CSCI 335 Software Design and Analysis III

Splay Trees/B-trees

Chapter 4

Ioannis Stamos

Amortized cost

- Consider a sequence of M operations(insert/delete/find)
  - What is the total cost?
  - What is the average cost per operation?

Amortized cost

- Consider a sequence of M operations (insert/delete/find):
- Example: binary search tree (regular)
  - M operations can cost in the worst case **O(M * N)**
  - Each operation will **not** cost more than **O(N)**
  - On average each operation costs **O(N)**

Amortized cost

- Consider a sequence of M operations (insert/delete/find):
- AVL tree:
  - A sequence of M operations will cost **O(M * logN)**
  - Each operation will **not** cost more than **O (logN)**
  - On average each operation costs **O (logN)**

Amortized cost

- Consider a sequence of M operations (insert/delete/find):
- Suppose that total cost is **O(M * f(N))**

  irrespective of the actual sequence of operations.
- The average cost is **O(f(N))** for each operation.
- This is called **amortized running time**.
- Caveat:

Individual operations in the sequence can be expensive though !

Splay tree

- A tree with amortized running time of O (logN)
- Some operations could be more expensive
- How can we achieve this?

Splay tree

- The trick is to rebalance the tree after a **find()** operation
- Bring the item returned by find() to the root while applying rotations on the way to the root.

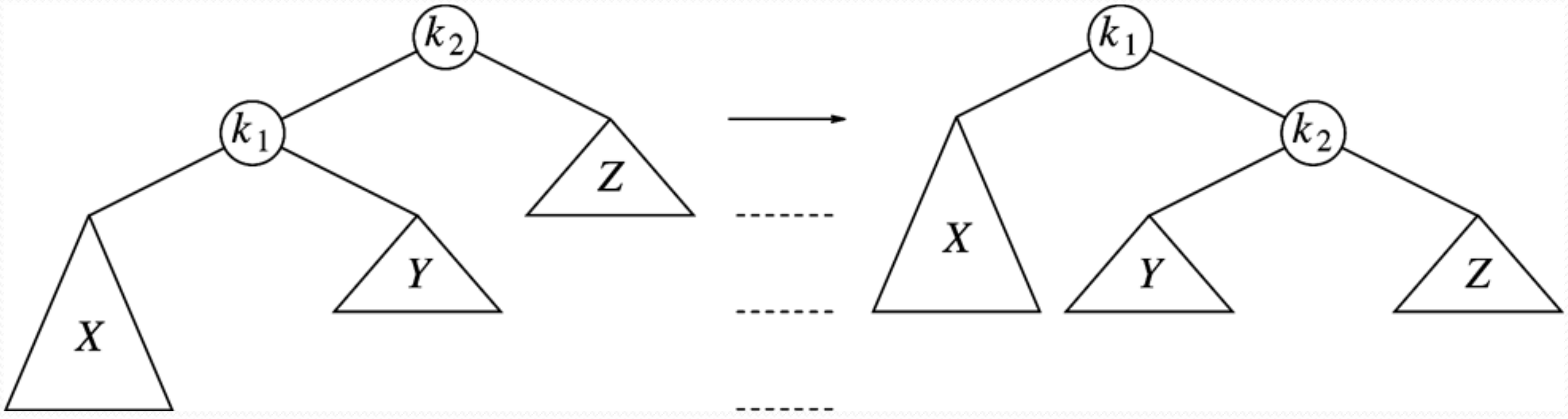
Splay tree

- Successive **finds()** of same element will be pretty fast
- Other items are coming **closer** to the root
- No need to store height information at each node
- **Amortized cost** of sequence of M operations is O(logN)

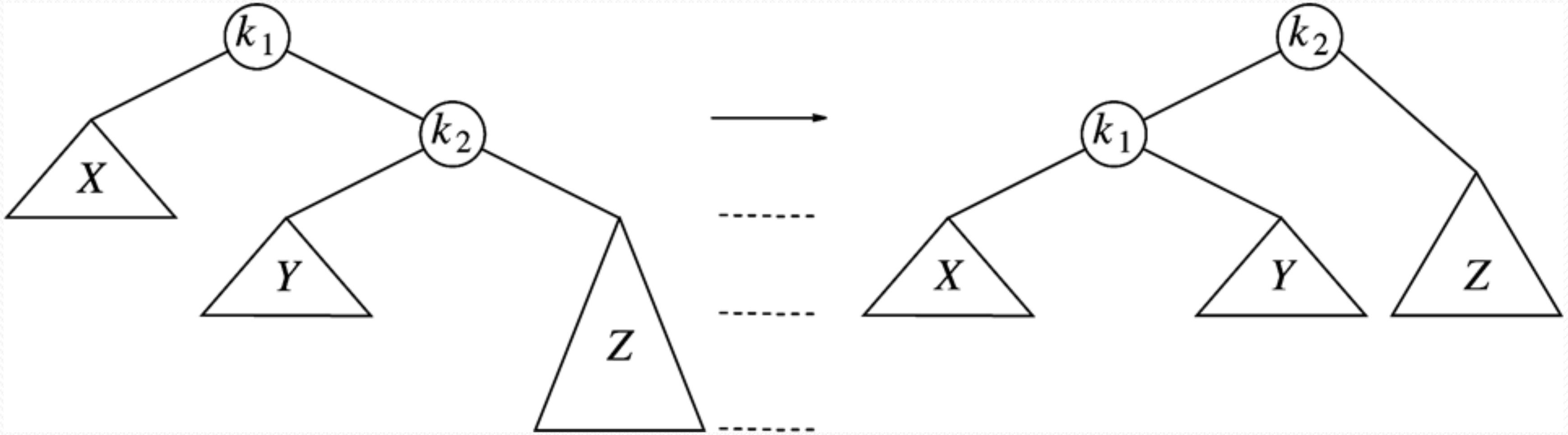
Splay Trees: a simple idea that does not work

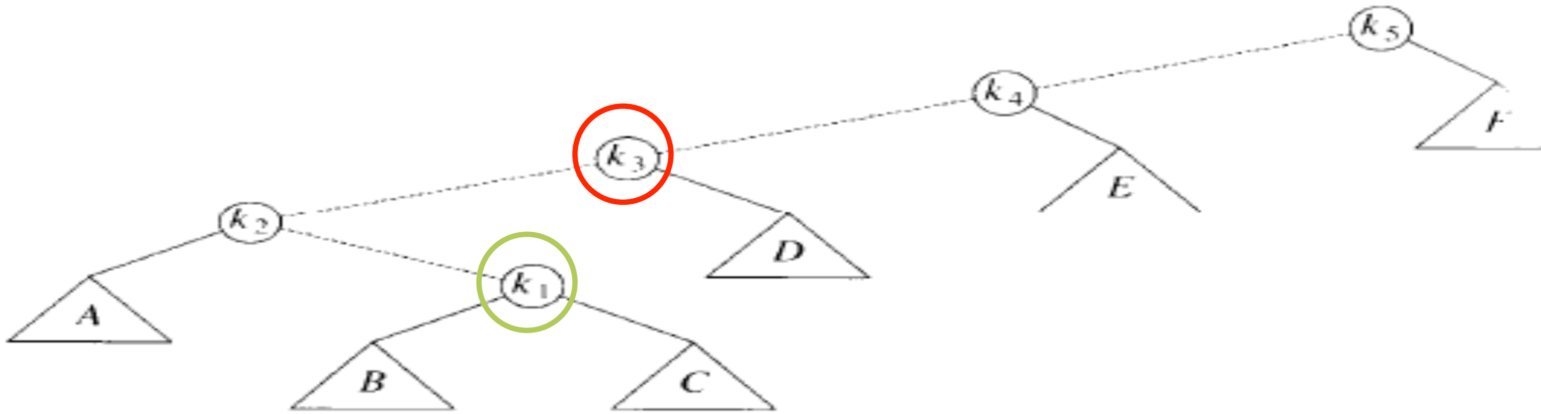
- After find() perform single-rotations bottom-up

Single rotation (case 1)

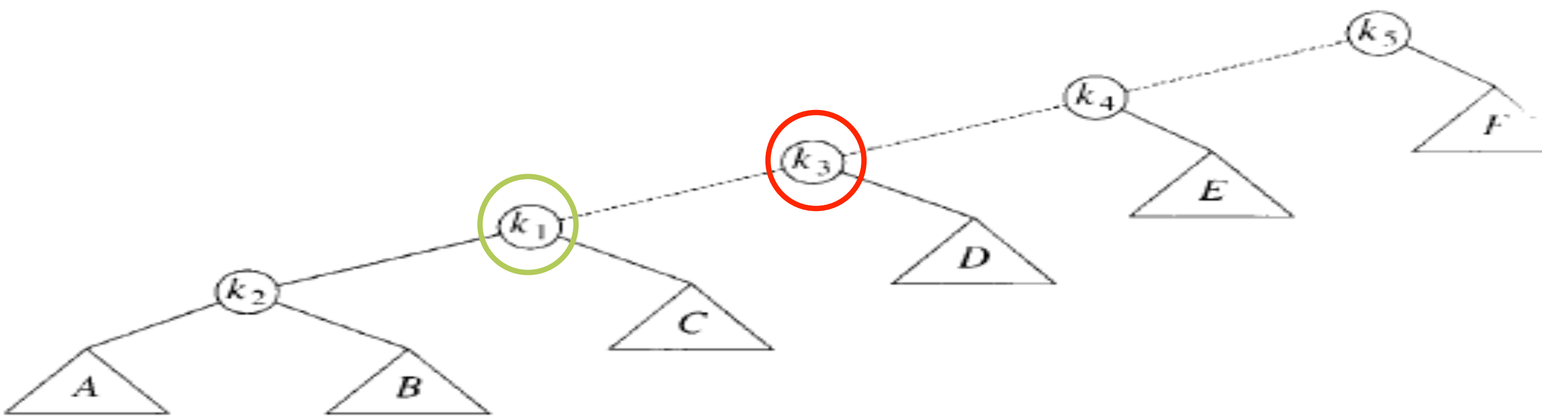


Single rotation (case 4)

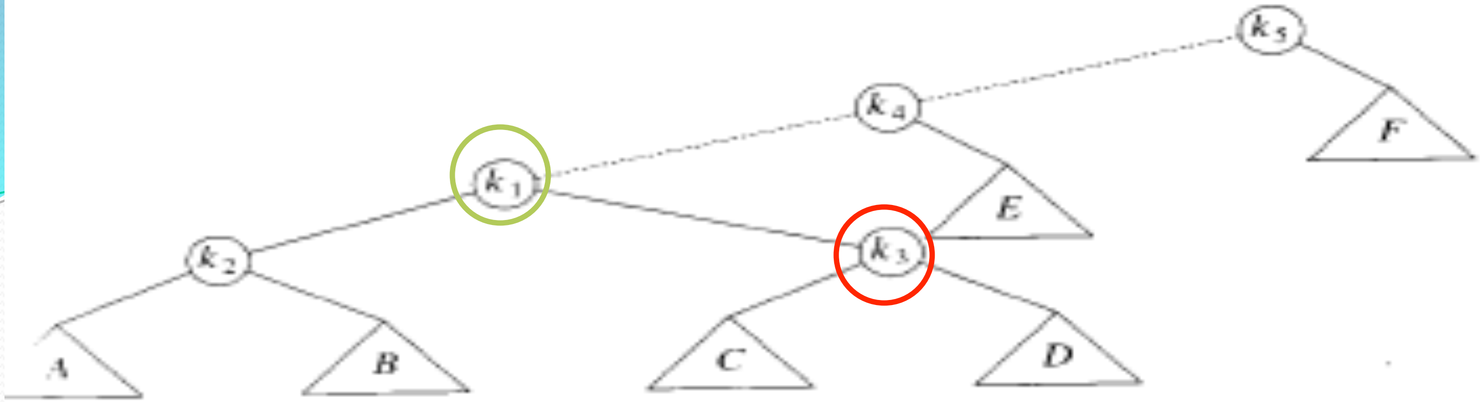




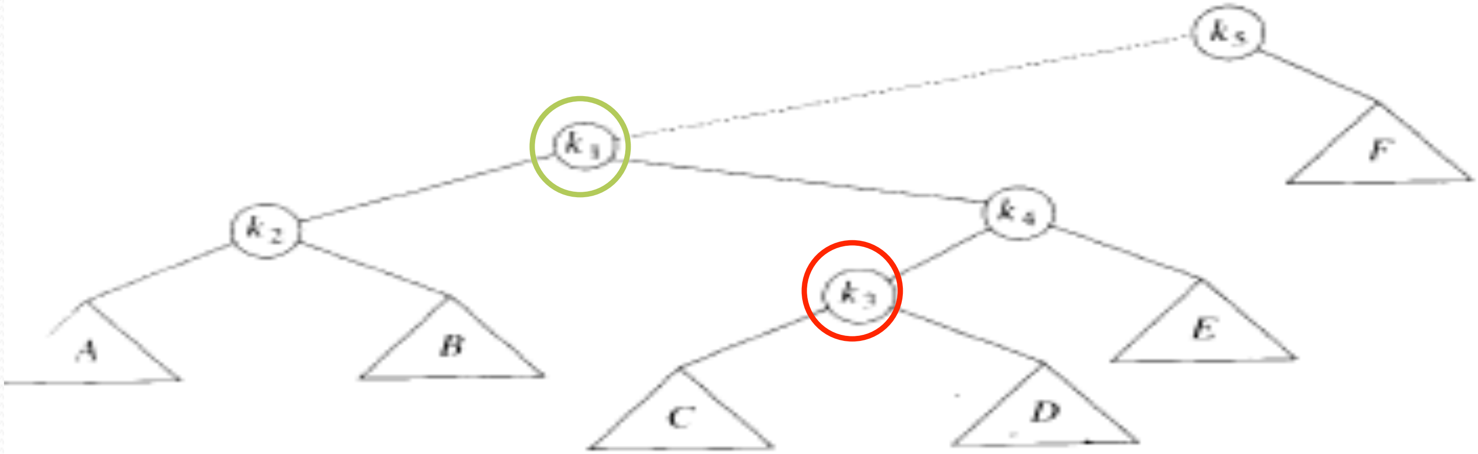
The access path is dashed. First, we would perform a single rotation between k_1 and
parent, obtaining the following tree.

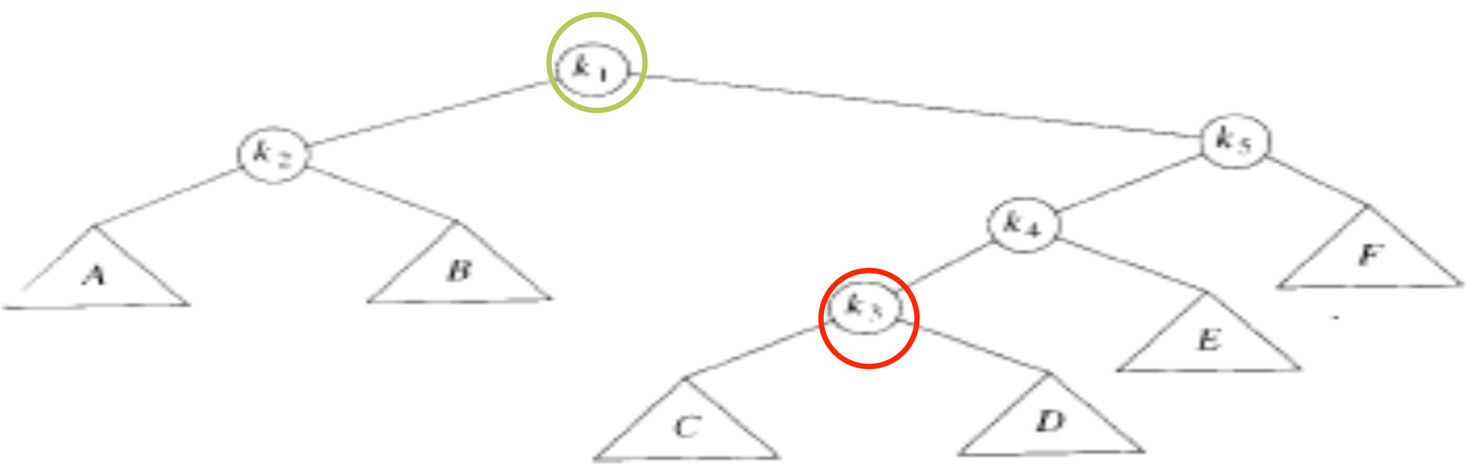


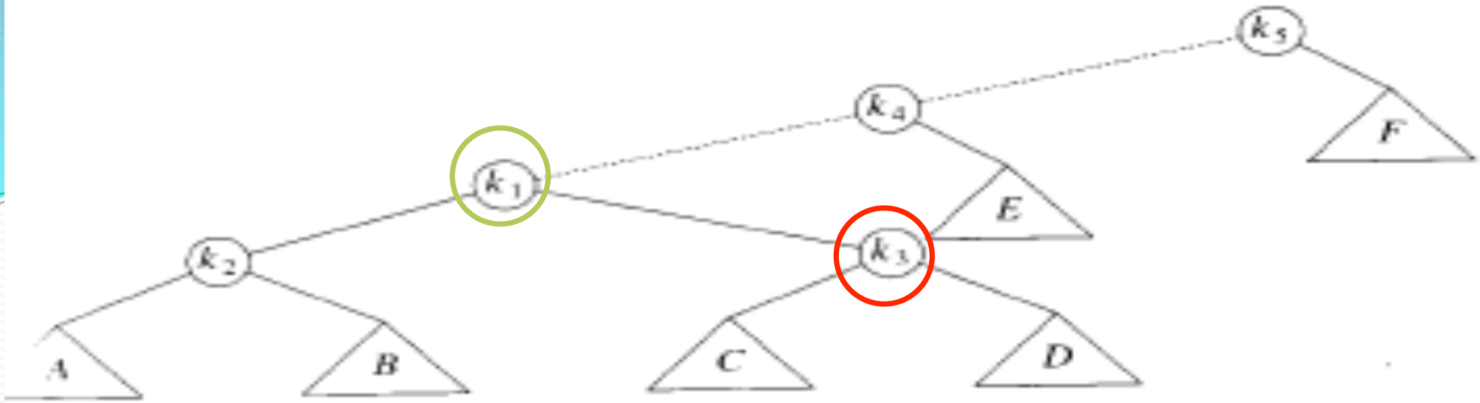
Then, we rotate between k_1 and k_3, obtaining the next tree.



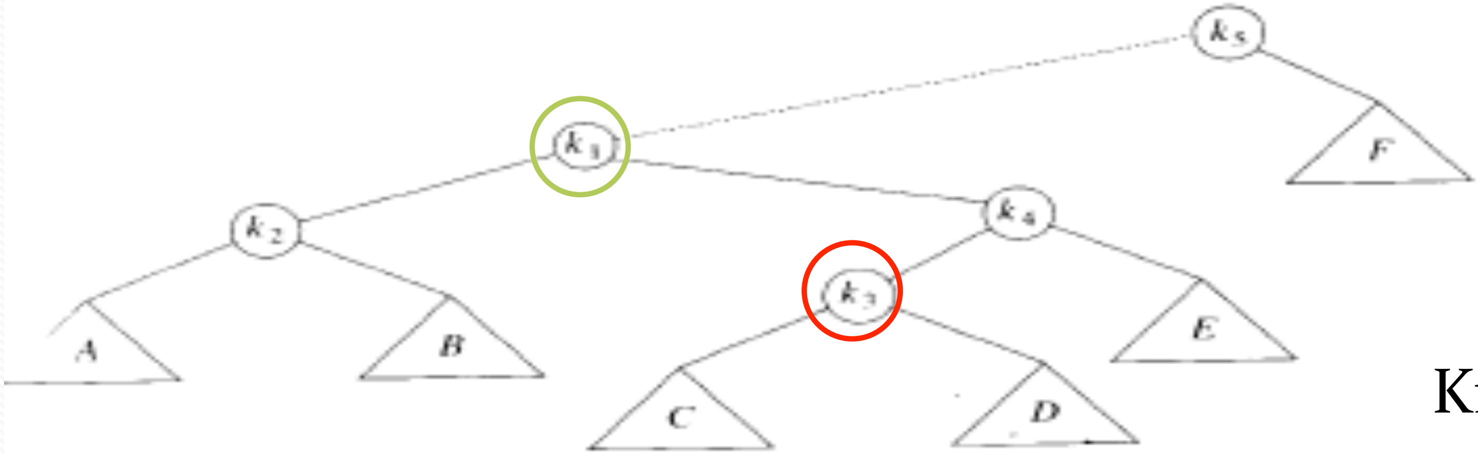
en two more rotations are performed until we reach the root.

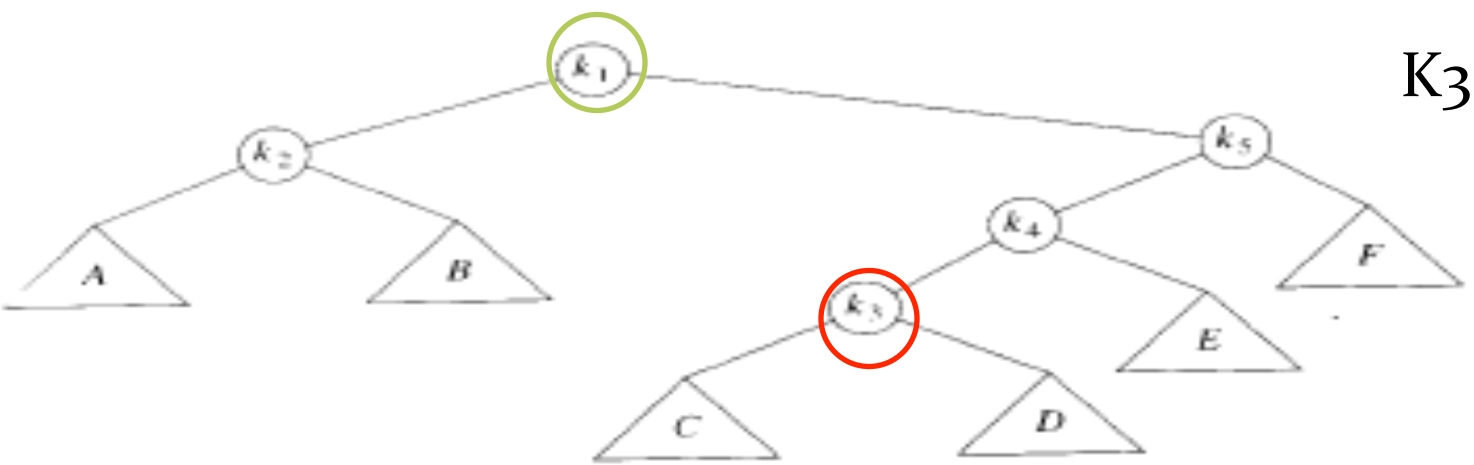






…en two more rotations are performed until we reach the root.





K1 went up

K3 went down

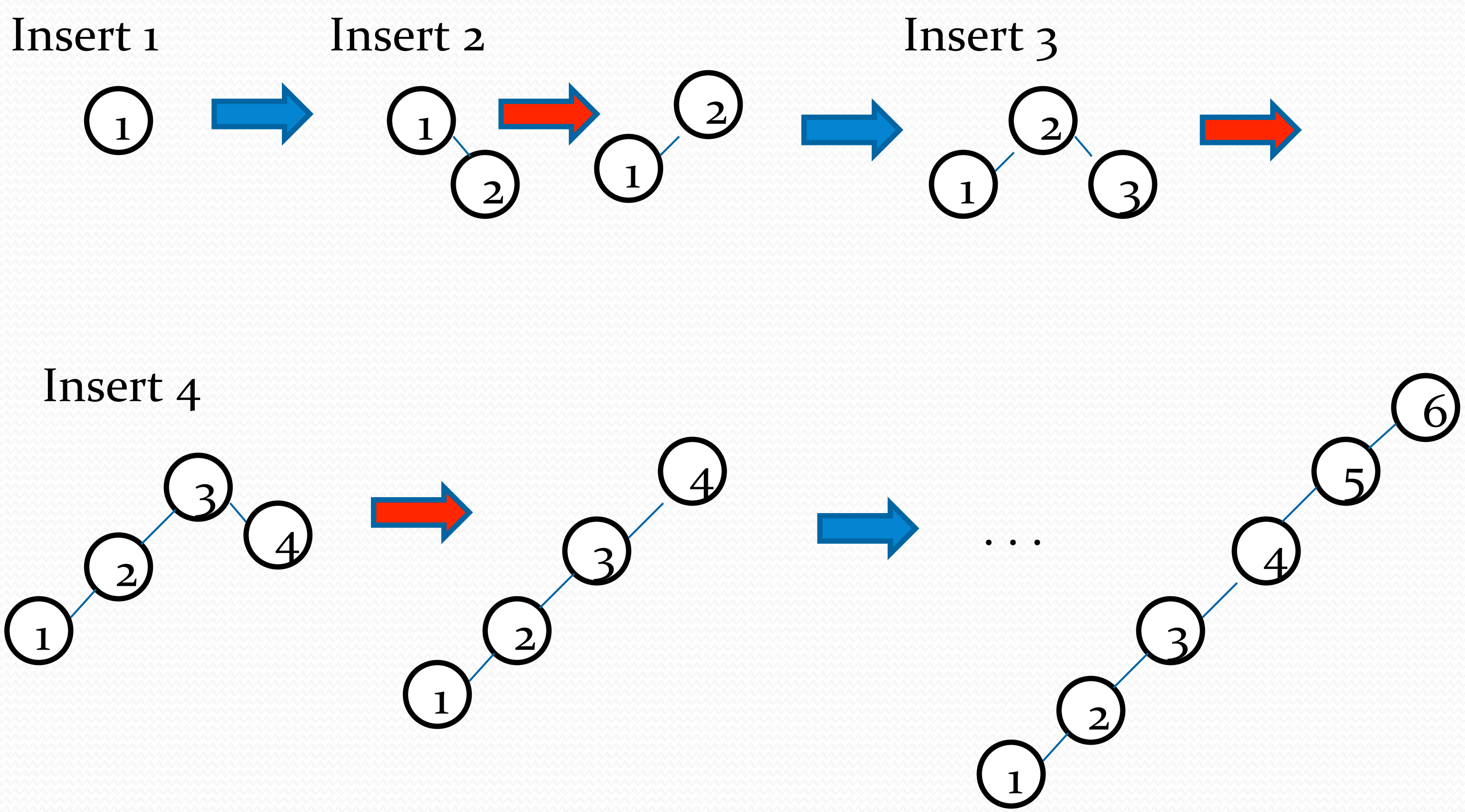
Splay Trees: a simple idea that does not work

- There is a sequence of M operations requiring $\Omega(N)$ time (**amortized**).
- => we want logarithmic amortized time

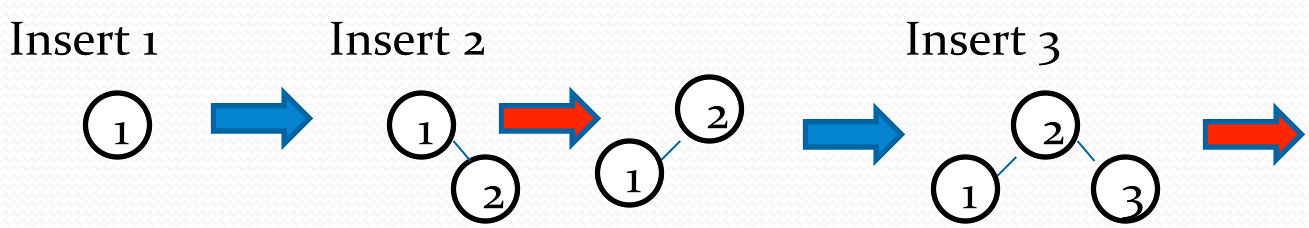
Splay Trees: M operations requiring $\Omega(N)$ time

- Consider inserting 1,2,3, … , N into an initially empty tree
  - Note that you splay on insertion (i.e. single AVL rotation)=>
  - Only left children
- Total time to build tree is O (N) (not bad)

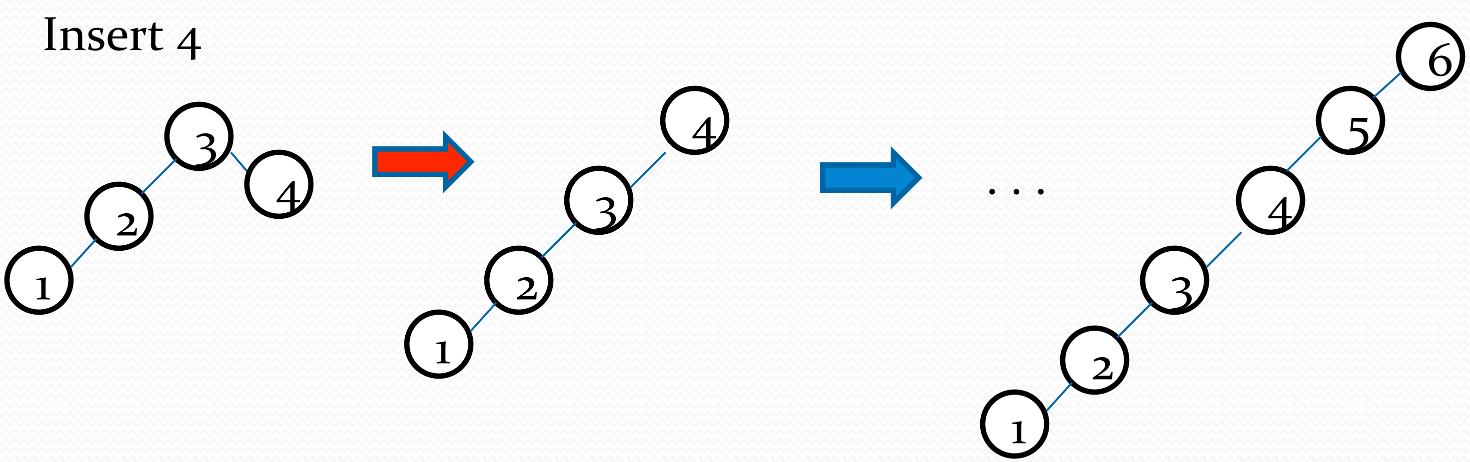
Insert 1, 2, 3, 4, 5, 6, ...



Insert 1, 2, 3, 4, 5, 6, … , N

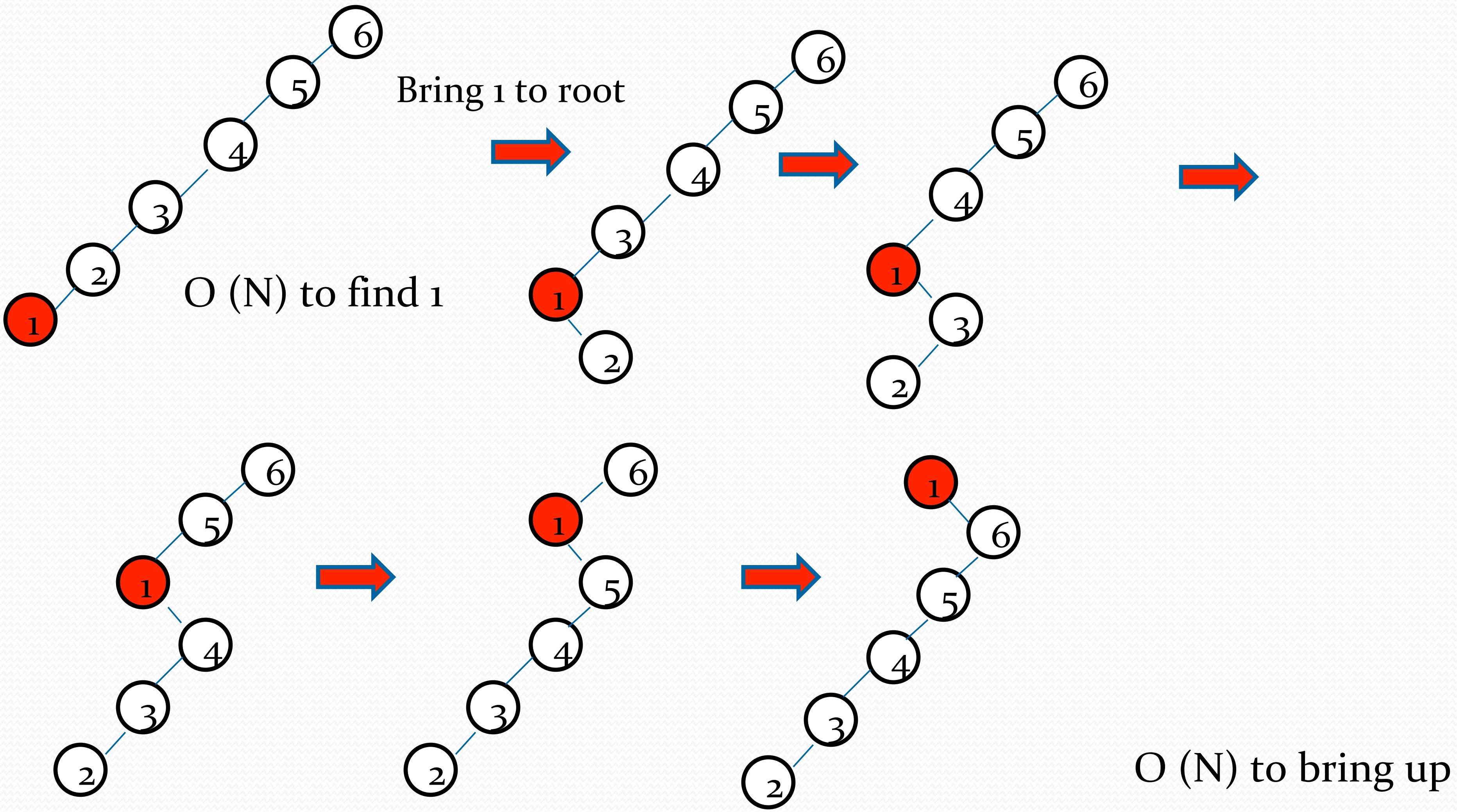


Total time to build tree is O (N)



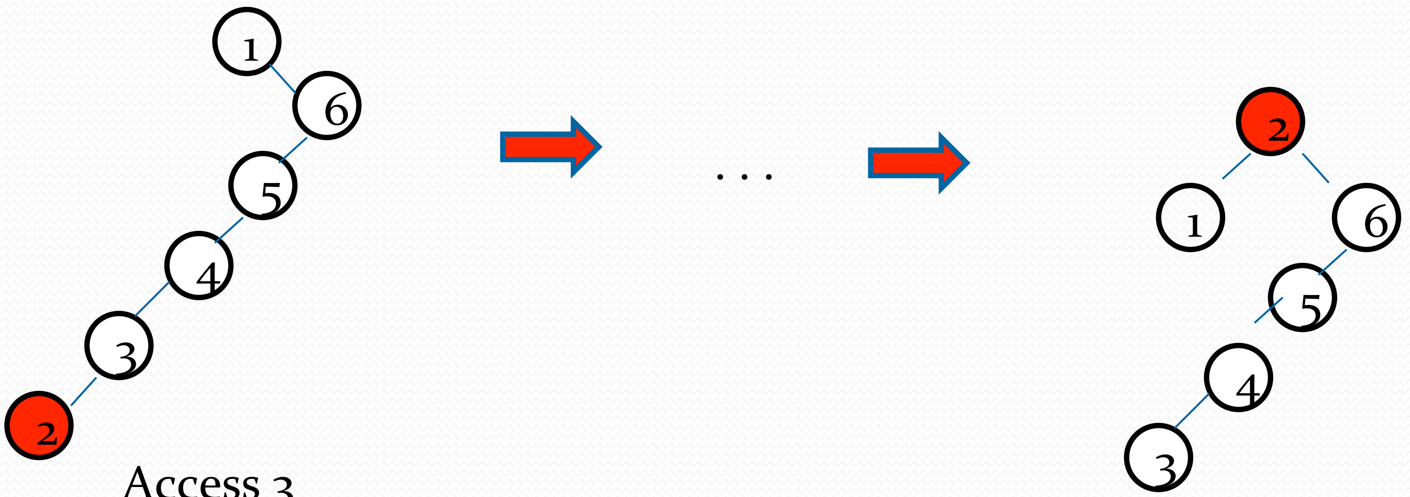
Access 1, 2, 3, 4, 5, 6

Access 1

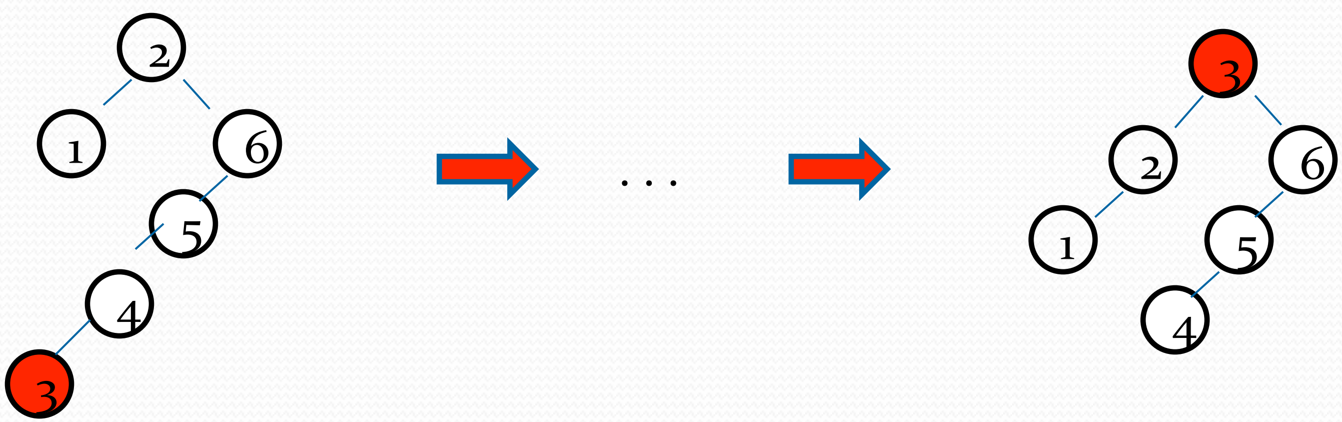


Access 1, 2, 3, 4, 5, 6

Access 2



Access 3



Splay Trees:

M operations requiring $\Omega(N)$ time

- Accessing the sequence 1,2,... is Ω (N^2) though...
- Run example:
  - Access 1 (time O(N)), then perform sequence of single rotations (time O(N))
  - Access 2 time O(N-1)...

Splaying – the correct way!

Rotate bottom-up on access, along access path

Let **X** a non-root node on access path at which we are rotating

if (parent of **X** is the root) { single rotation with root }

else {

Let **P** be parent of **X**, and **G** be the grandparent

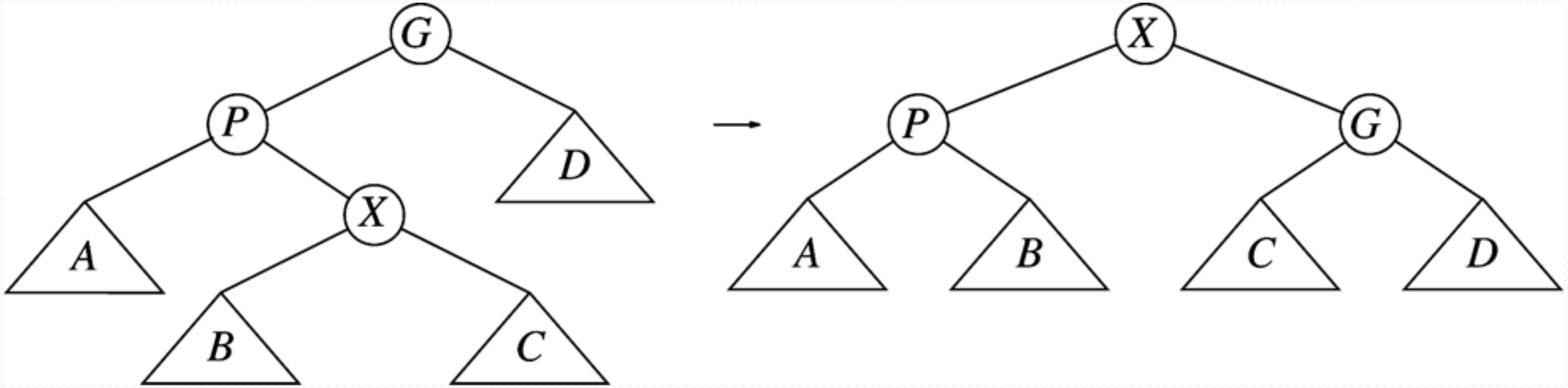
Two cases (plus symmetries) to consider:

- Case 1 (zig-zag): double rotation (see figure)
- Case2 (zig-zig): (see figure)

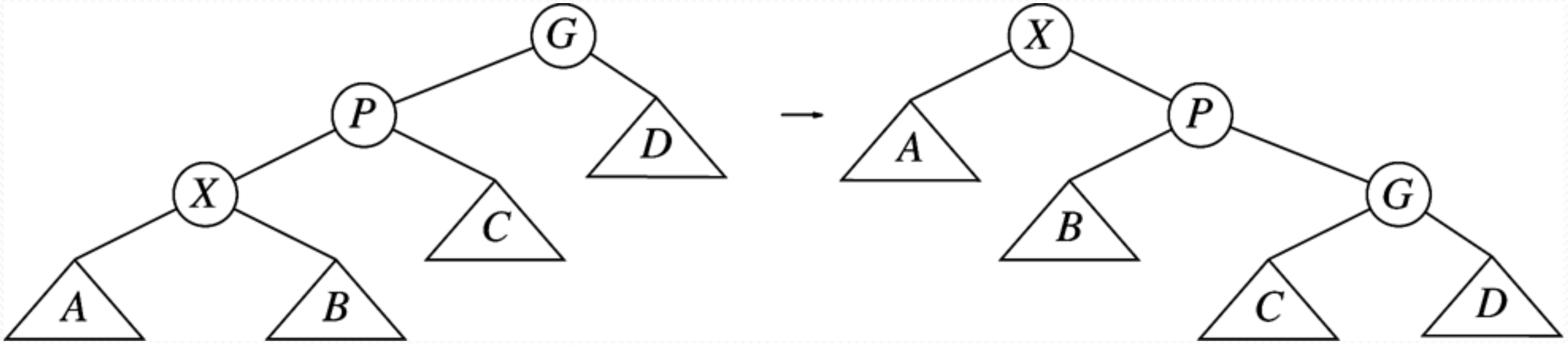
}

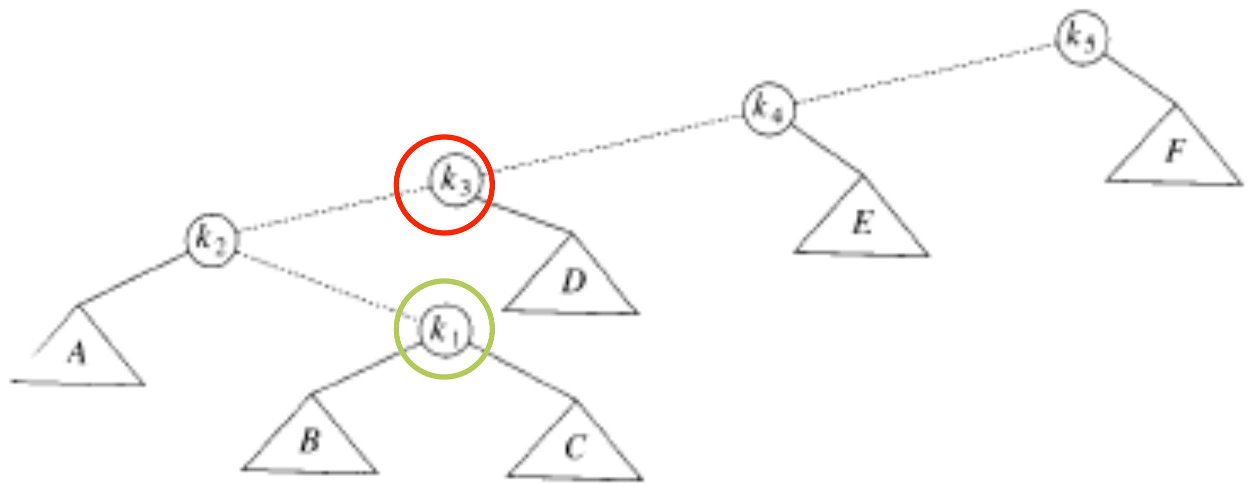
Applet: https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

Zig-zag

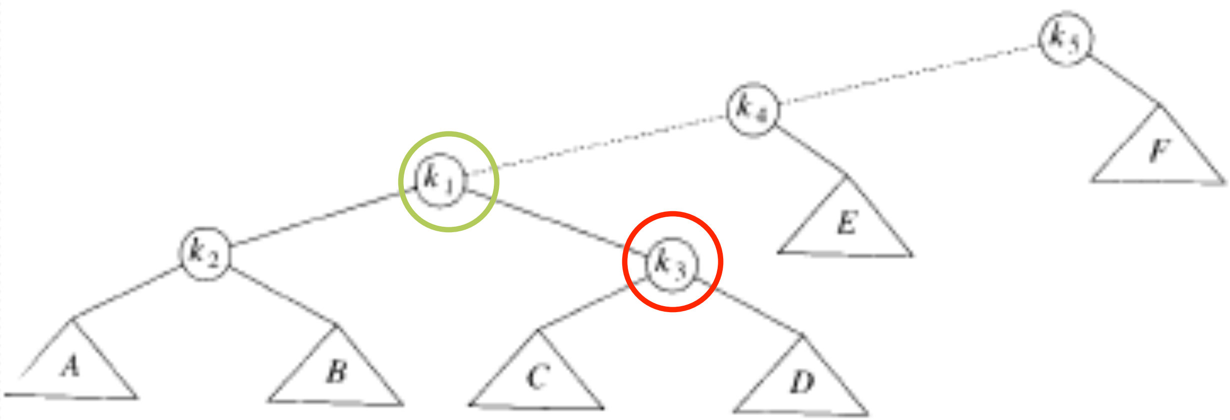


Zig-Zig

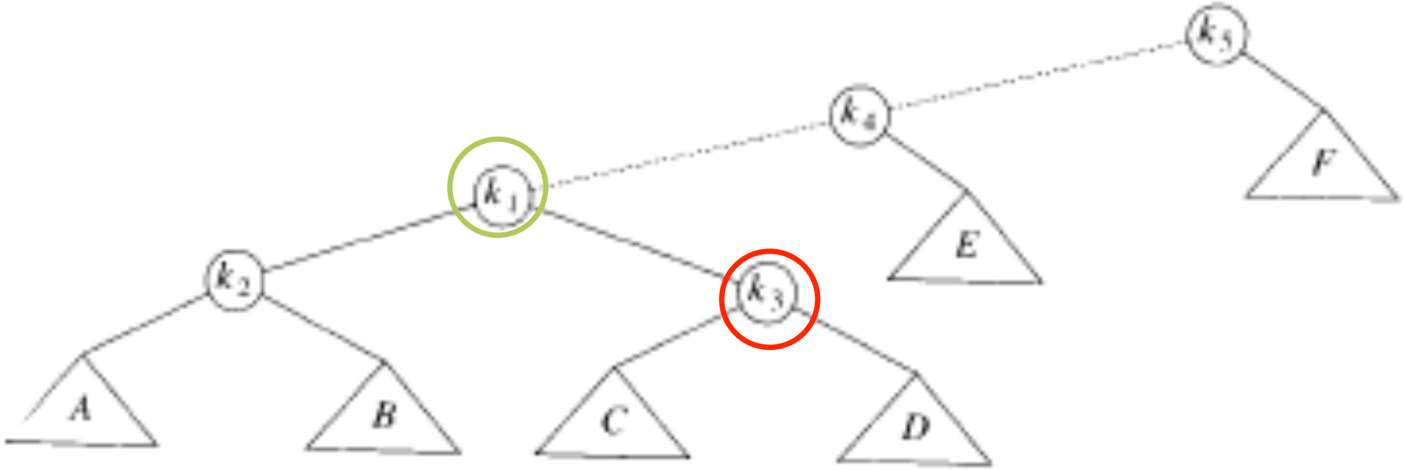




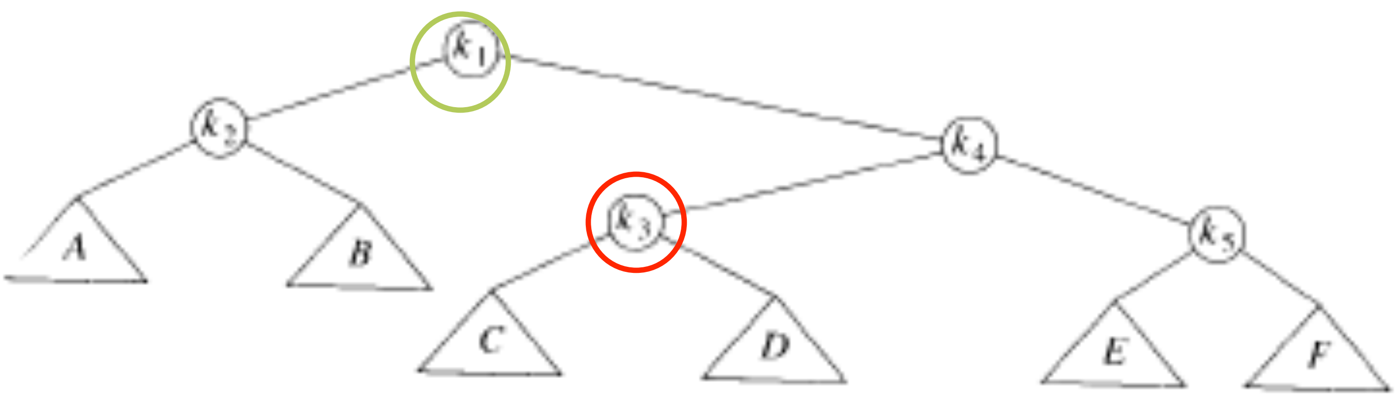
The first splay step is at k_1, and is clearly a *zig-zag*, so we perform a standard AVL double rotation using k_1, k_2, and k_3. The resulting tree follows.

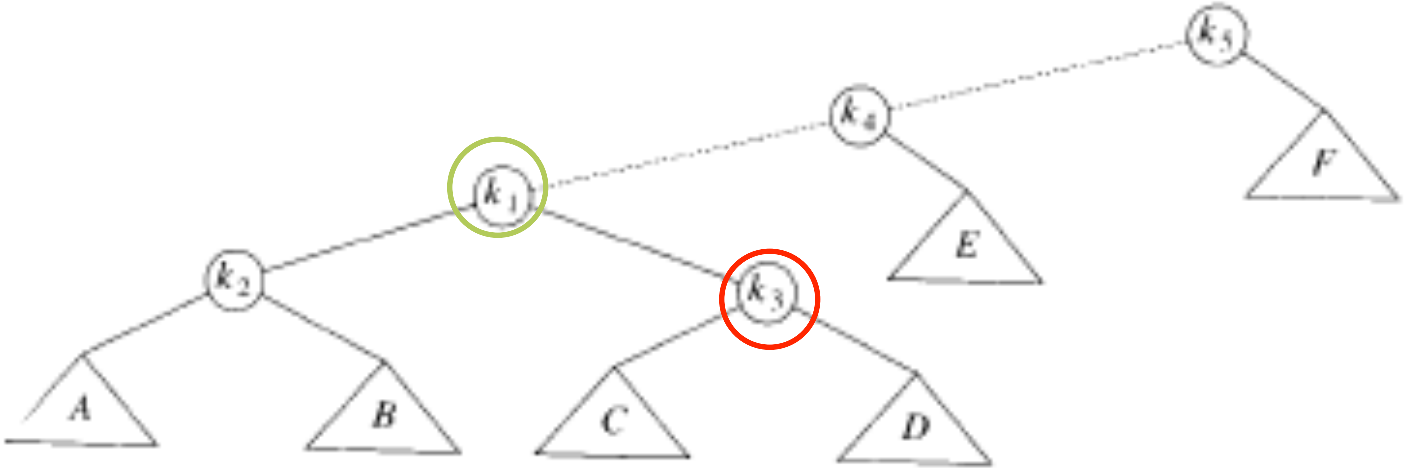


The next splay step at k_1 is a zig-zig

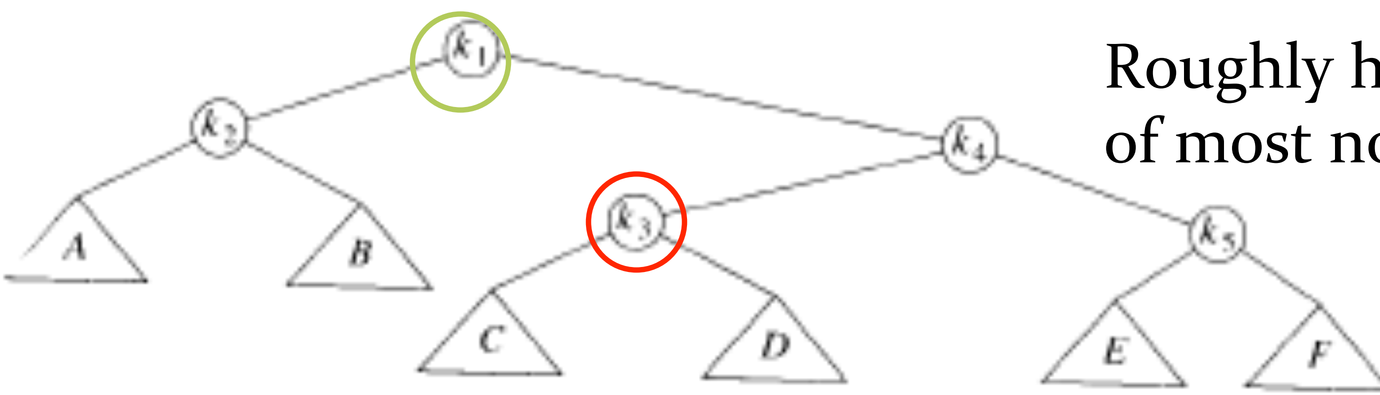


The next splay step at k_1 is a zig-zig, so we do the zig-zig rotation with k_1, k_4, and k_5, obtaining the final tree.



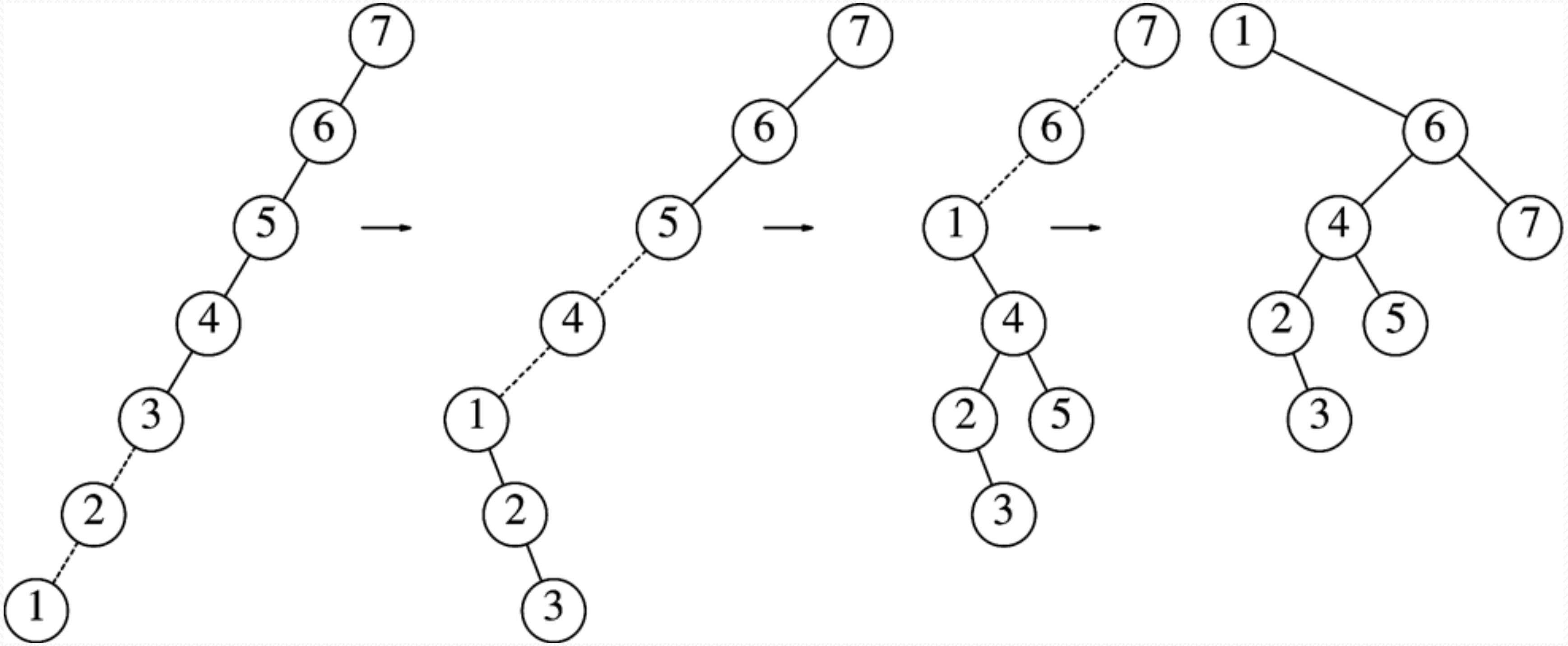


The next splay step at k_1 is a zig-zig, so we do the zig-zig rotation with k_1, k_4, and k_5, obtaining the final tree.

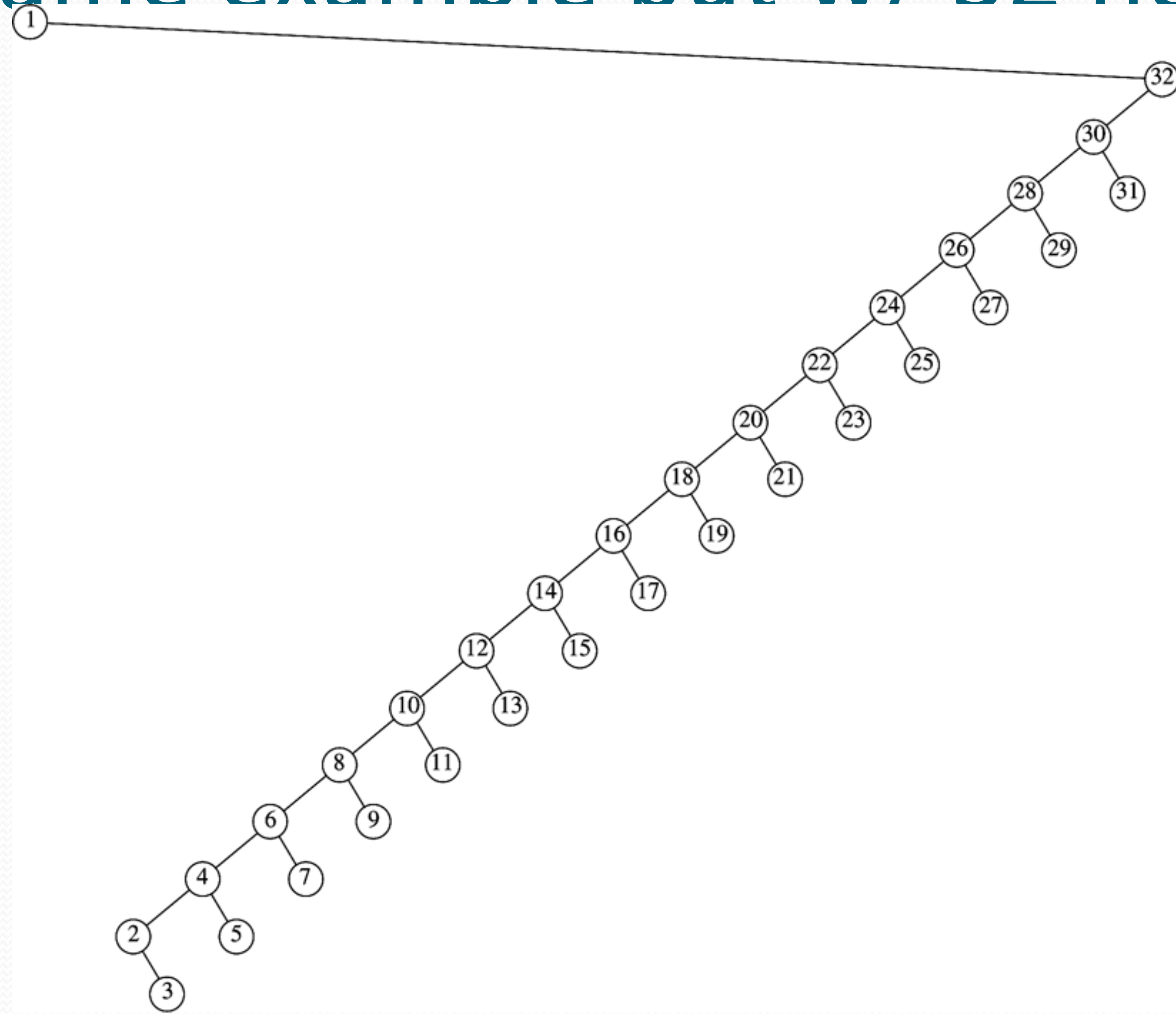


Roughly halving depth of most nodes…

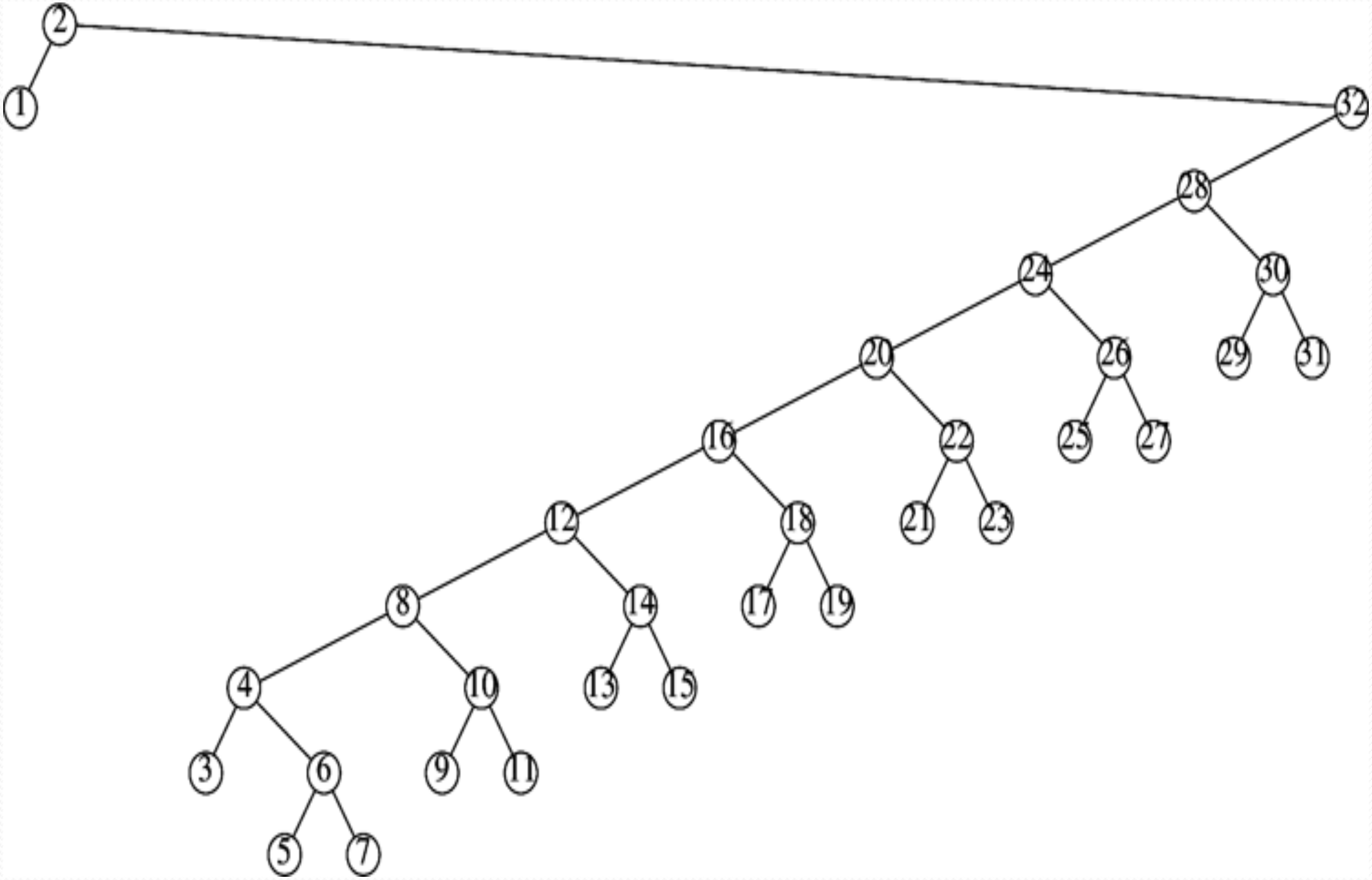
Example
Insert 1,…,7 -> then access 1



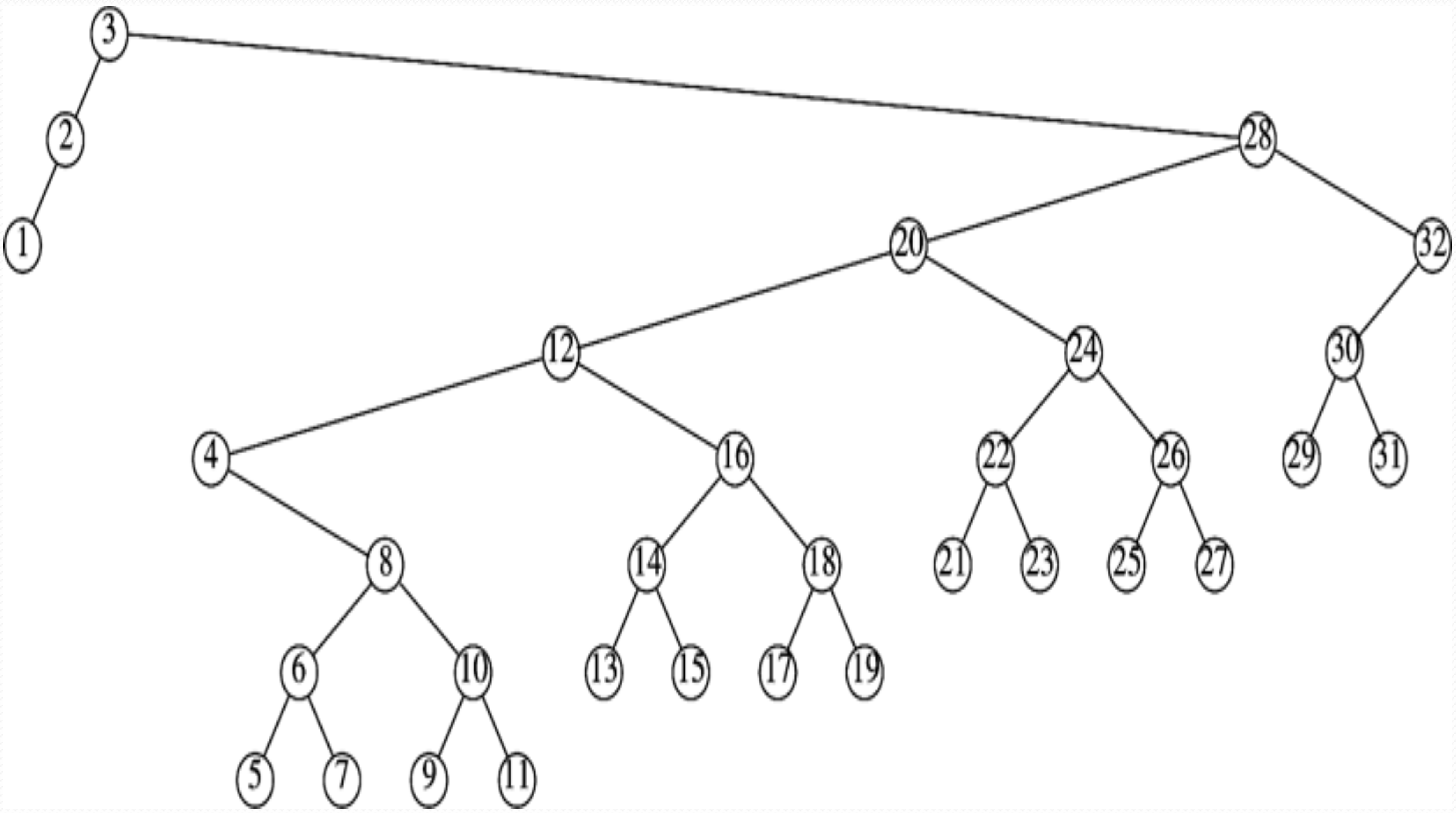
Same example but w/ 32 nodes



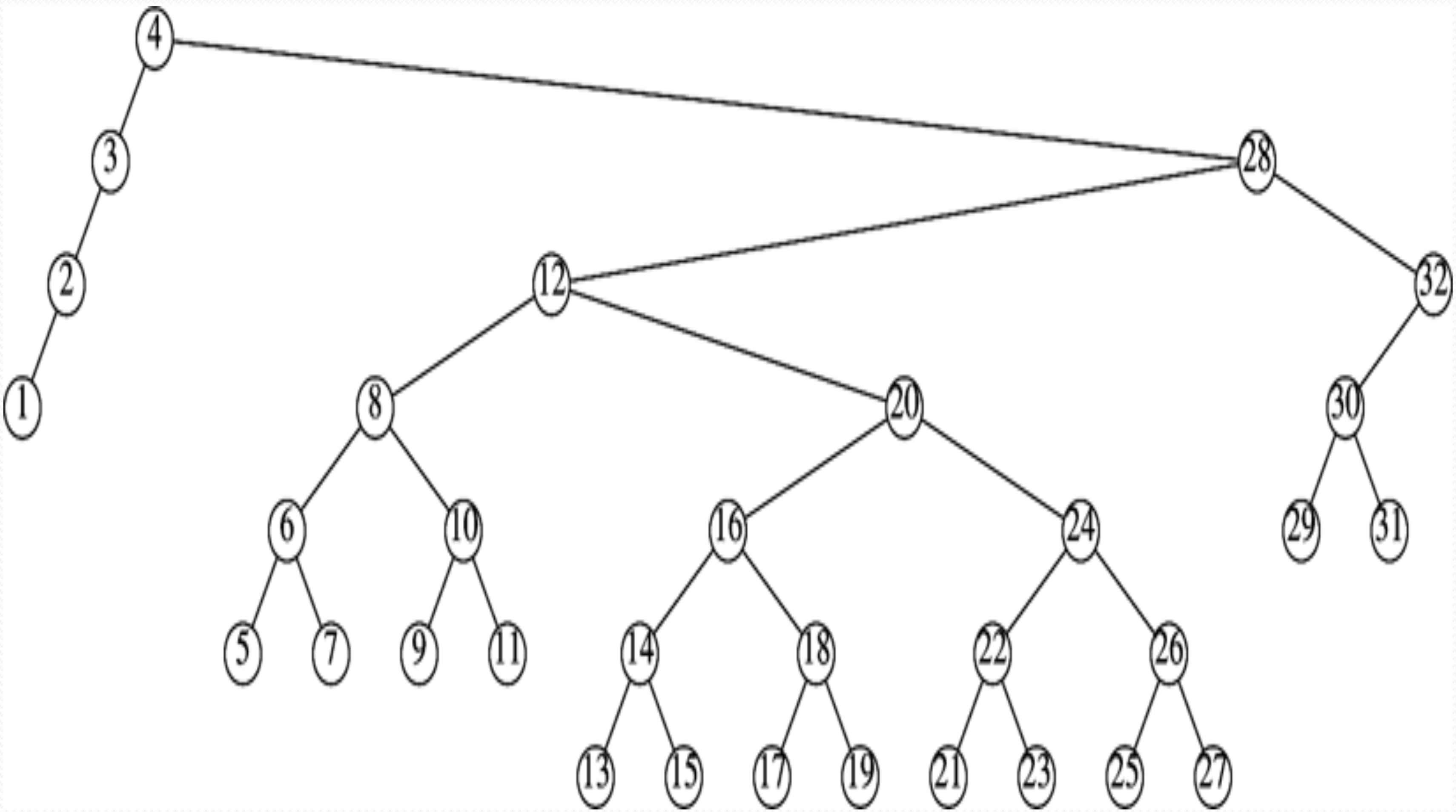
Same example but w/ 32 nodes



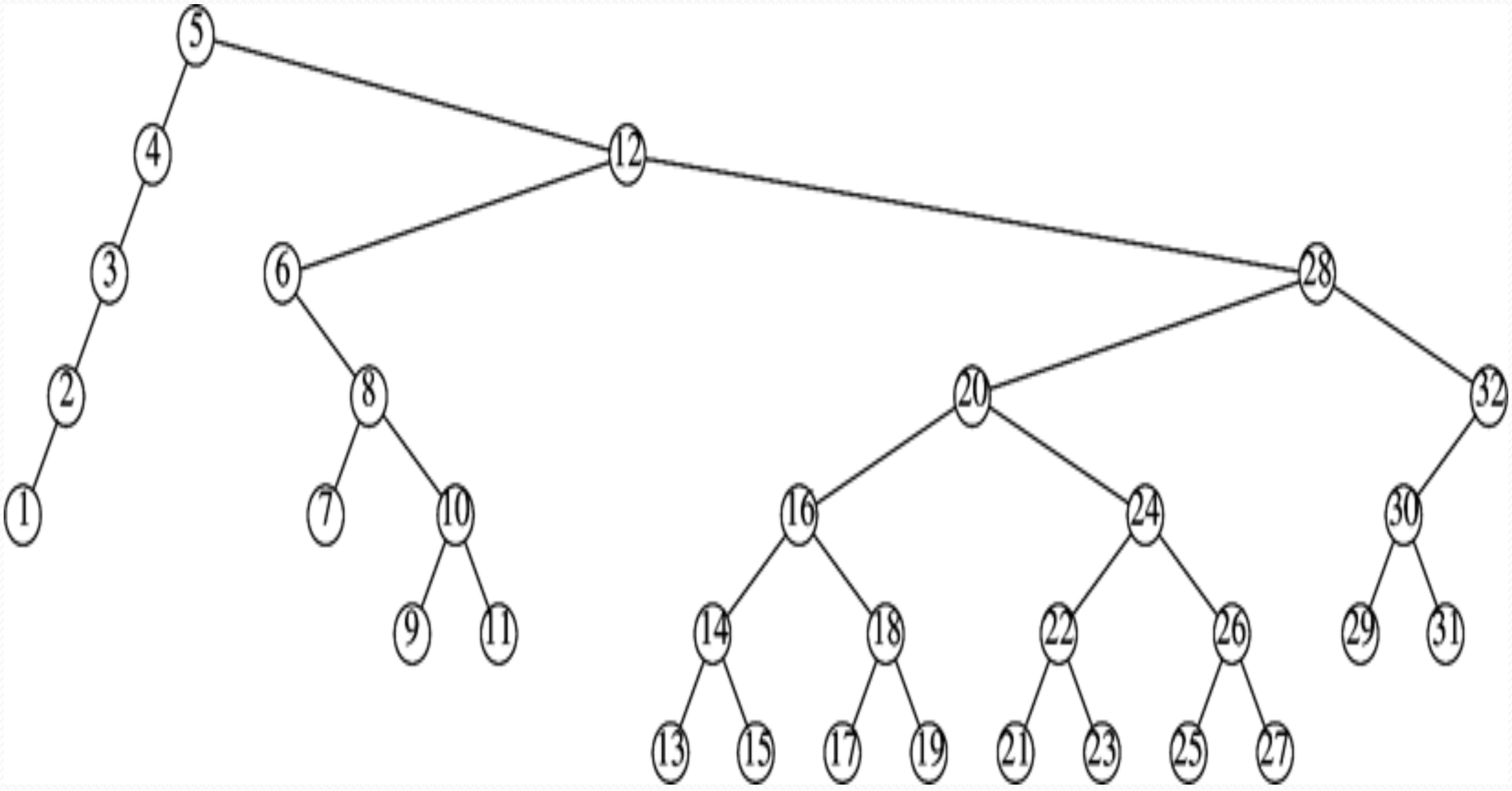
Same example but w/ 32 nodes



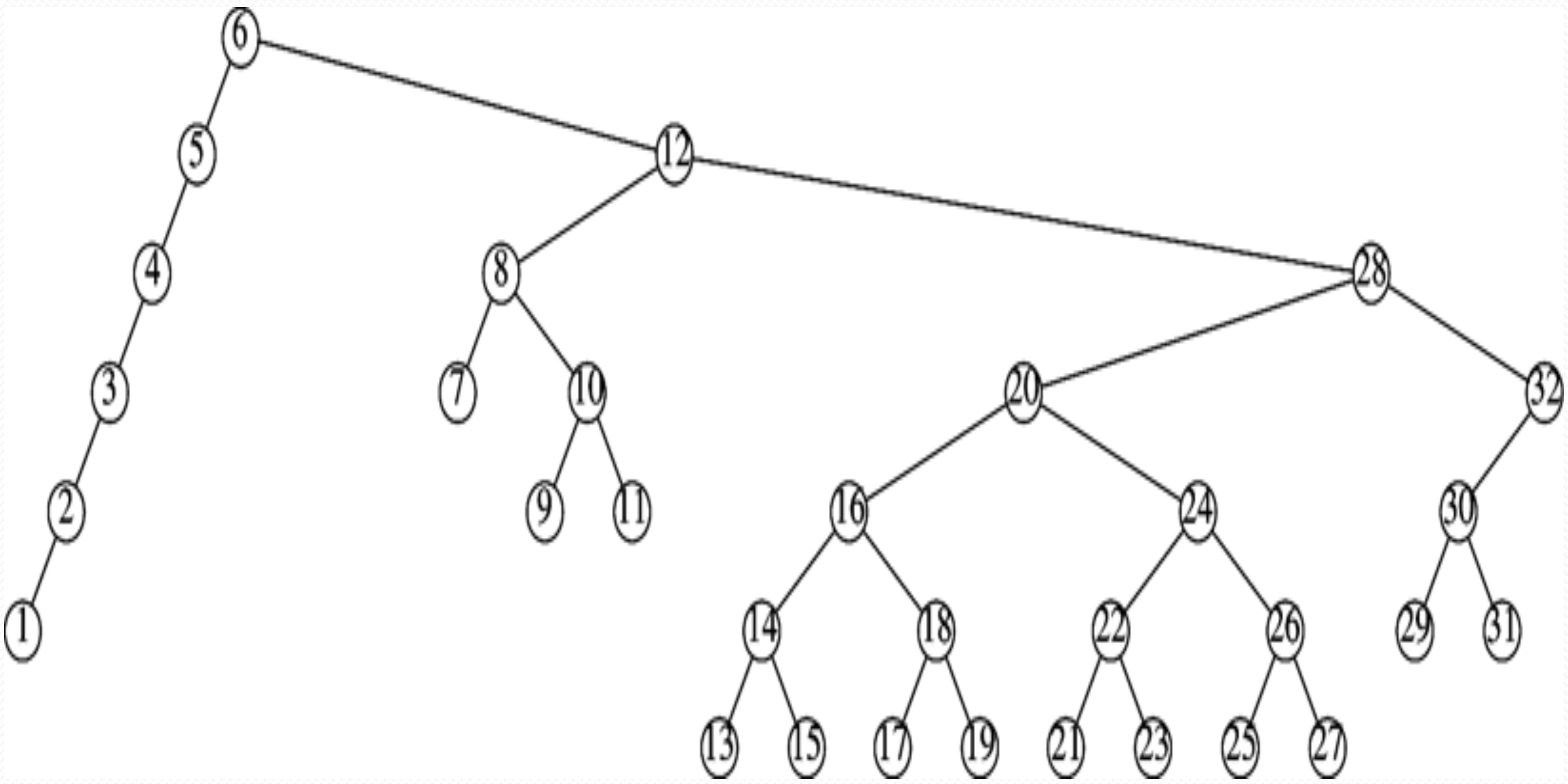
Same example but w/ 32 nodes



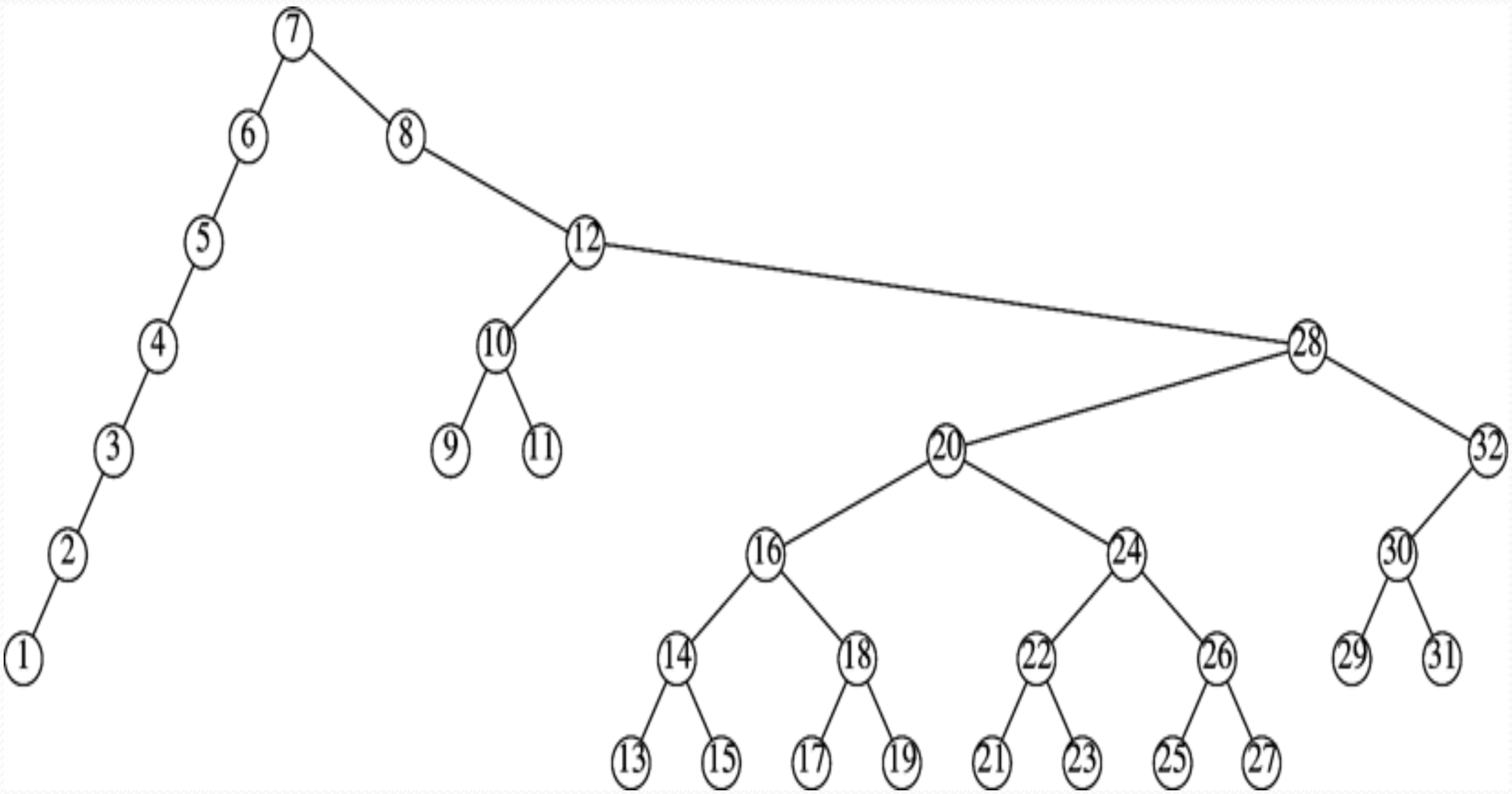
Same example but w/ 32 nodes



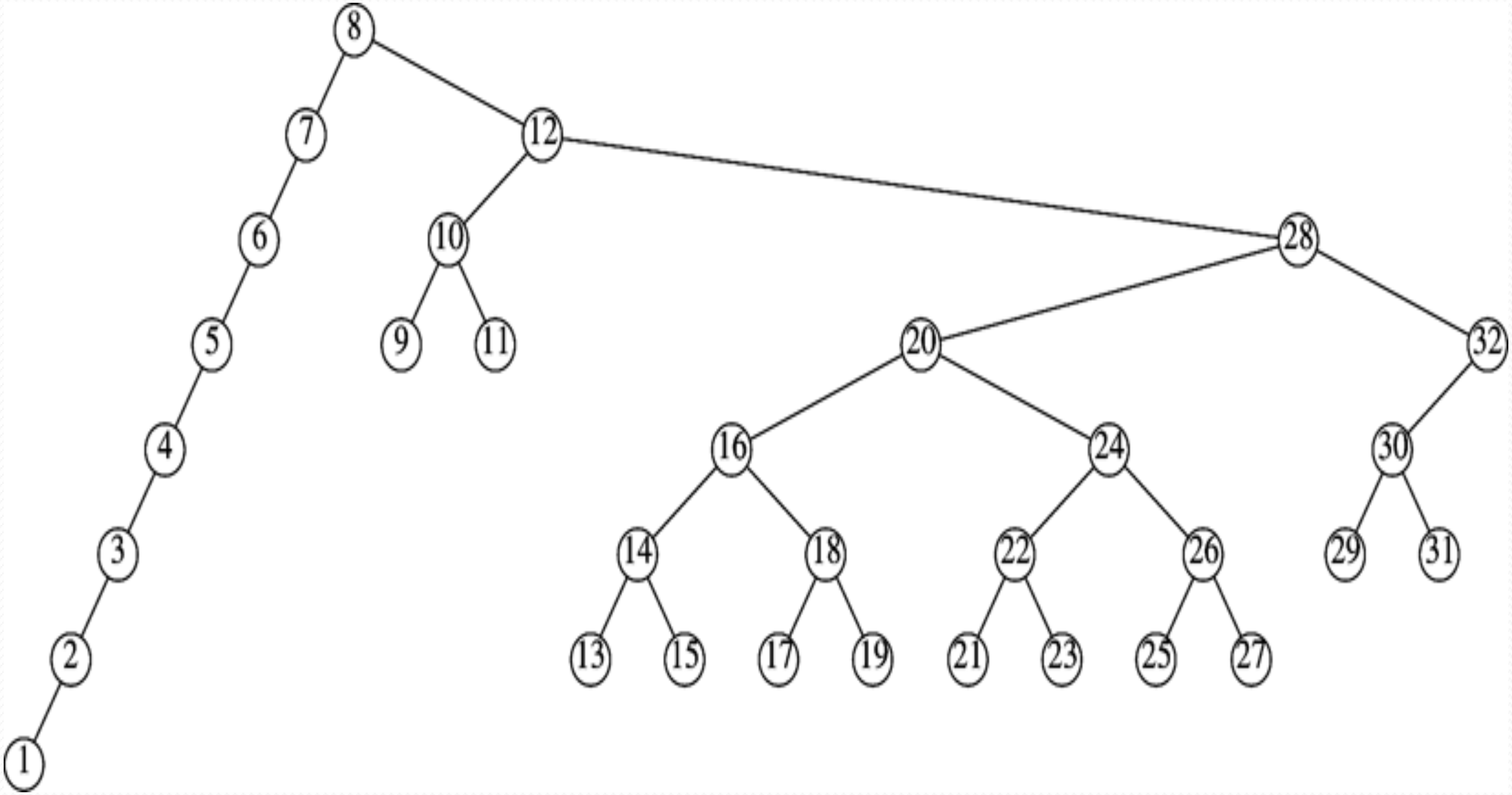
Same example but w/ 32 nodes



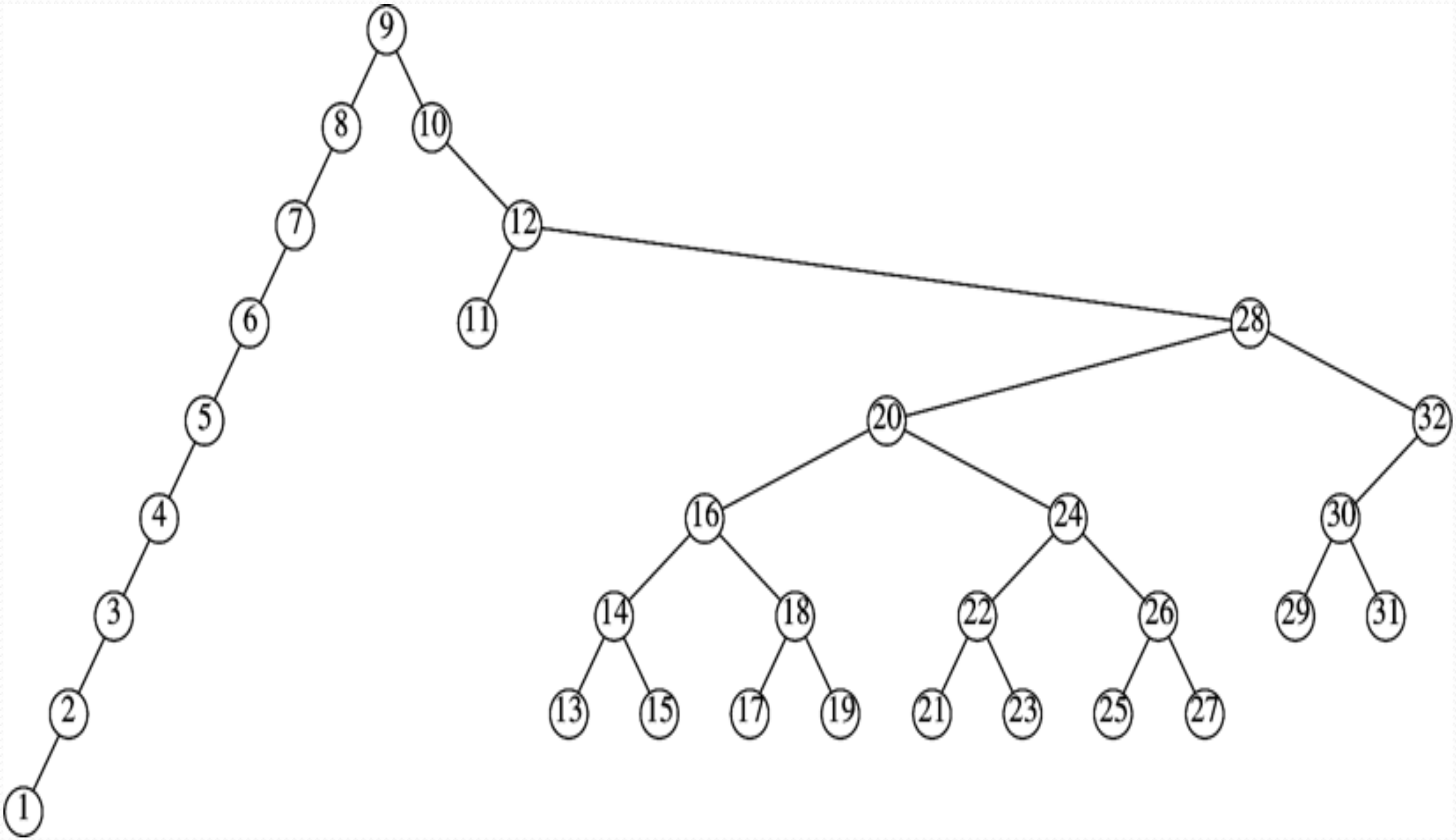
Same example but w/ 32 nodes



Same example but w/ 32 nodes



Same example but w/ 32 nodes



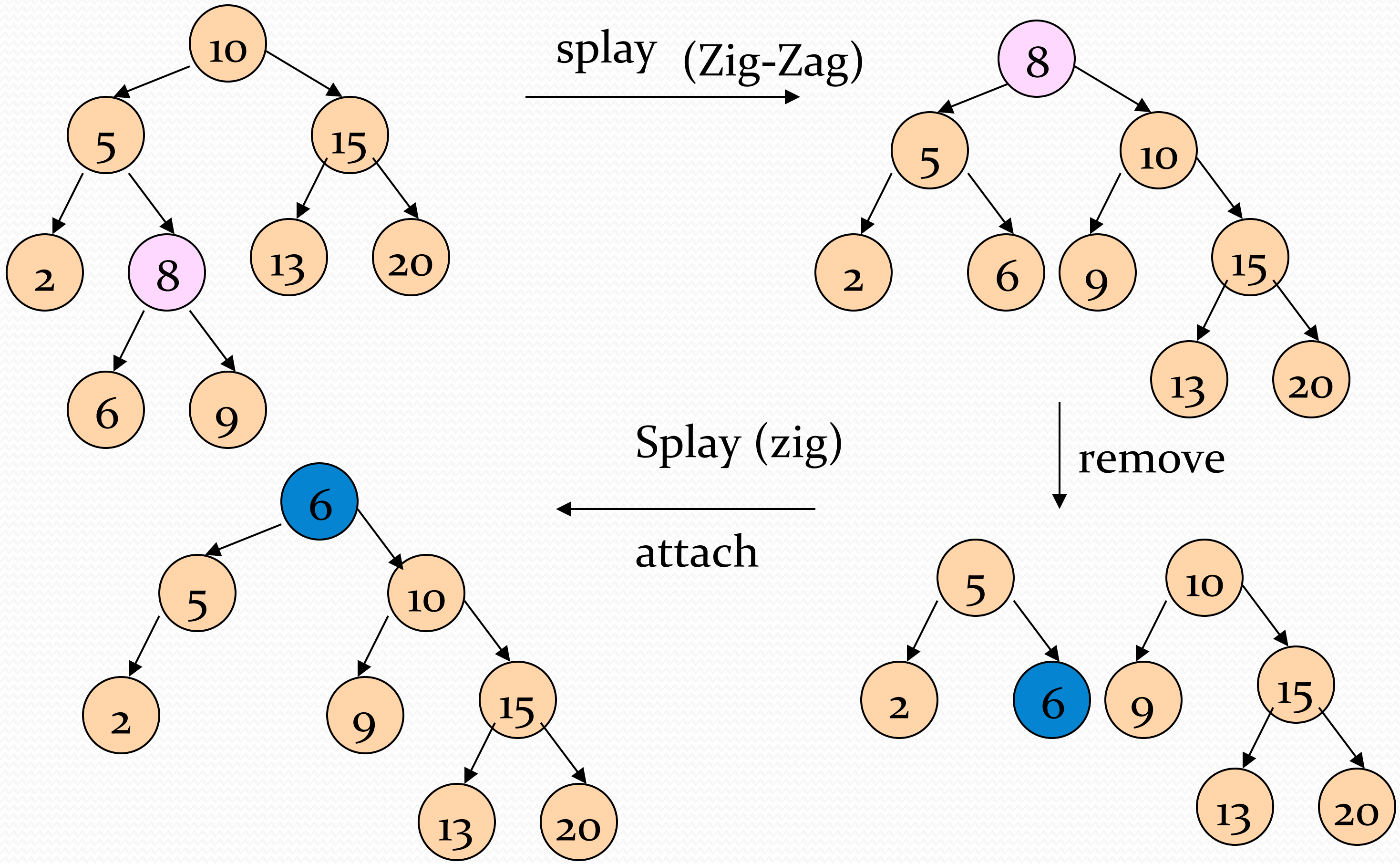
Analysis

- Hard (see later chapter)
- In the example: Access of 2 -> N/4 of root, …, up-to logN of root
- Fundamental properties:
  - When access paths are long, longer search time, but rotations good for future operations
  - When access is cheap, rotations not as good.
  - It can be proved that time is O(logN) per operation (amortized)
- Deletion?
- Much simpler to program with fewer cases
- No need to store balance information

Deletion

- Access, bring node to top
- Delete creating two subtrees
- Access largest element in left tree, bring it to root with no right child
- Patch in right tree as right child

Example Deletion of 8



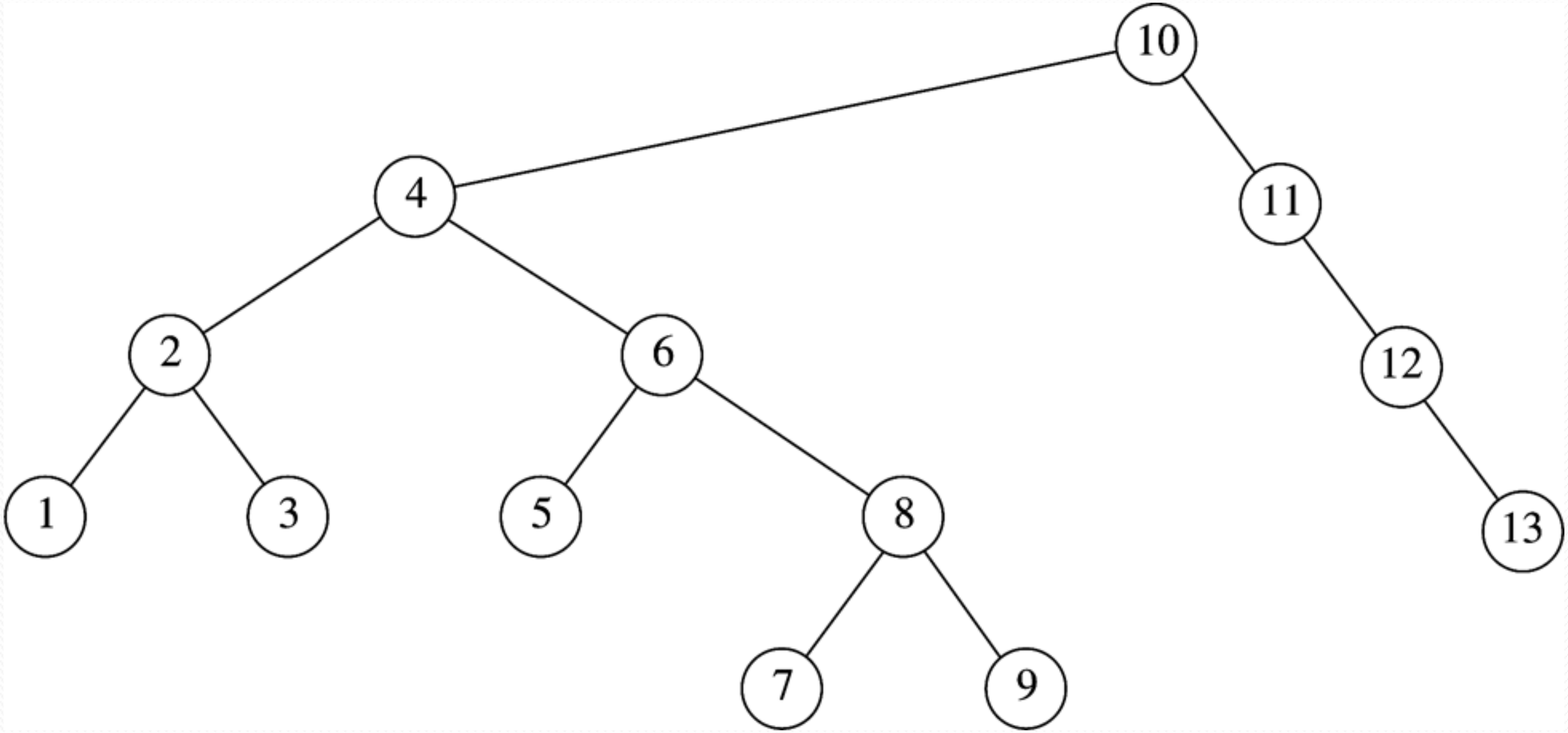
From Univ. of Washington

Exercise from book

- 4.26 (Double rotation implementation without two single rotations)
- 4.27, 4.28

For exercises

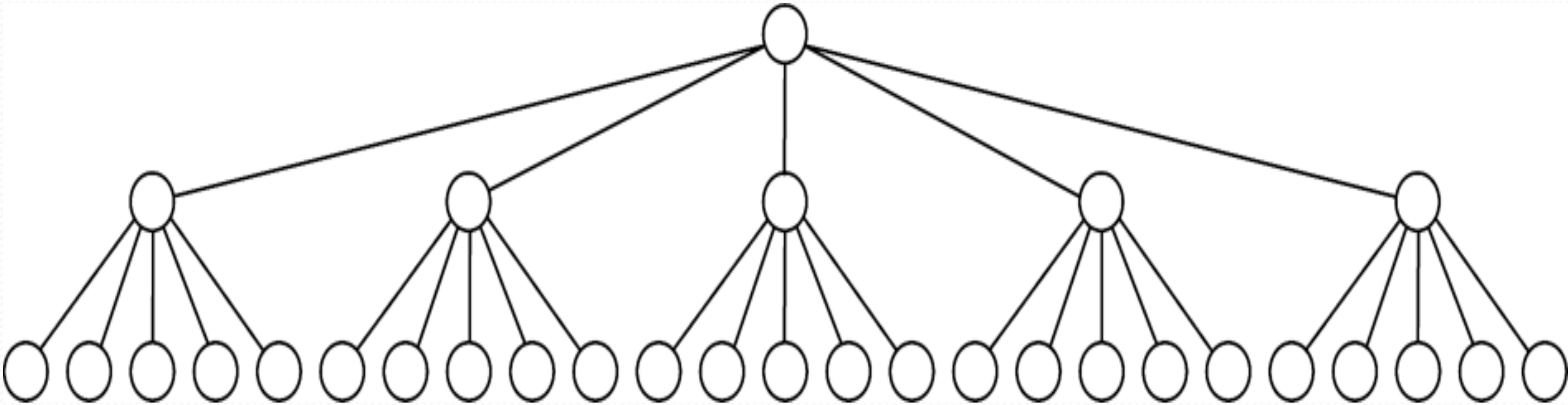
Find: 3, 9, 1, 5
Delete: 6



B-trees

- Scenario: Tree is large and can't fit into memory
- Fact: Disk I/O is much slower than machine instruction (one disk access ~ 4M instructions)
- Assume that we have 10M records, each of 256 bytes, and key is 32 bytes
- Solution?
  - AVL? (worst case is **1.44 logN** , but each operation is costly)
  - We need even smaller trees
    - **M-ary tree has height $\log_M(N)$**

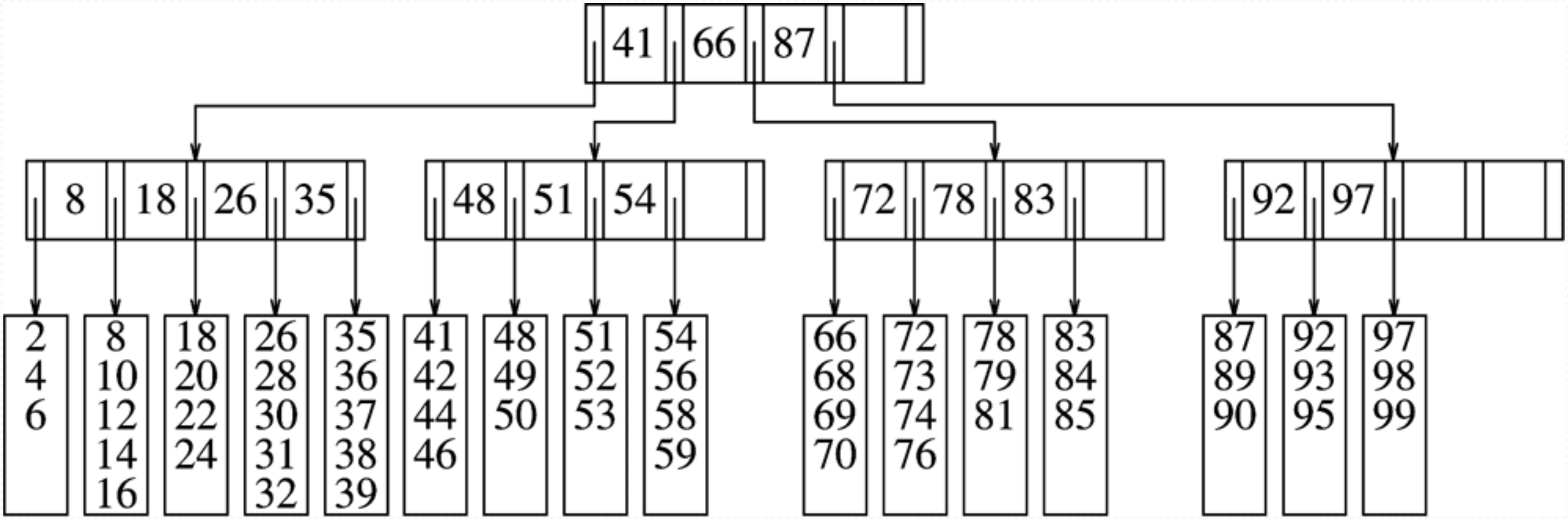
5-ary tree of 31 nodes



Definition of B-tree

- M-ary tree such that:
  - Data is stored at leaves
  - The nonleaf nodes store up to M-1 keys to guide searching. <u>Key i represents smallest key in subtree i+1</u>
  - Root is either a leaf or has between two and M children
  - All nonleaf nodes (except root) have between M/2 and M children (up to half full).
  - All leaves are at same depth and have between L/2 and L data items for some L (up to half full).
- M, and L are determined based on disk block (one access should load a whole node).

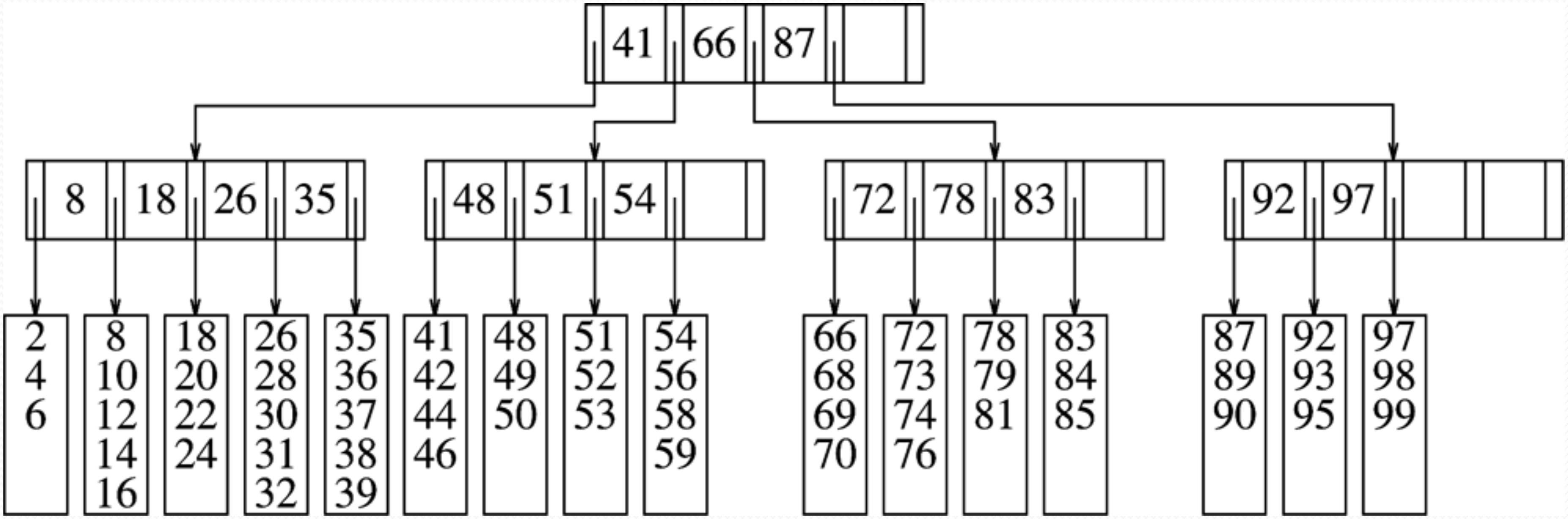
Example B-tree of order 5 (M=5)



Example

- Block size is 8,192 bytes
- Key is 32 bytes
- Internal nodes hold M-1 keys => 32(M-1) bytes plus M branches (4 bytes per branch) => total is 36M-32 bytes.
- Since 8,192 <= 36M-32 => M = 228
- Data record is 256 bytes => 32 records on a block => L=32
- So each leaf should hold between 16 to 32 records
- Each internal node should have at least 114 branches
- 10 million records => 625,000 leafs (10 million/16). In worst case leaves on level 4 (why?)
  - On average number of accesses is **$\log_{M/2}$ (N)**
  - Root and first levels could also be cached in memory…

Example B-tree of order 5

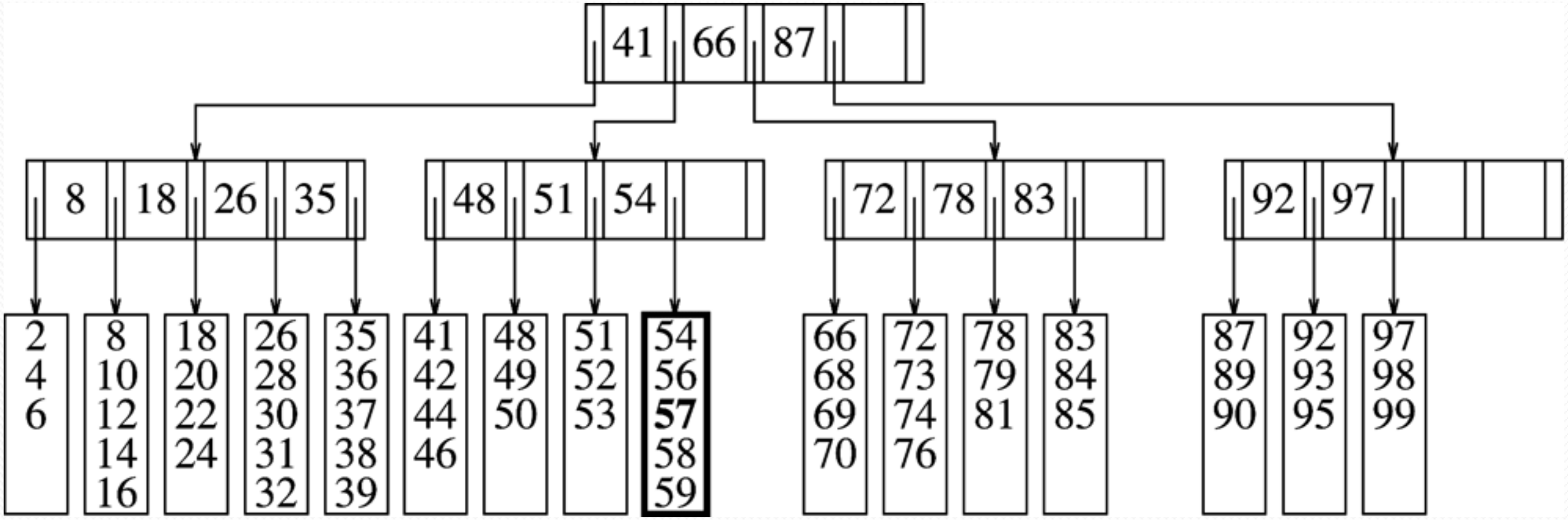


Insertion

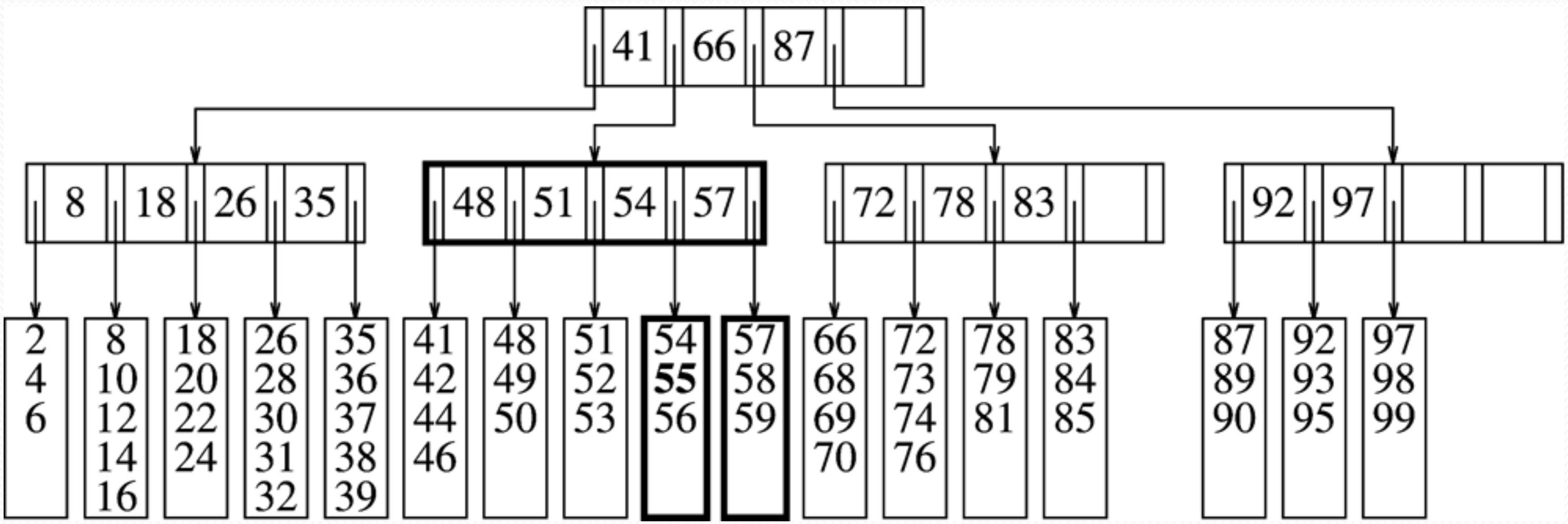
To insert:

- Put it in the appropriate leaf.
- If the leaf is full, break it in two, adding a child to the parent.
- If this puts the parent over the limit, split upwards recursively.
- If you need to split the root, add a new one with two children. This is the only way you add depth.

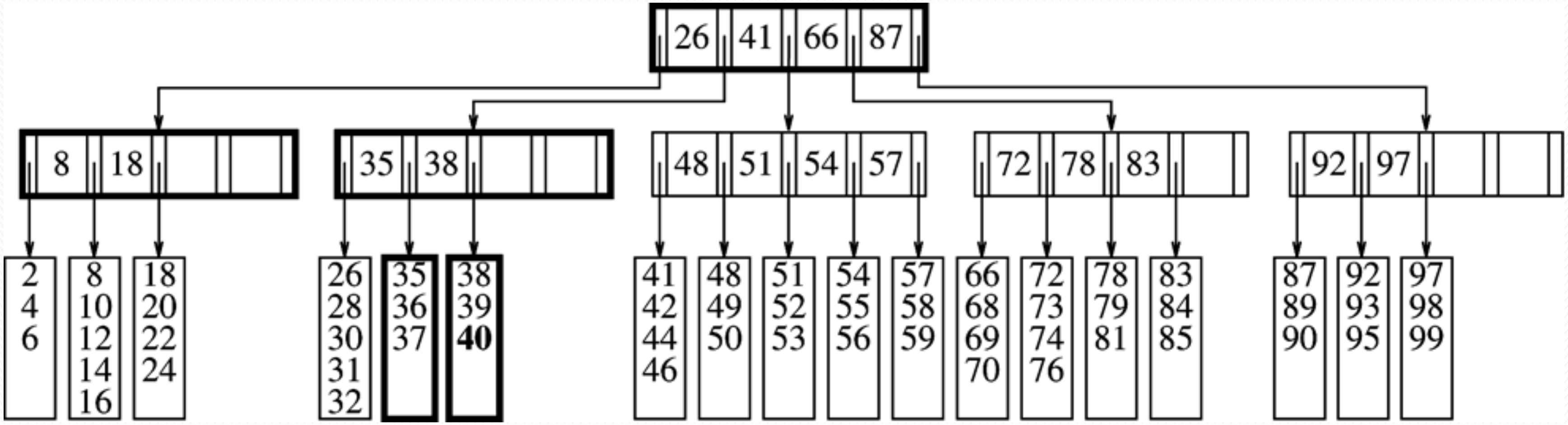
Example of insertion (57)



Example of insertion (55)



Example of insertion (40)



deletion

Delete from appropriate leaf.

- If the leaf is below its minimum, adopt from a neighbor if possible.
- If that's not possible, you can merge with the neighbor. This causes the parent to lose a branch and you continue upward recursively.

Example of deletion (99)