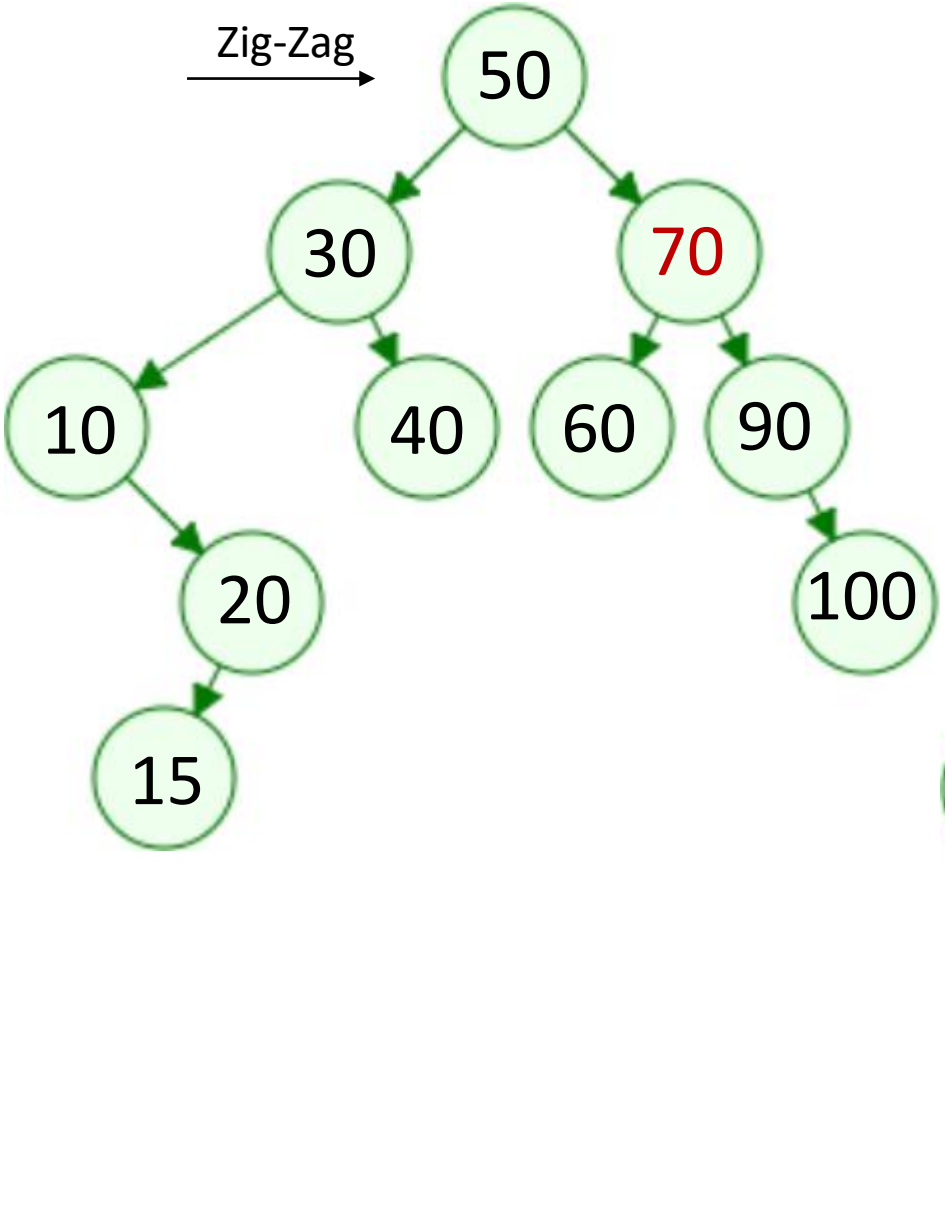

Zig-Zig
80
60
90
50
70
100
30
10
40
20
15

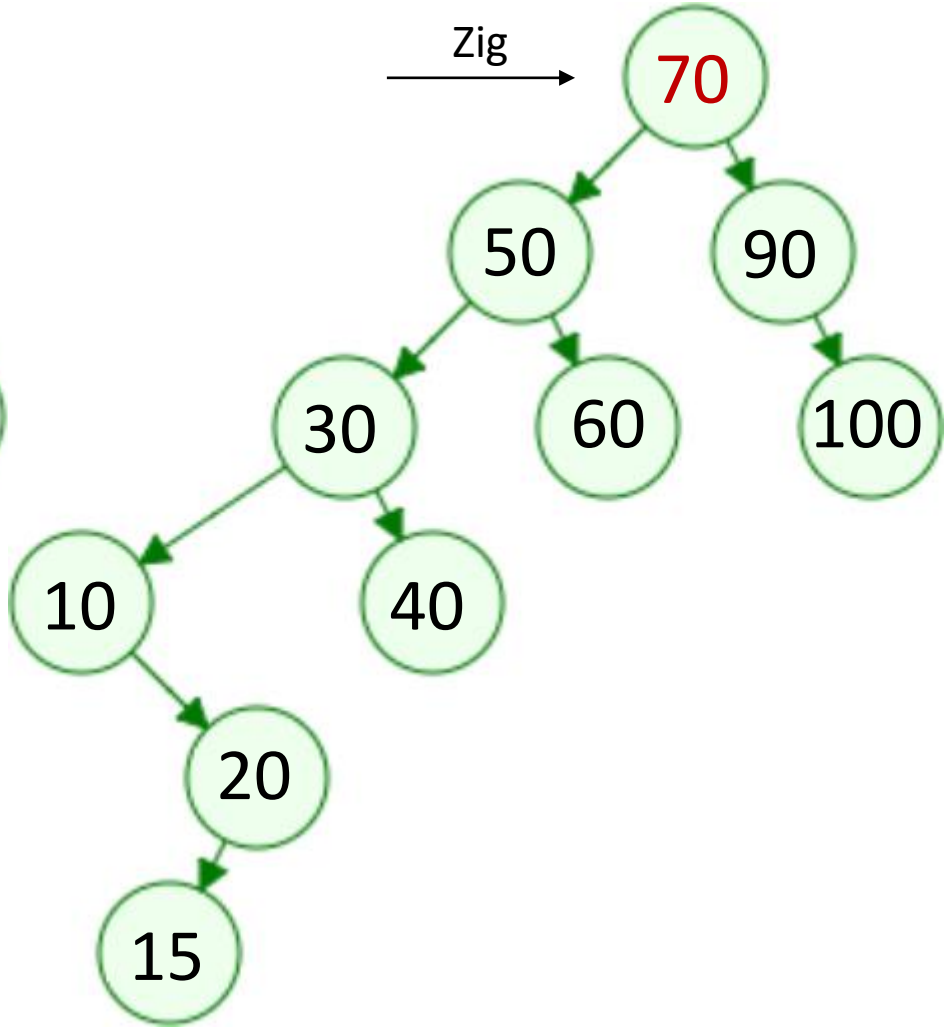
Example – Un-successful Searching (Bottom-up)

- Search 80.



Contd…

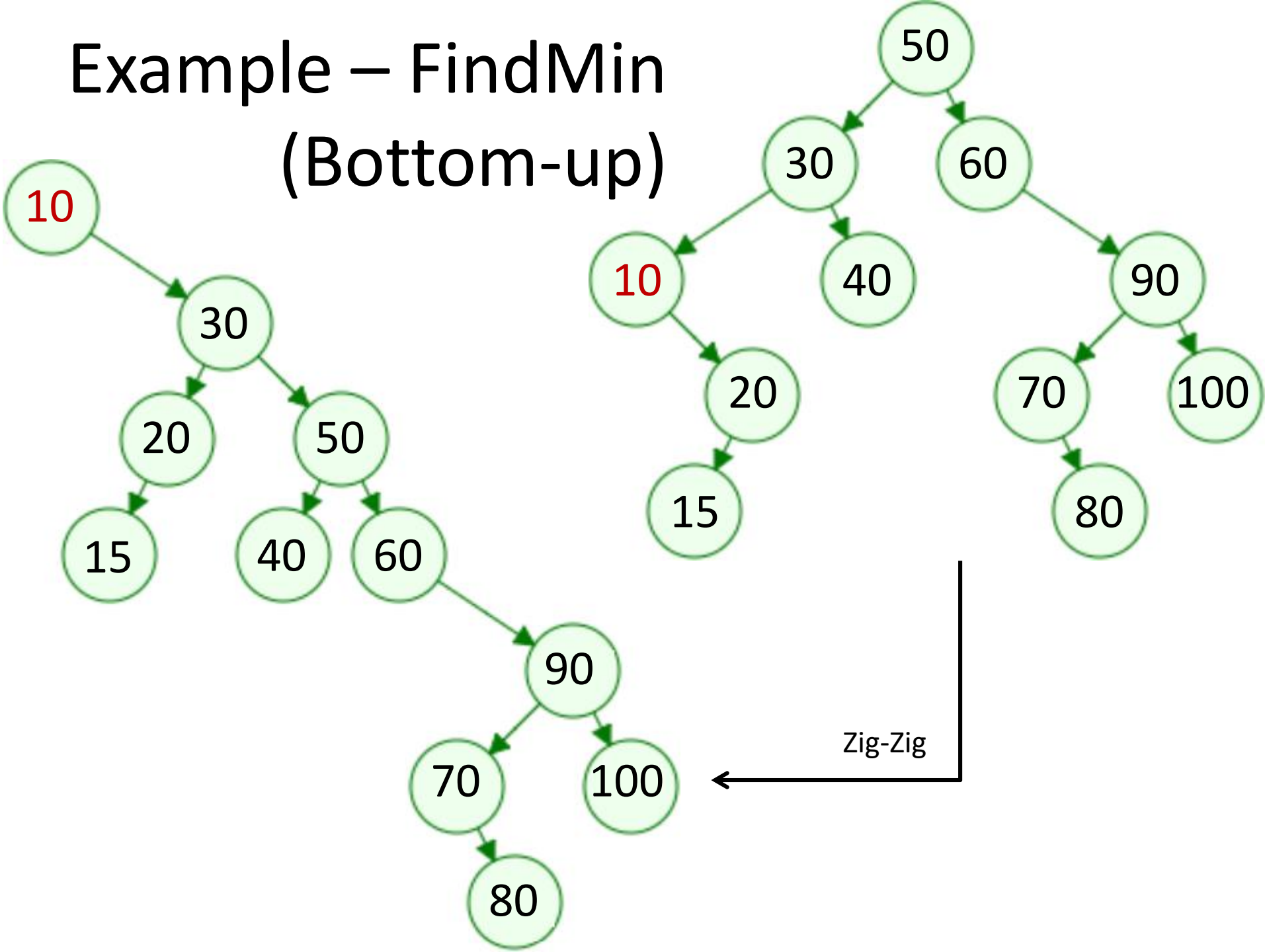

Zig-Zag
50
30
70
10
40
60
90
20
100
15


Zig
70
50
90
30
60
100
10
40
20
15

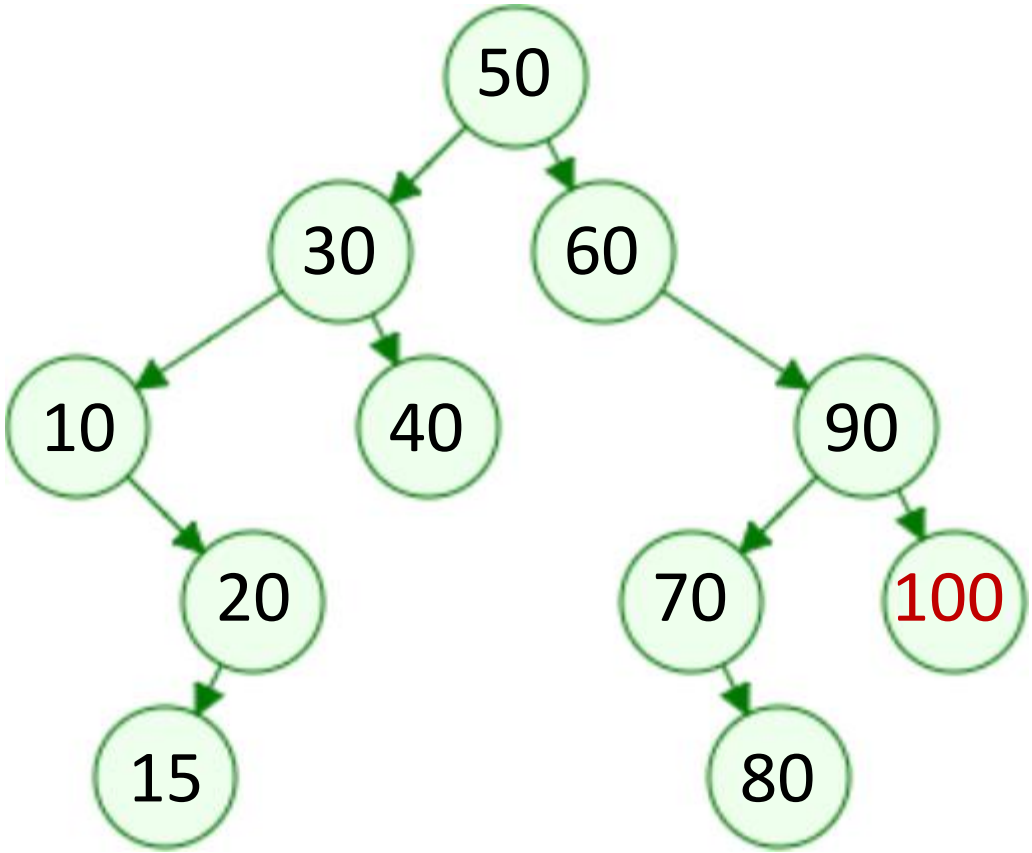
Operations

- FindMin
  - Splay at the minimum node.
- FindMax
  - Splay at the maximum node.

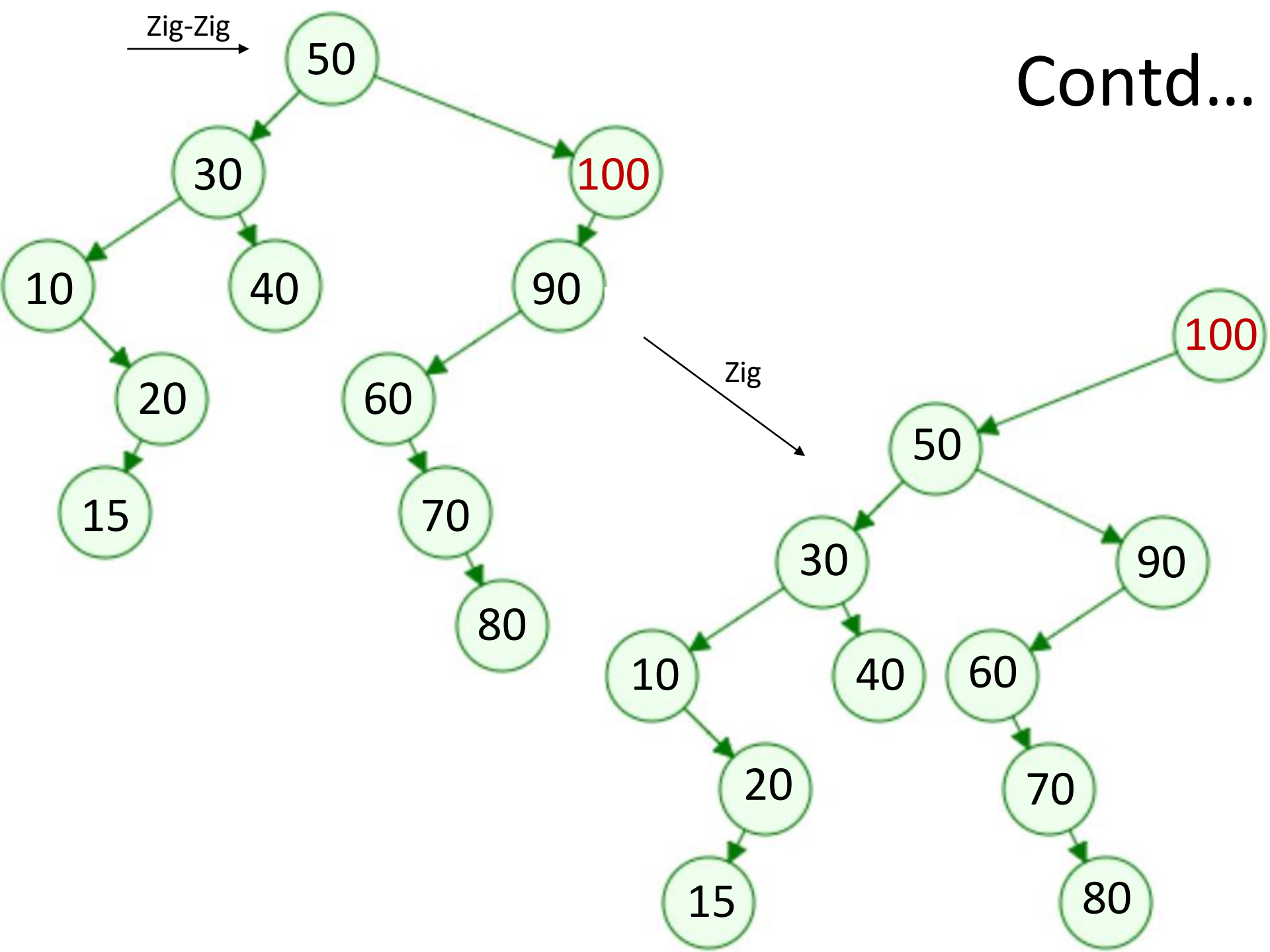
Example – FindMin (Bottom-up)



Example – FindMax (Bottom-up)



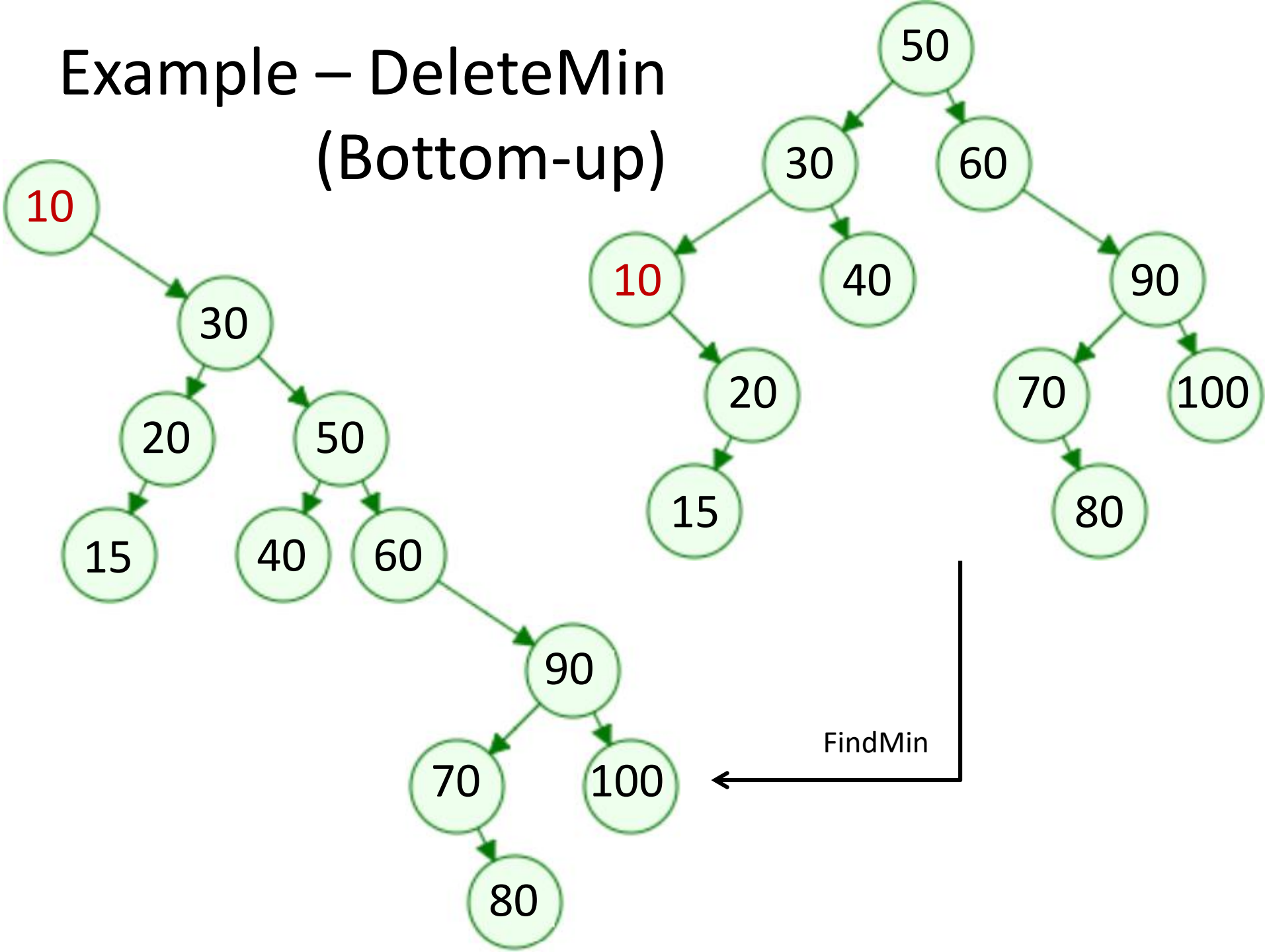
Contd…


Zig-Zig
50
30
100
10
40
90
20
60
15
70
80
Zig
100
50
30
90
10
40
60
20
70
15
80

Contd…

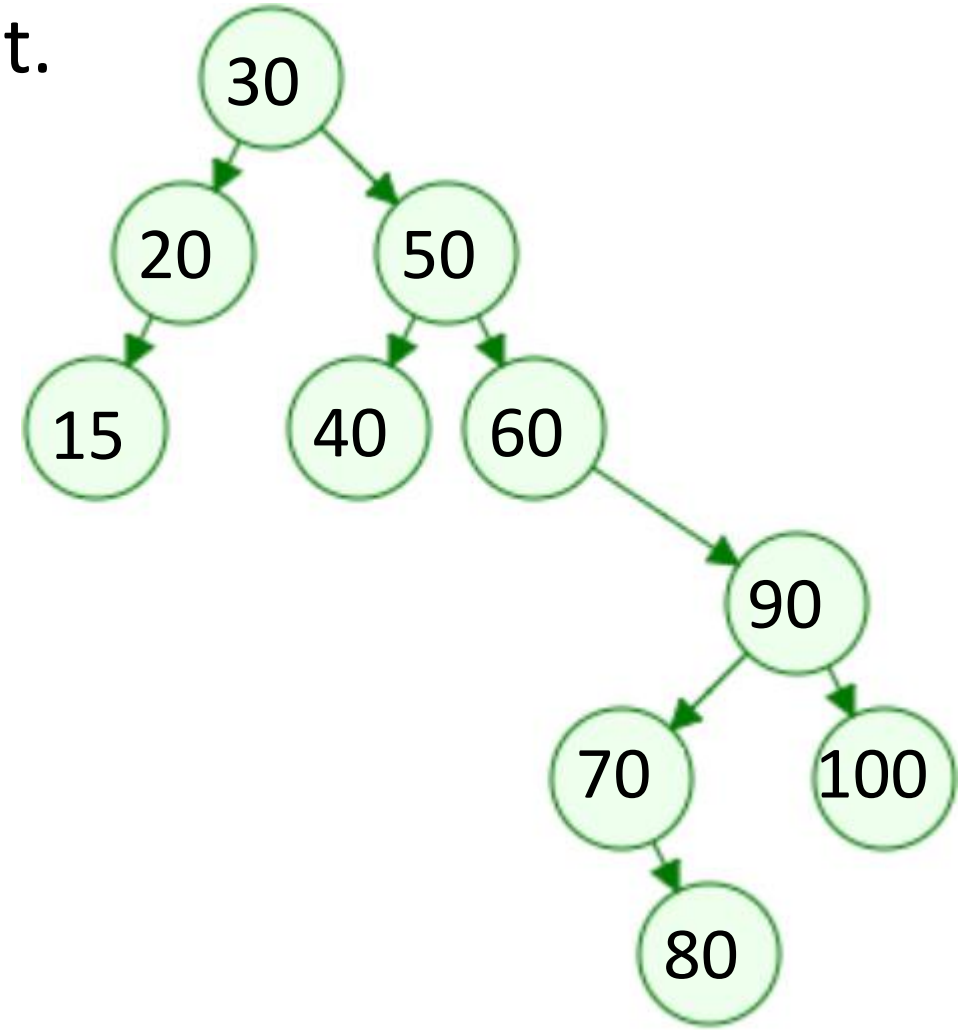
- DeleteMin
  - FindMin.
  - Use the right child as the new root and delete the node containing the minimum.
- DeleteMax
  - FindMax.
  - Use the left child as the new root and delete the node containing the maximum.

Example – DeleteMin (Bottom-up)

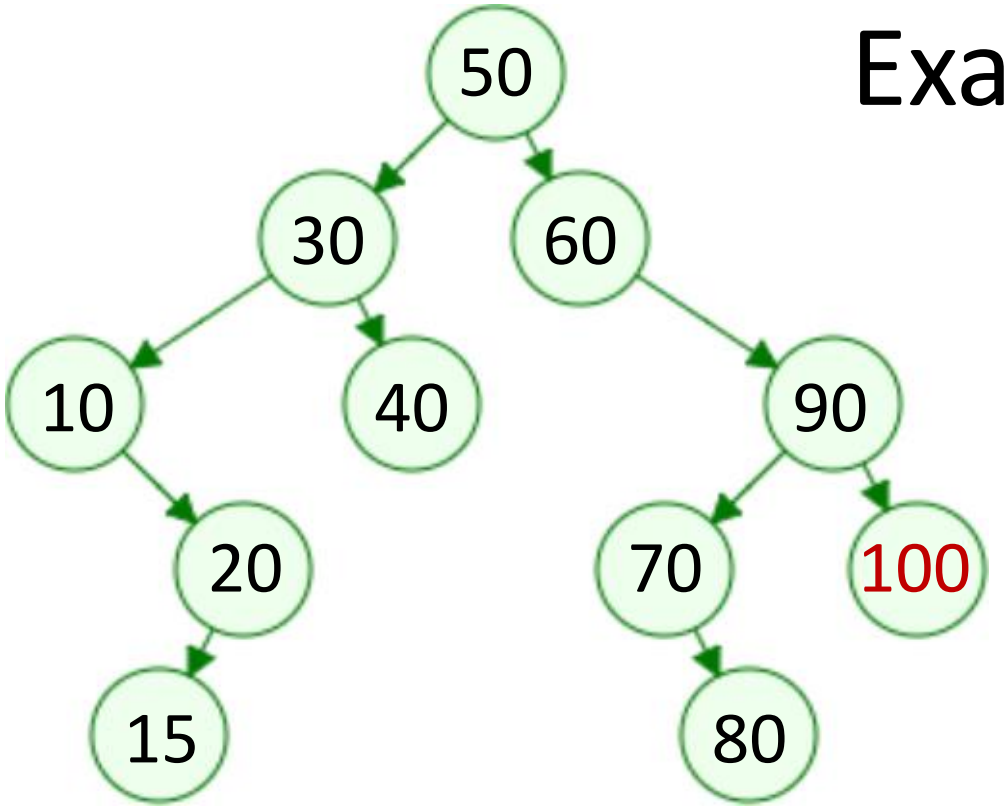


Contd…

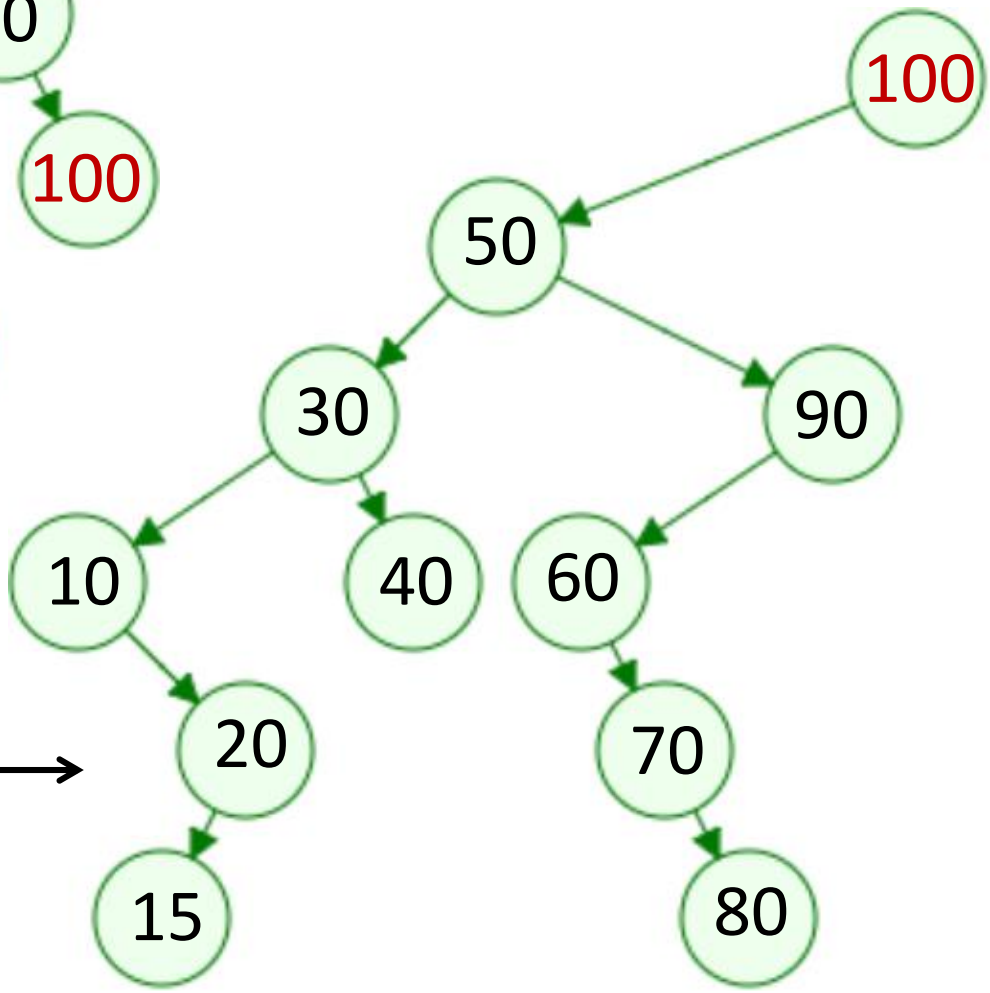
- Make right child of old root the new root and delete the old root.



Example – DeleteMax (Bottom-up)

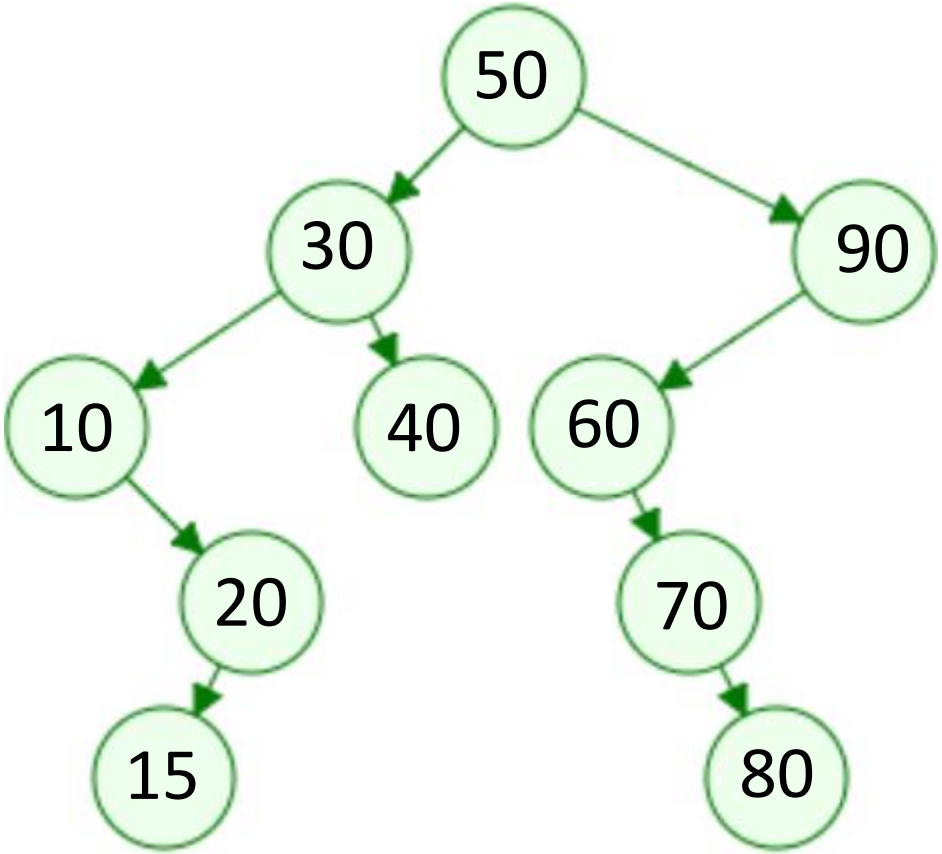


FindMax



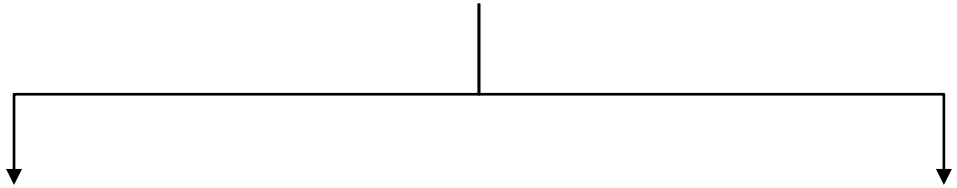
Contd…

- Make left child of old root the new root and delete the old root.



Contd…

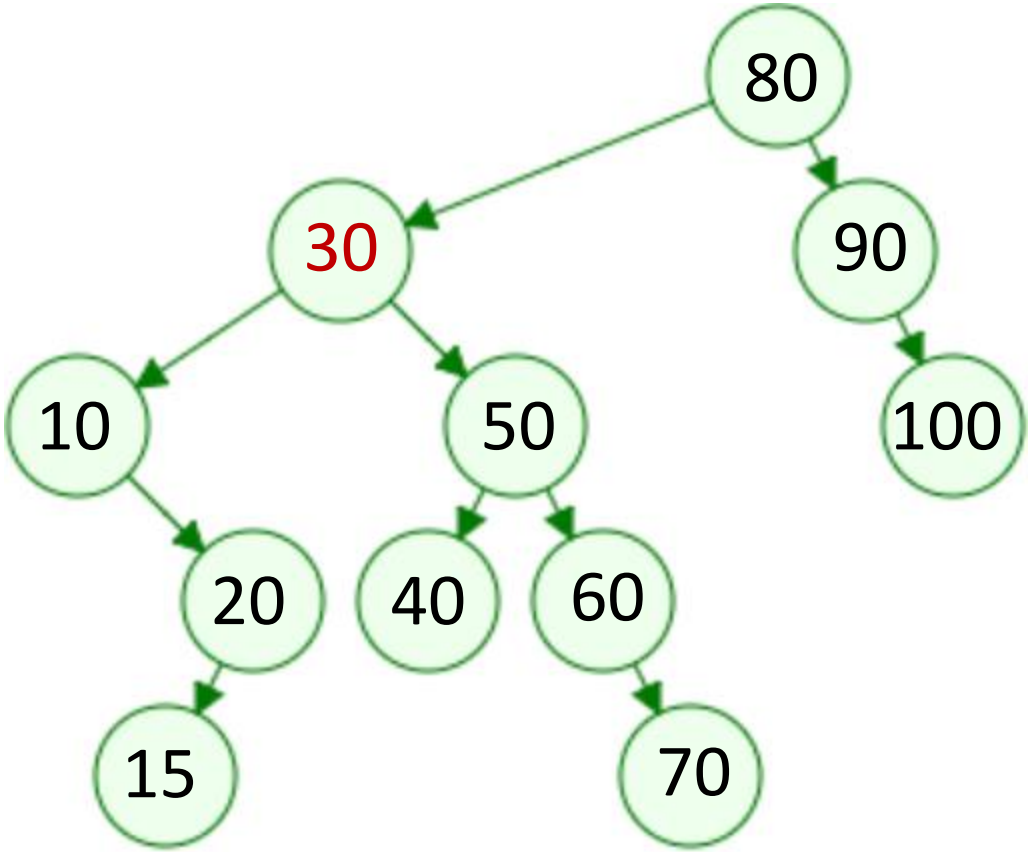
- Deletion
  - Splay at the node to be deleted.
  - Delete the root leaving two subtrees L (left) and R (right).



  - Find the largest element in L using a FindMax.
  - Make R the right child of L's root.
  - Find the smallest element in R using a FindMin.
  - Make L the left child of R's root.

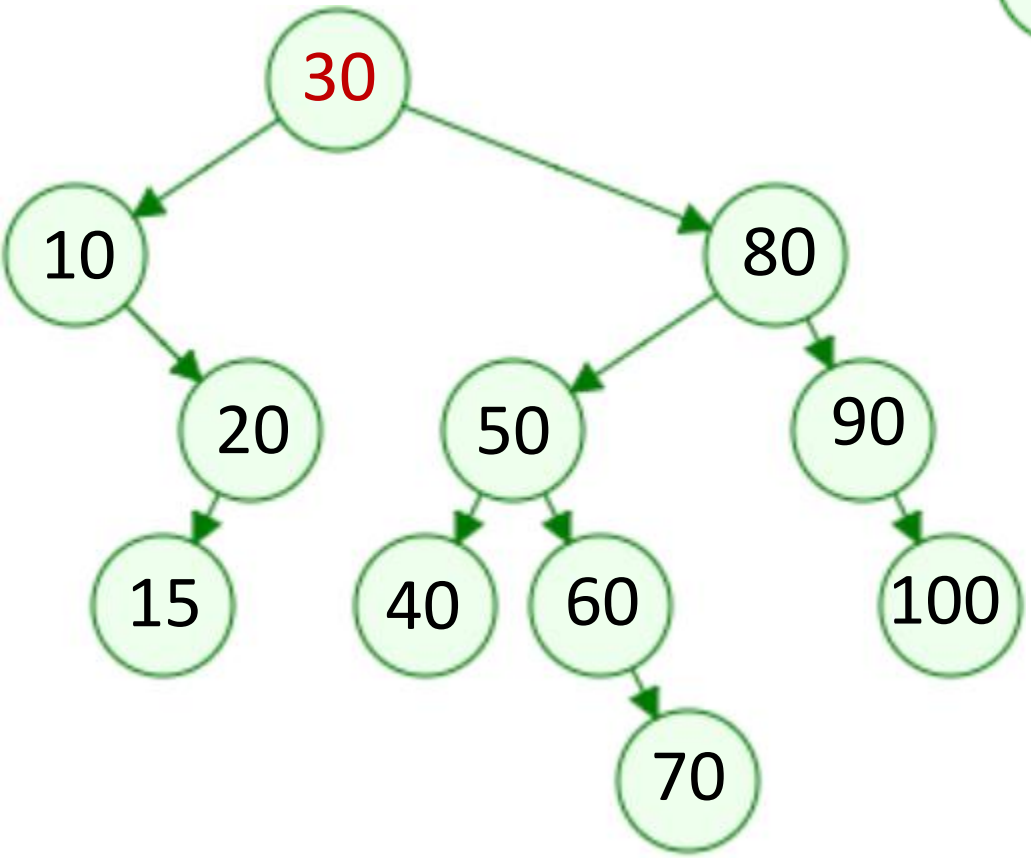
Example – Deletion (Bottom-up)

- Delete 30

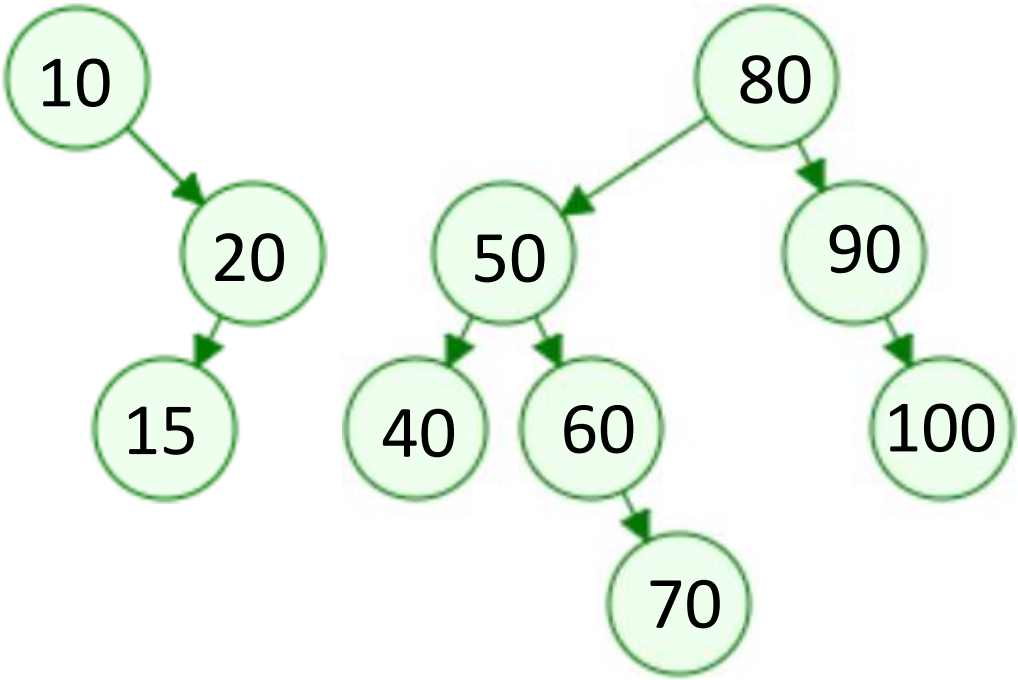


Contd…

- Search 30



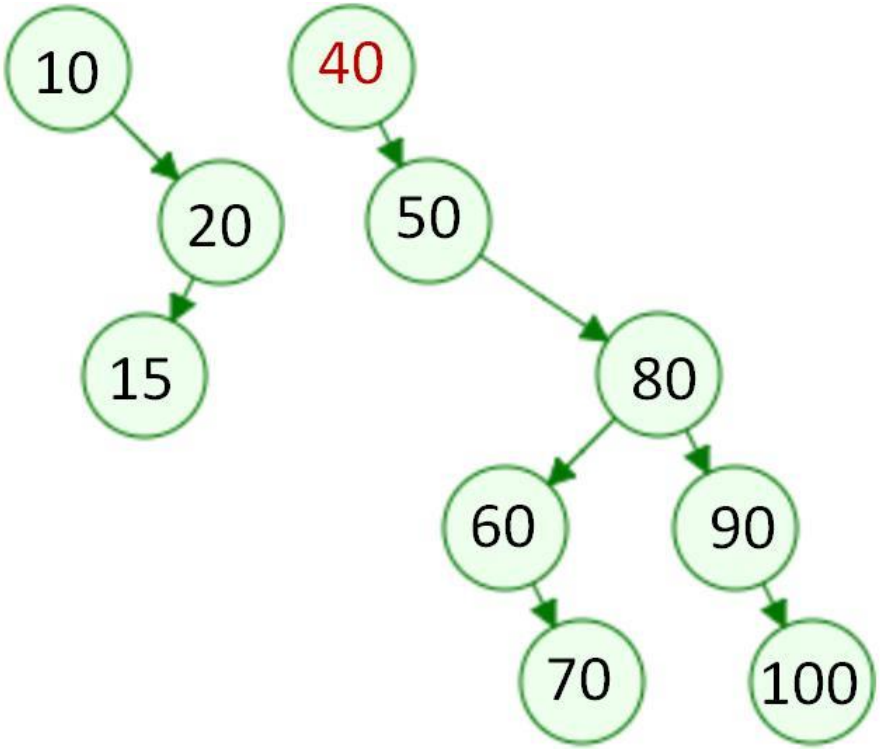
Delete 30



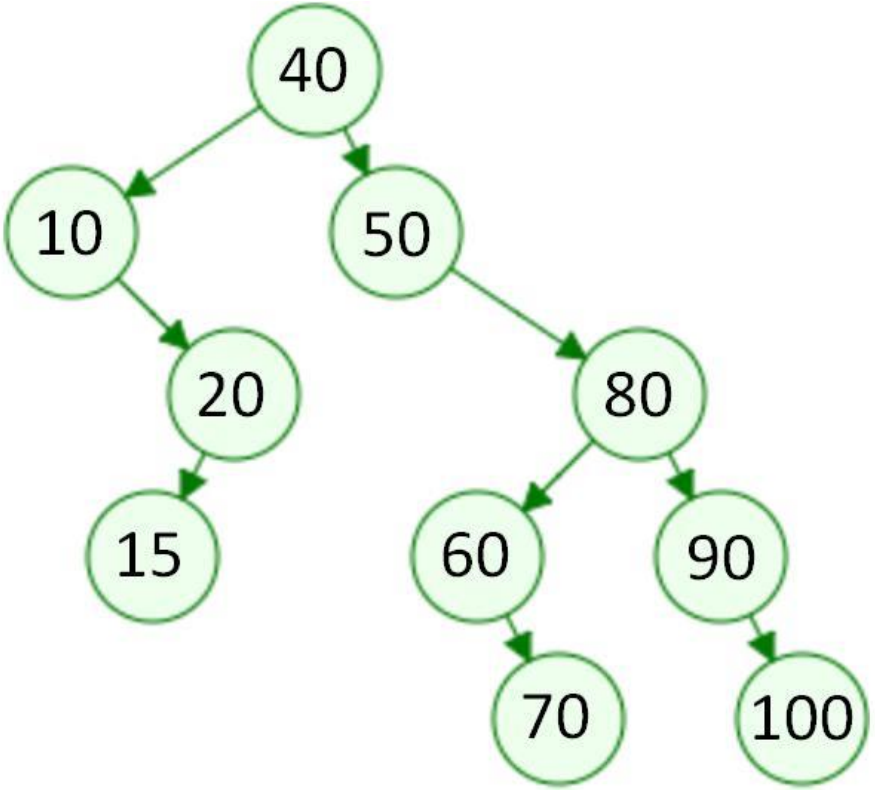
Delete 30

Contd…

FindMin in R

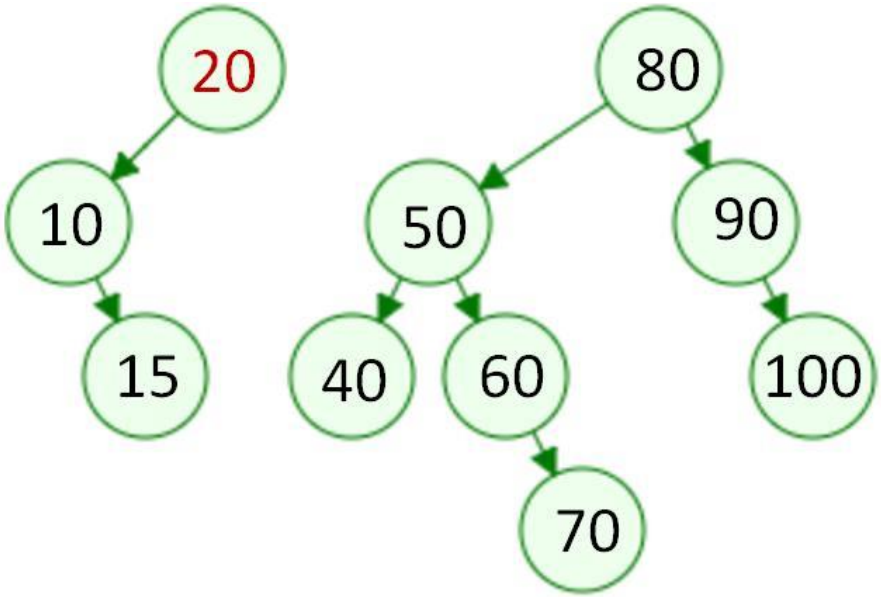


- Join

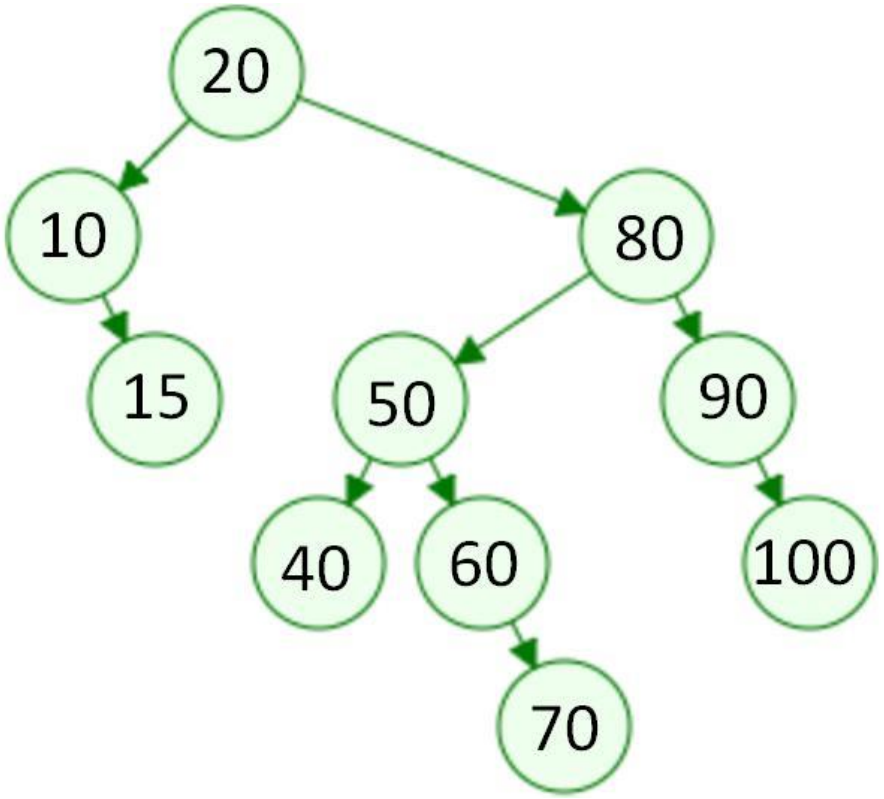


Contd…

FindMax in L



- Join



Bottom-Up Splay

```
splay(node *x)
1.  { while( x->parent )
2.    { if( !x->parent->parent )
3.       {  if( x->parent->left == x )
4.             right_rotate( x->parent )
5.          else
6.             left_rotate( x->parent ) }
7.      else if(x->parent->left == x &&
                      x->parent->parent->left == x->parent )
8.      {  right_rotate( x->parent->parent )
9.         right_rotate( x->parent ) }
```

Contd…

```
10.   else if( x->parent->right == x &&
                    x->parent->parent->right == x->parent )
11.   {   left_rotate( x->parent->parent )
12.       left_rotate( x->parent ) }
13.   else if( x->parent->left == x &&
                    x->parent->parent->right == x->parent )
14.   {   right_rotate( x->parent )
15.       left_rotate( x->parent ) }
16.   else
17.   {   left_rotate( x->parent )
18.       right_rotate( x->parent ) } } }
```

Further Readings

- http://web.onda.com.br/abveiga/capitulo12-ingles.pdf
- http://www.cs.cmu.edu/afs/cs/academic/class/15451-f00/www/lectures/lect0921
- https://www.cs.cornell.edu/courses/cs3110/2013sp/recitations/rec08-splay/rec08.html
- APPLICATION OF SPLAY TREES TO DATA COMPRESSION
  http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.1924&rep=rep1&type=pdf