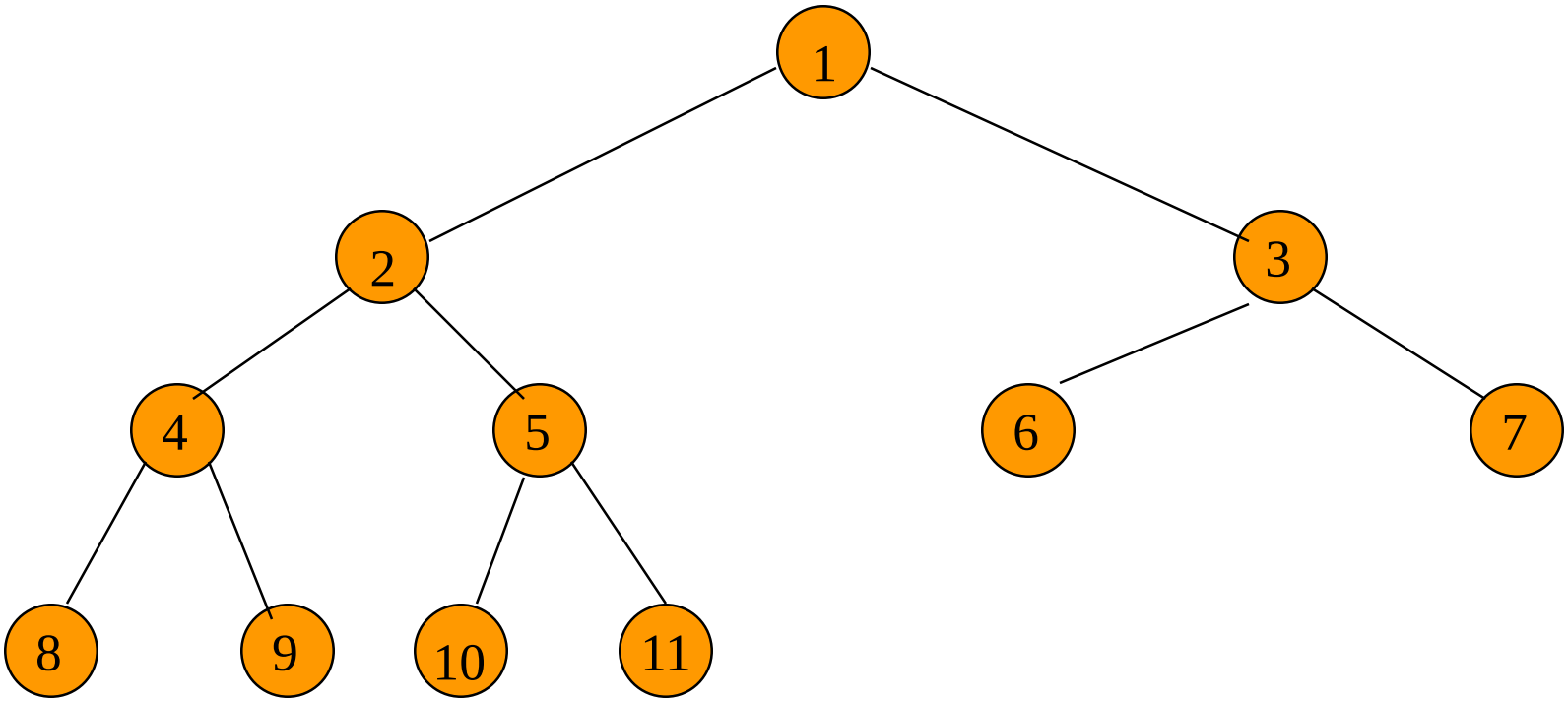
Chapter Seven

Several Extensions and Applications with Trees

Section 7-1

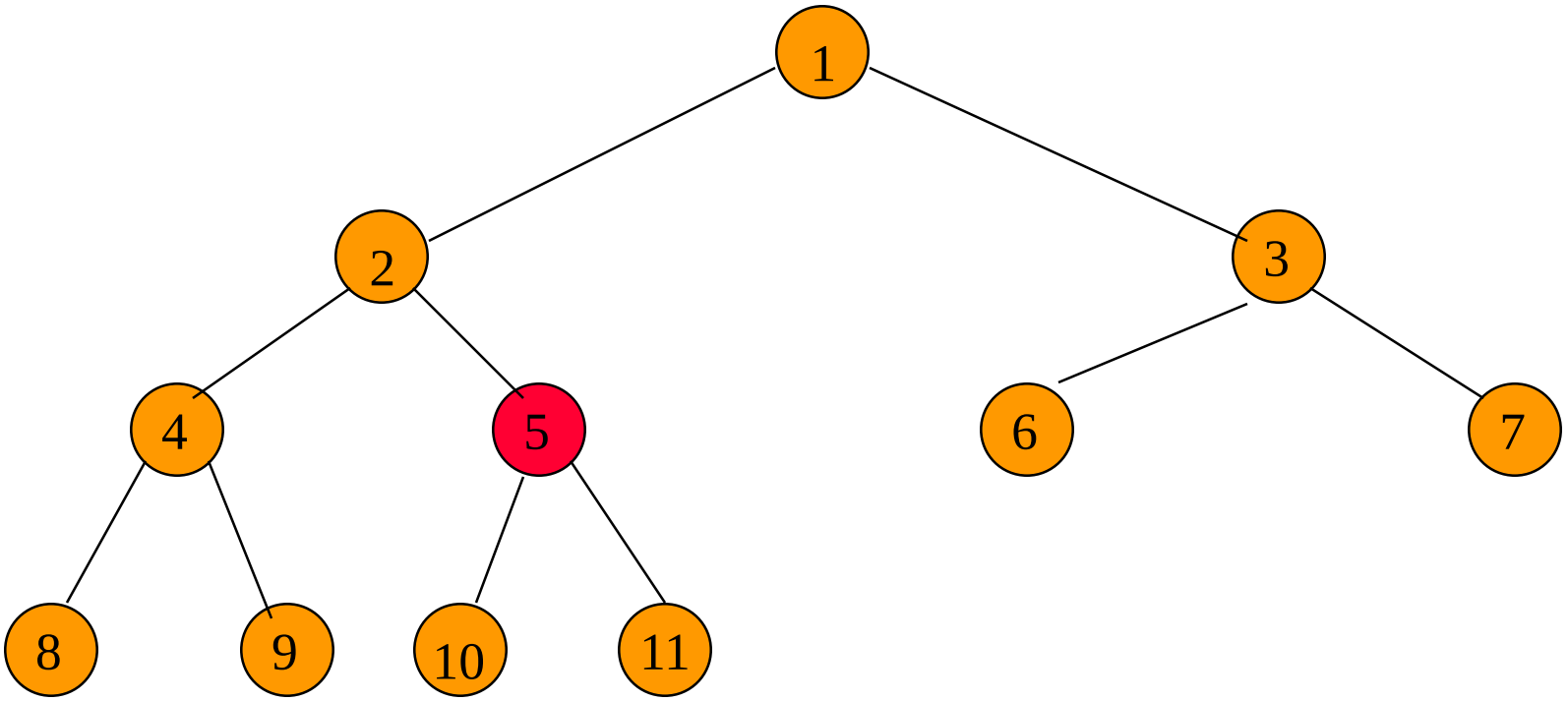
An Extension on Heaps

Initializing A Max Heap



input array = [-, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

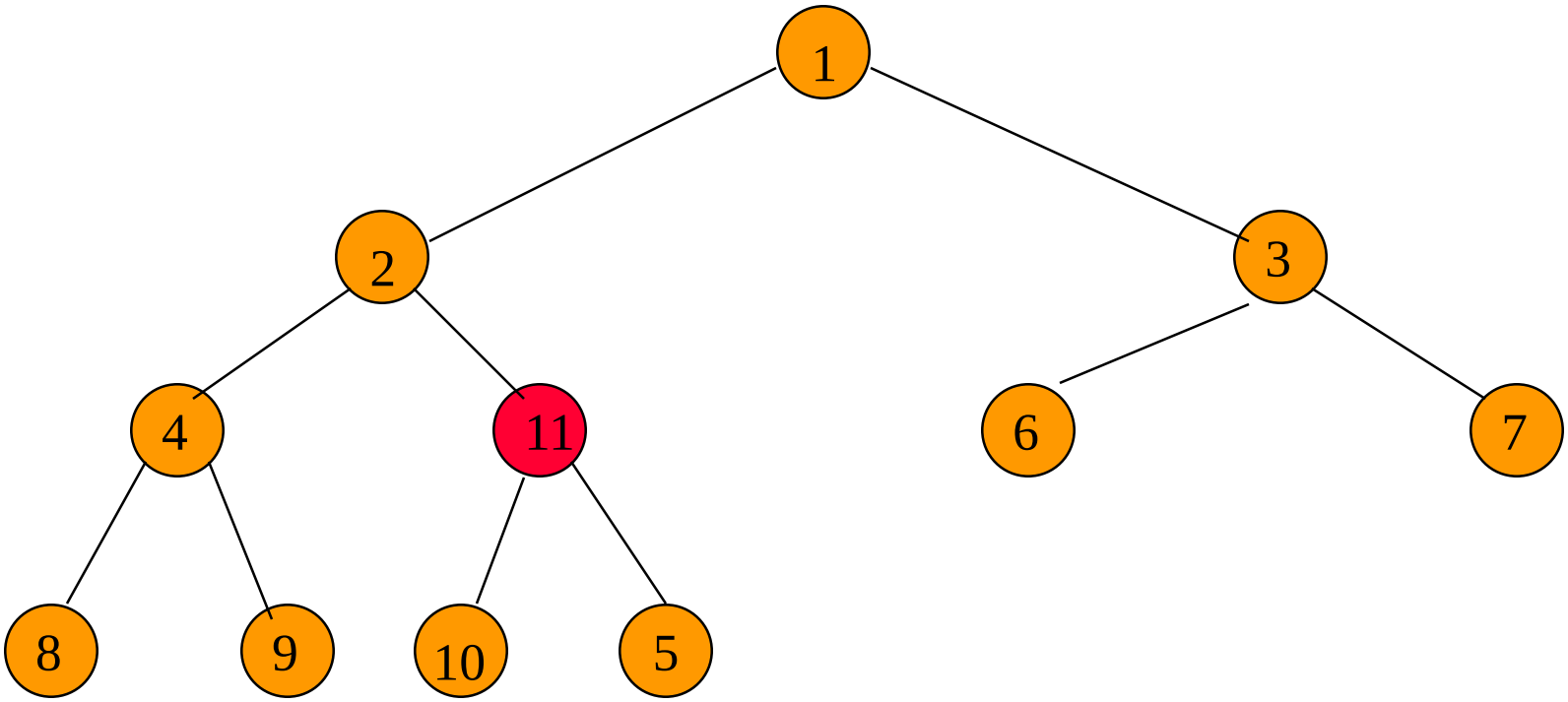
Initializing A Max Heap



Start at rightmost array position that has a child.

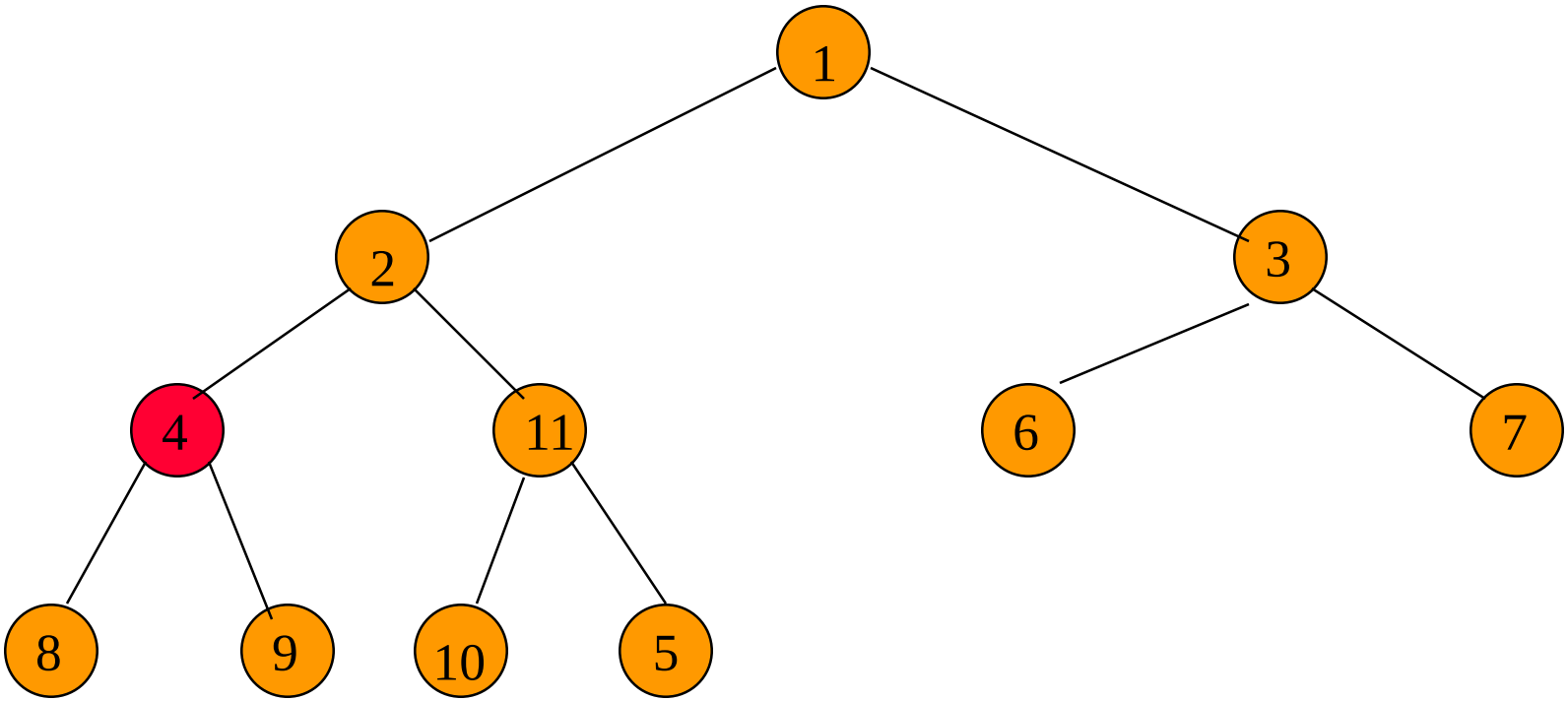
Index is n/2.

Initializing A Max Heap

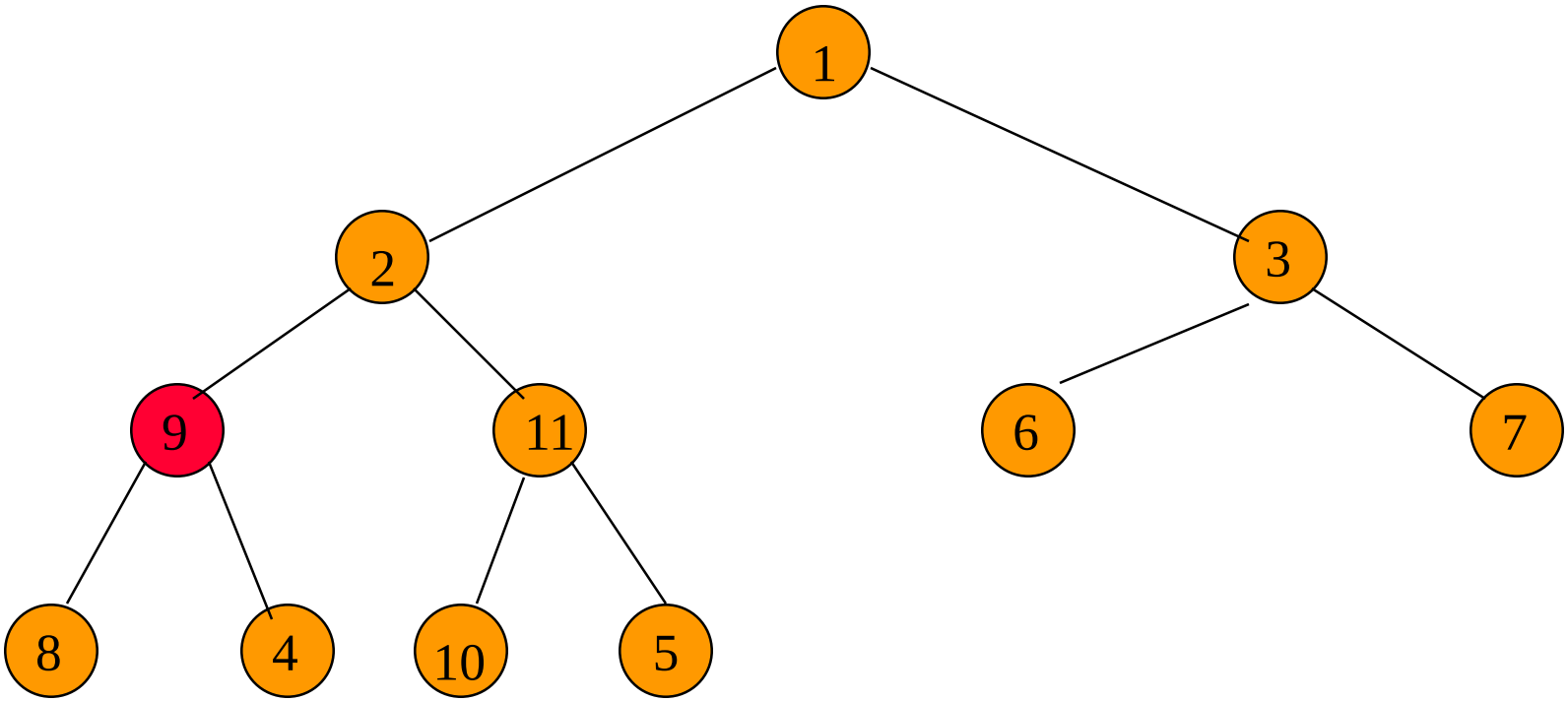


Move to next lower array position.

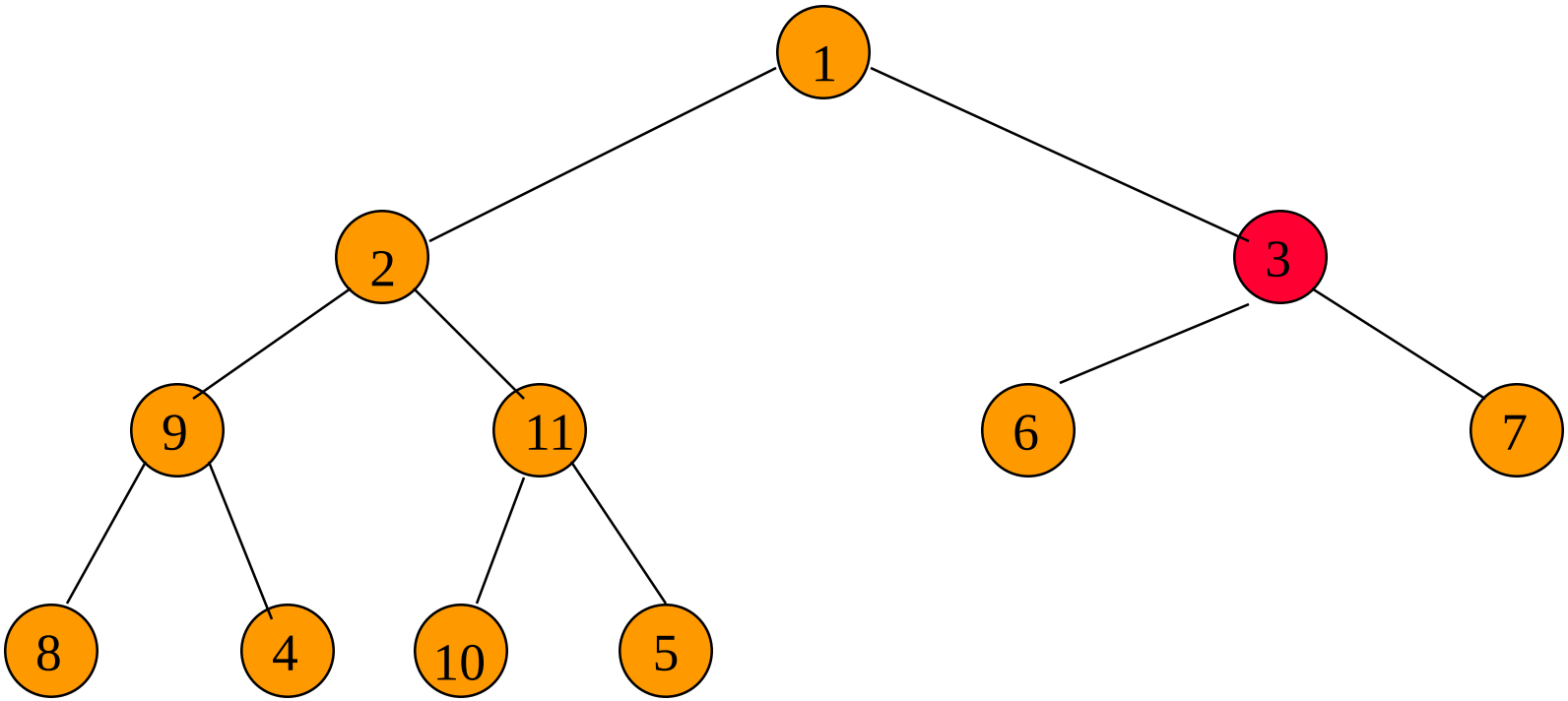
Initializing A Max Heap



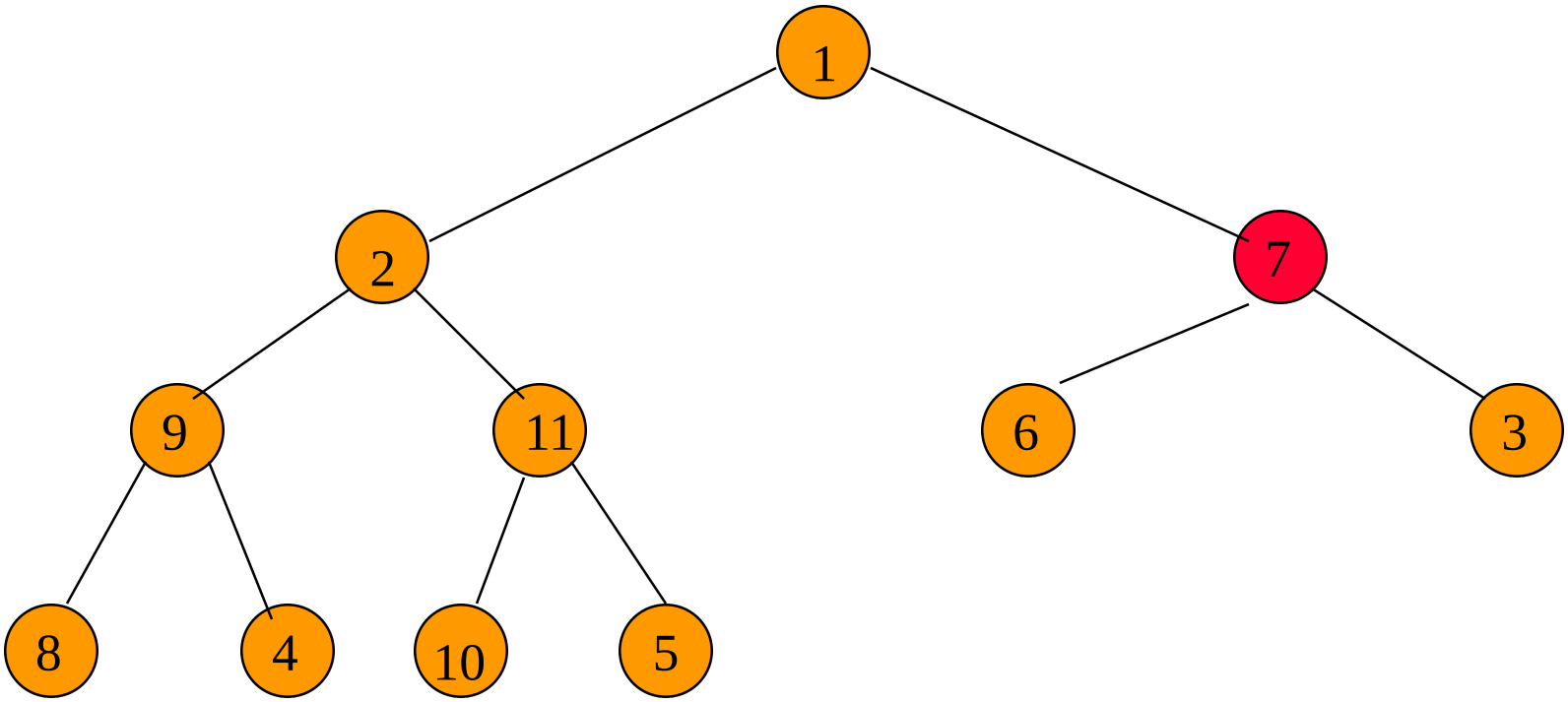
Initializing A Max Heap



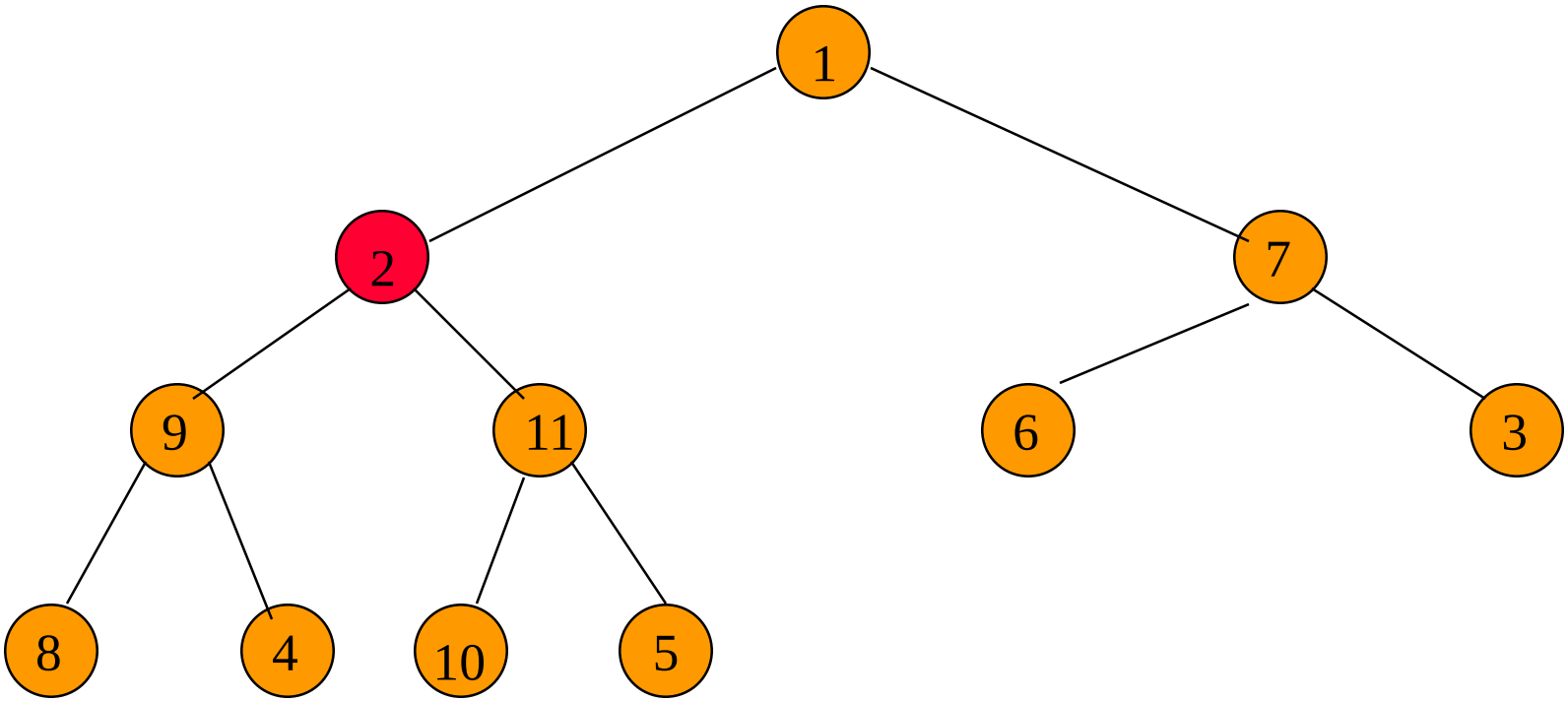
Initializing A Max Heap



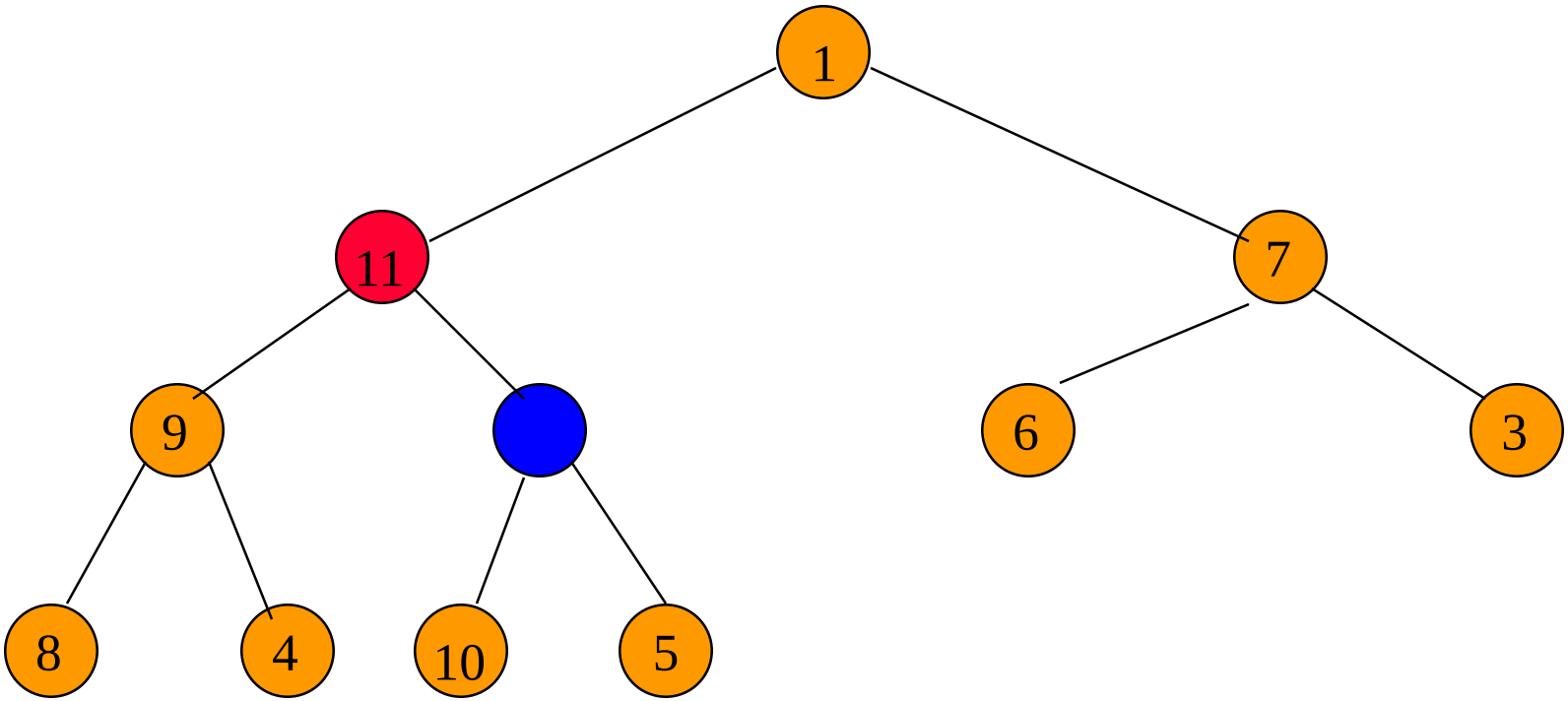
Initializing A Max Heap



Initializing A Max Heap

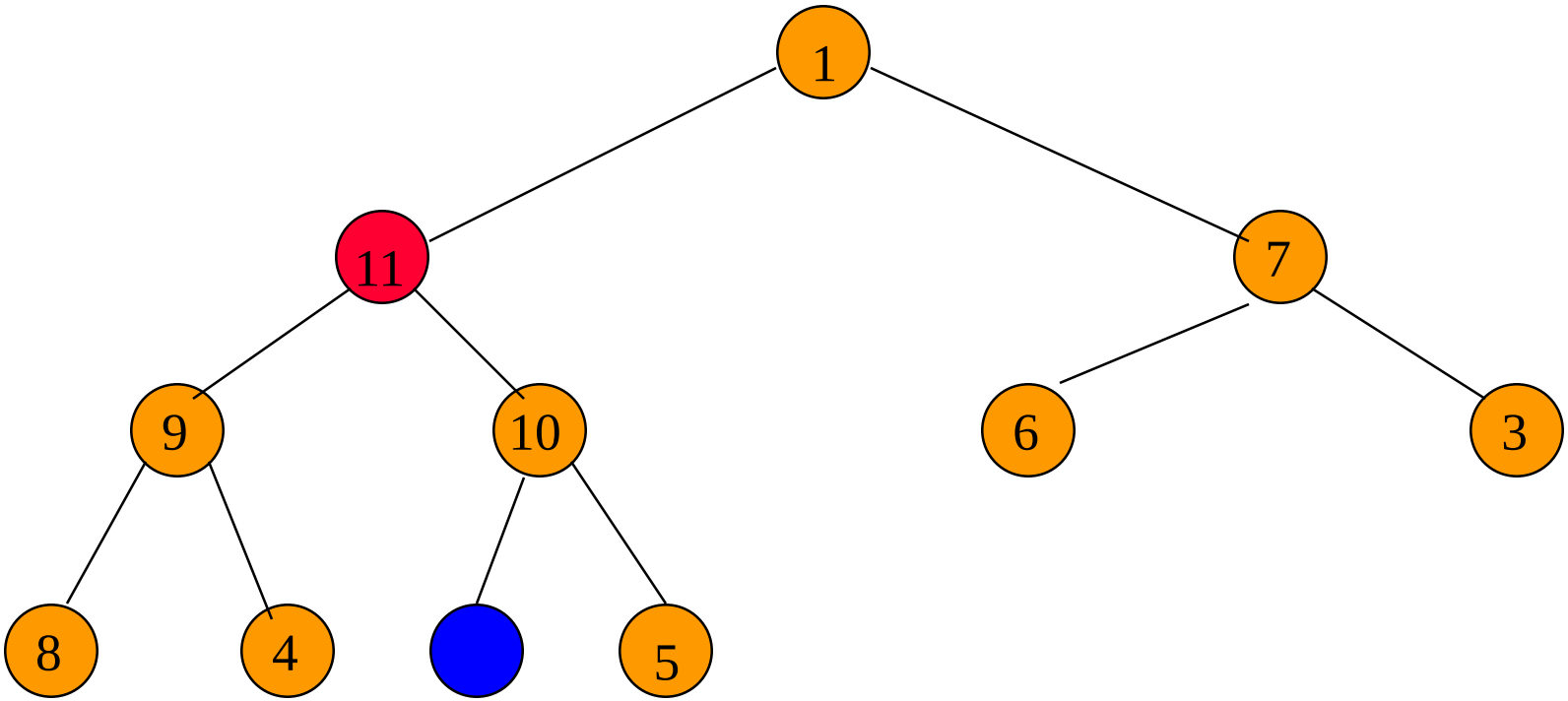


Initializing A Max Heap



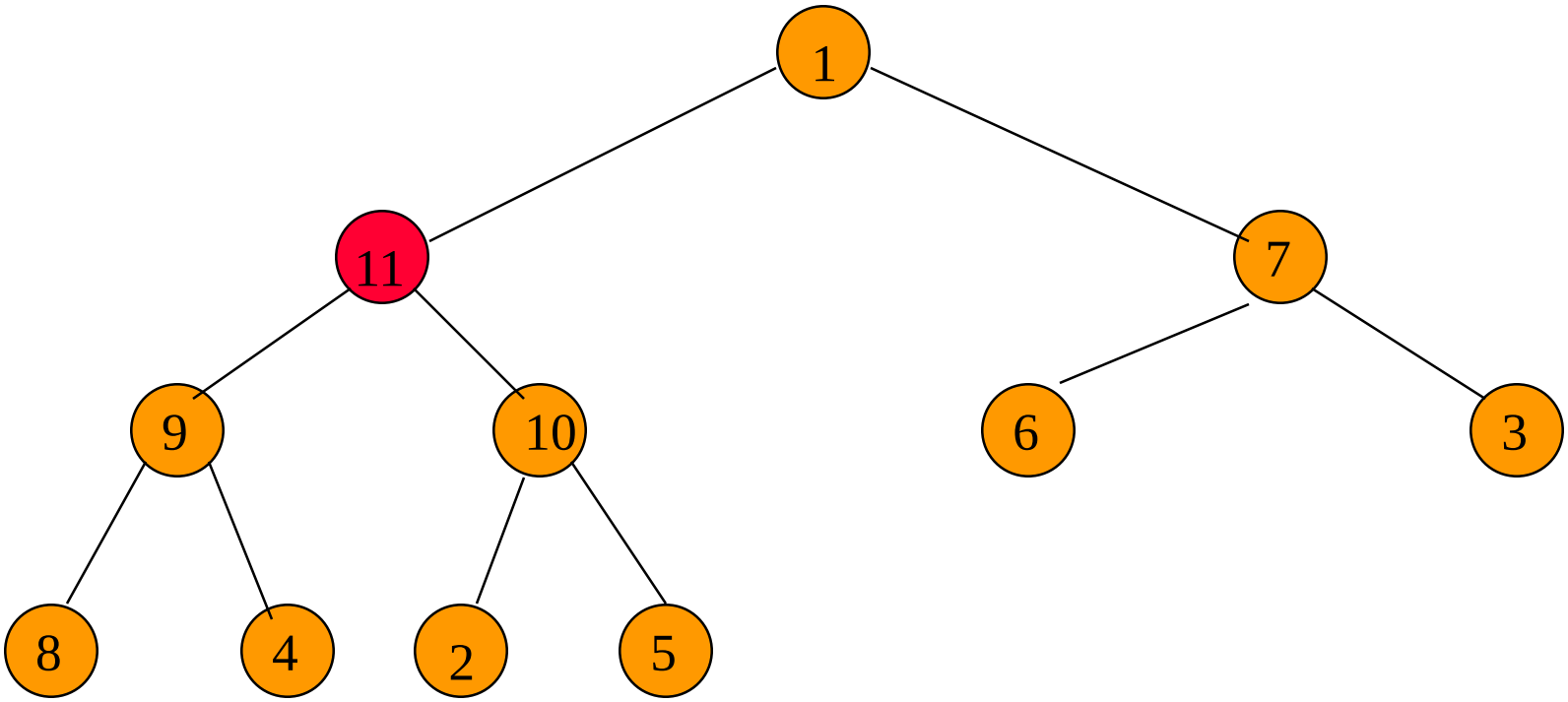
Find a home for 2.

Initializing A Max Heap



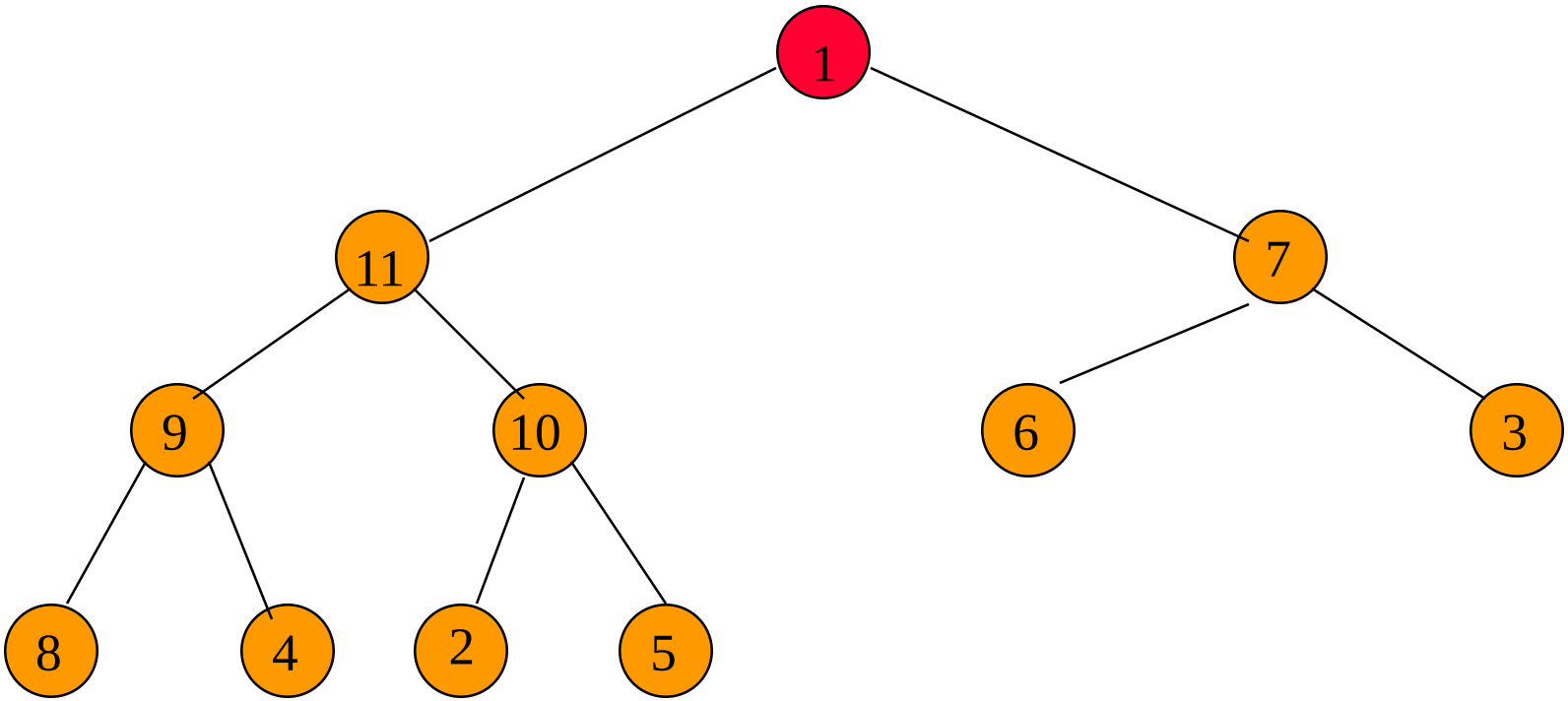
Find a home for 2.

Initializing A Max Heap



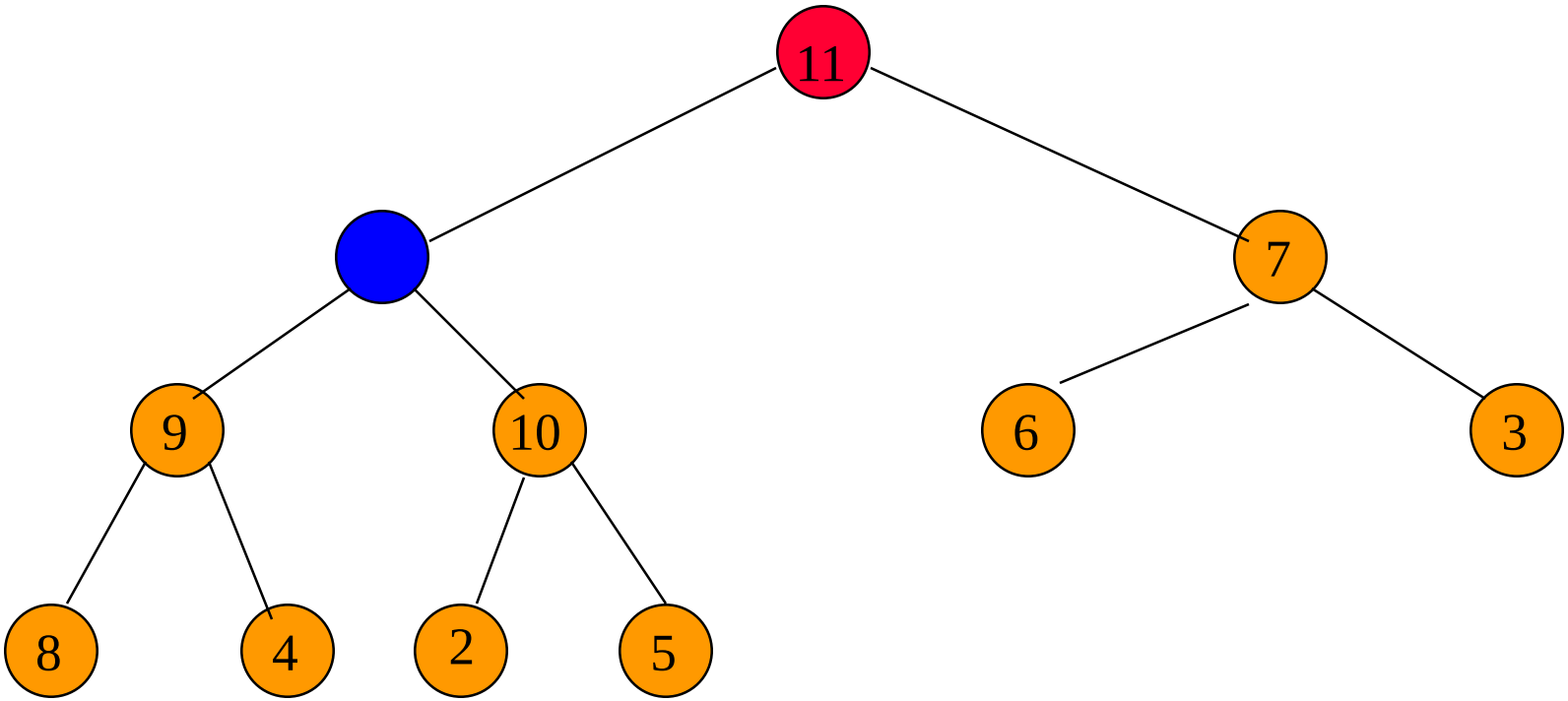
Done, move to next lower array position.

Initializing A Max Heap



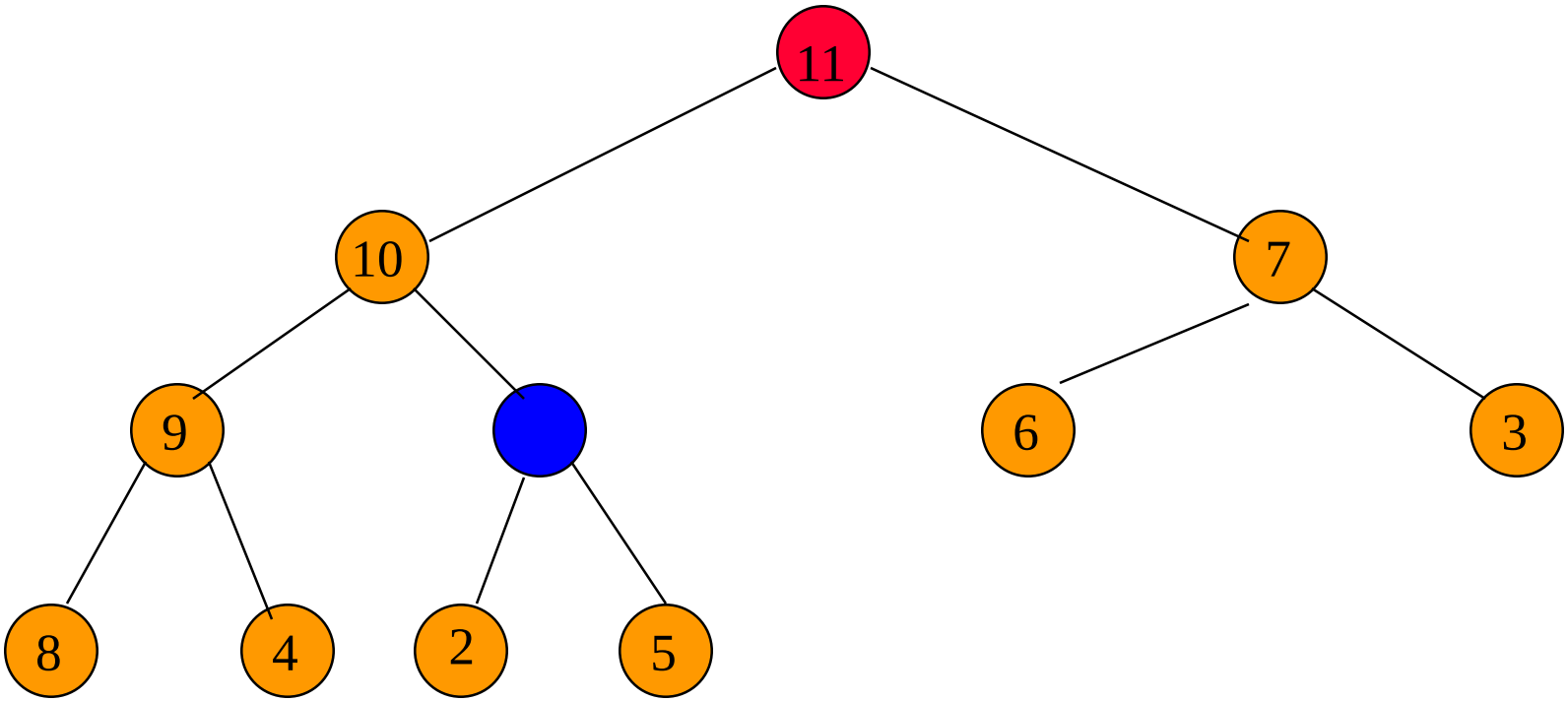
Find home for 1.

Initializing A Max Heap



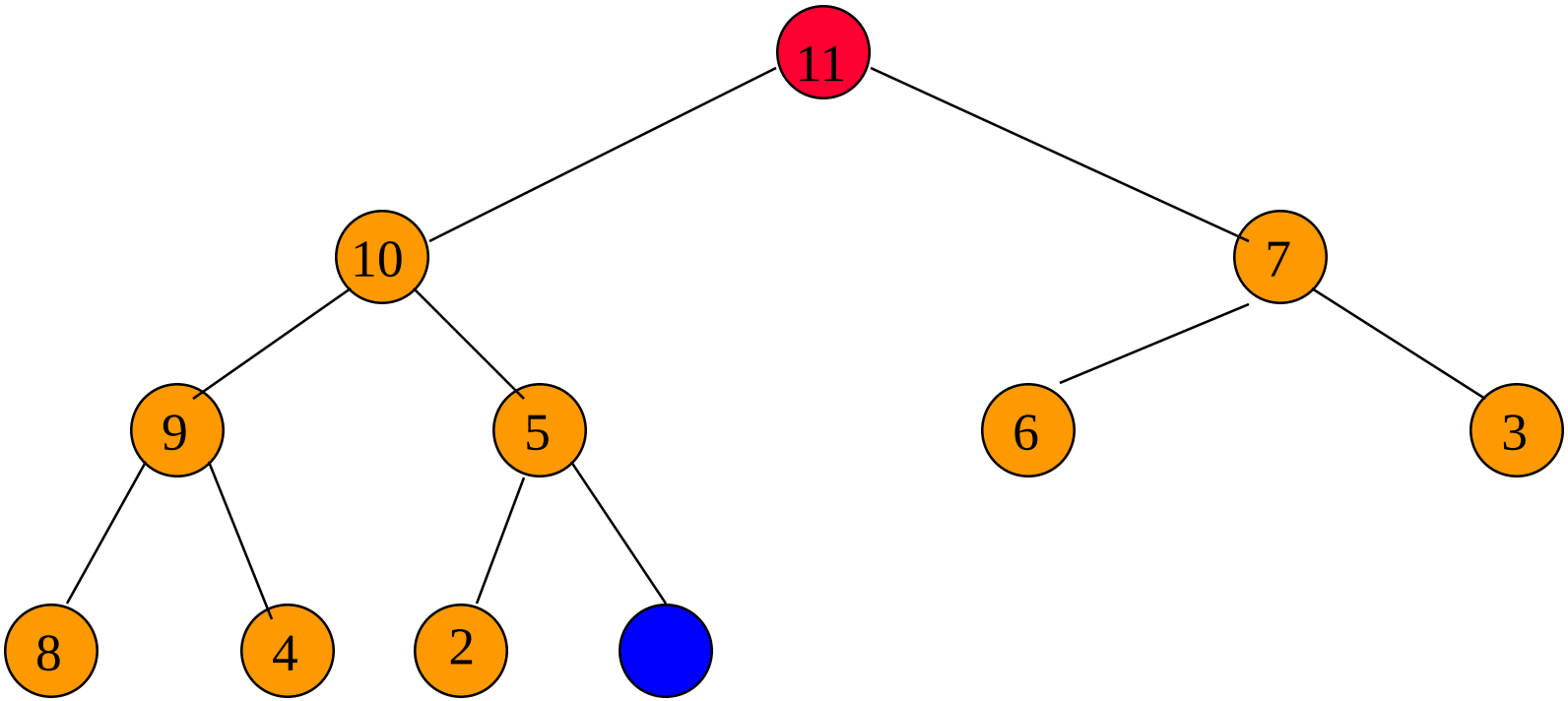
Find home for 1.

Initializing A Max Heap



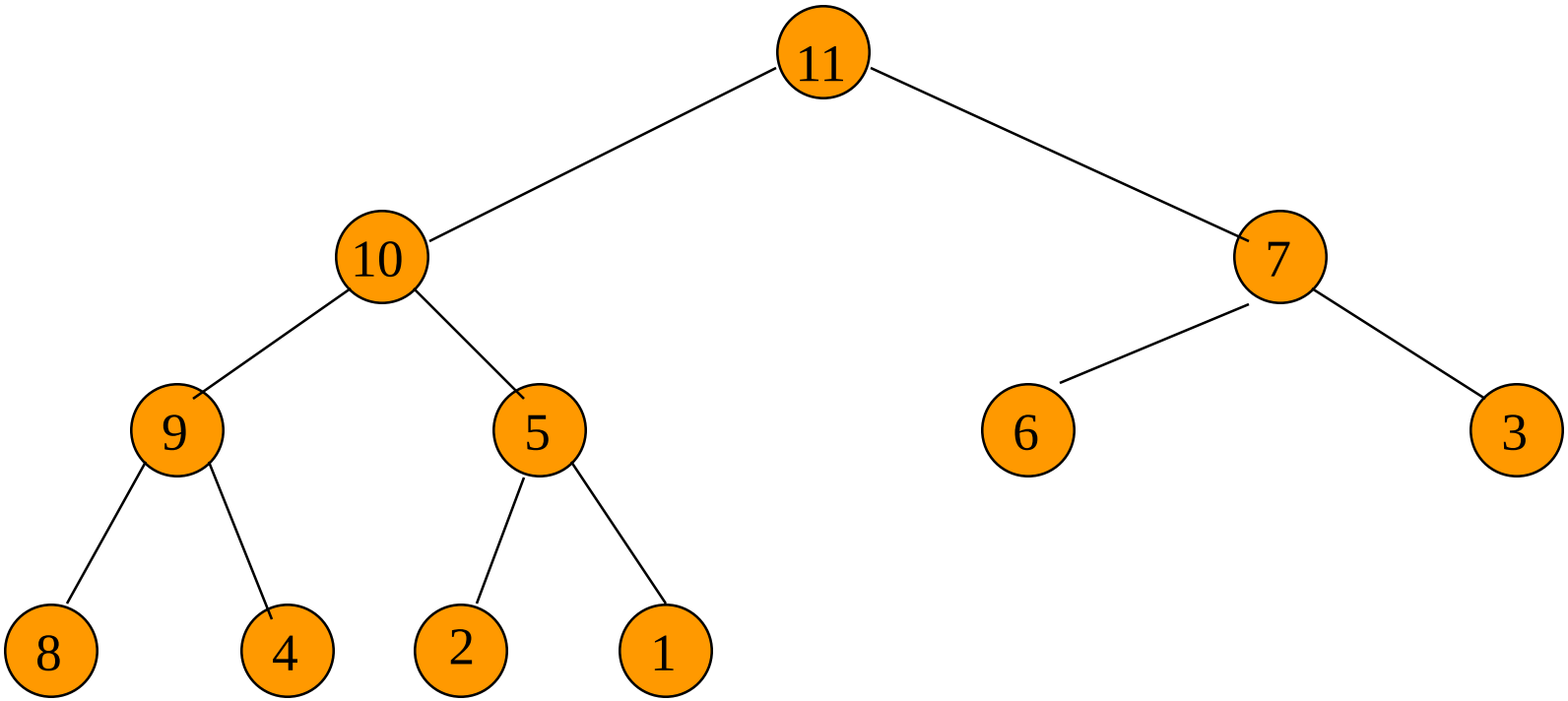
Find home for 1.

Initializing A Max Heap



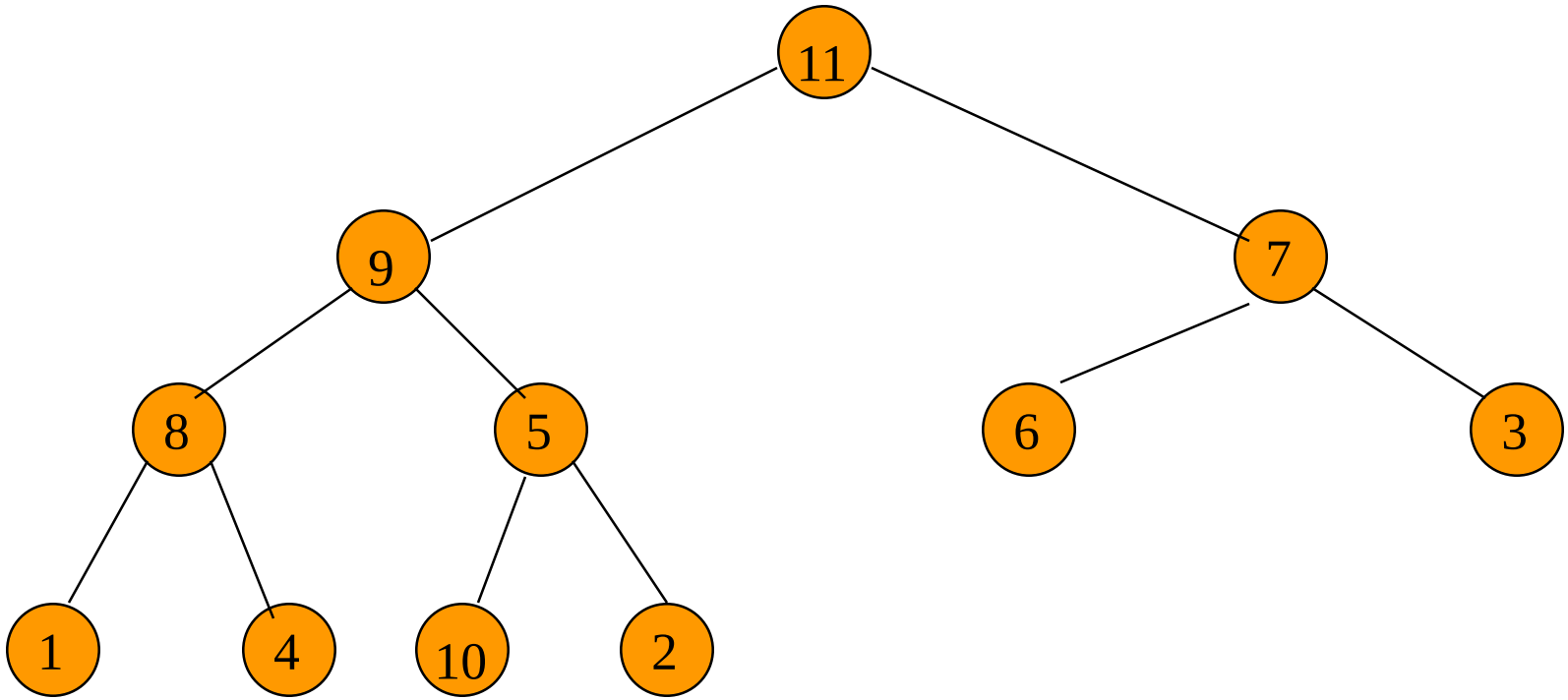
Find home for 1.

Initializing A Max Heap



Done.

Time Complexity



Height of heap = h.

Number of subtrees with root at level j is <= 2^{j-1}.

Time for each subtree is O(h-j+1).

Complexity

Time for level j subtrees is $<= 2^{j-1}(h-j+1) = t(j)$.

Total time is $t(1) + t(2) + \ldots + t(h-1) = O(n)$.

Leftist Trees

Linked binary tree.

Can do everything a heap can do and in the same asymptotic complexity.

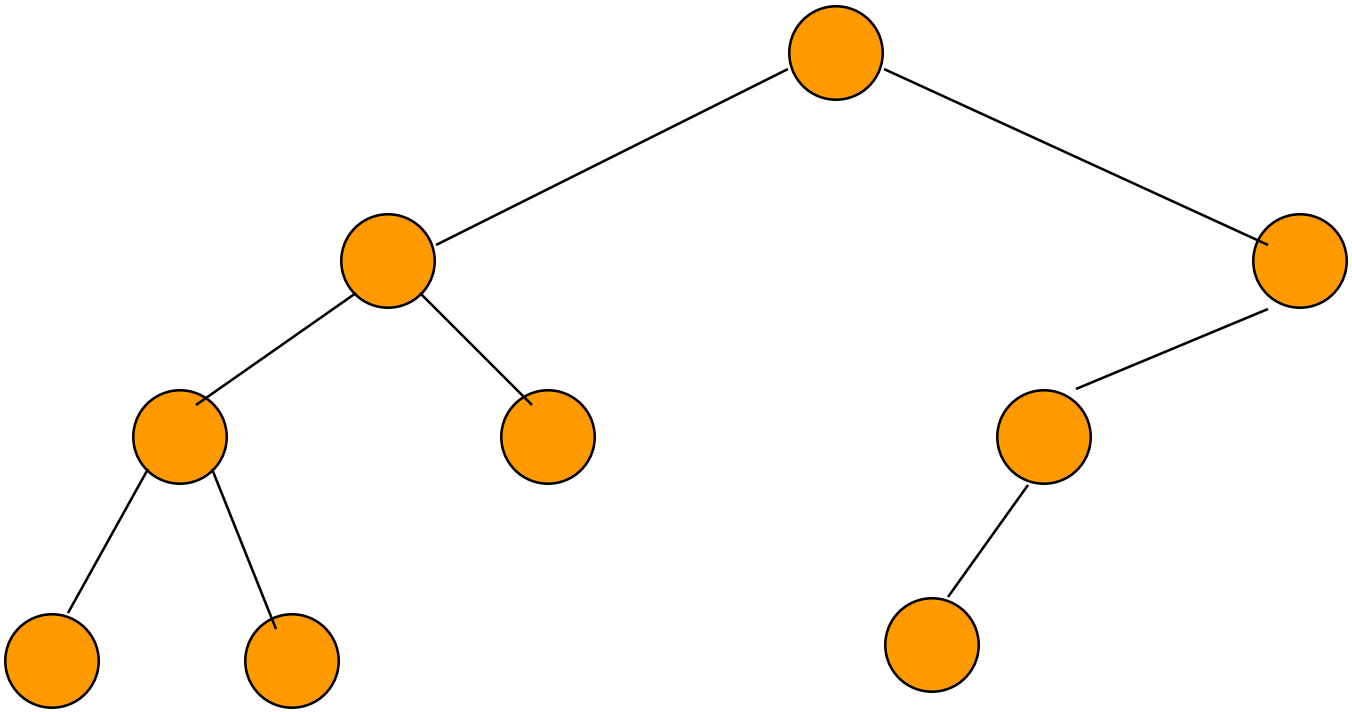
Can meld two leftist tree priority queues in O(log n) time.

Extended Binary Trees

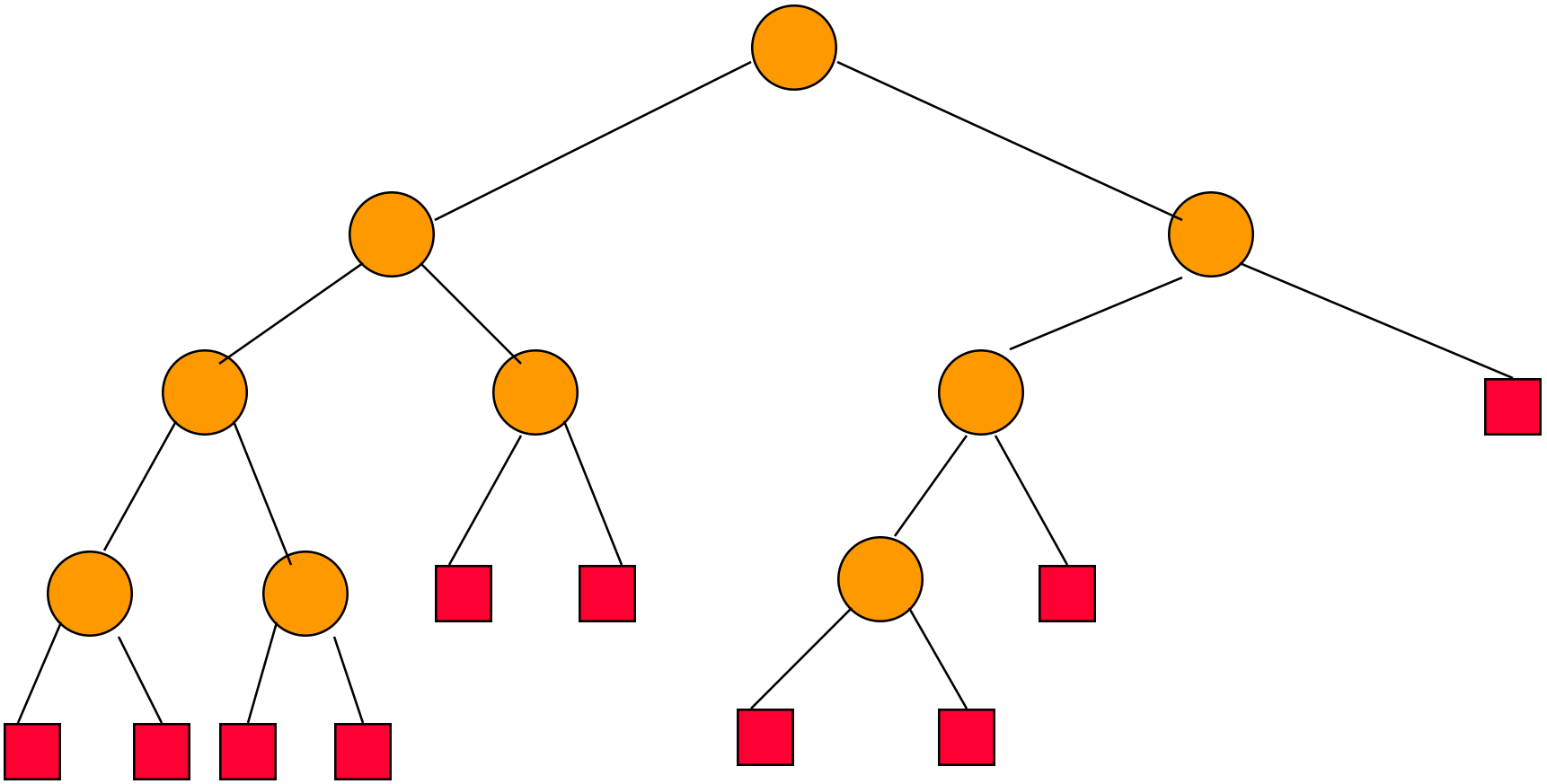
Start with any binary tree and add an external node wherever there is an empty subtree.

Result is an extended binary tree.

A Binary Tree



An Extended Binary Tree

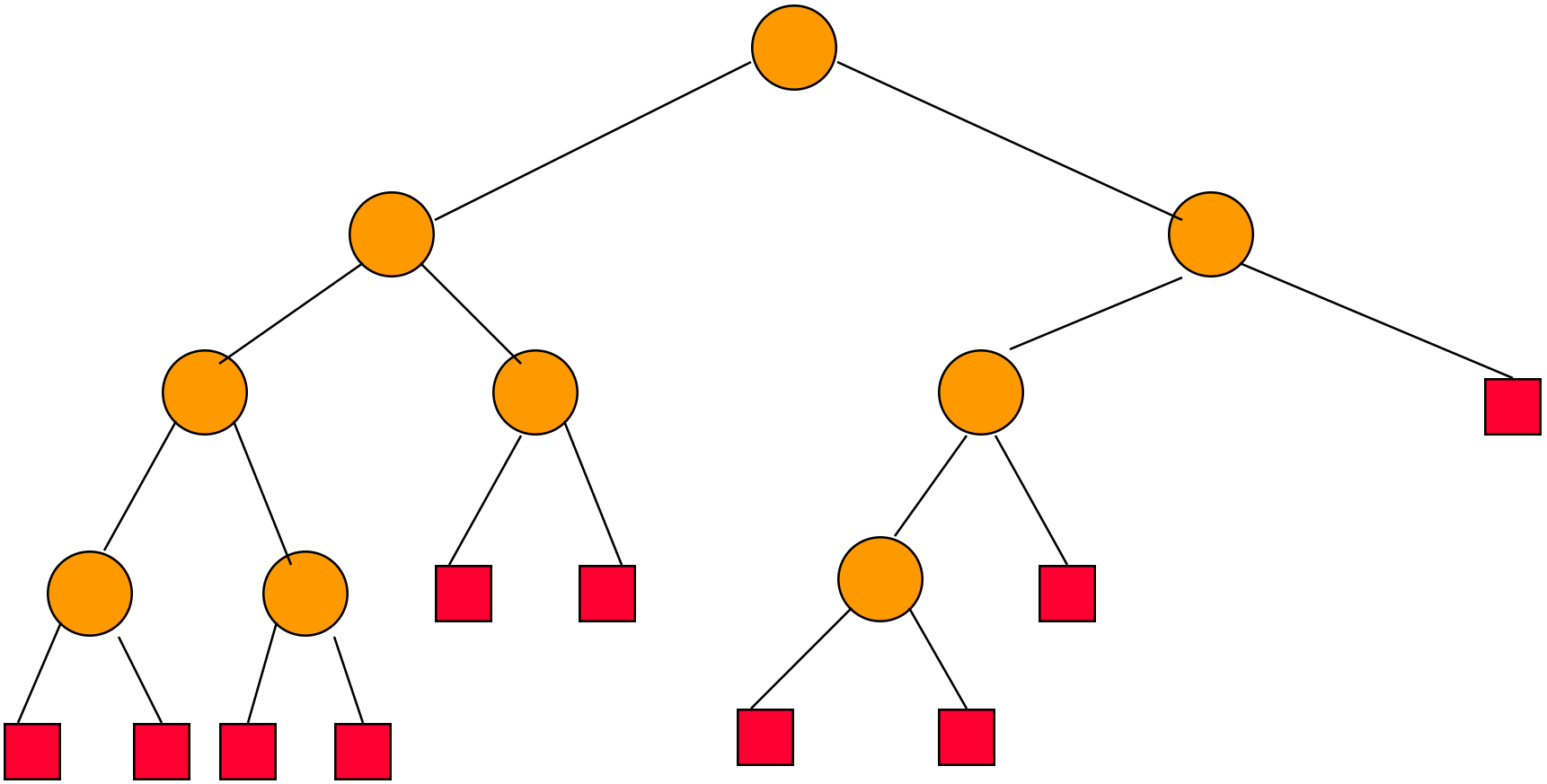


number of external nodes is n+1

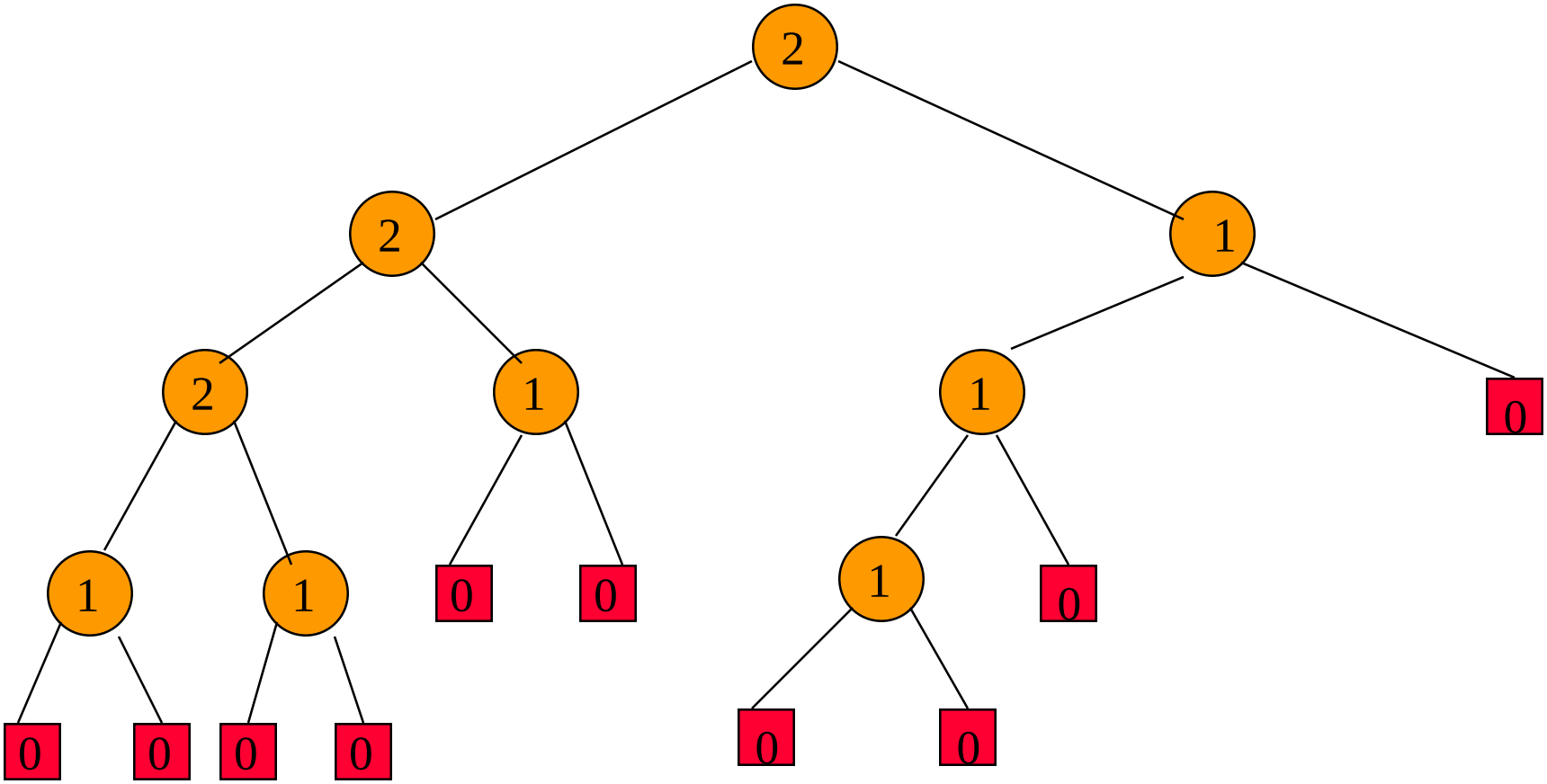
The Function s()

For any node x in an extended binary tree, let s(x) be the length of a shortest path from x to an external node in the subtree rooted at x.

s() Values Example



s() Values Example



Properties Of s()

If x is an external node, then s(x) = 0.

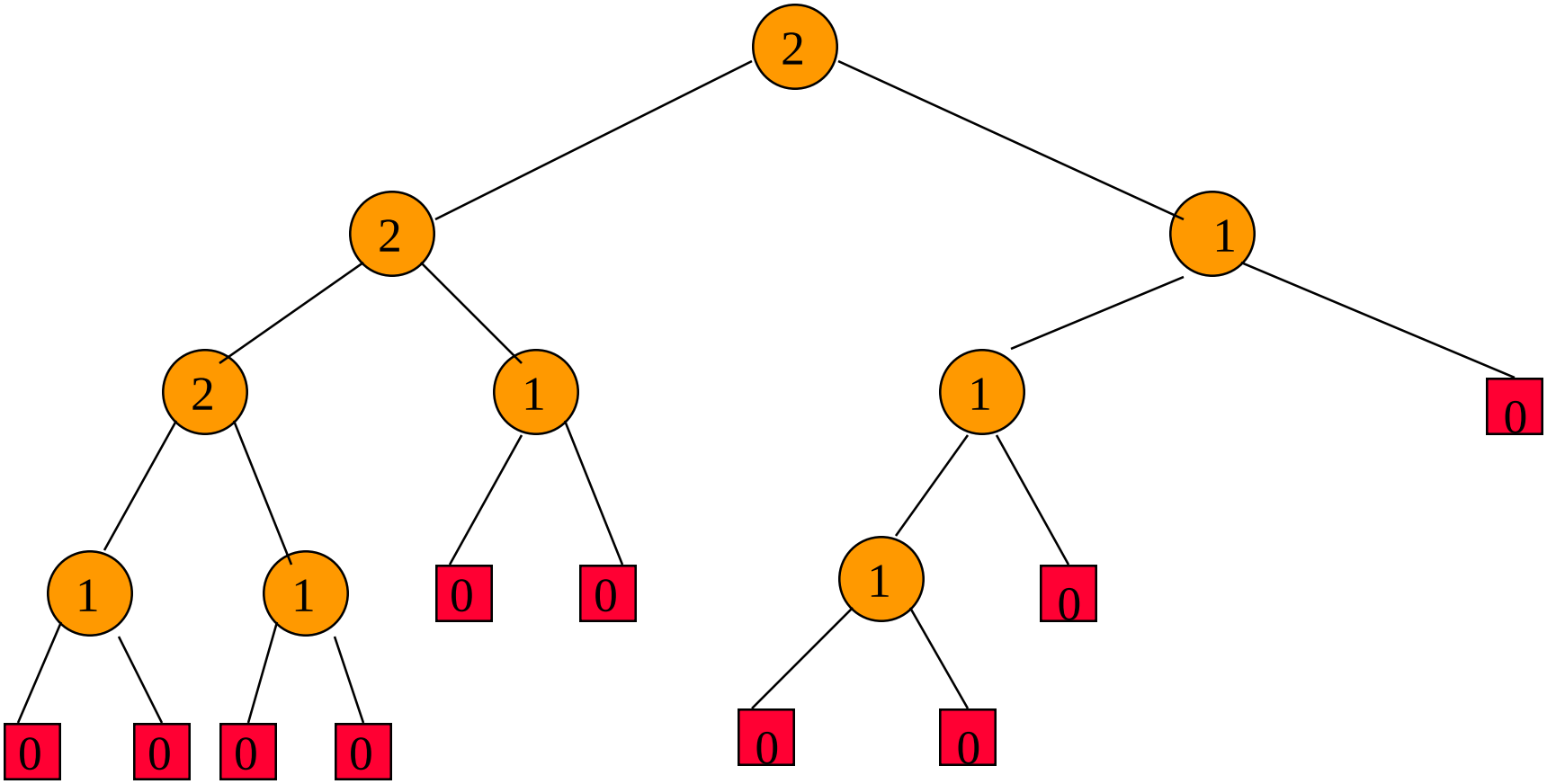
Otherwise,

s(x) = min {s(leftChild(x)),
s(rightChild(x))} + 1

Height Biased Leftist Trees

A binary tree is a (height biased) leftist tree iff for every internal node x,
s(leftChild(x)) >= s(rightChild(x))

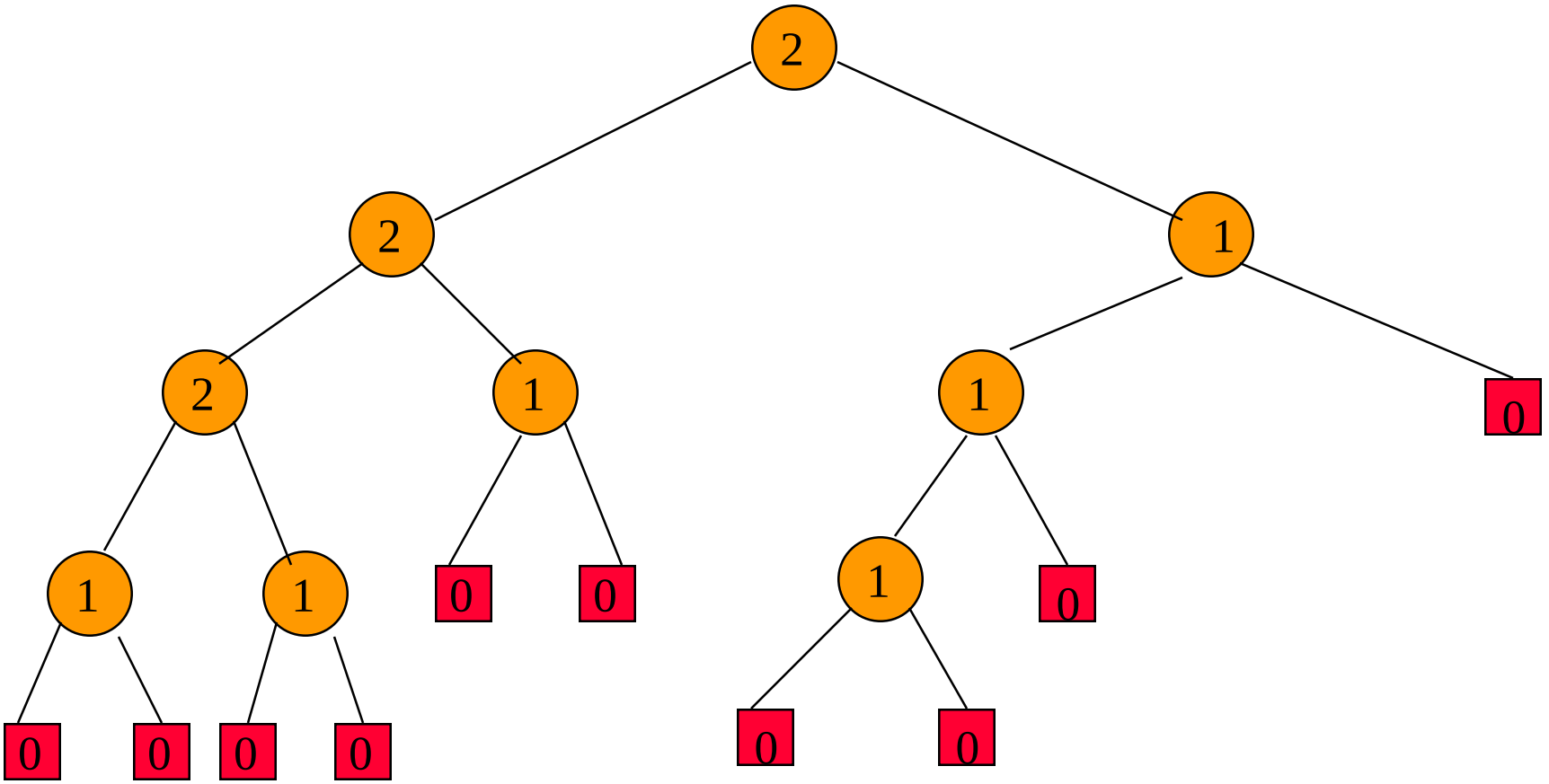
A Leftist Tree



Leftist Trees--Property 1

In a leftist tree, the rightmost path is a shortest root to external node path and the length of this path is s(root).

A Leftist Tree



Length of rightmost path is 2.

Leftist Trees—Property 2

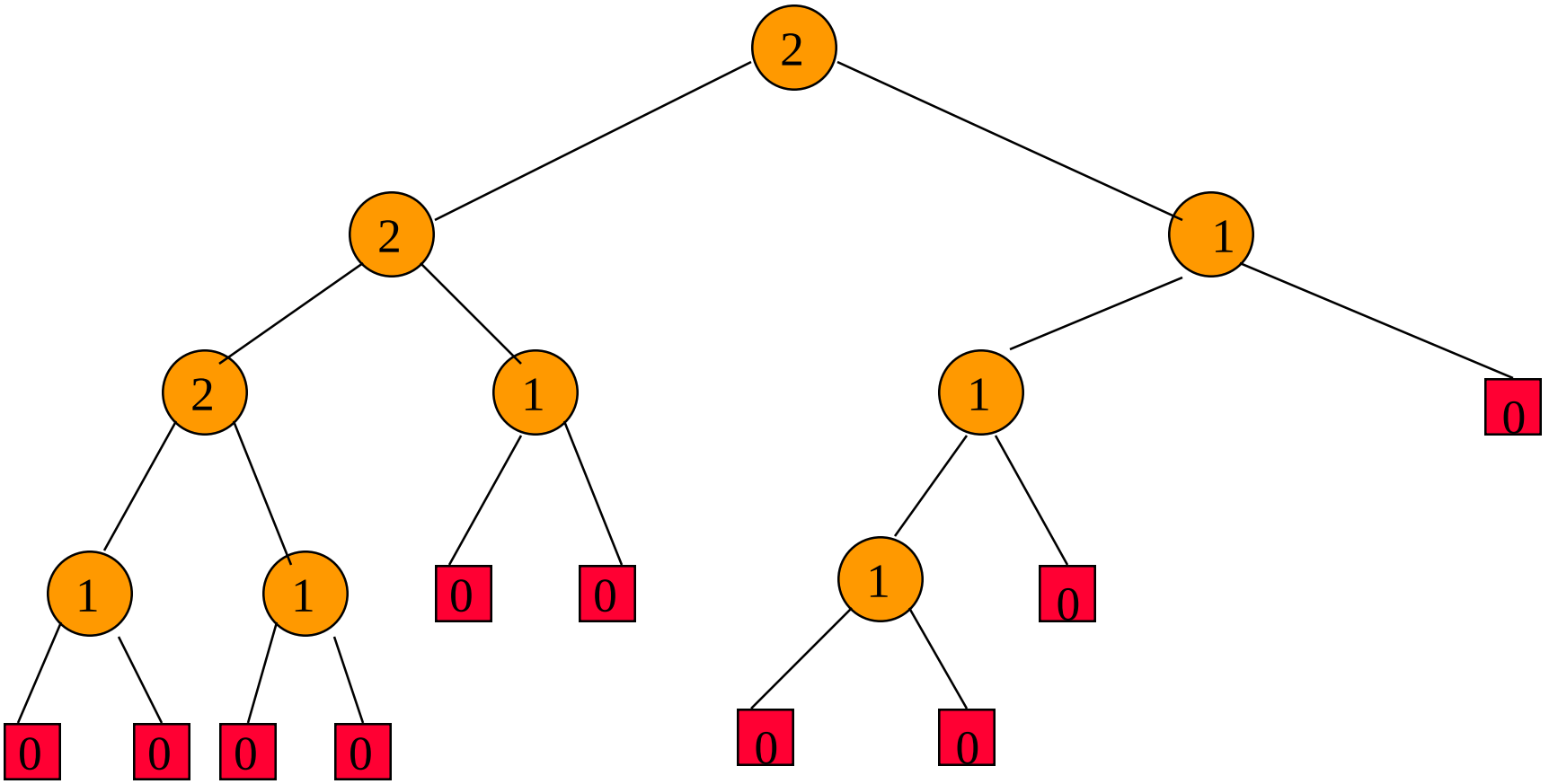
The number of internal nodes is at least

$2^{s(root)} - 1$

Because levels 1 through s(root) have no external nodes.

So, $s(root) <= \log(n+1)$

A Leftist Tree



Levels 1 and 2 have no external nodes.

Leftist Trees—Property 3

Length of rightmost path is $O(\log n)$, where n is the number of nodes in a leftist tree.

Follows from Properties 1 and 2.

Leftist Trees As Priority Queues

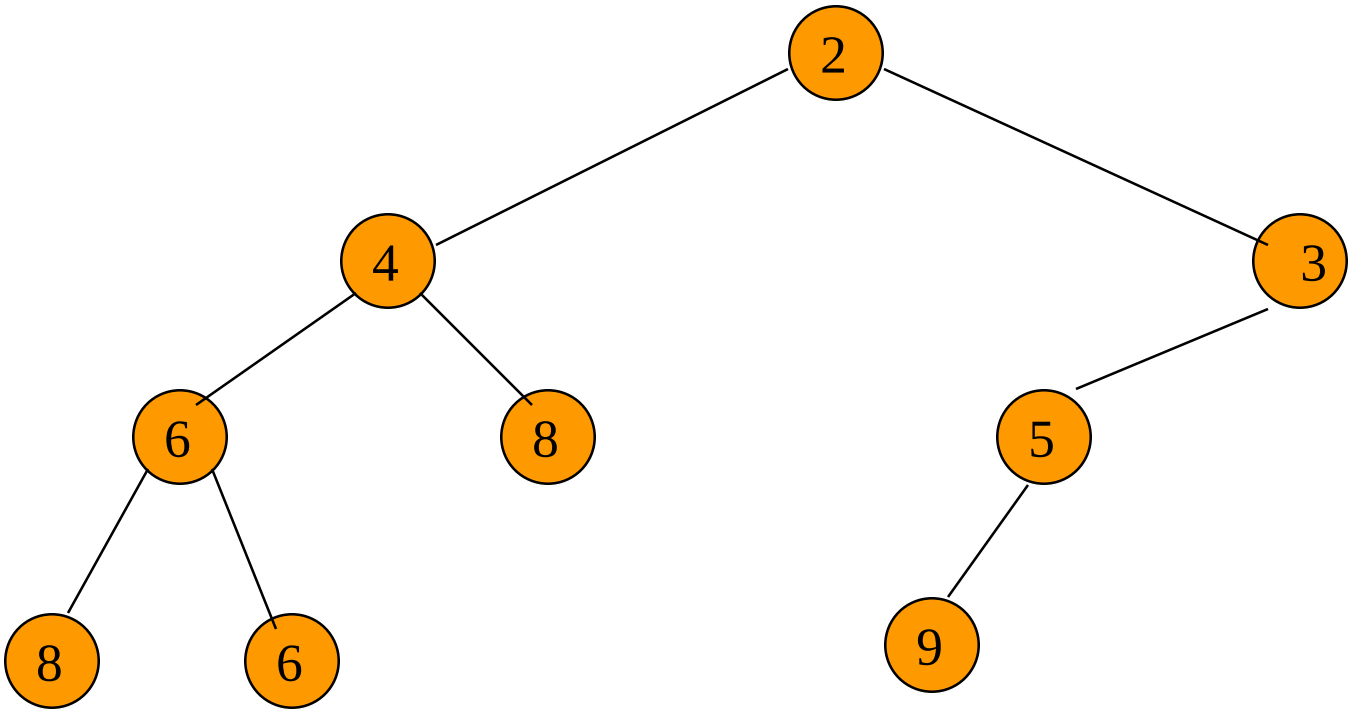
Min leftist tree … leftist tree that is a min tree.

Used as a min priority queue.

Max leftist tree … leftist tree that is a max tree.

Used as a max priority queue.

A Min Leftist Tree



Some Min Leftist Tree Operations

empty()

size()

top()

push()

pop()

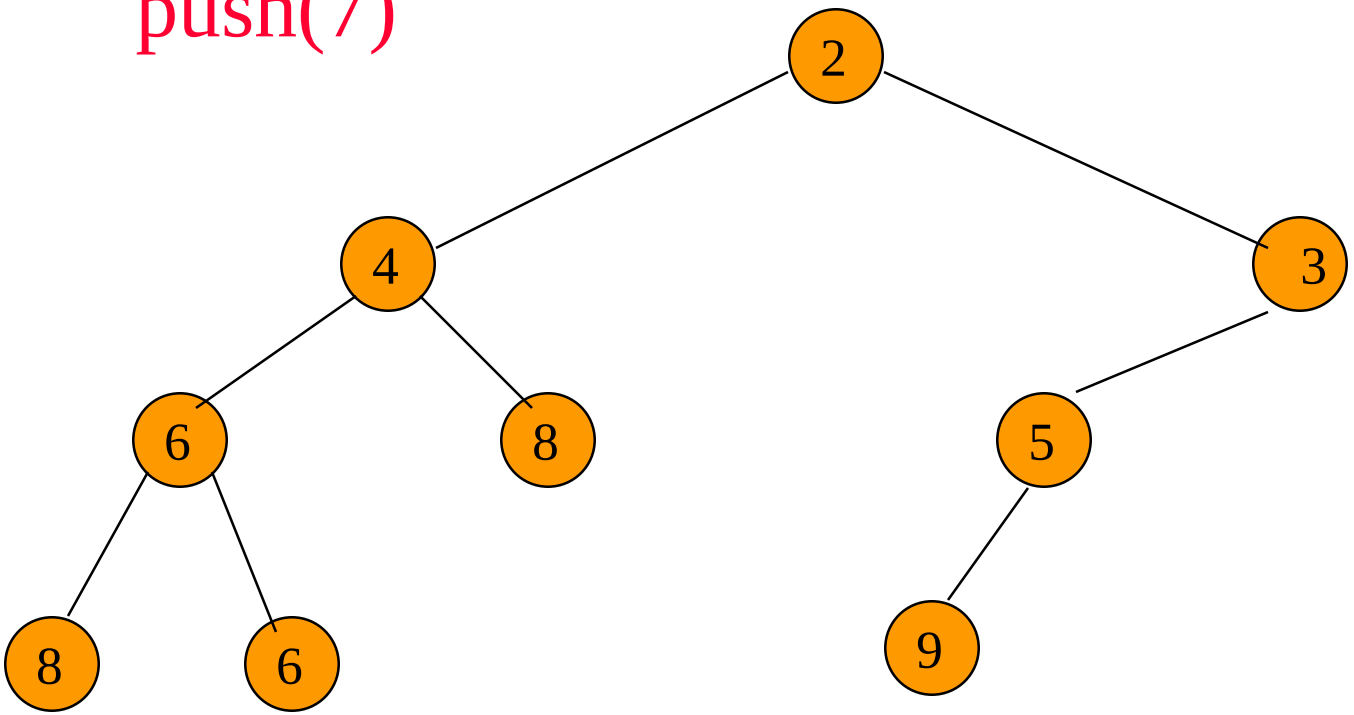
meld()

initialize()

push() and pop() use meld().

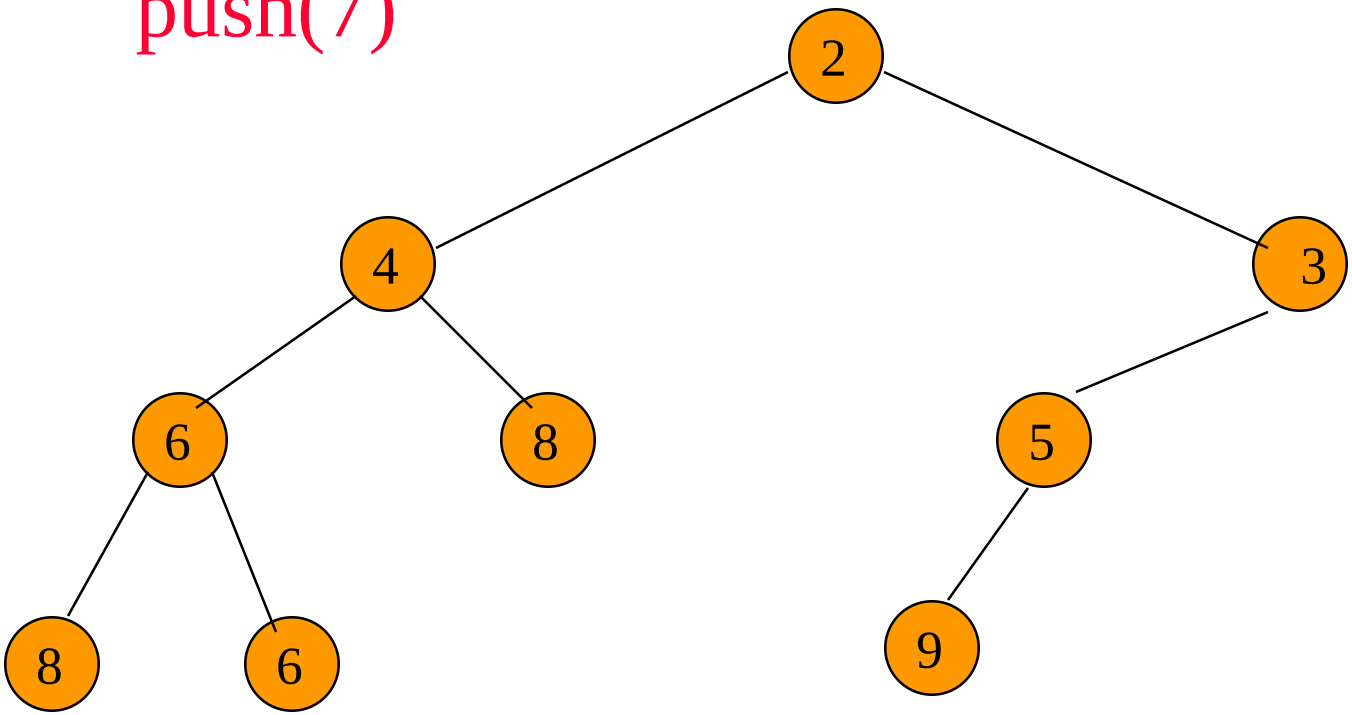
Push Operation

push(7)



Push Operation

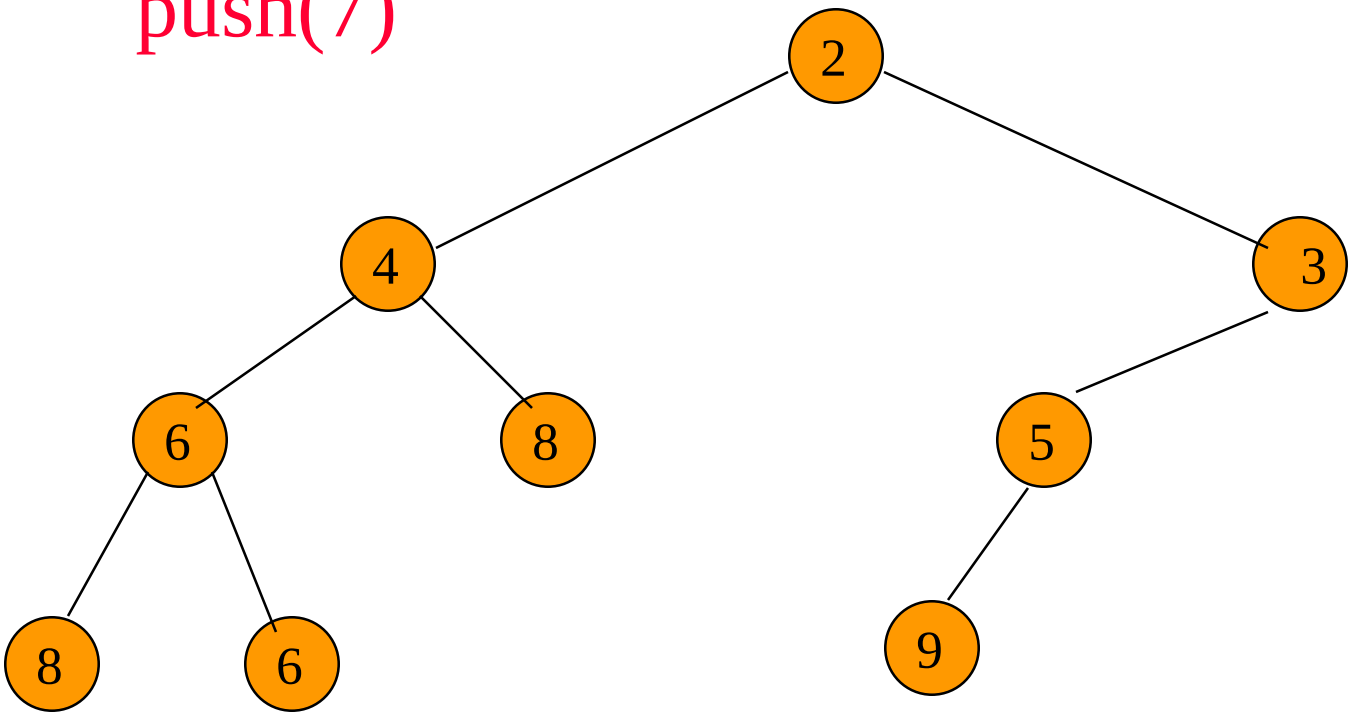
push(7)



Create a single node min leftist tree.

Push Operation

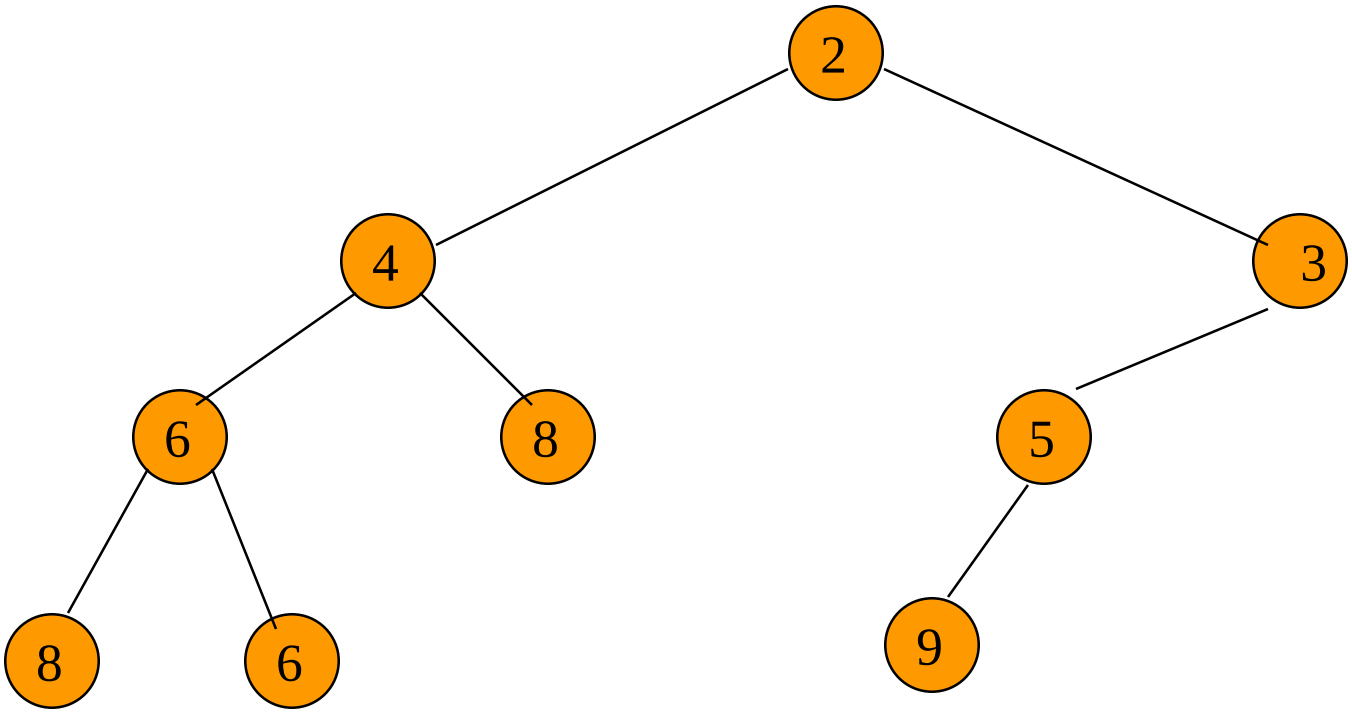
push(7)



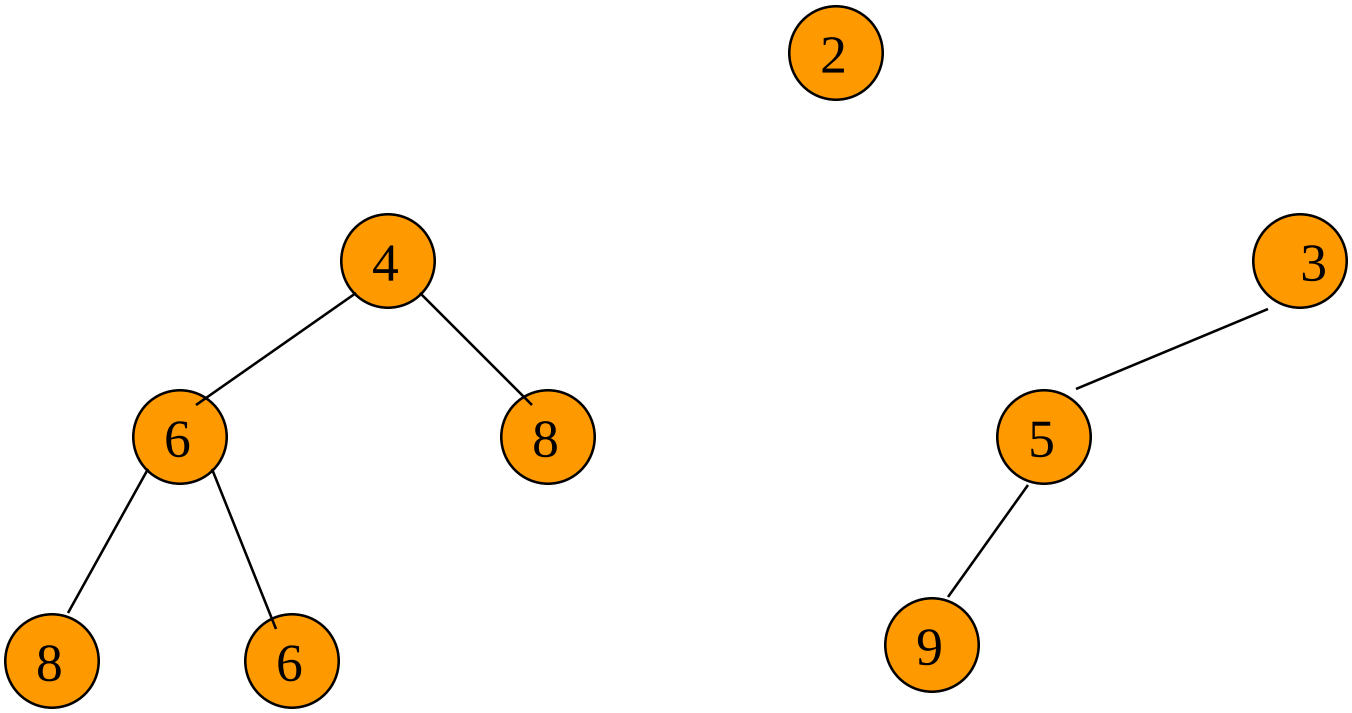
Create a single node min leftist tree.

Meld the two min leftist trees.

Remove Min (pop)

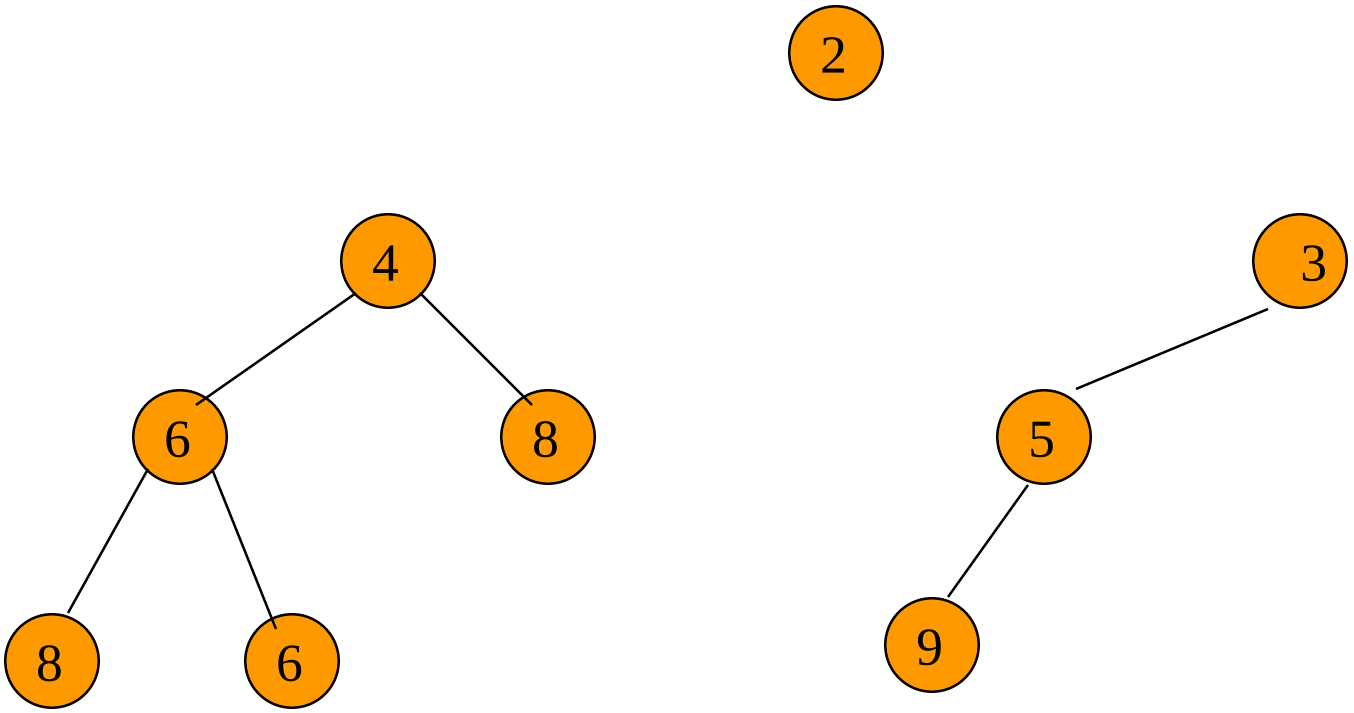


Remove Min (pop)



Remove the root.

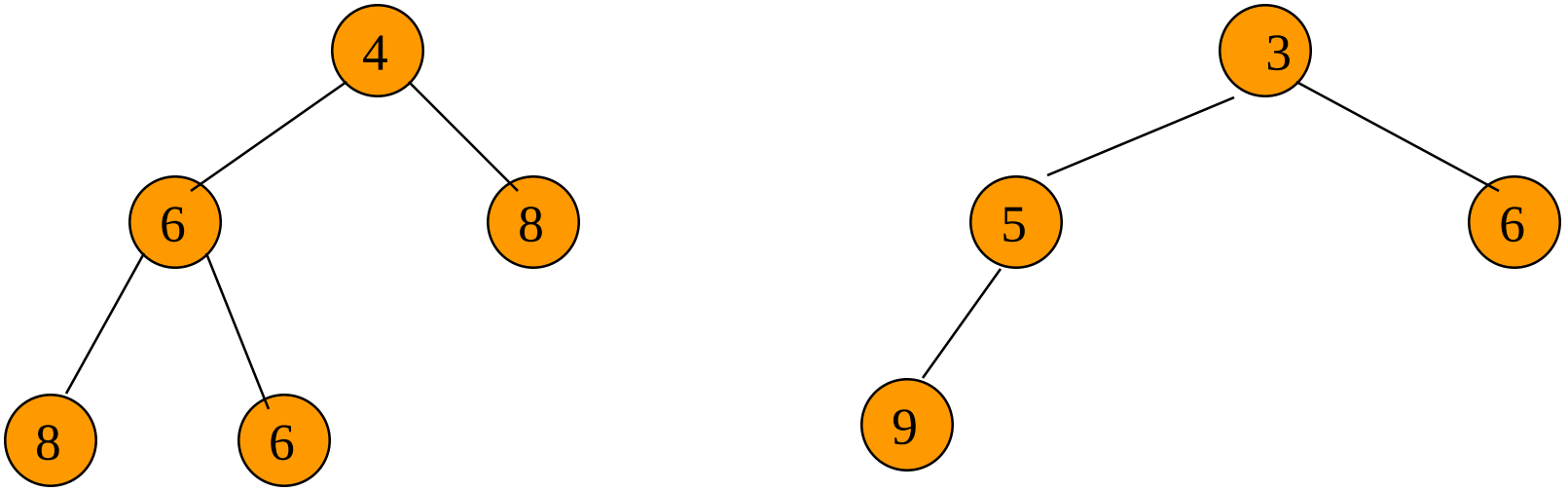
Remove Min (pop)



Remove the root.

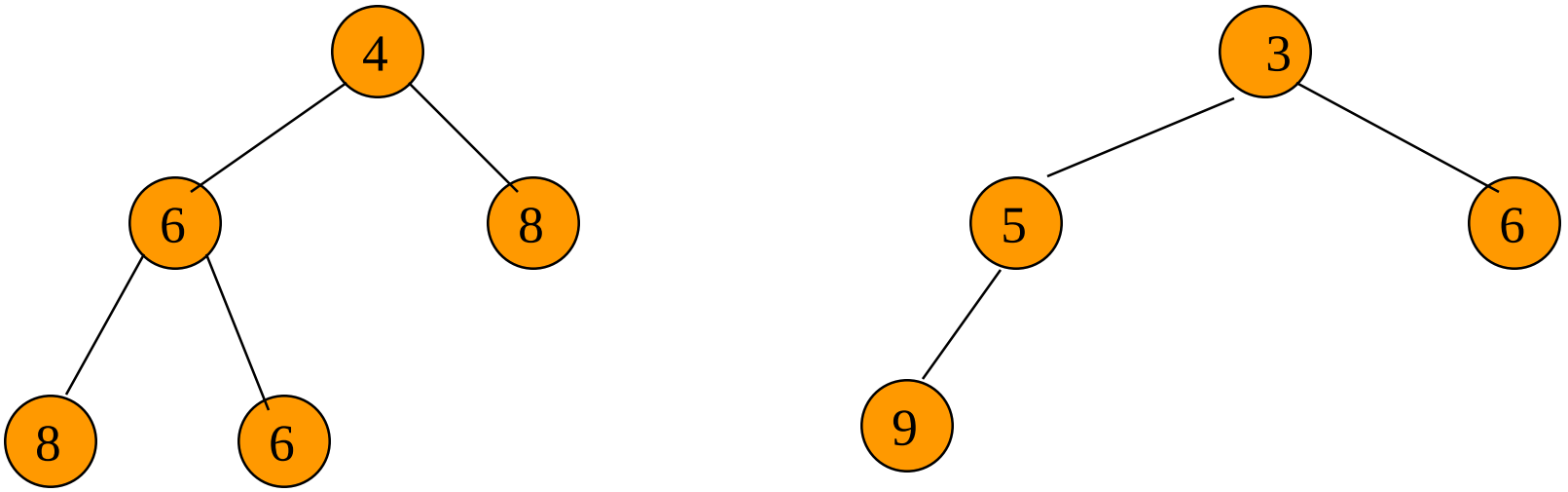
Meld the two subtrees.

Meld Two Min Leftist Trees



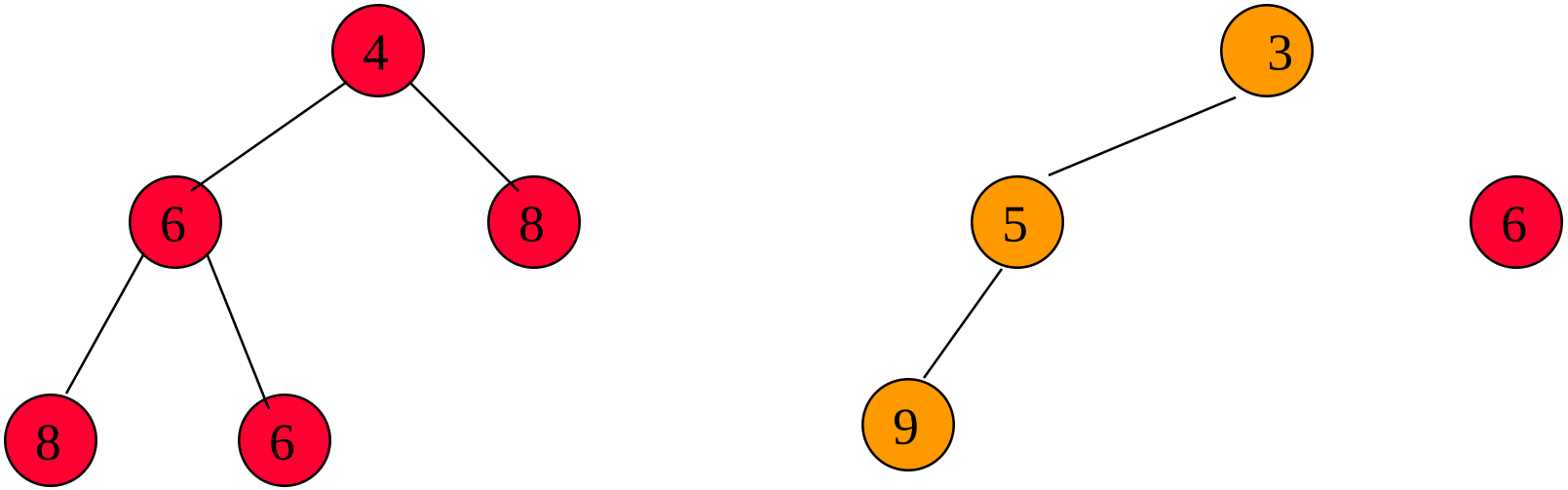
Traverse only the rightmost paths so as to get logarithmic performance.

Meld Two Min Leftist Trees



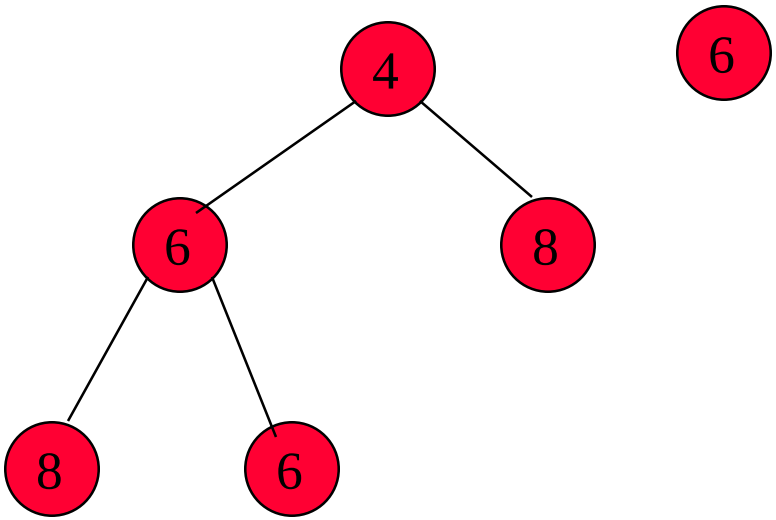
Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

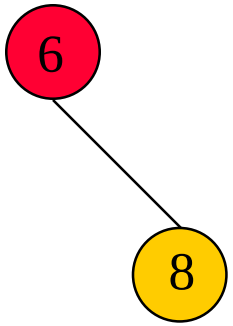
Meld Two Min Leftist Trees

Meld right subtree of tree with smaller root and all of other tree.

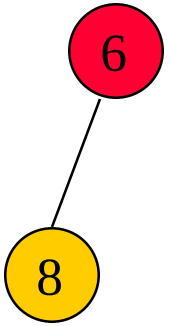
Right subtree of 6 is empty. So, result of melding right subtree of tree with smaller root and other tree is the other tree.

Meld Two Min Leftist Trees

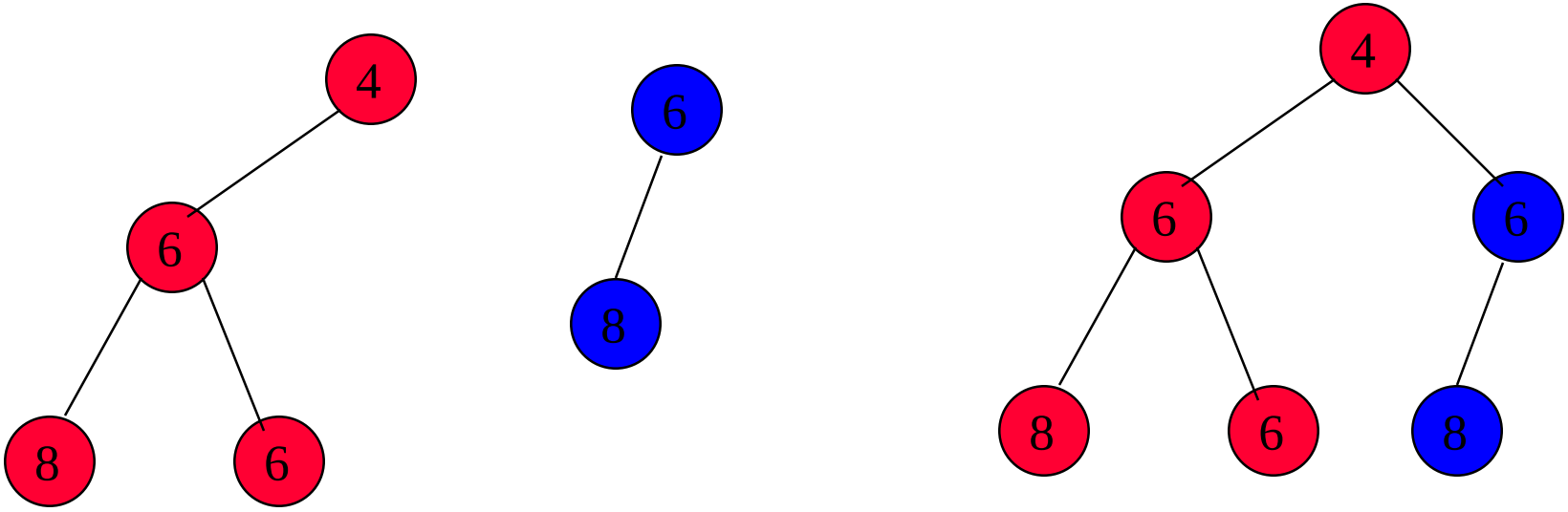
Make melded subtree right subtree of smaller root.



Swap left and right subtree if s(left) < s(right).



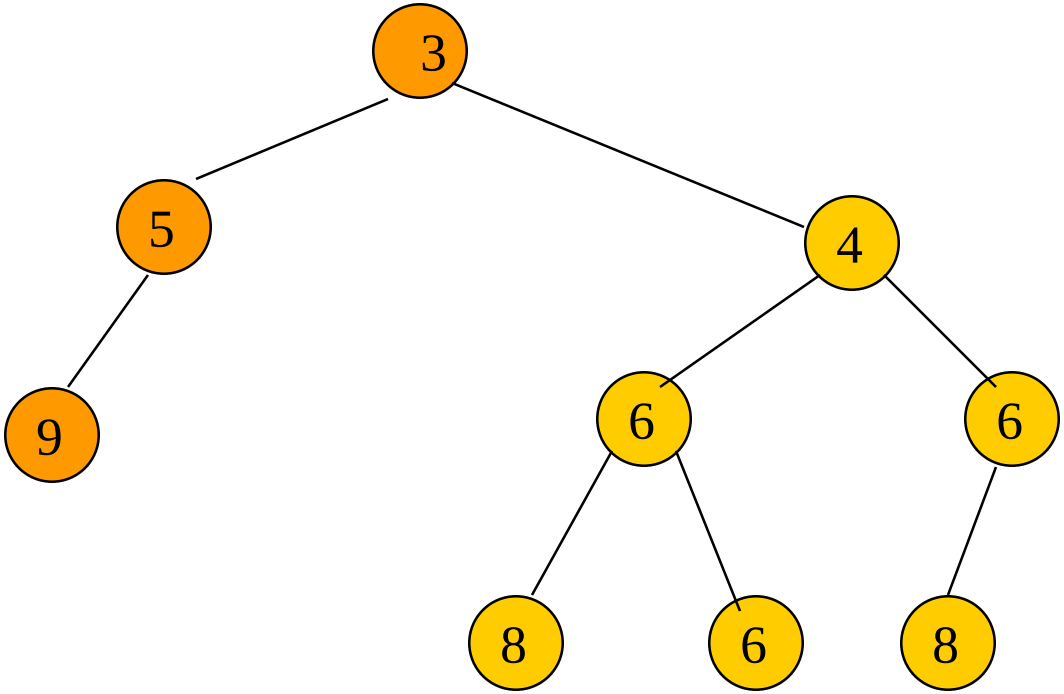
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if s(left) < s(right).

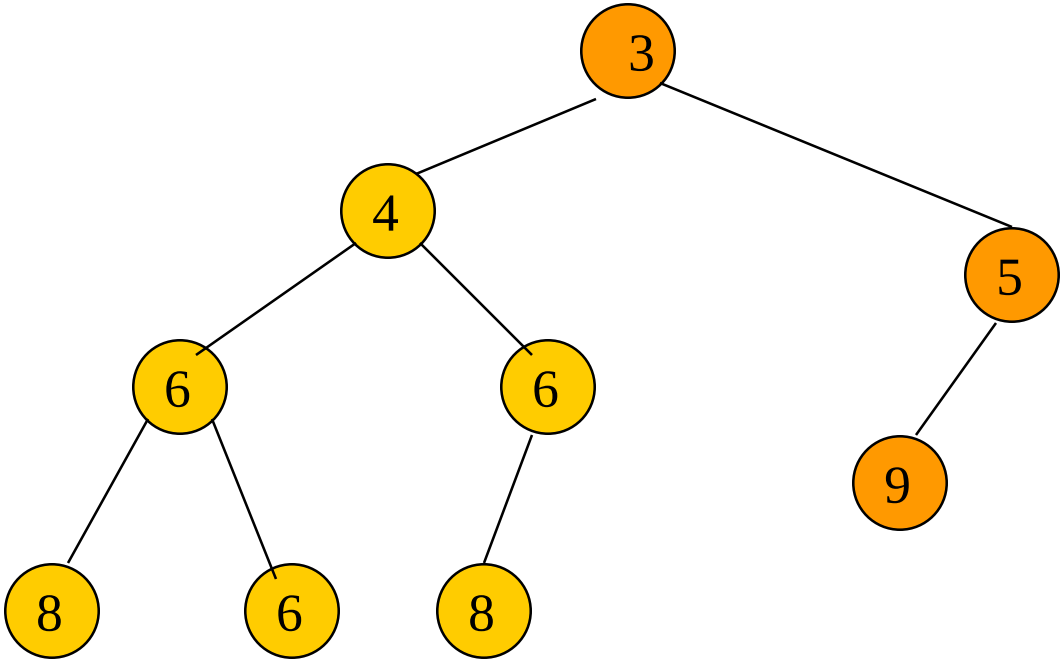
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if s(left) < s(right).

Meld Two Min Leftist Trees



Initializing In O(n) Time

- create n single node min leftist trees and place them in a FIFO queue
- repeatedly remove two min leftist trees from the FIFO queue, meld them, and put the resulting min leftist tree into the FIFO queue
- the process terminates when only 1 min leftist tree remains in the FIFO queue
- analysis is the same as for heap initialization

Section 7-2

An Introduction to
Binary Search Tree

Binary Search Trees

- Dictionary Operations:
  - IsEmpty()
  - Get(key)
  - Insert(key, value)
  - Delete(key)

Complexity Of Dictionary Operations
Get(), Insert() and Delete()

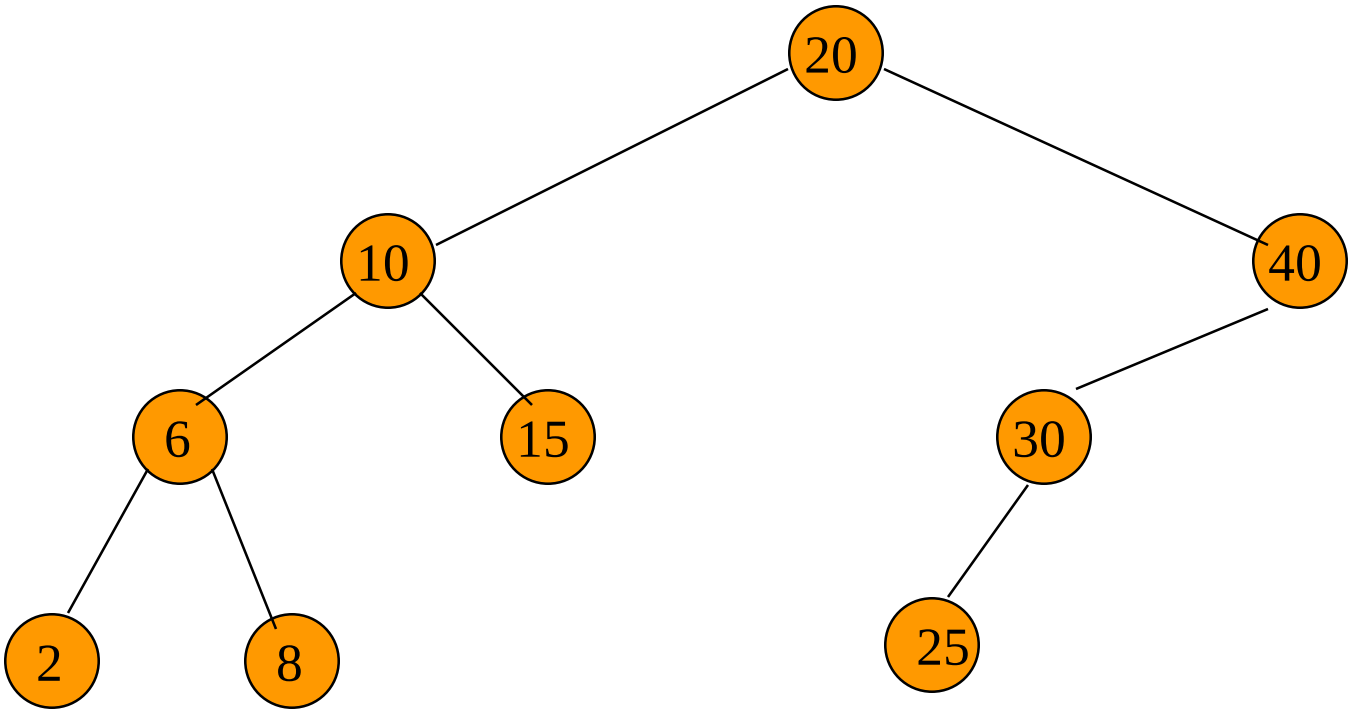
Data Structure	Worst Case	Expected
Hash Table	O(n)	O(1)
Binary Search Tree	O(n)	O(log n)
Balanced Binary Search Tree	O(log n)	O(log n)

n is number of elements in dictionary

Definition Of Binary Search Tree

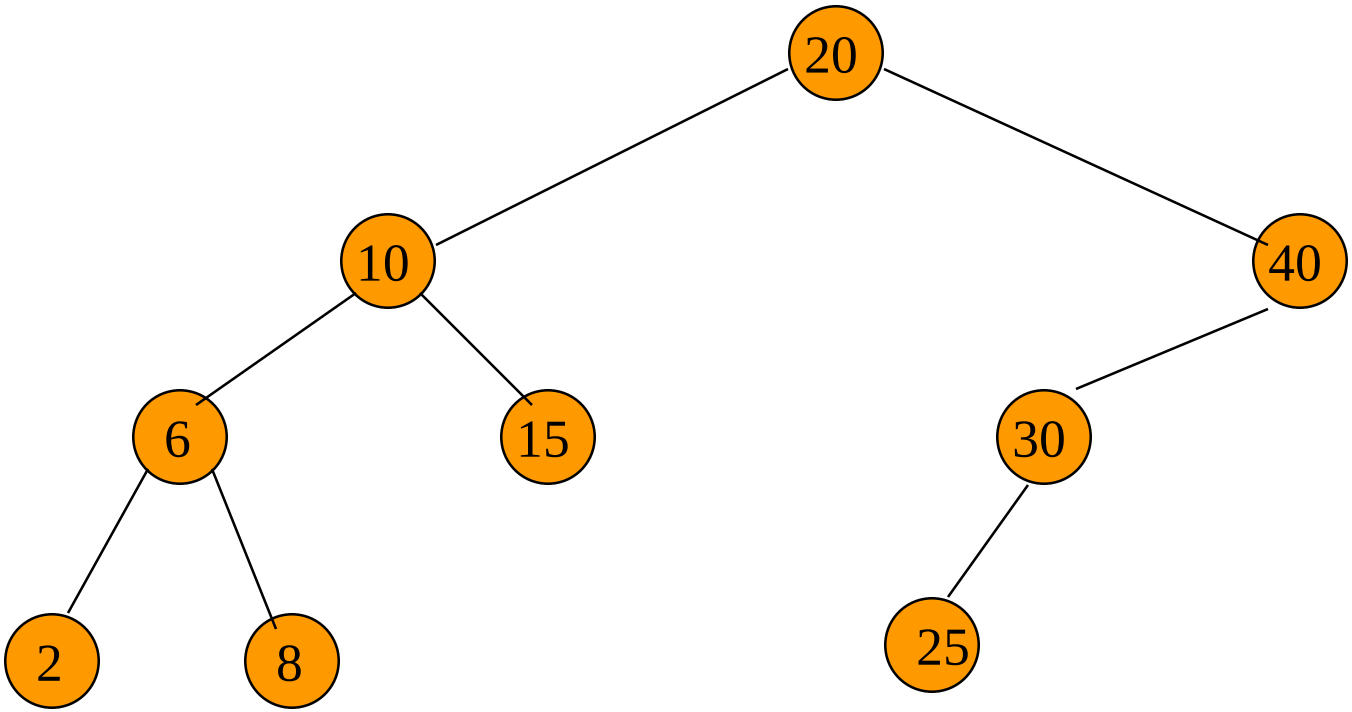
- A binary tree.
- Each node has a (key, value) pair.
- For every node x, all keys in the left subtree of x are smaller than that in x.
- For every node x, all keys in the right subtree of x are greater than that in x.

Example Binary Search Tree



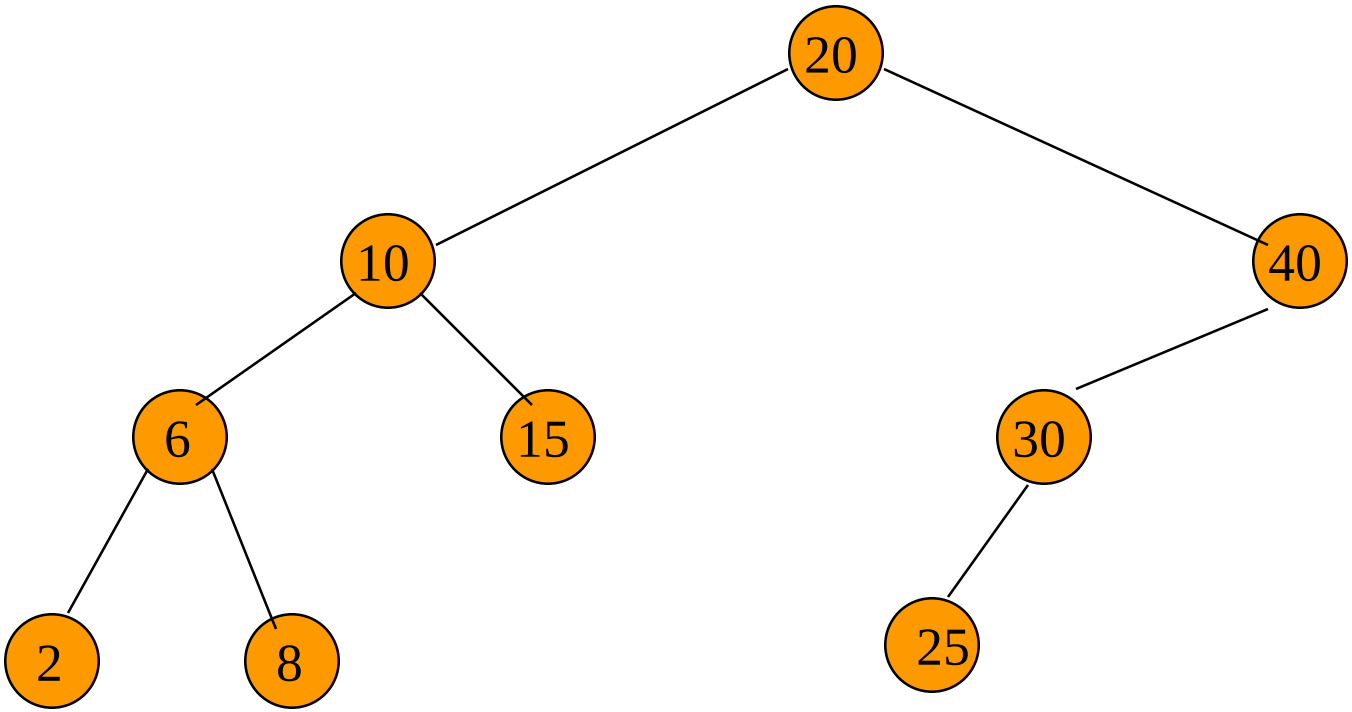
Only keys are shown.

The Operation Ascend()



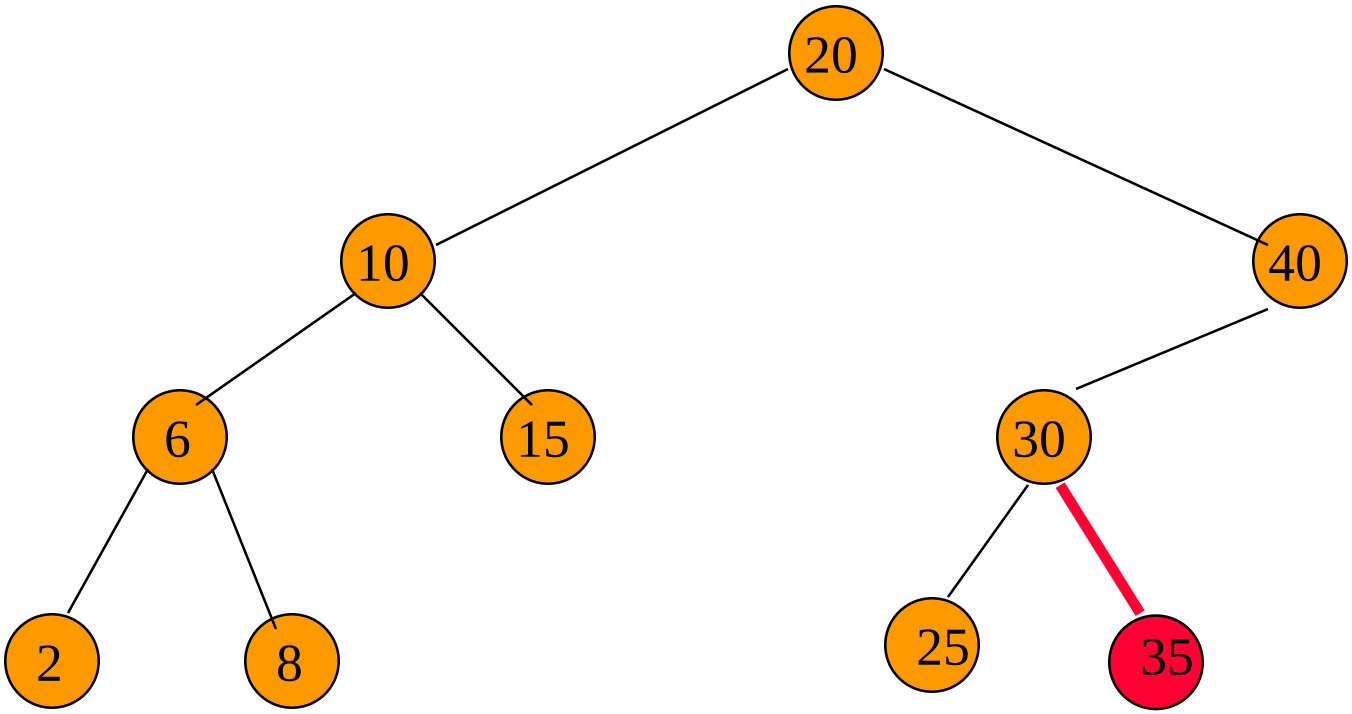
Do an inorder traversal. O(n) time.

The Operation Get()



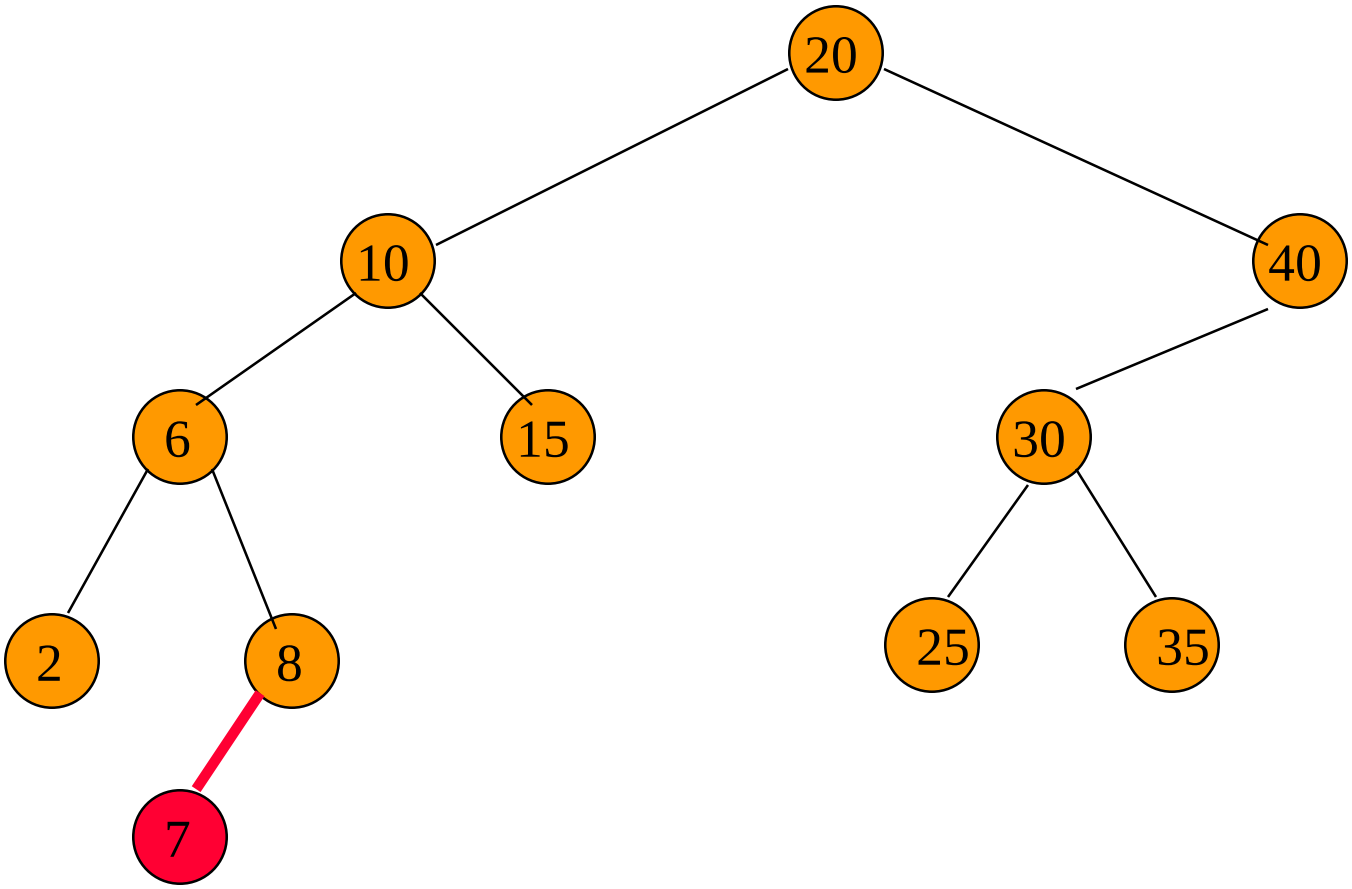
Complexity is O(height) = O(n), where n is number of nodes/elements.

The Operation Insert()



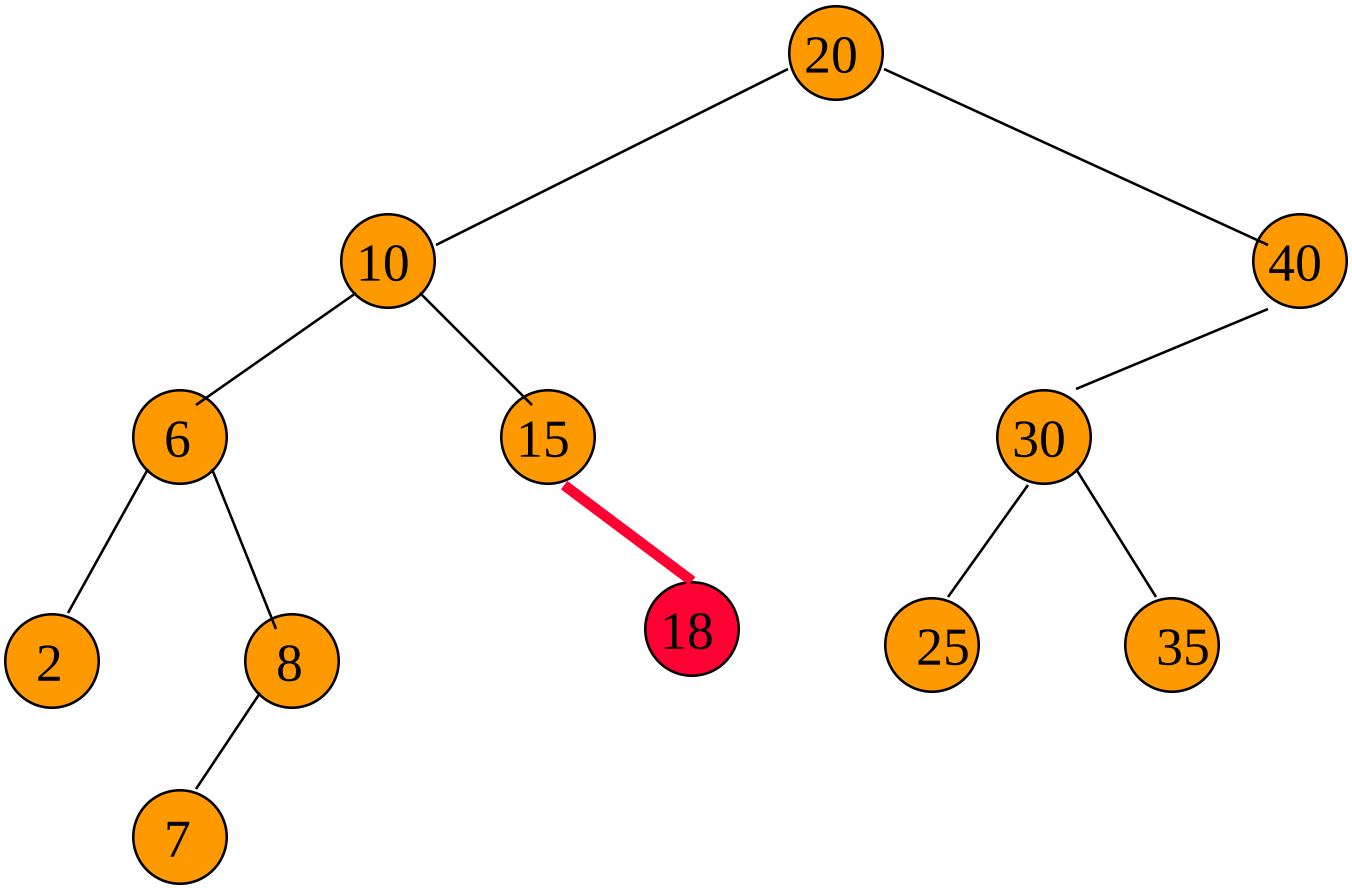
Insert a pair whose key is 35.

The Operation Insert()



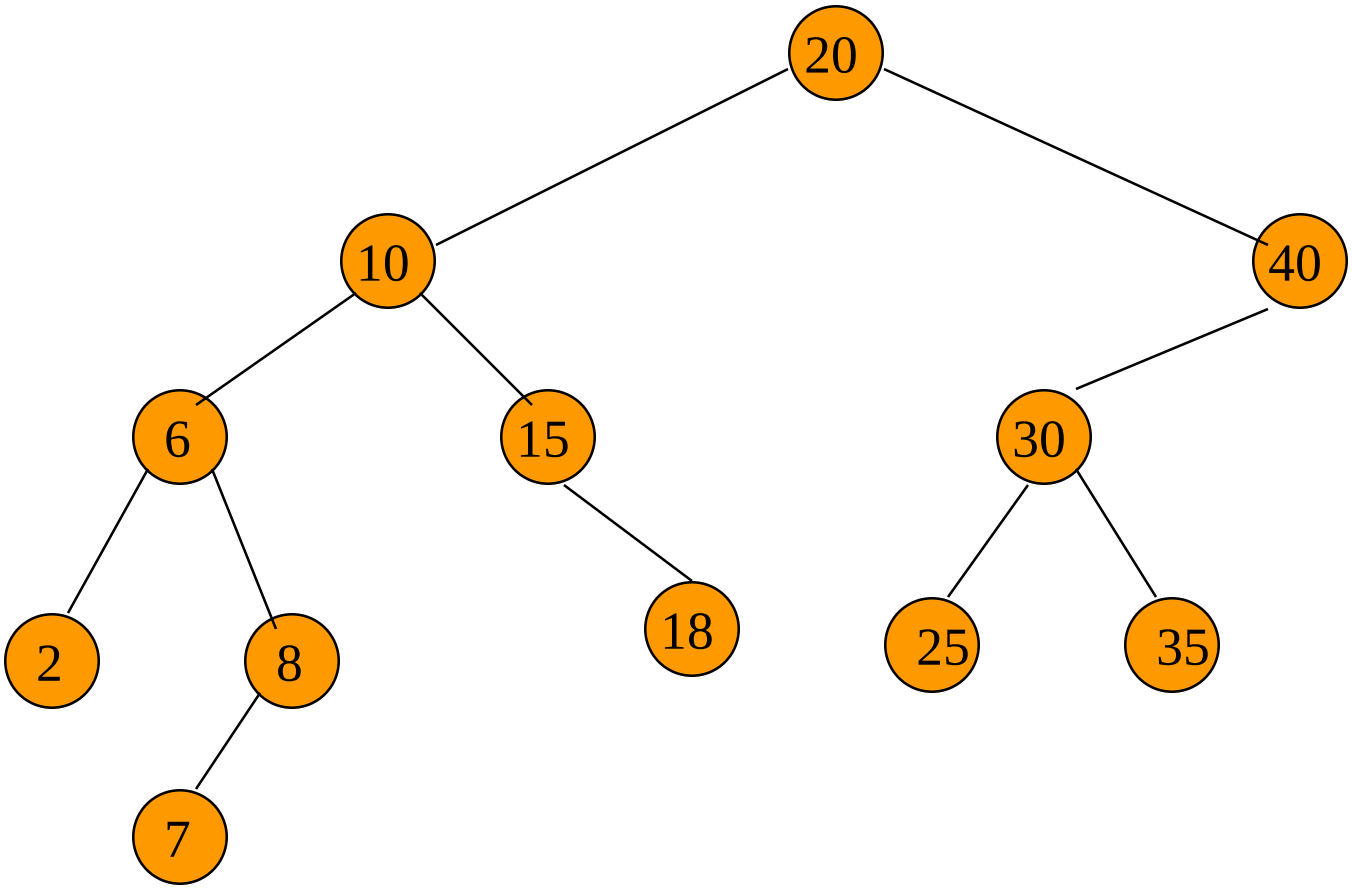
Insert a pair whose key is 7.

The Operation Insert()



Insert a pair whose key is 18.

The Operation Insert()



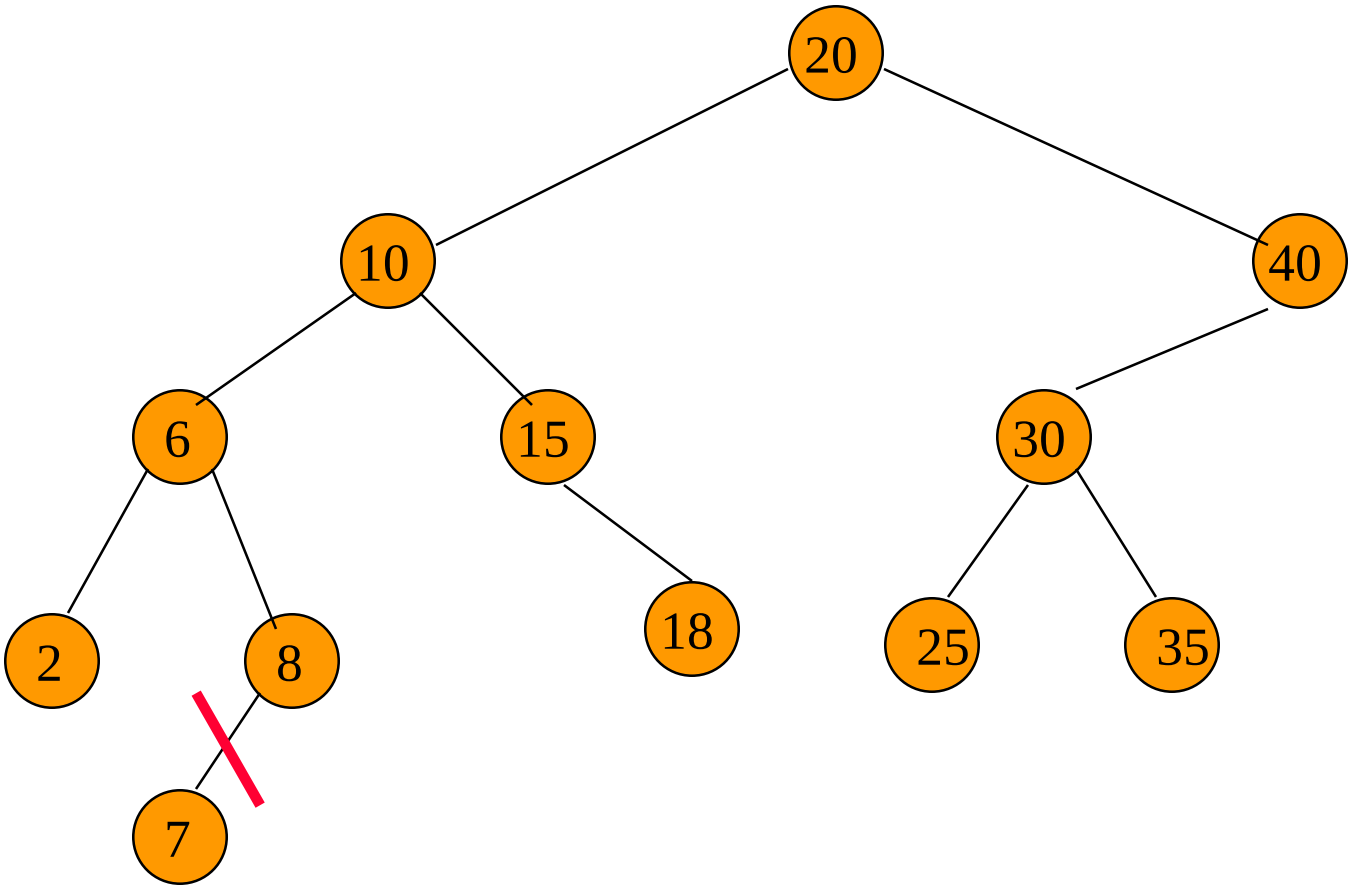
Complexity of Insert() is O(height).

The Operation Delete()

Four cases:

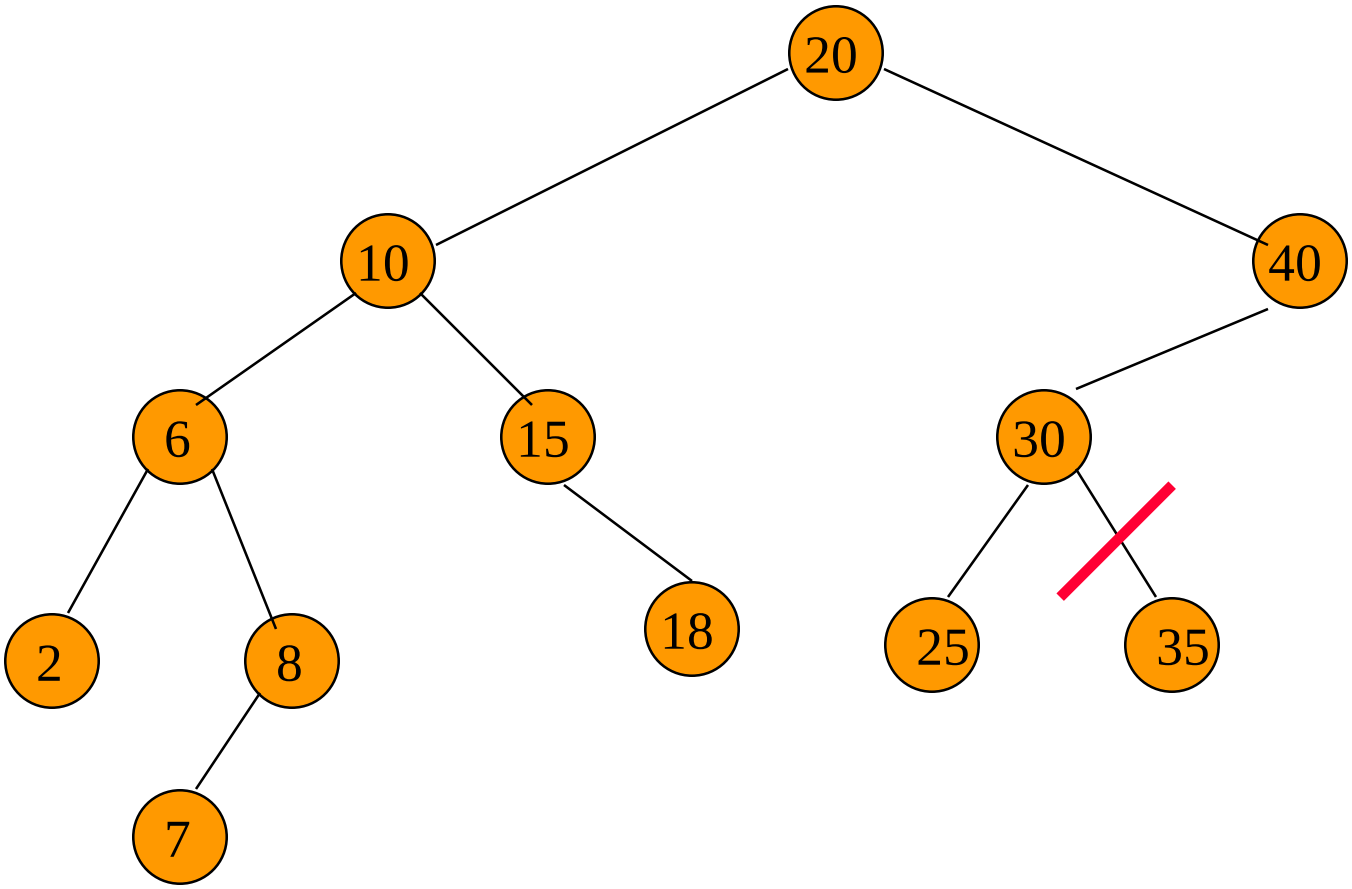
- No element with delete key.
- Element is in a leaf.
- Element is in a degree 1 node.
- Element is in a degree 2 node.

Delete From A Leaf



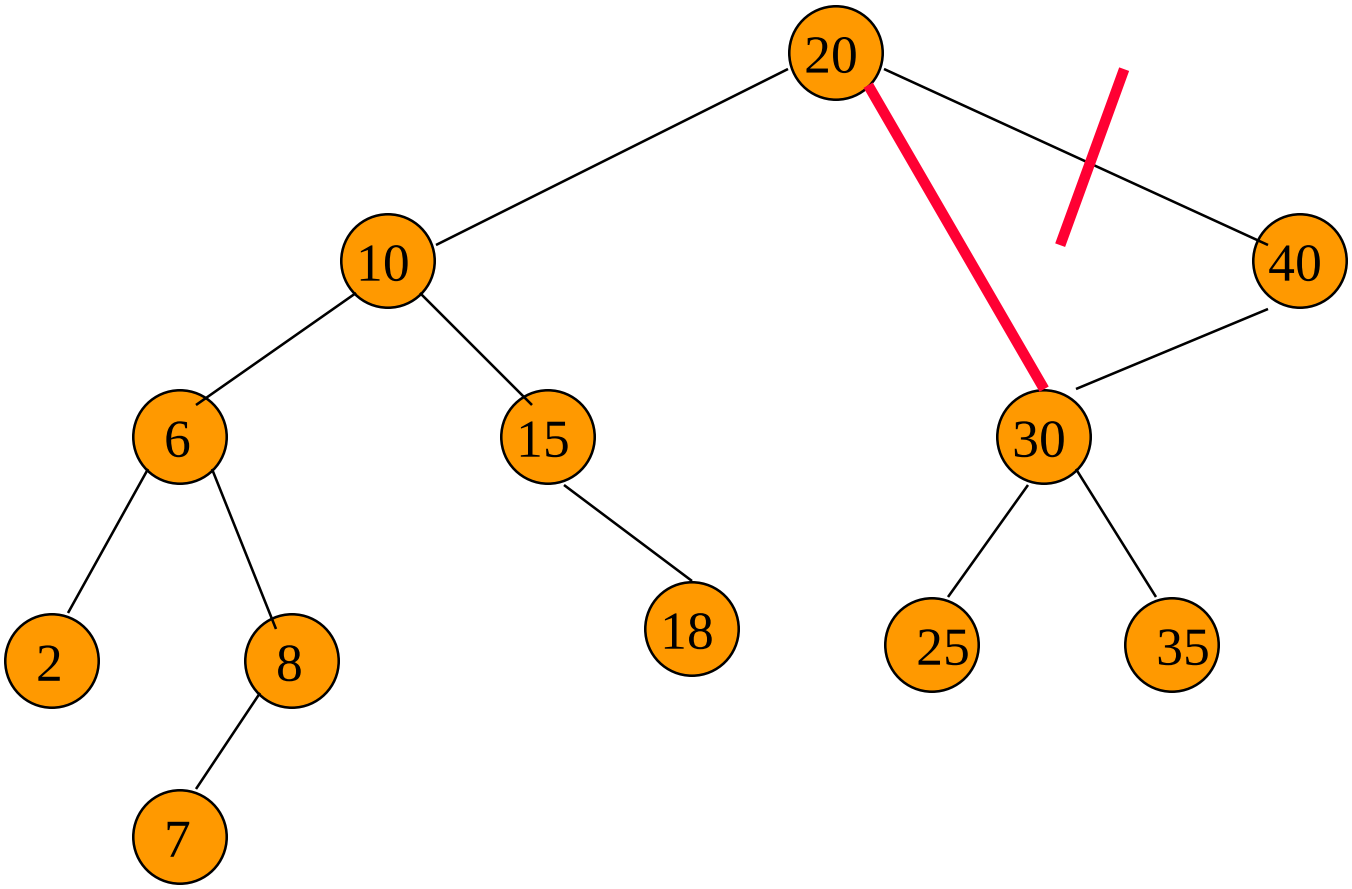
Delete a leaf element. key = 7

Delete From A Leaf (contd.)



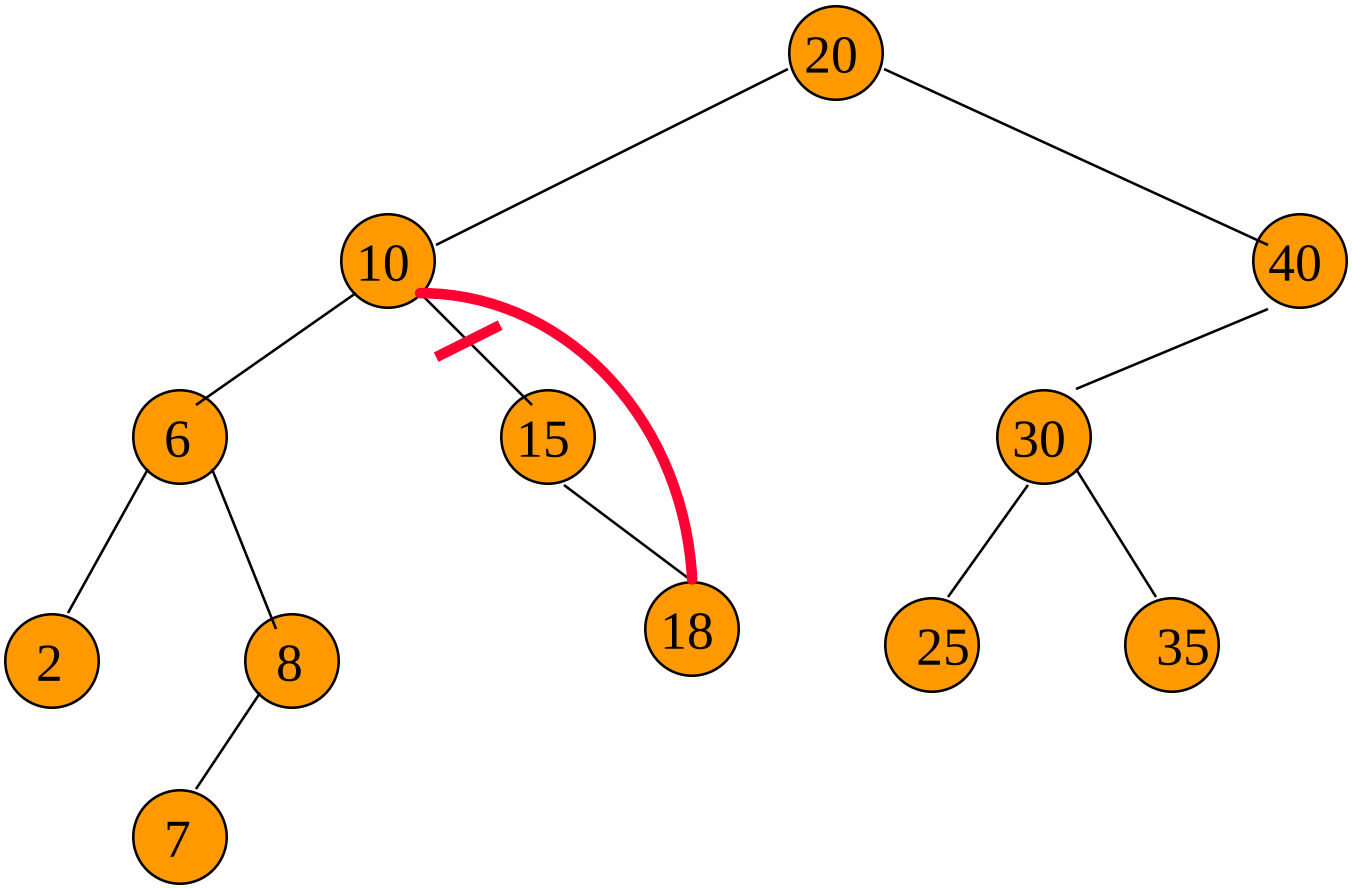
Delete a leaf element. key = 35

Delete From A Degree 1 Node



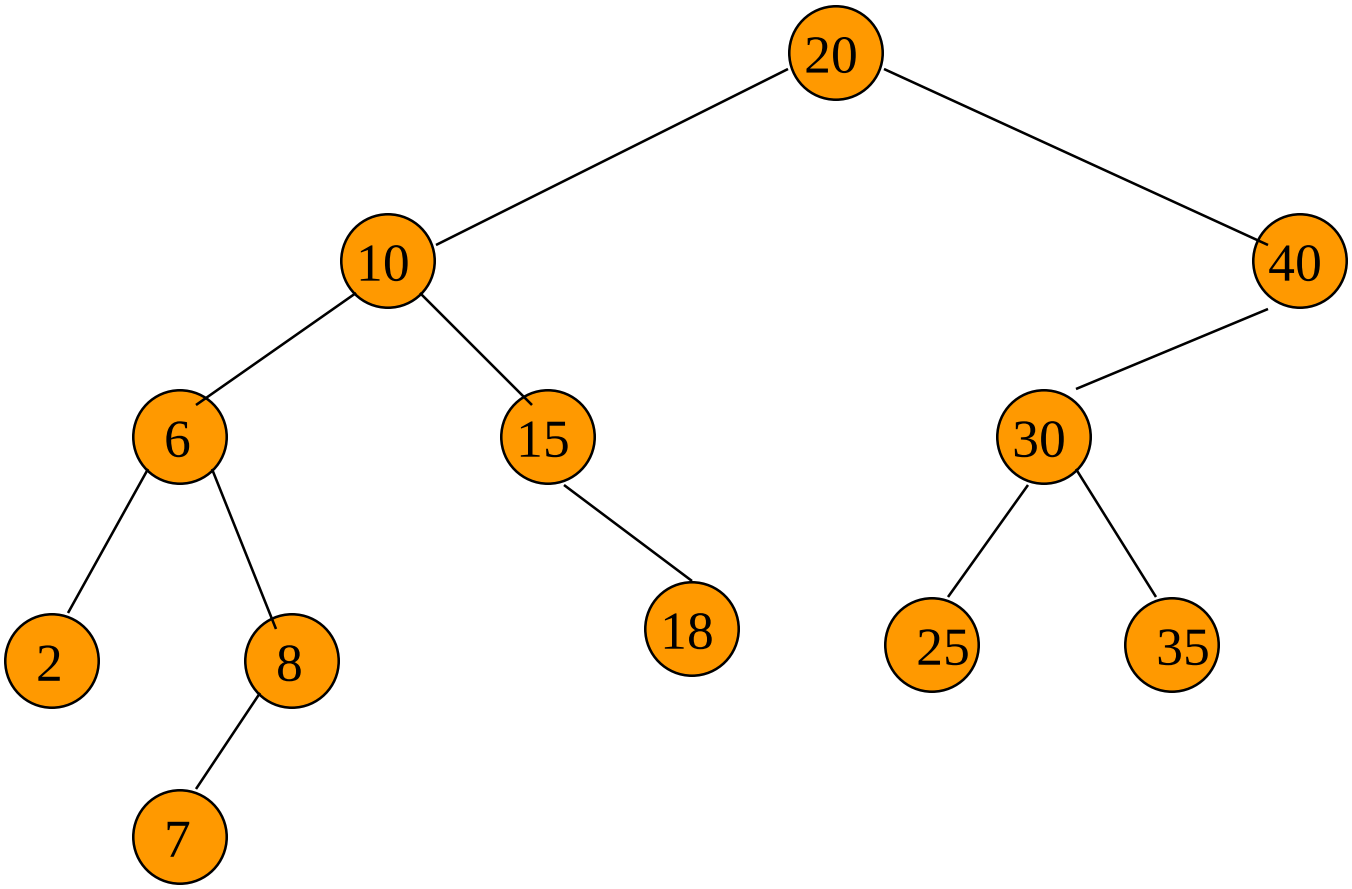
Delete from a degree 1 node. key = 40

Delete From A Degree 1 Node (contd.)



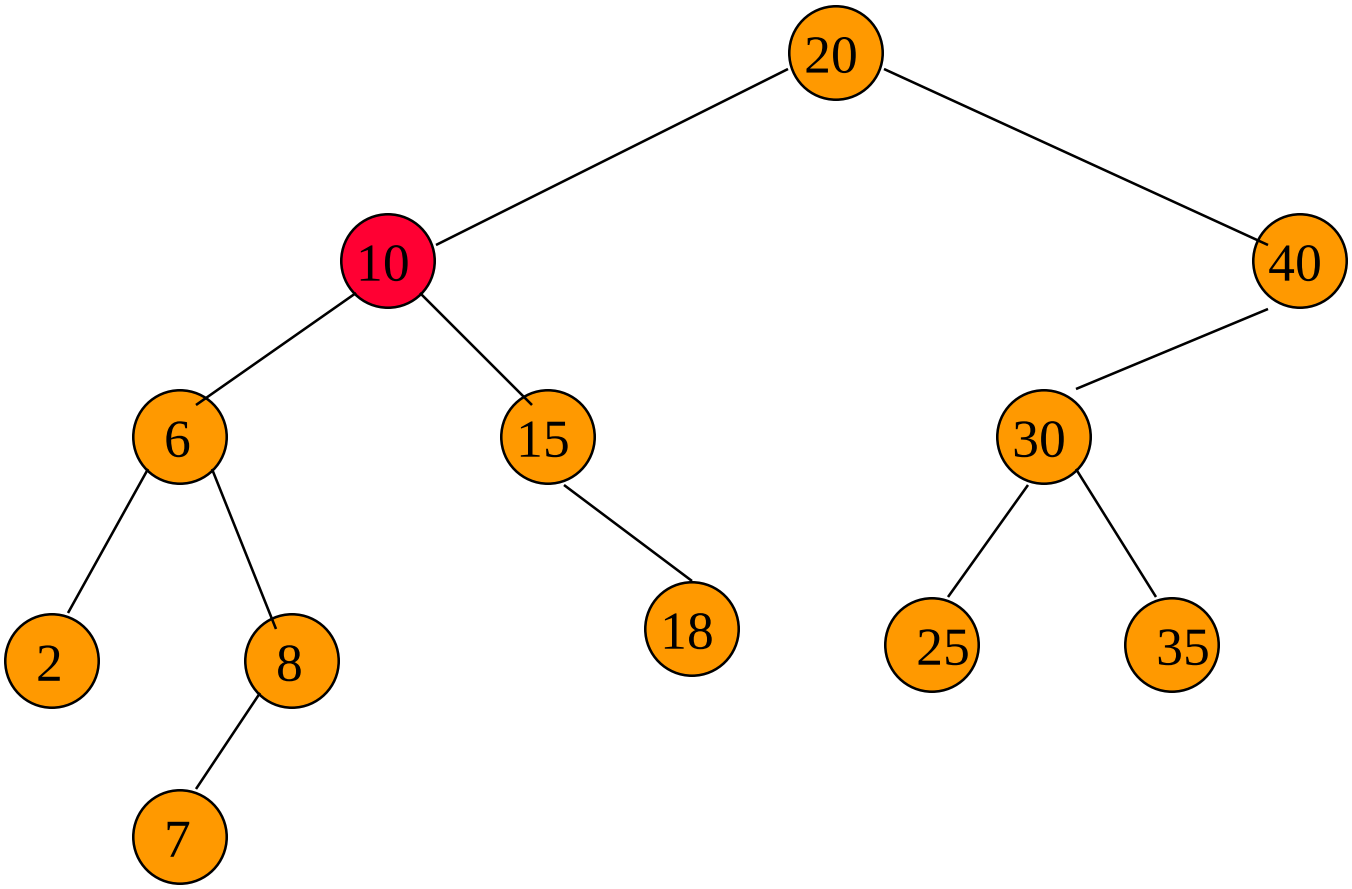
Delete from a degree 1 node. key = 15

Delete From A Degree 2 Node



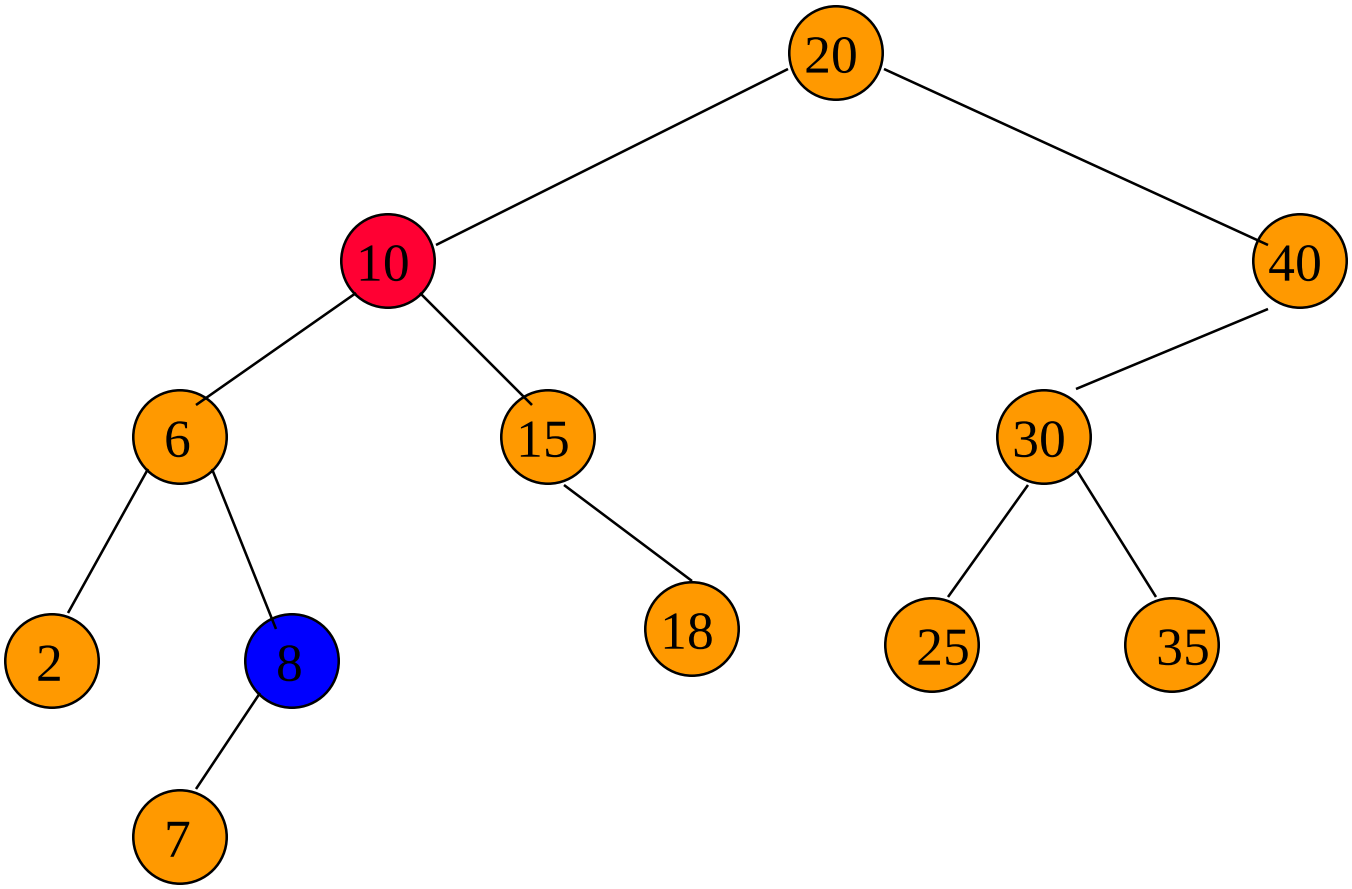
Delete from a degree 2 node. key = 10

Delete From A Degree 2 Node



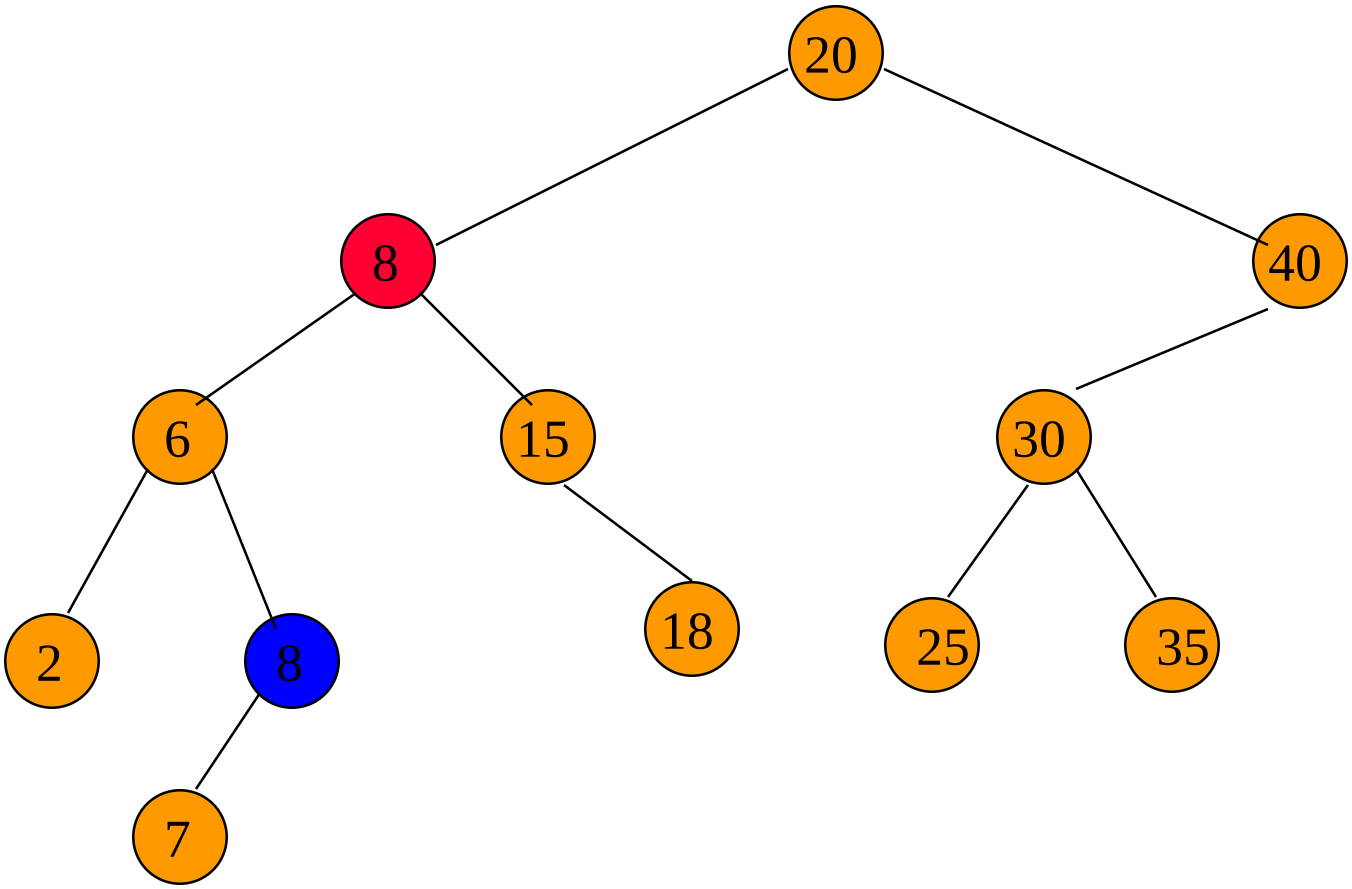
Replace with largest key in left subtree (or smallest in right subtree).

Delete From A Degree 2 Node



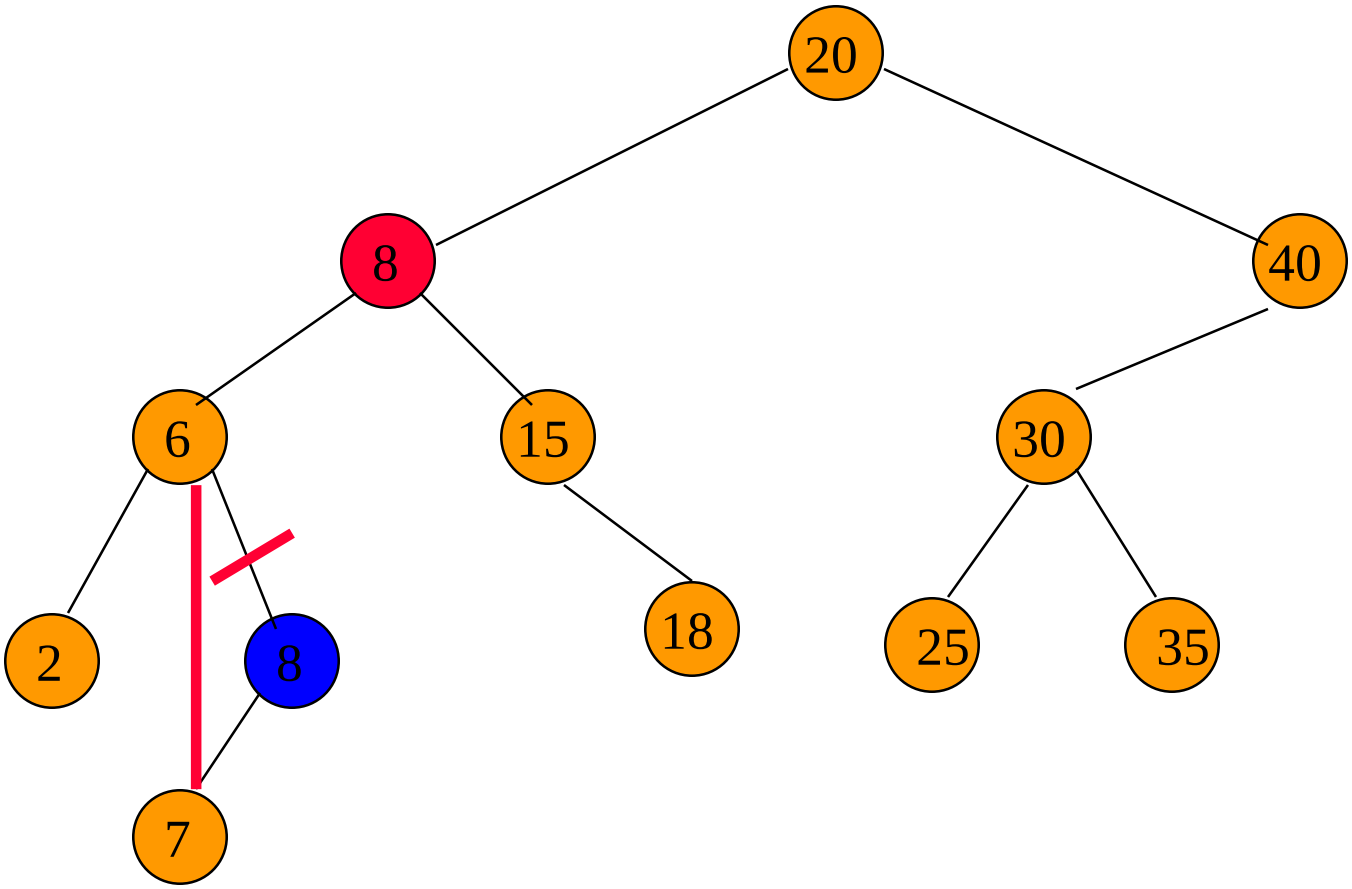
Replace with largest key in left subtree (or smallest in right subtree).

Delete From A Degree 2 Node



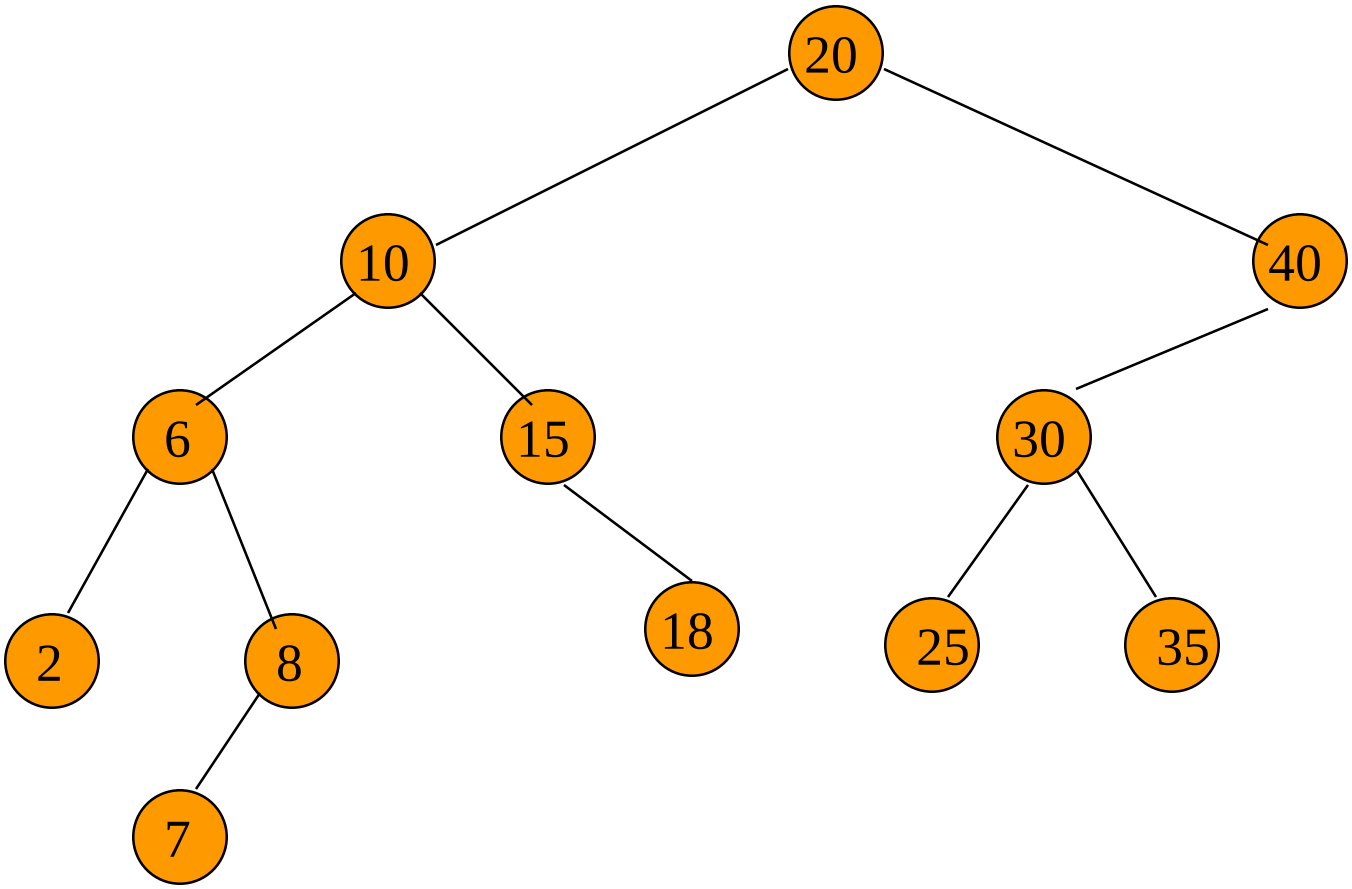
Replace with largest key in left subtree (or smallest in right subtree).

Delete From A Degree 2 Node



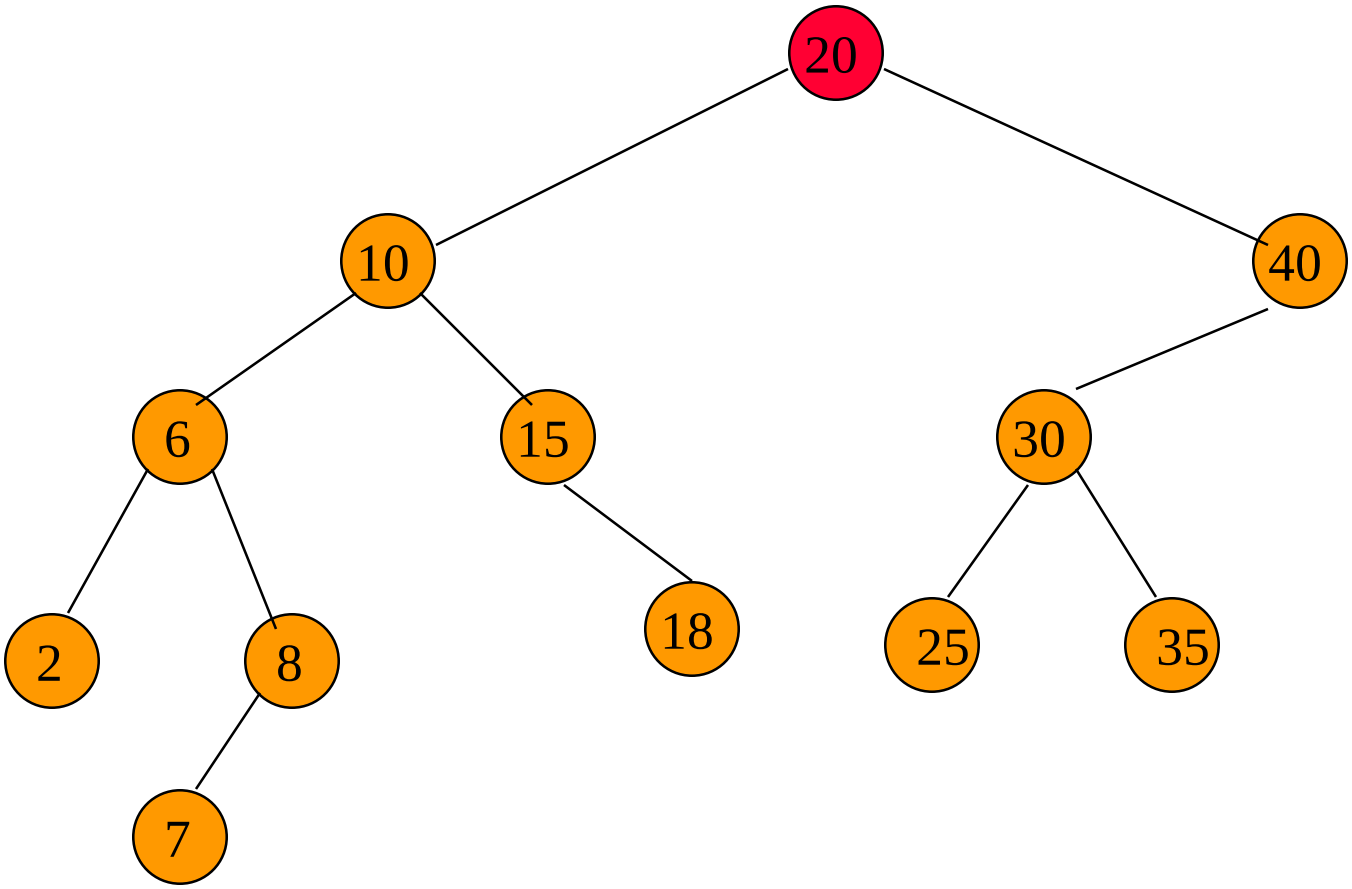
Largest key must be in a leaf or degree 1 node.

Another Delete From A Degree 2 Node



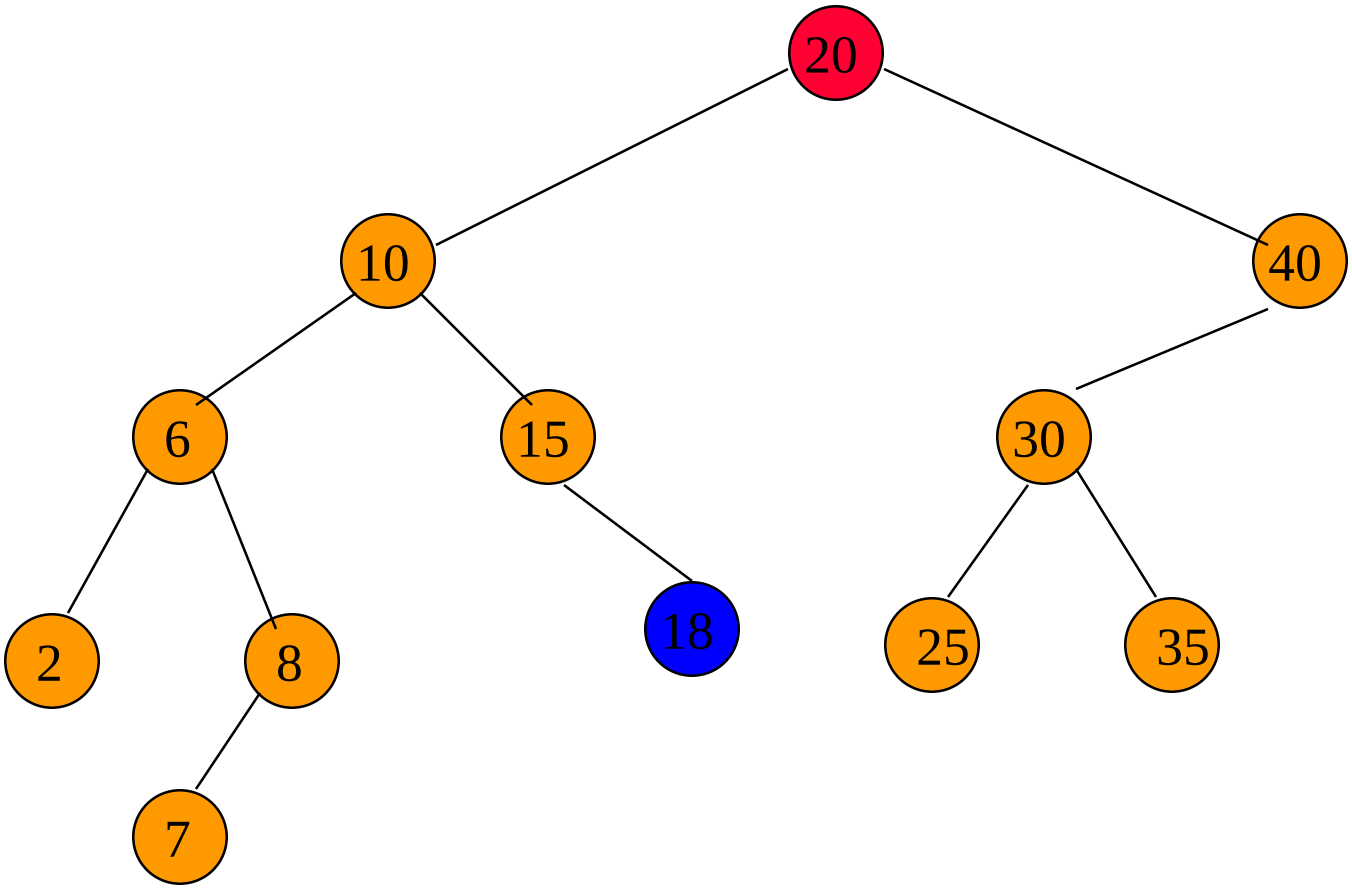
Delete from a degree 2 node. key = 20

Delete From A Degree 2 Node



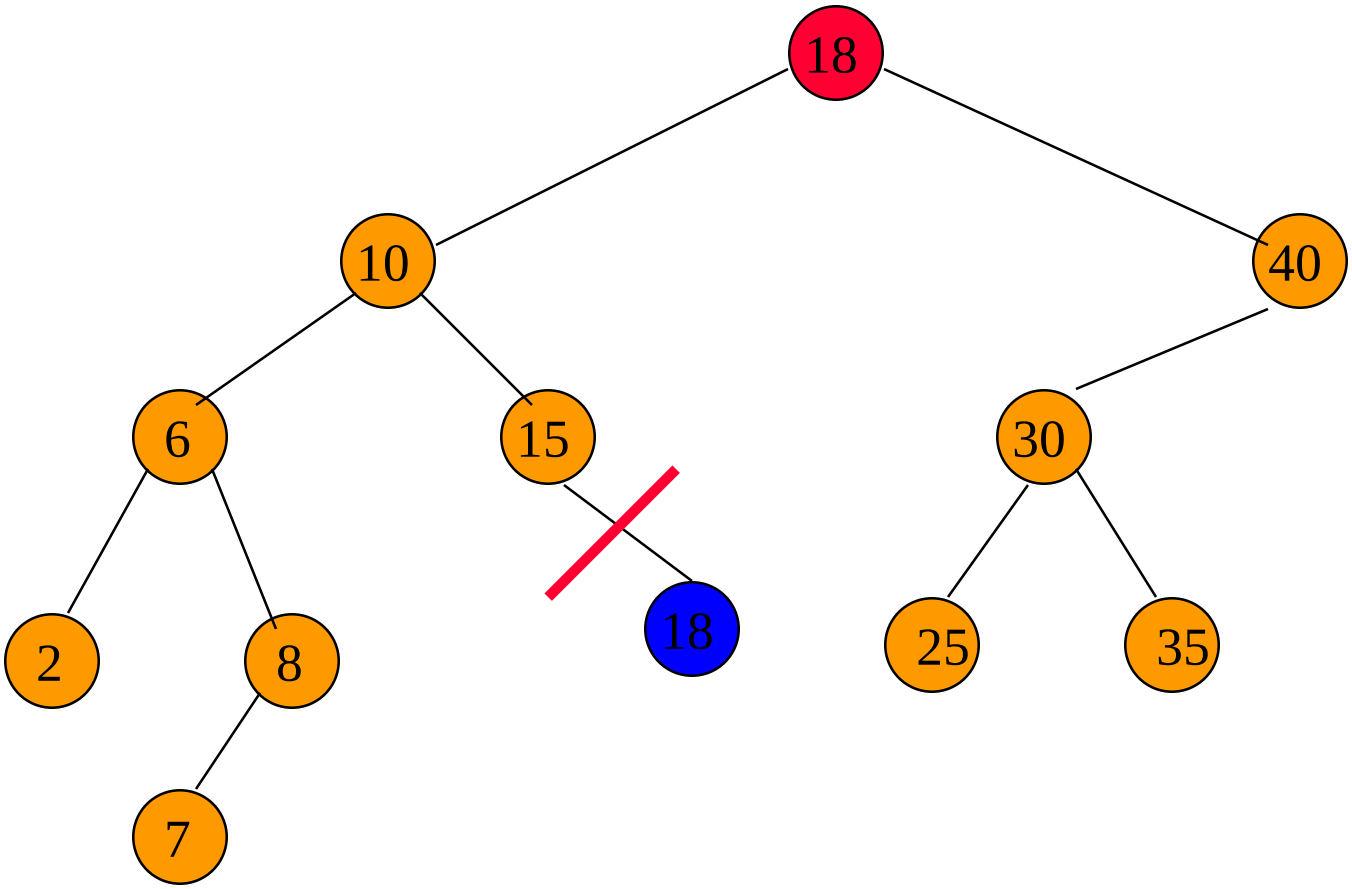
Replace with largest in left subtree.

Delete From A Degree 2 Node



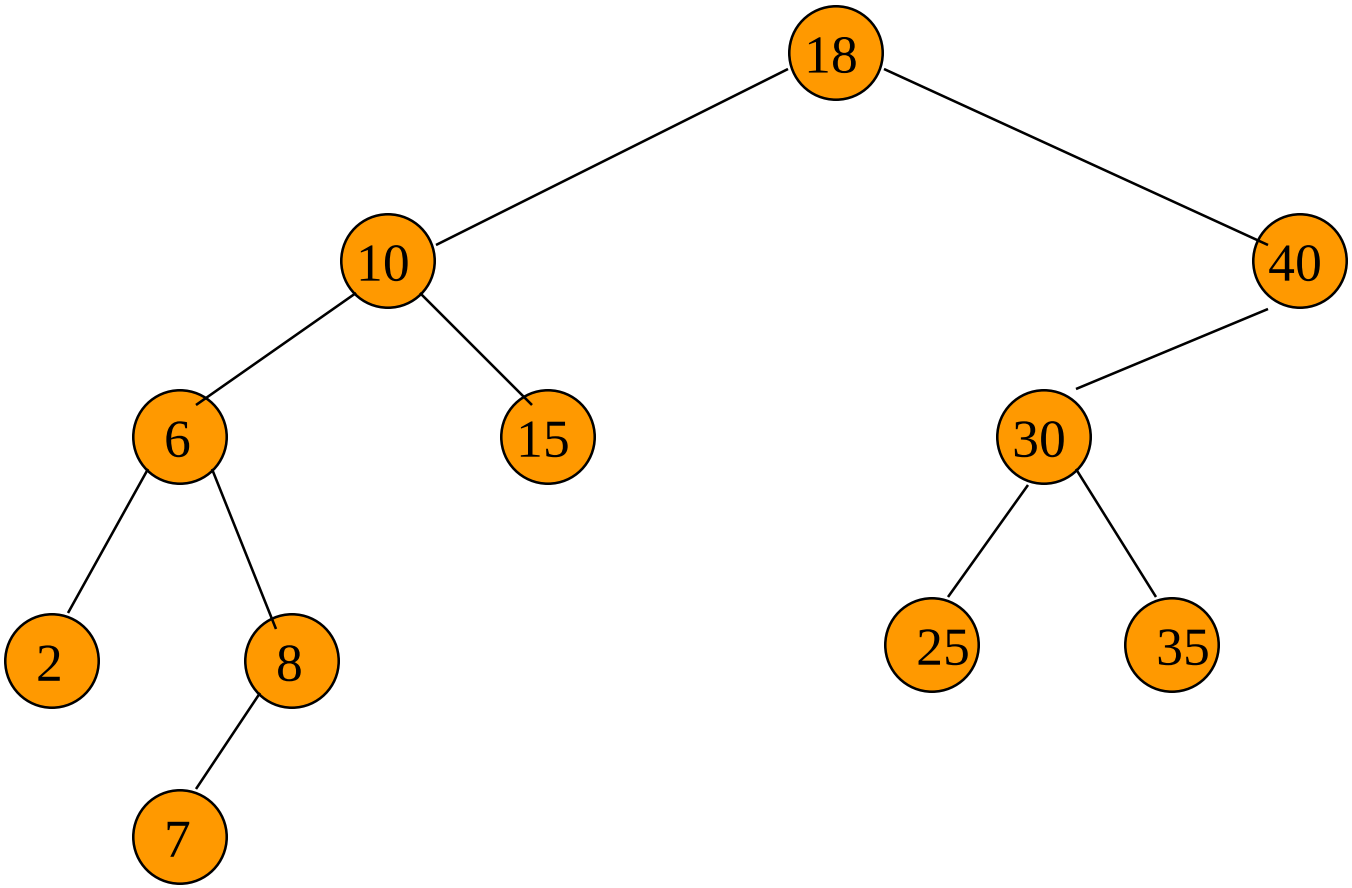
Replace with largest in left subtree.

Delete From A Degree 2 Node



Replace with largest in left subtree.

Delete From A Degree 2 Node

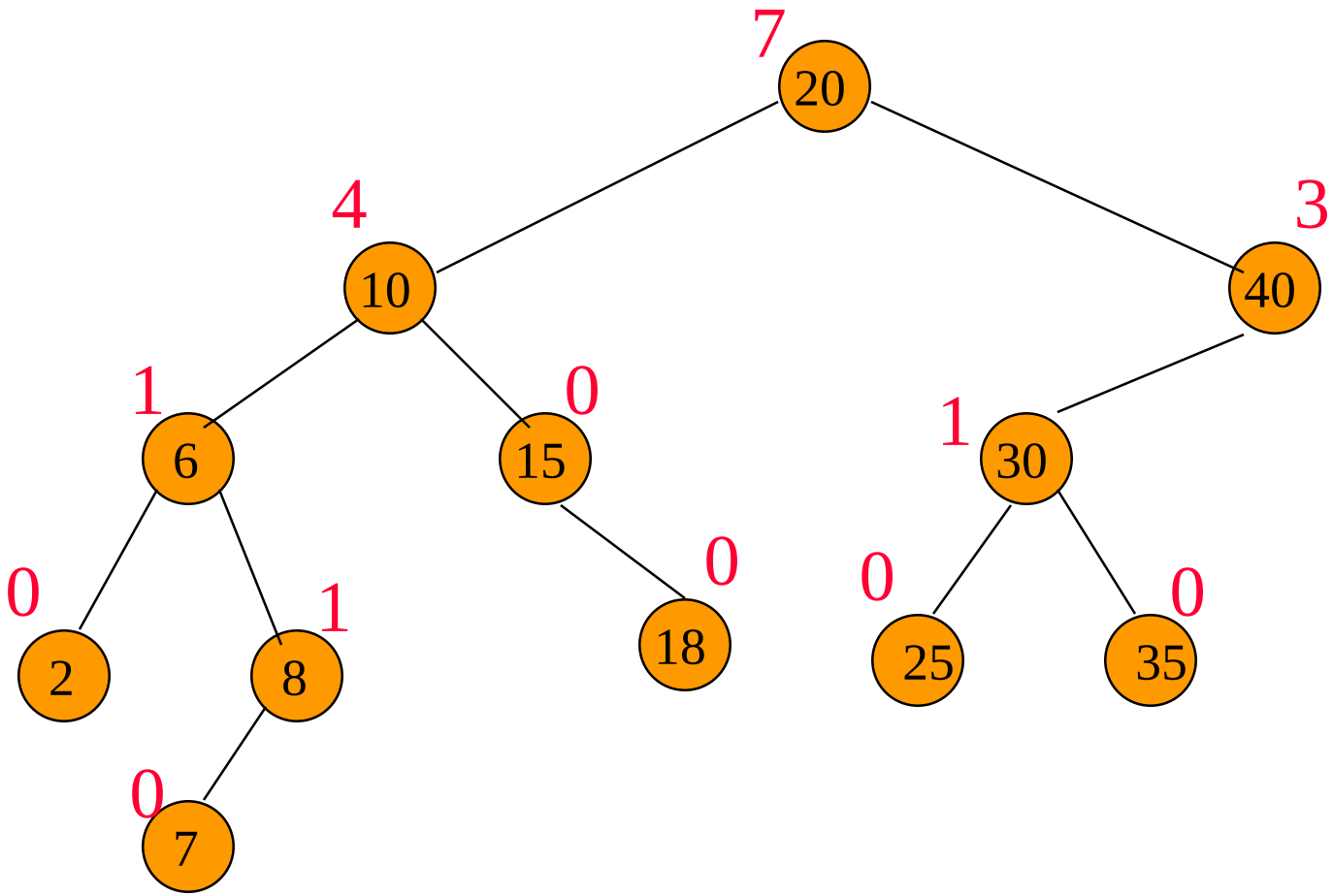


Complexity is O(height).

Indexed Binary Search Tree

- Binary search tree.
- Each node has an additional field.
  - leftSize = number of nodes in its left subtree

Example Indexed Binary Search Tree



leftSize values are in red

leftSize And Rank

Rank of an element is its position in inorder (inorder = ascending key order).

[2,6,7,8,10,15,18,20,25,30,35,40]

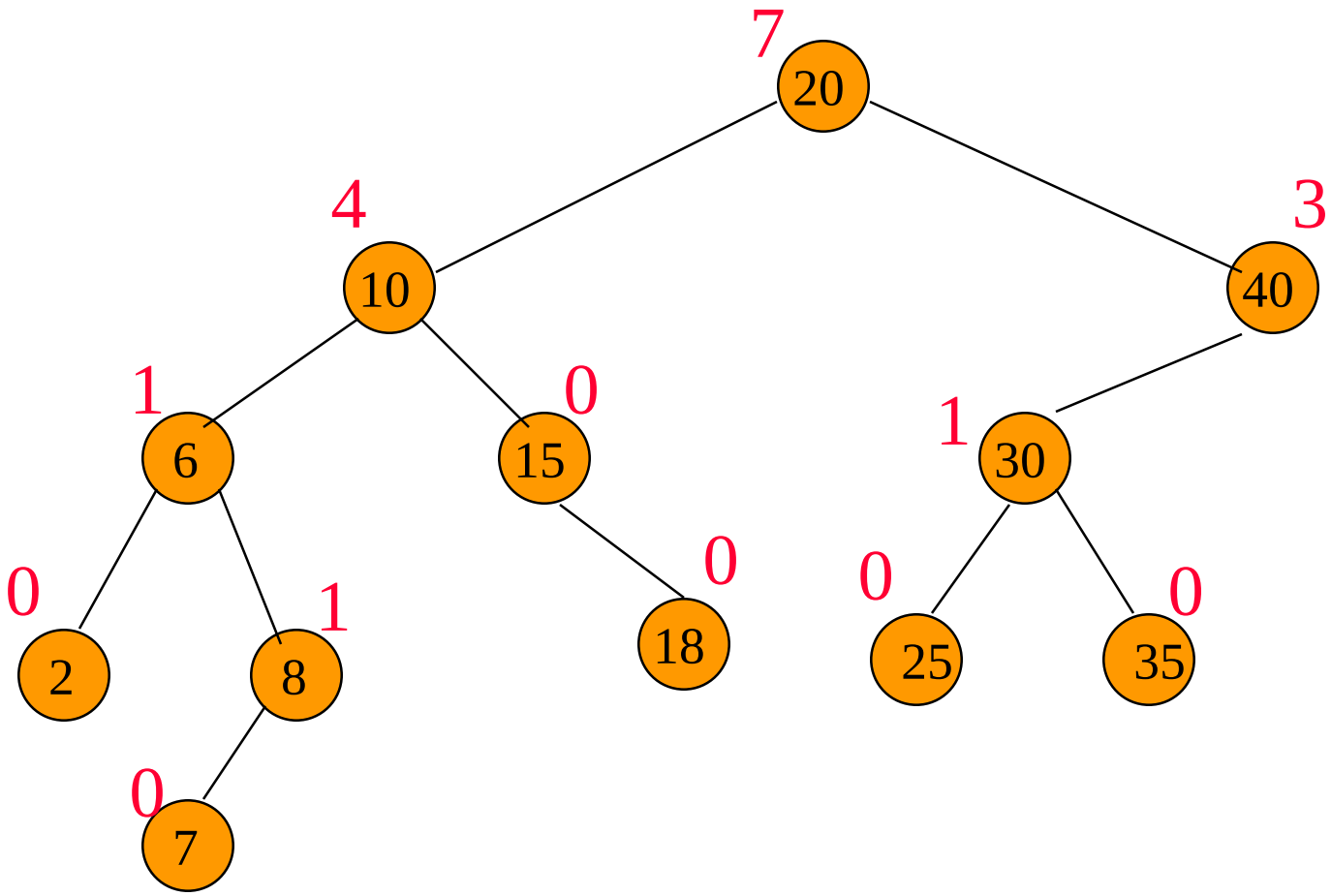
rank(2) = 0

rank(15) = 5

rank(20) = 7

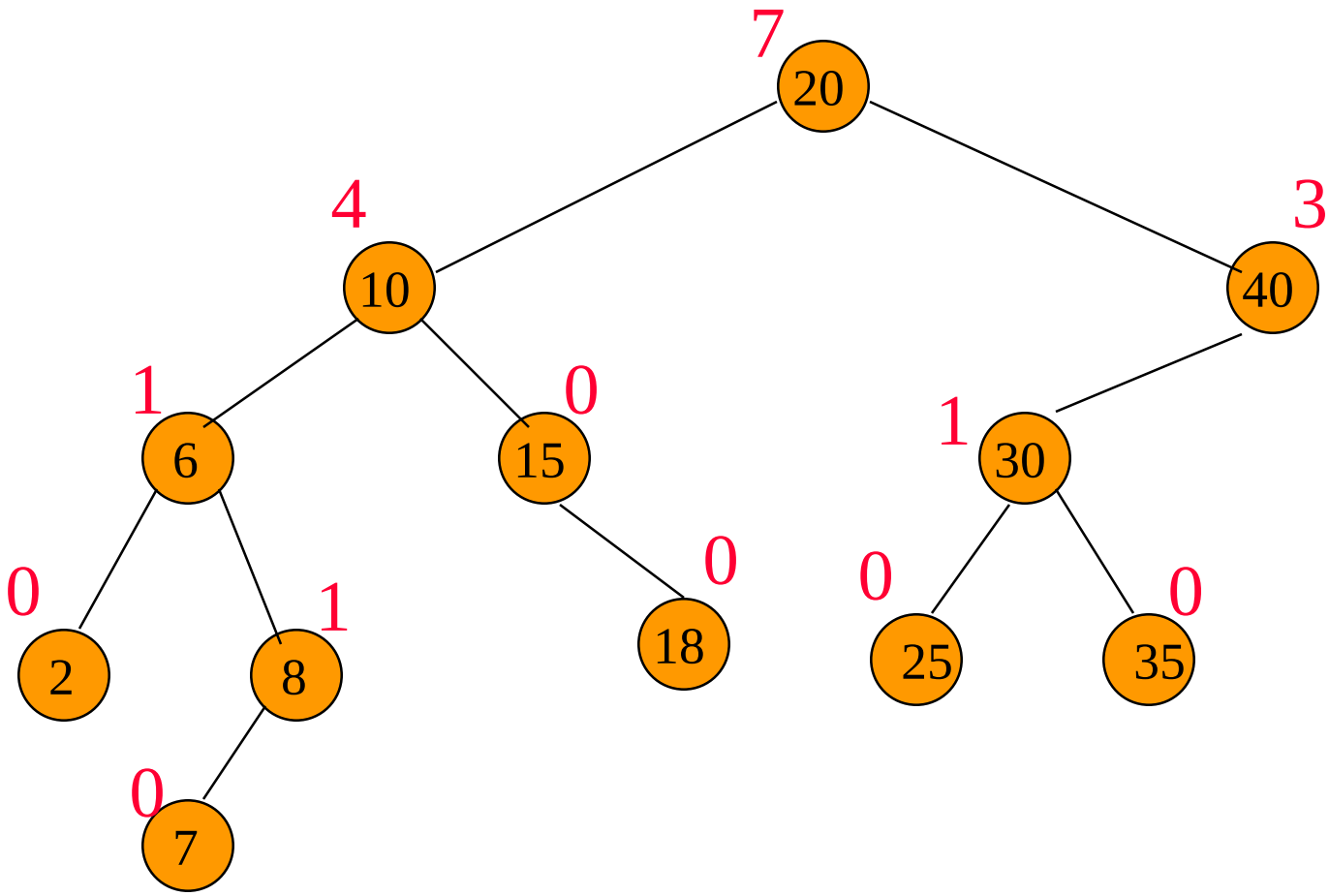
leftSize(x) = rank(x) with respect to elements in subtree rooted at x

leftSize And Rank



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

Get(index) And Delete(index)



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

Get(index) And Delete(index)

- if index = x.leftSize desired element is x.element
- if index < x.leftSize desired element is index'th element in left subtree of x
- if index > x.leftSize desired element is (index - x.leftSize-1)'th element in right subtree of x

Applications

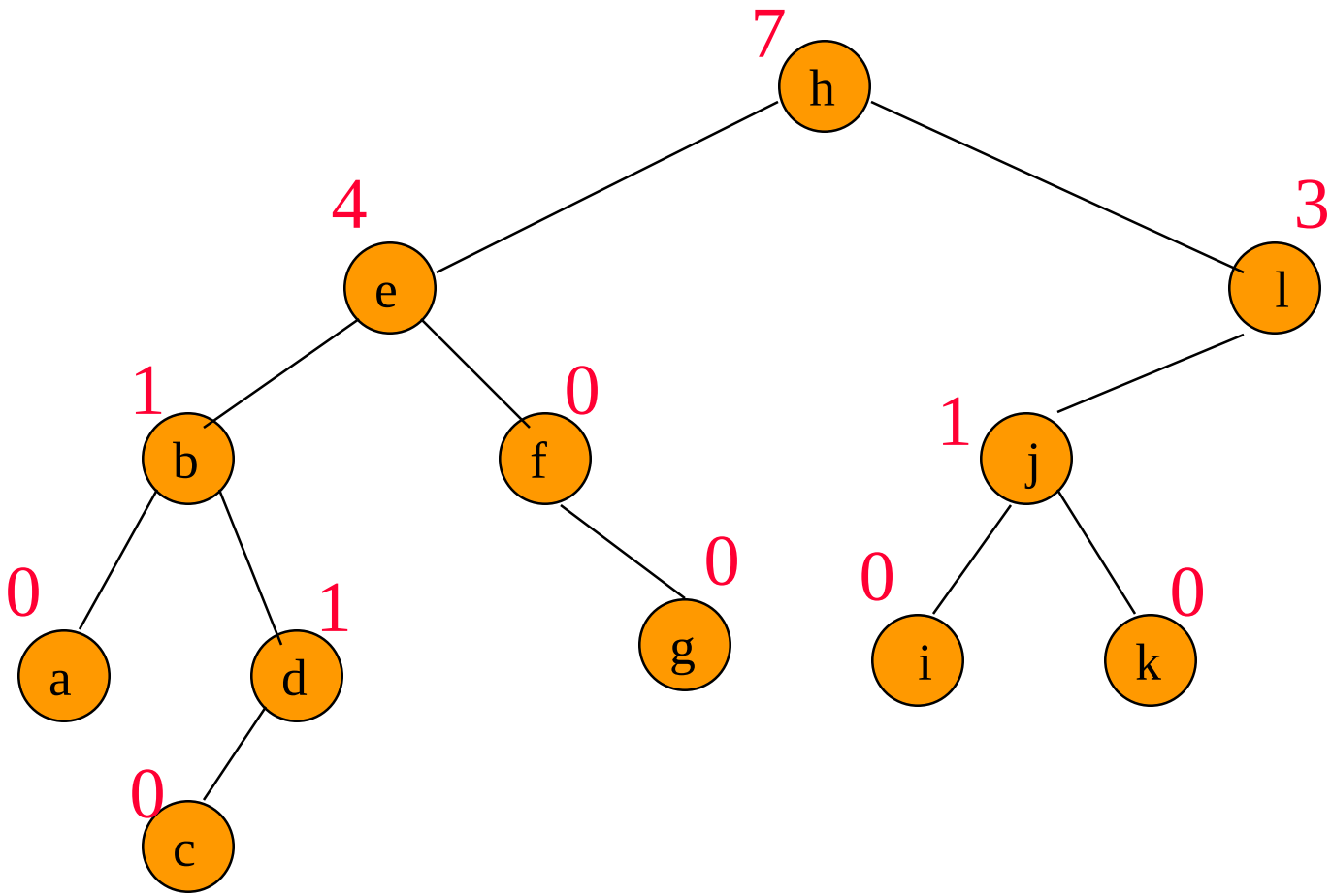
(Complexities Are For Balanced Trees)

Best-fit bin packing in O(n log n) time.

Representing a linear list so that Get(index), Insert(index, element), and Delete(index) run in O(log(list size)) time (uses an indexed binary tree, not indexed binary search tree).

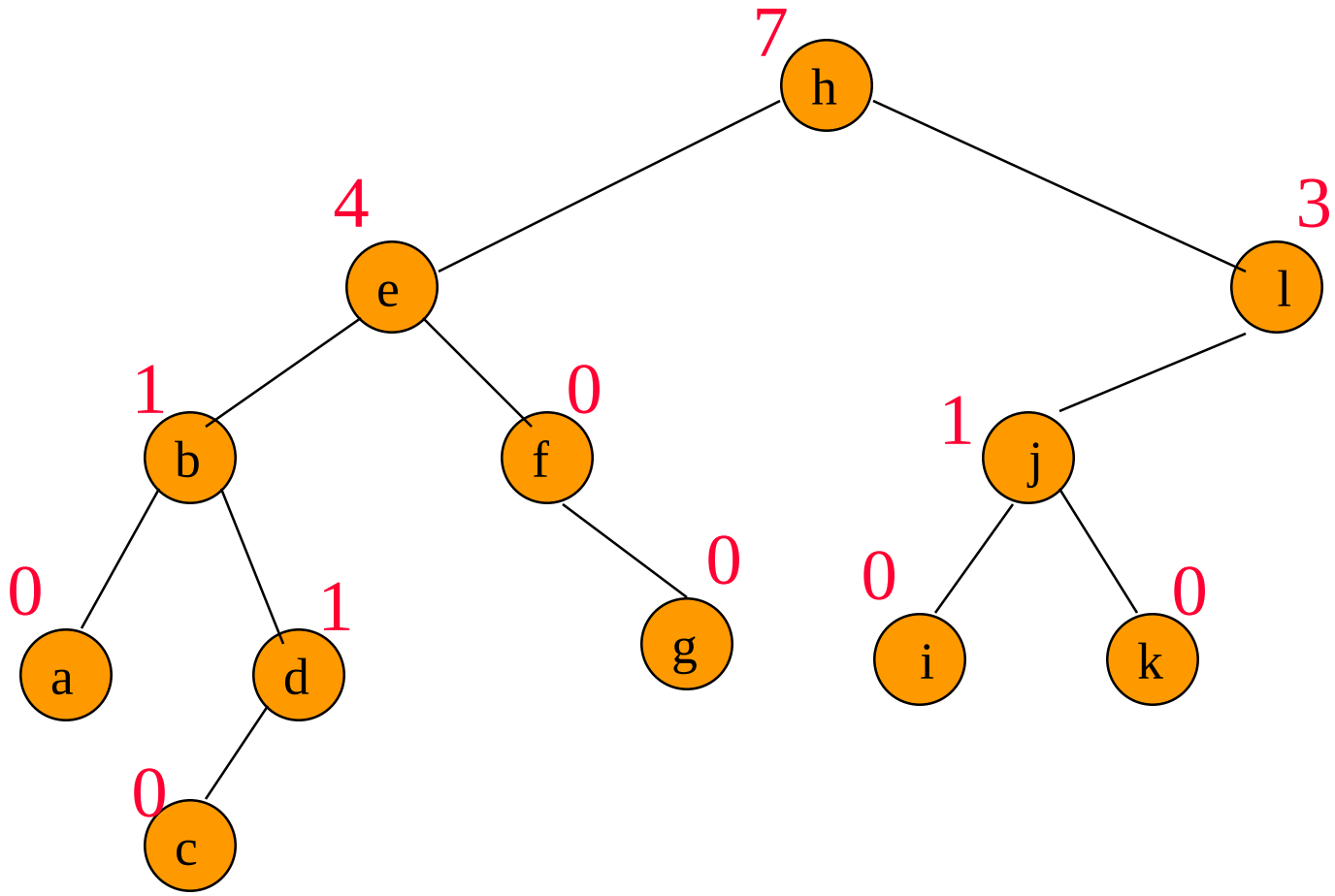
Can't use hash tables for either of these applications.

Linear List As Indexed Binary Tree



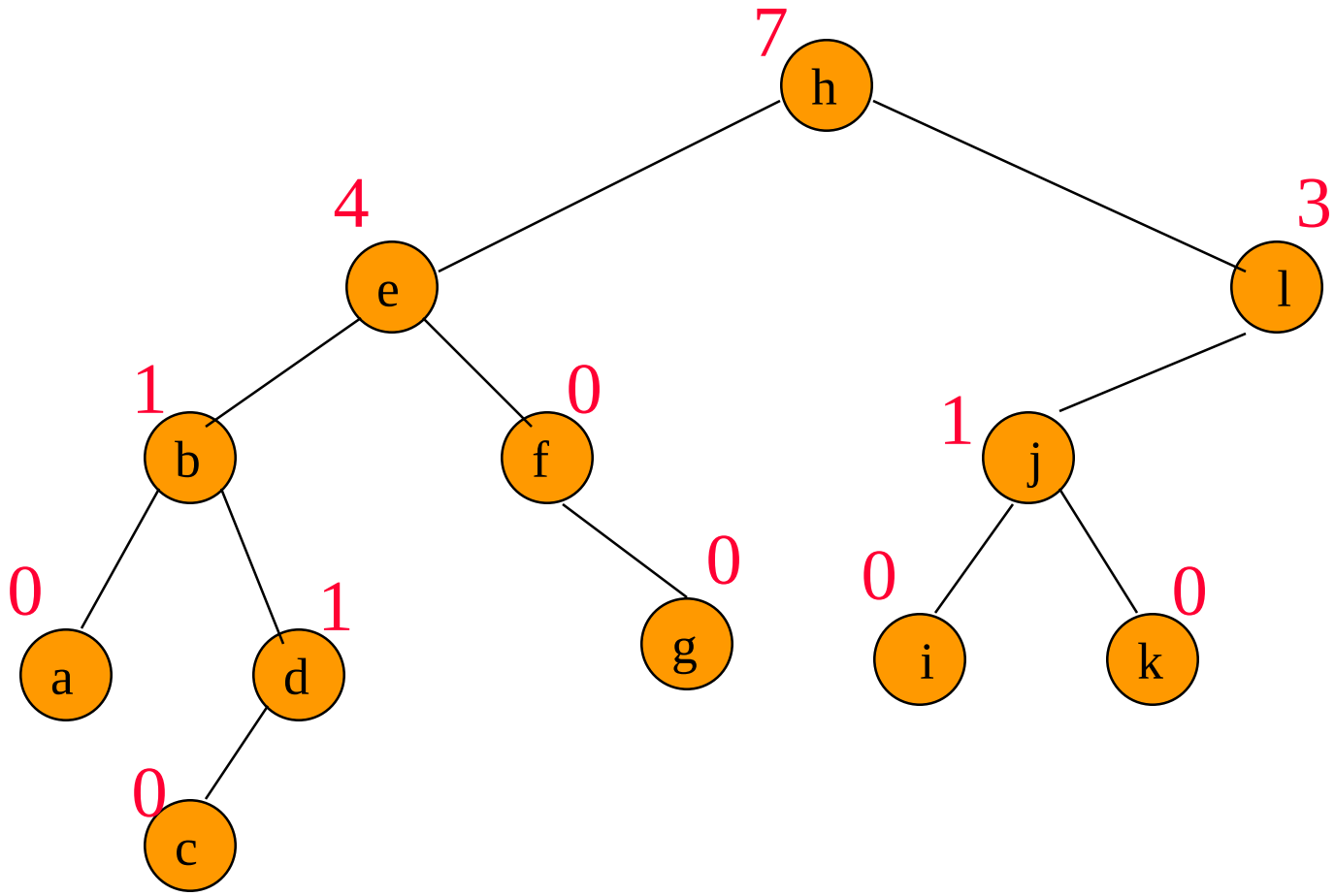
list = [a,b,c,d,e,f,g,h,i,j,k,l]

Insert(5,'m')



list = [a,b,c,d,e,f,g,h,i,j,k,l]

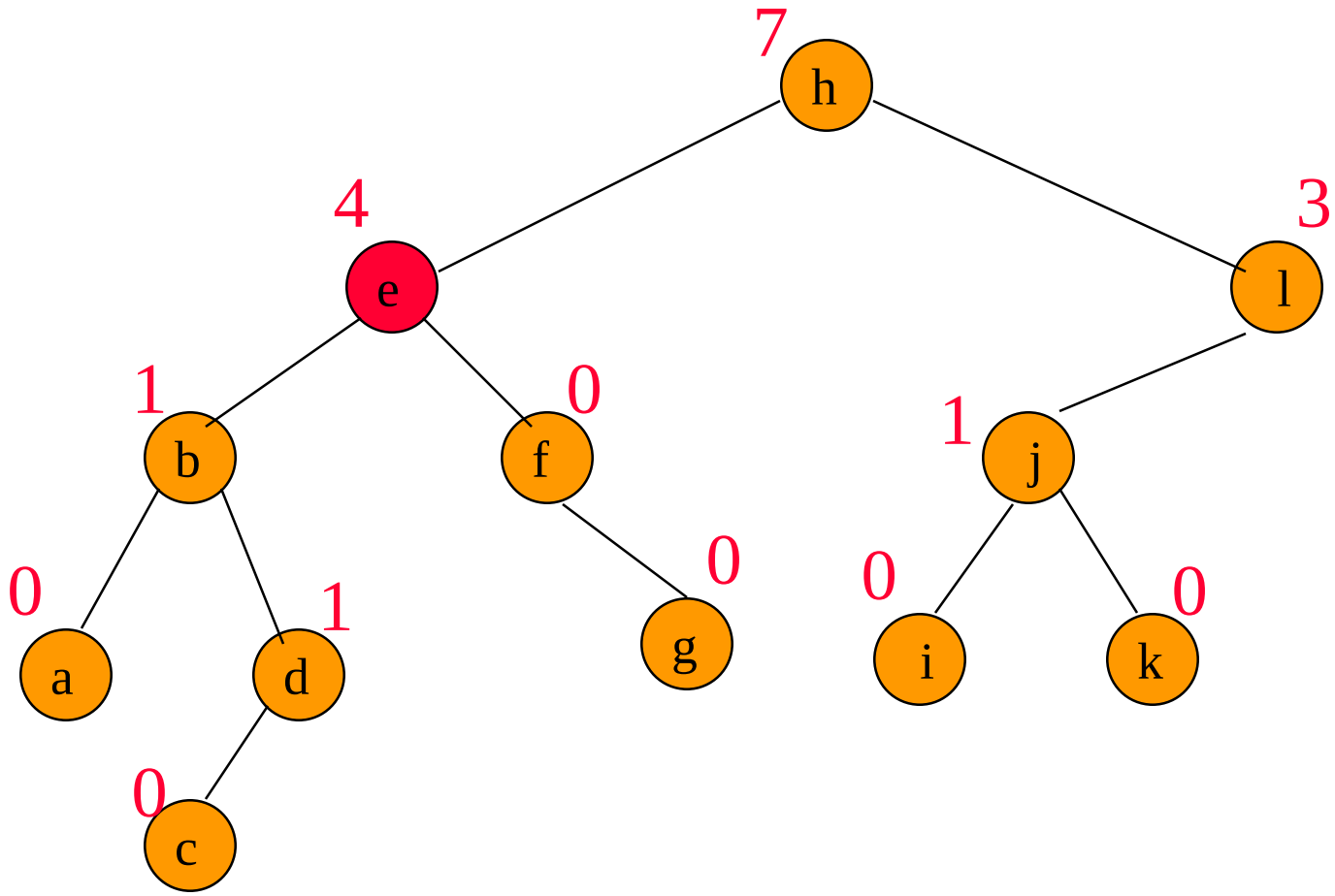
Insert(5,'m')



list = [a,b,c,d,e, m,f,g,h,i,j,k,l]

find node with element 4 (e)

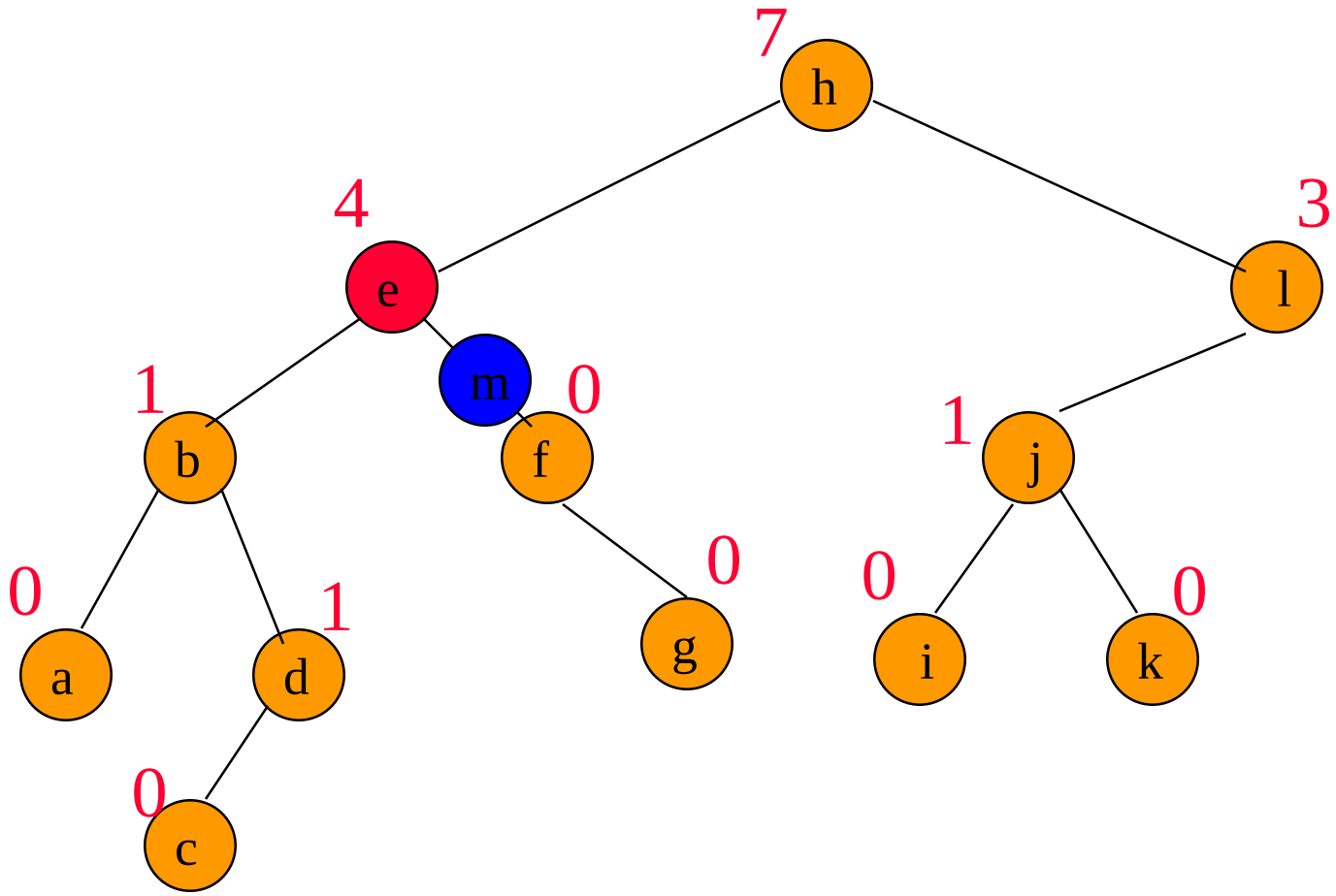
Insert(5,'m')



list = [a,b,c,d,e, m,f,g,h,i,j,k,l]

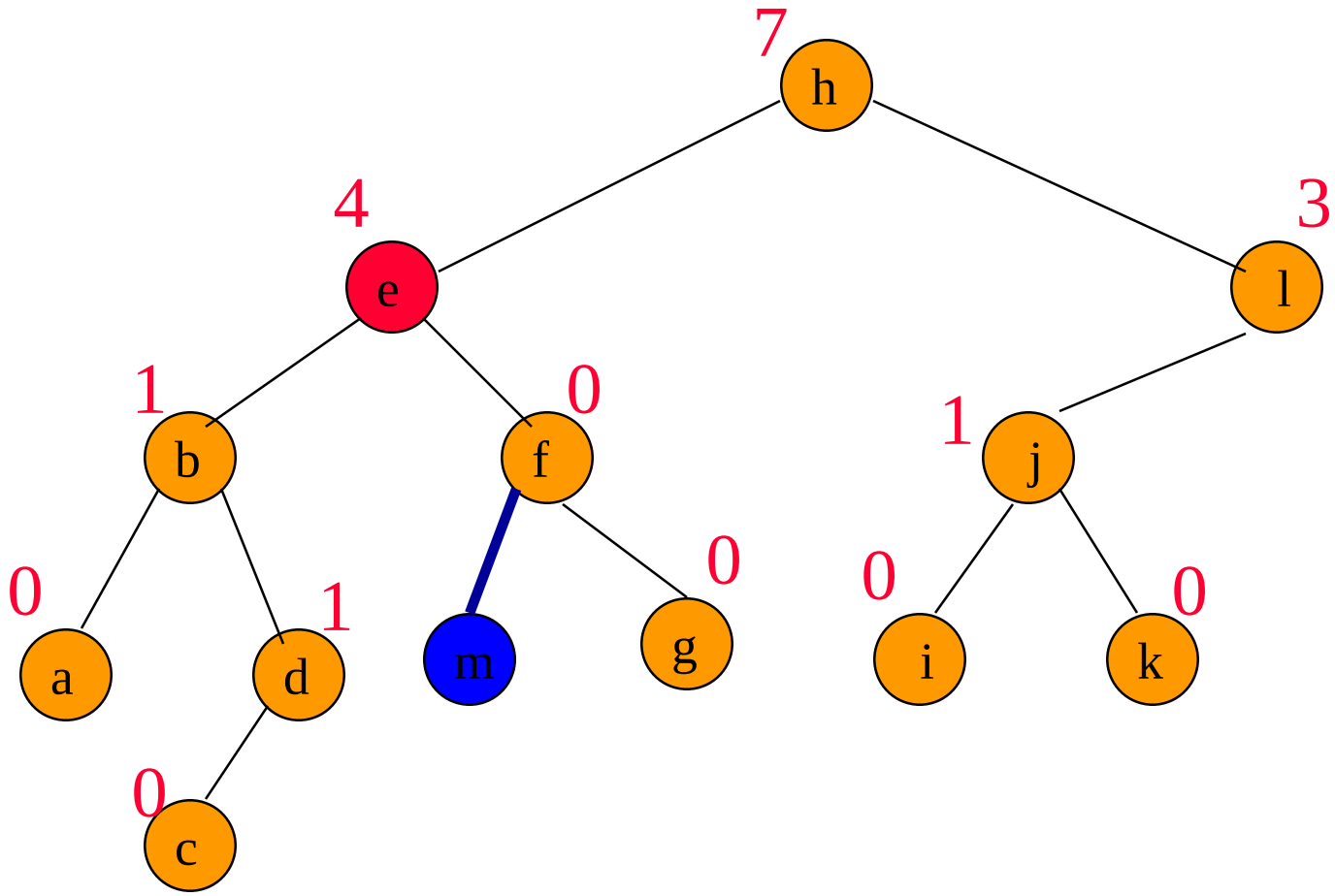
find node with element 4 (e)

Insert(5,'m')



add m as right child of e; former right subtree of e becomes right subtree of m

Insert(5,'m')



add m as leftmost node in right subtree of e

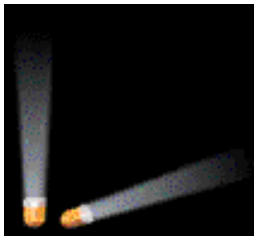
Insert(5,'m')

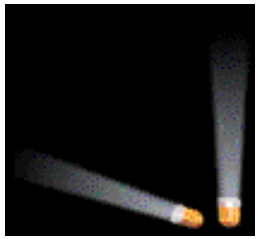
- Other possibilities exist.
- Must update some leftSize values on path from root to new node.
- Complexity is O(height).

Section 7-3

An Introduction to Selection Trees

Selection Trees





Winner trees.

Loser Trees.

Winner Trees

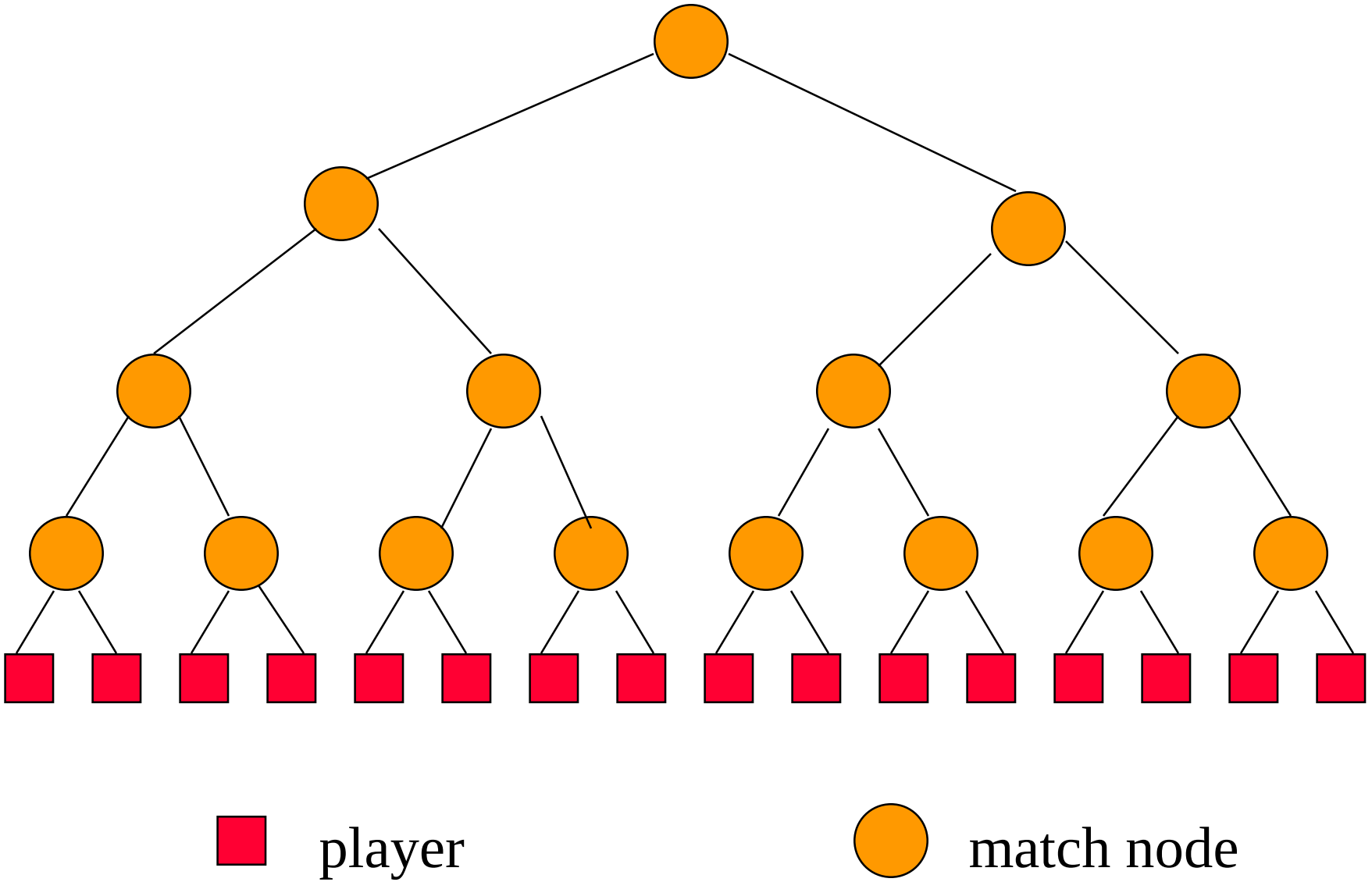
Complete binary tree with n external nodes and n - 1 internal nodes.

External nodes represent tournament players.

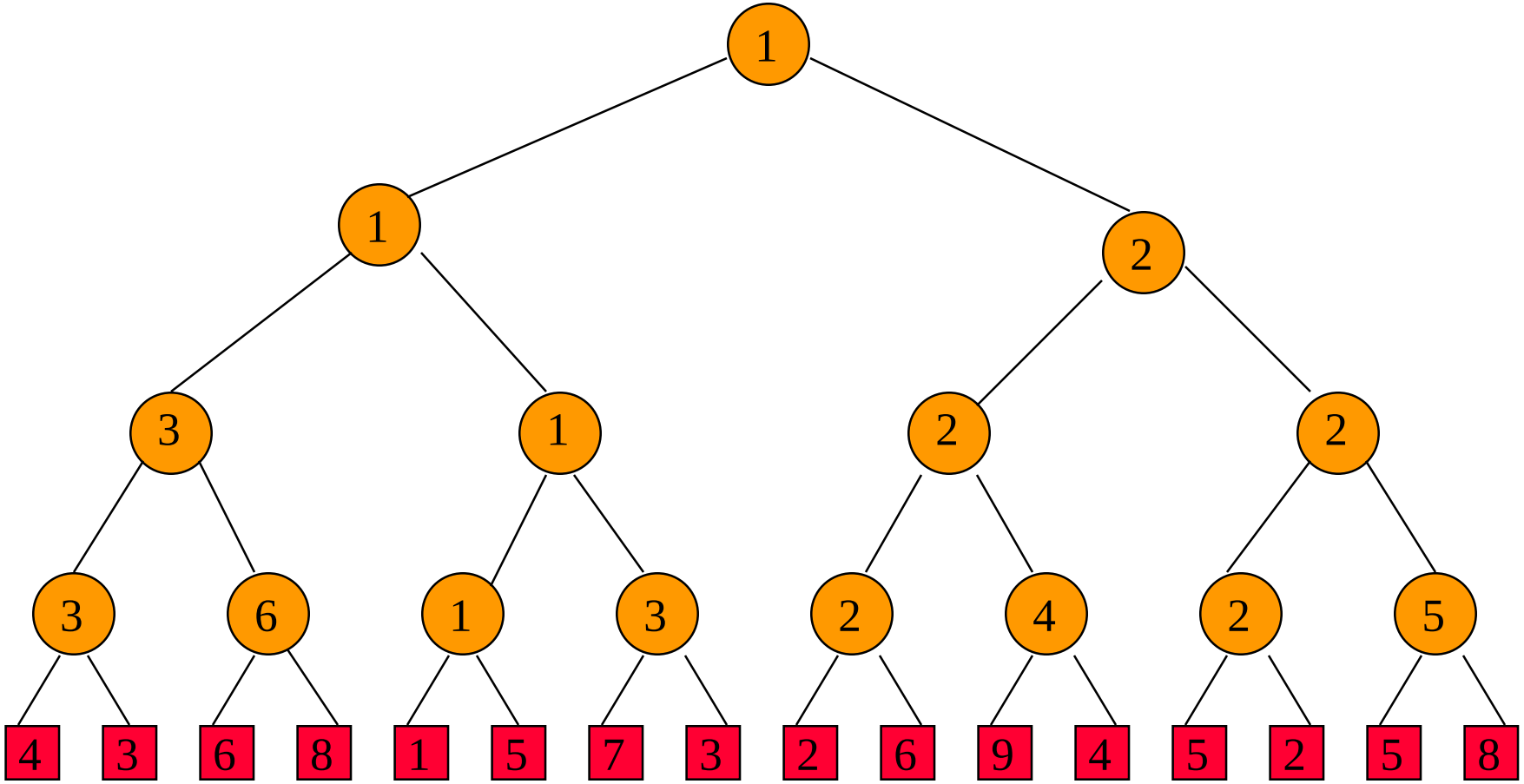
Each internal node represents a match played between its two children; the winner of the match is stored at the internal node.

Root has overall winner.

Winner Tree For 16 Players

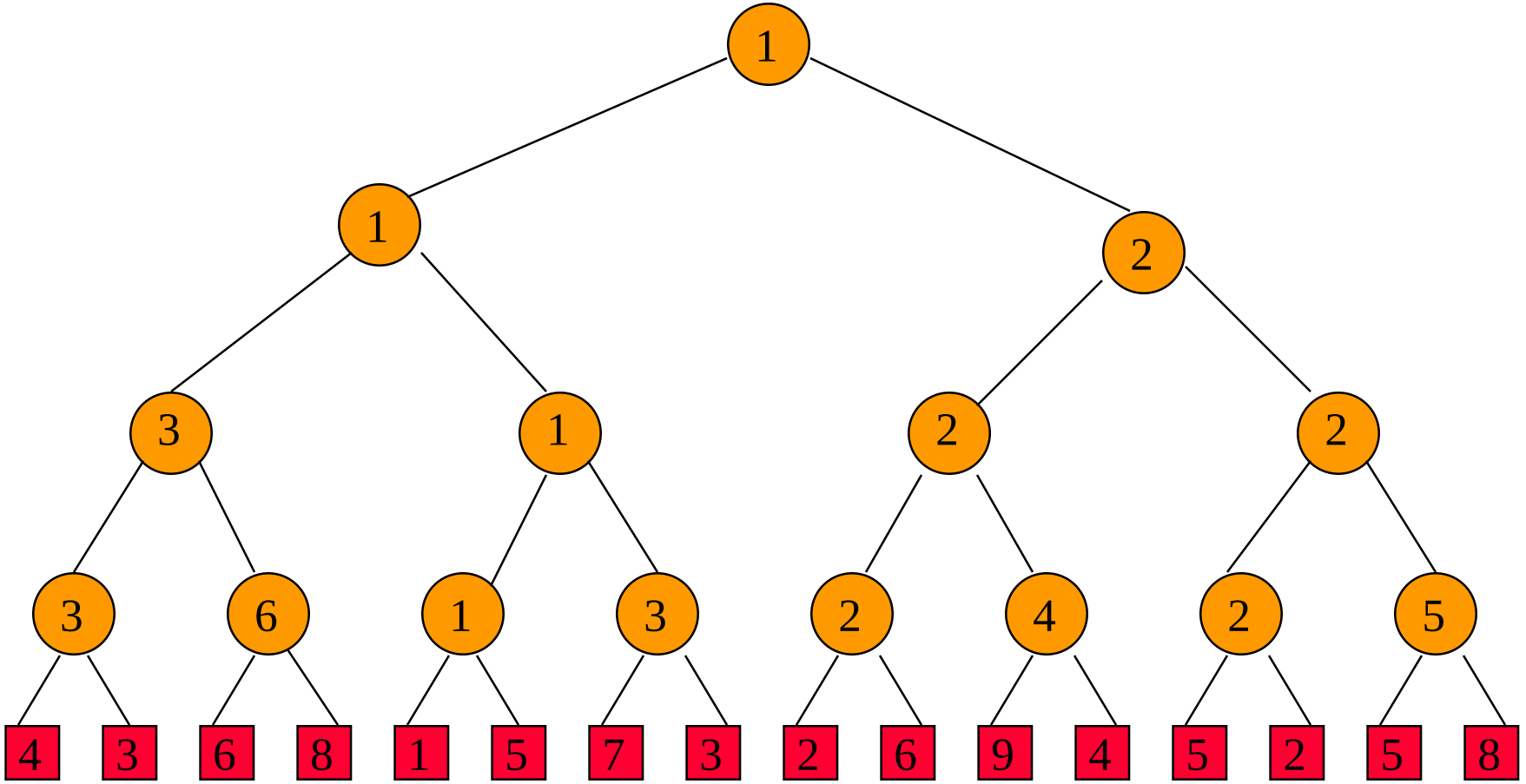


Winner Tree For 16 Players



Smaller element wins => min winner tree.

Winner Tree For 16 Players



height is $\log_2 n$ (excludes player level)

Complexity Of Initialize

- O(1) time to play match at each match node.
- n - 1 match nodes.
- O(n) time to initialize n player winner tree.

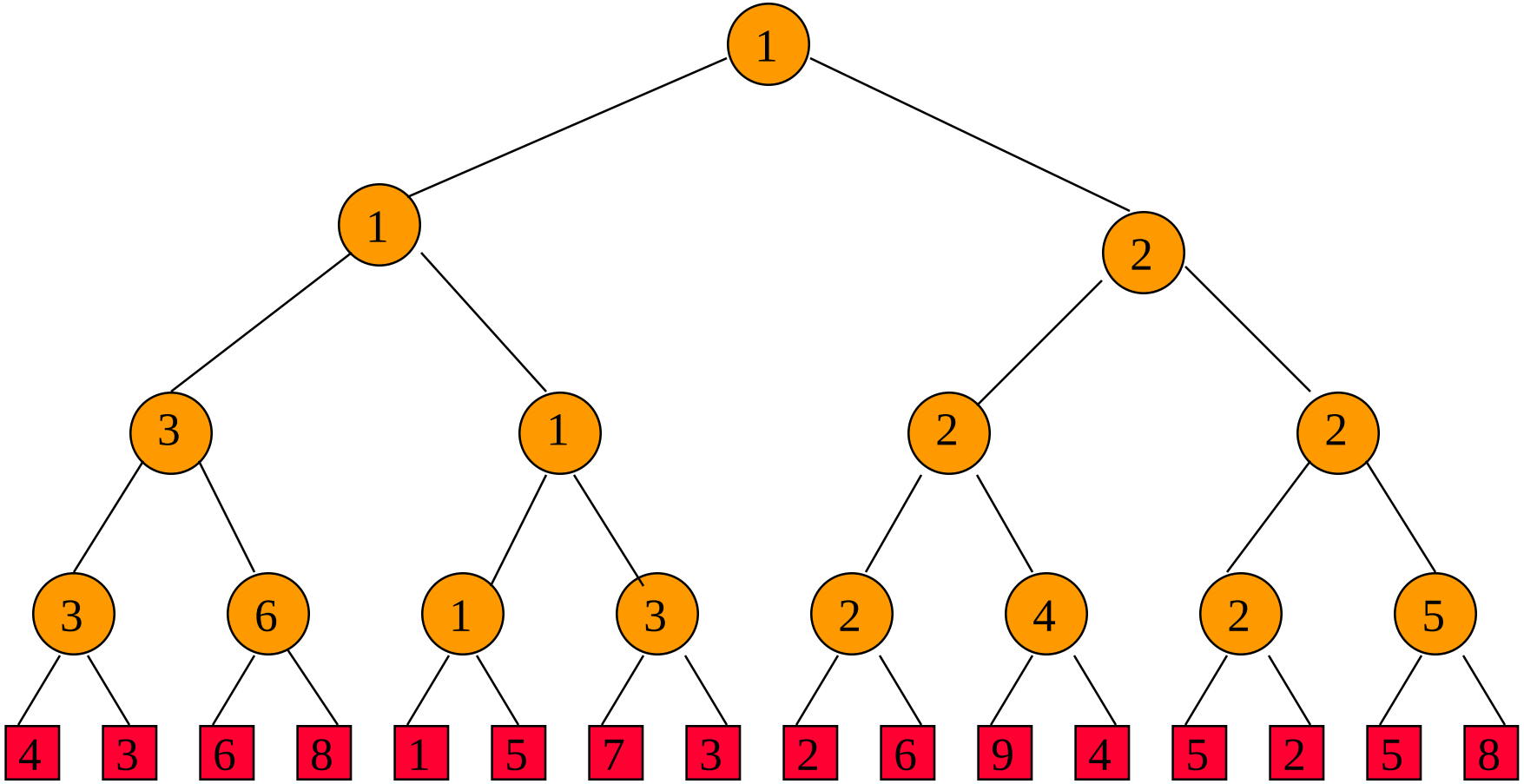
Applications

Sorting.

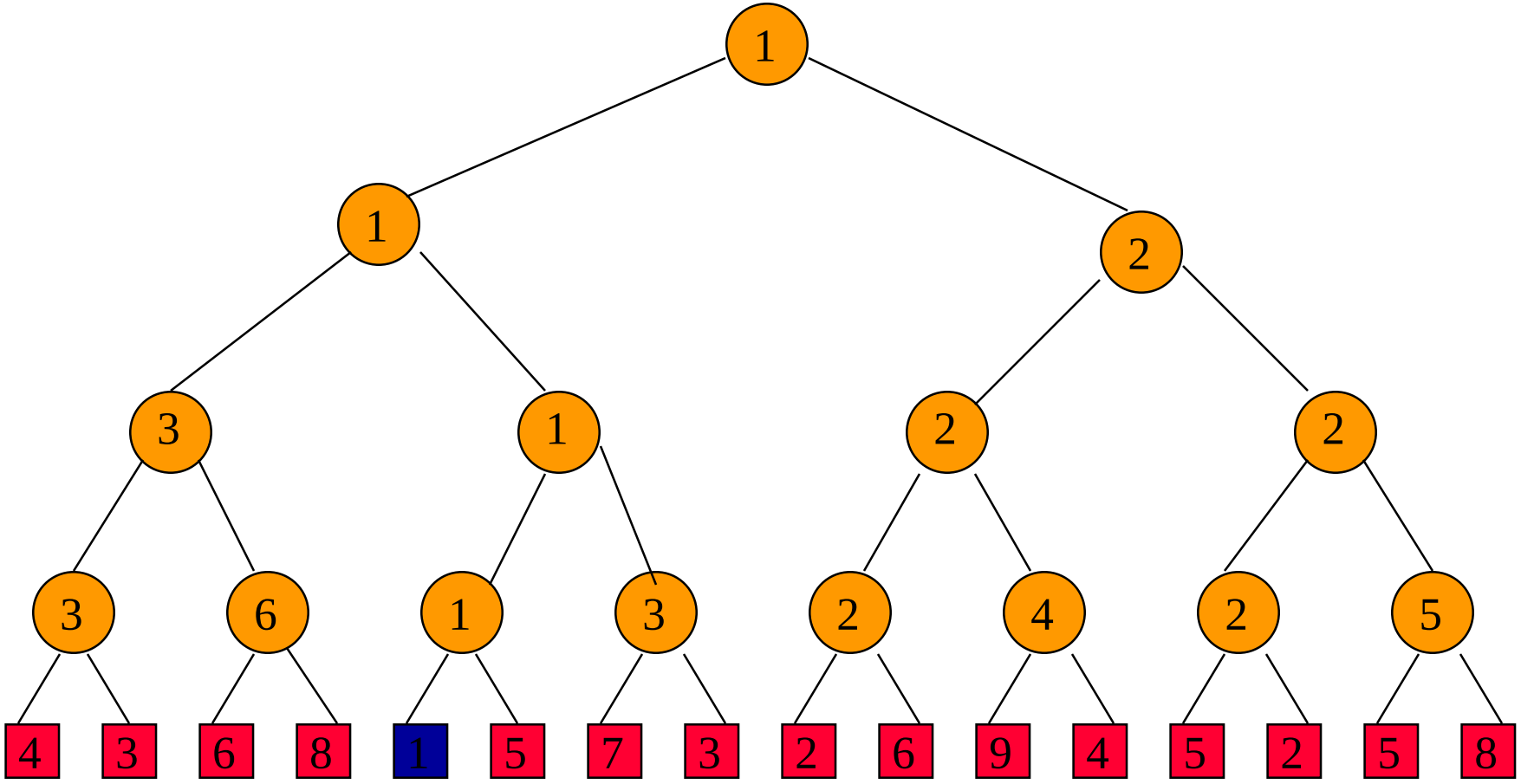
Insert elements to be sorted into a winner tree.

Repeatedly extract the winner and replace by a large value.

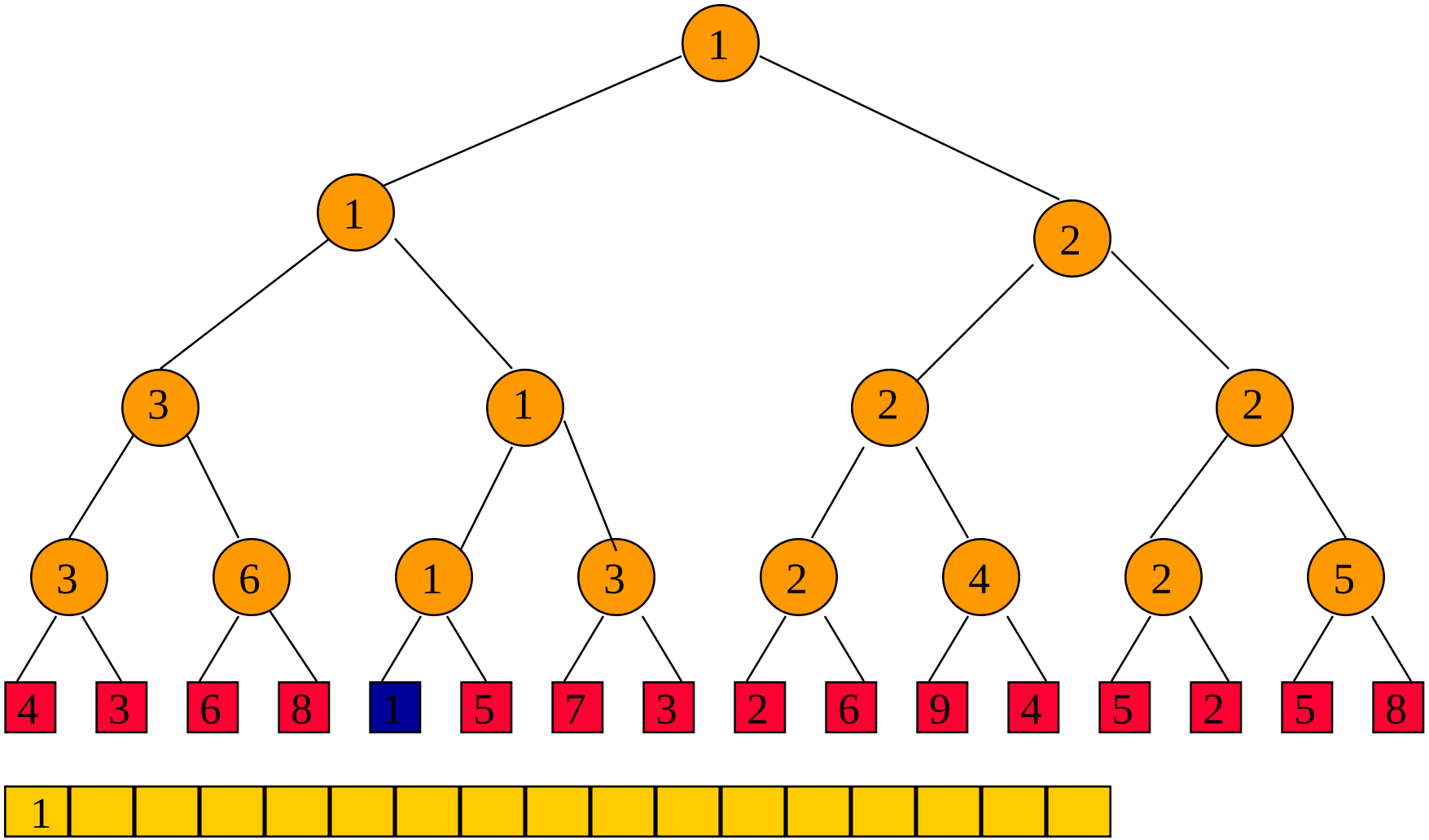
Sort 16 Numbers



Sort 16 Numbers

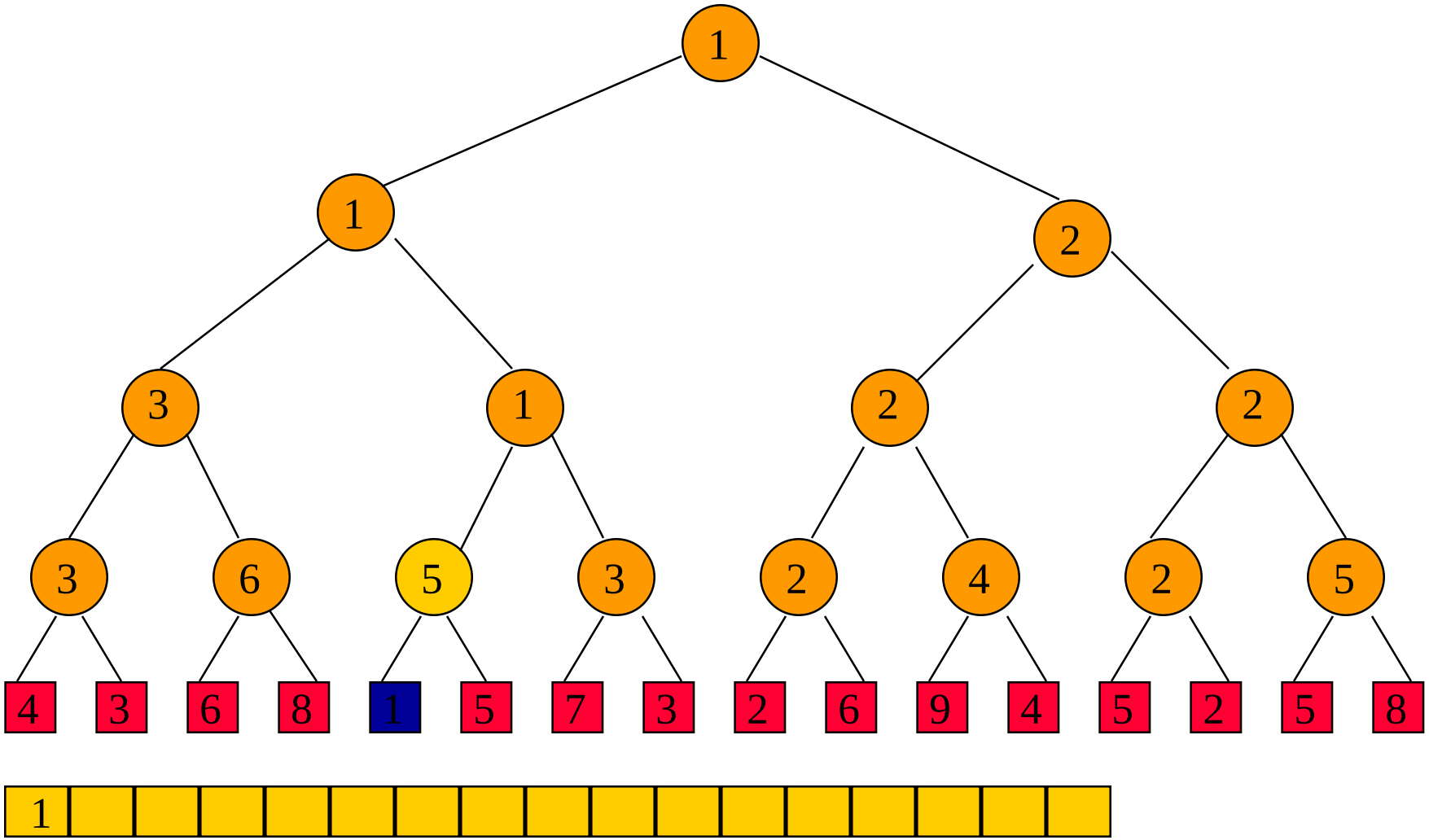


Sort 16 Numbers



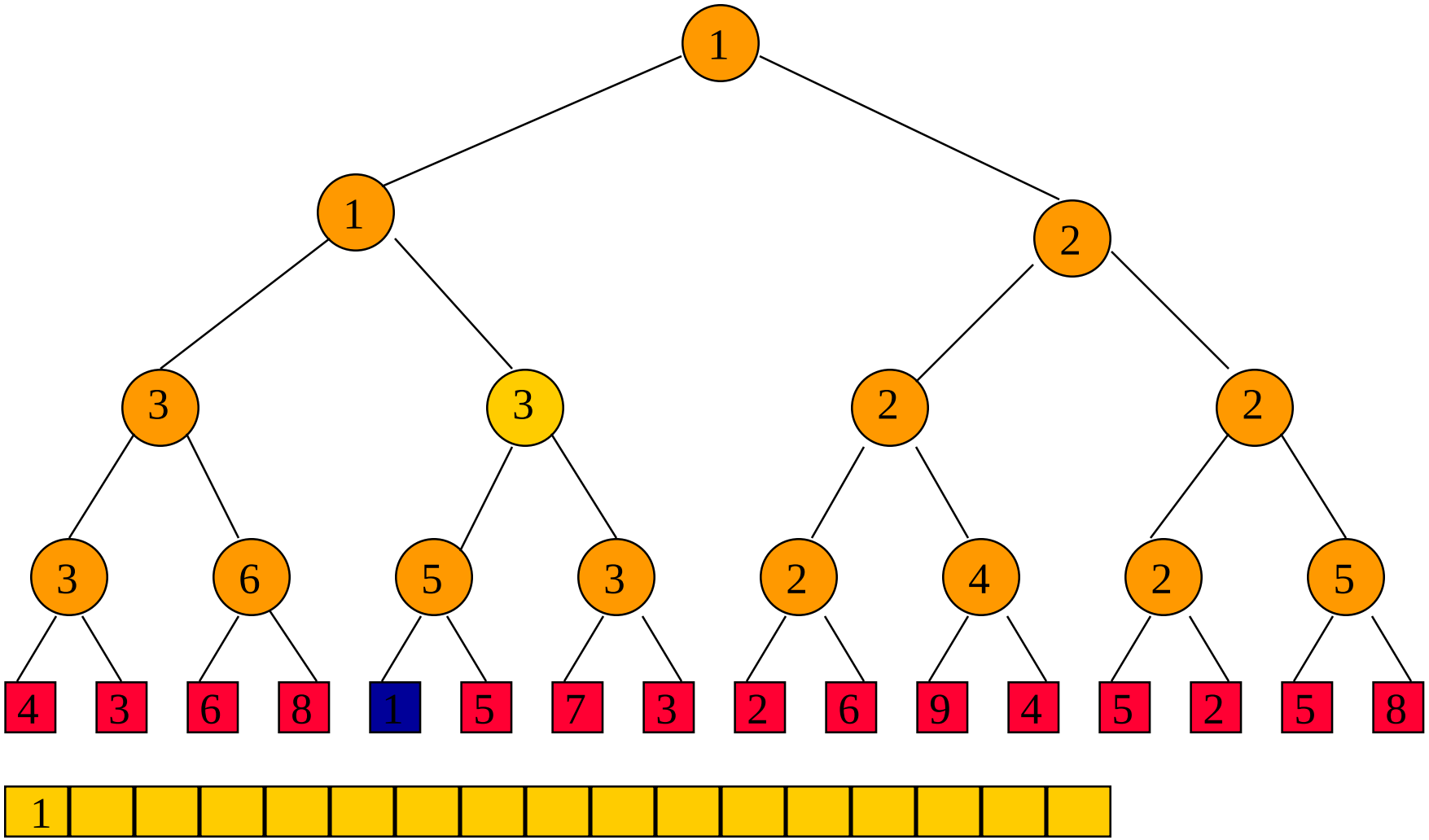
Sorted array.

Sort 16 Numbers



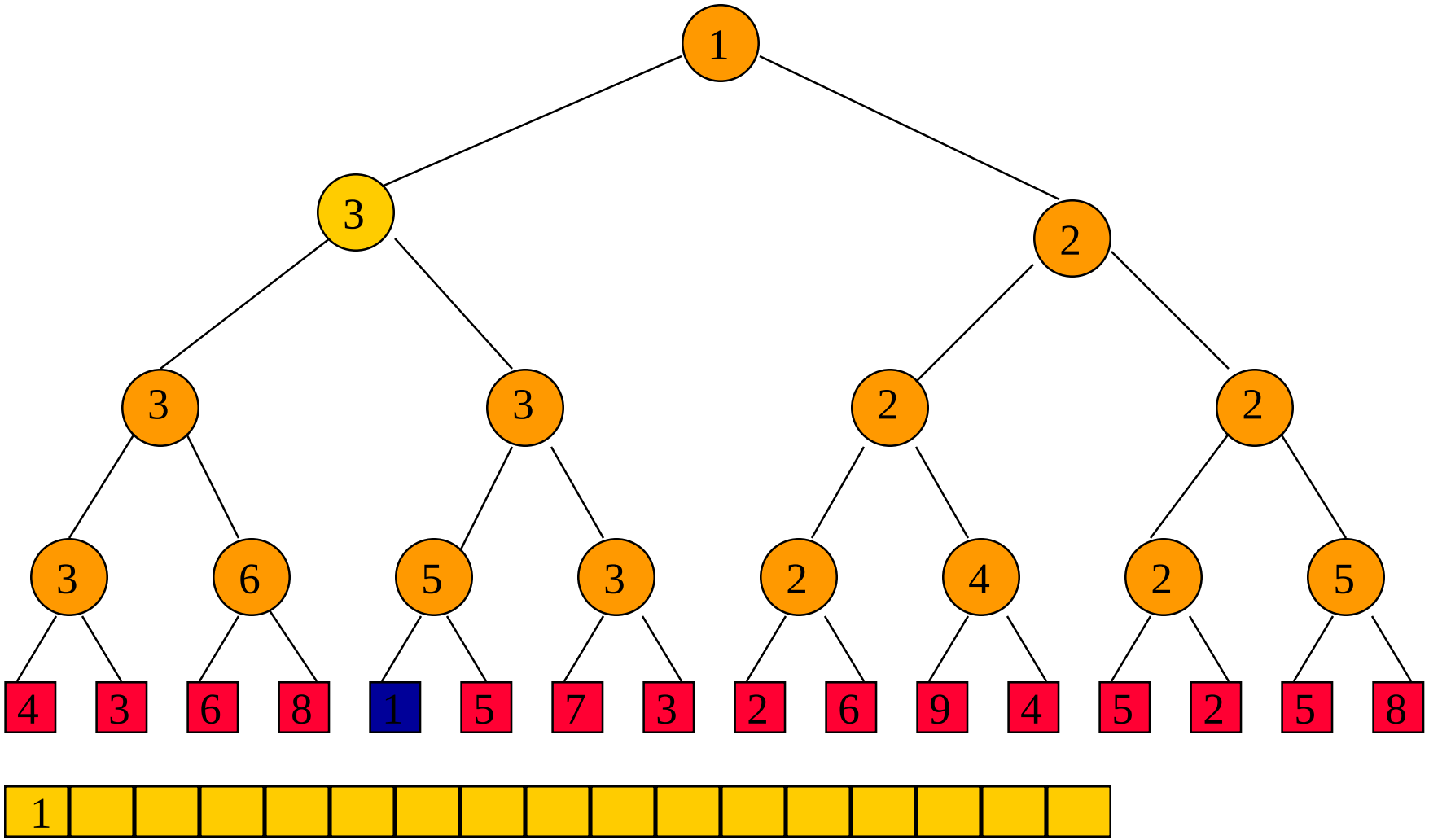
Sorted array.

Sort 16 Numbers



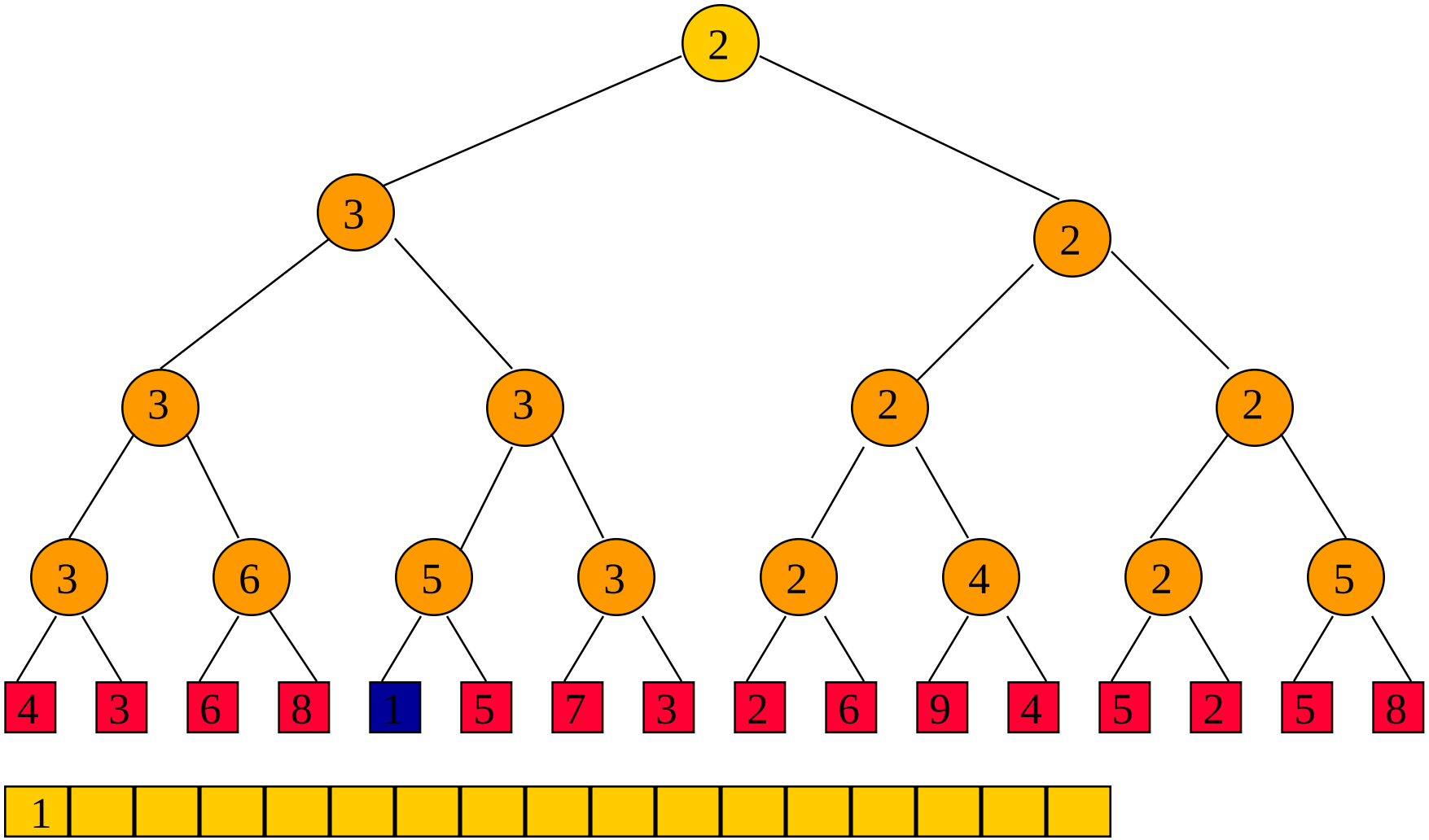
Sorted array.

Sort 16 Numbers



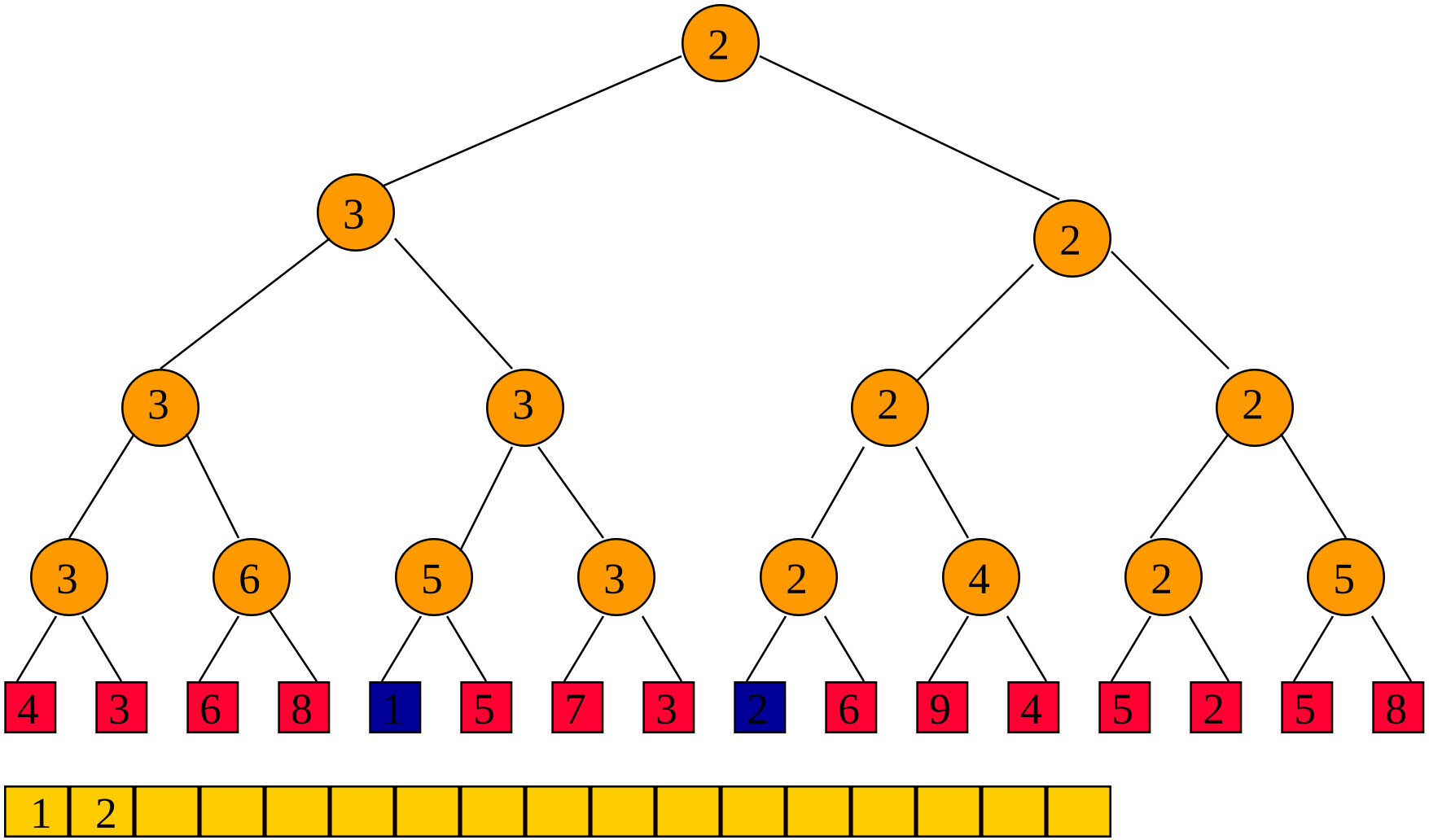
Sorted array.

Sort 16 Numbers



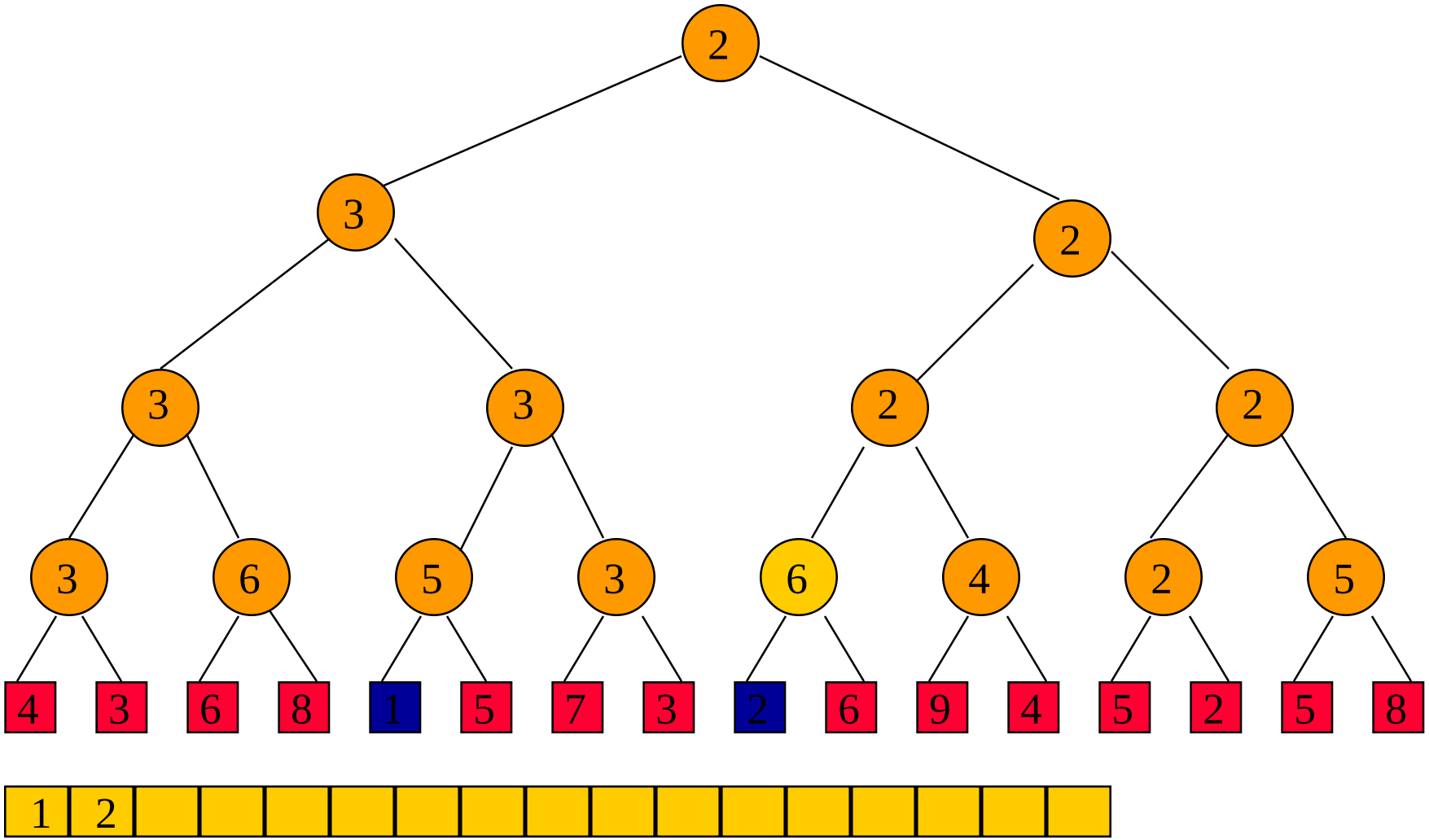
Sorted array.

Sort 16 Numbers



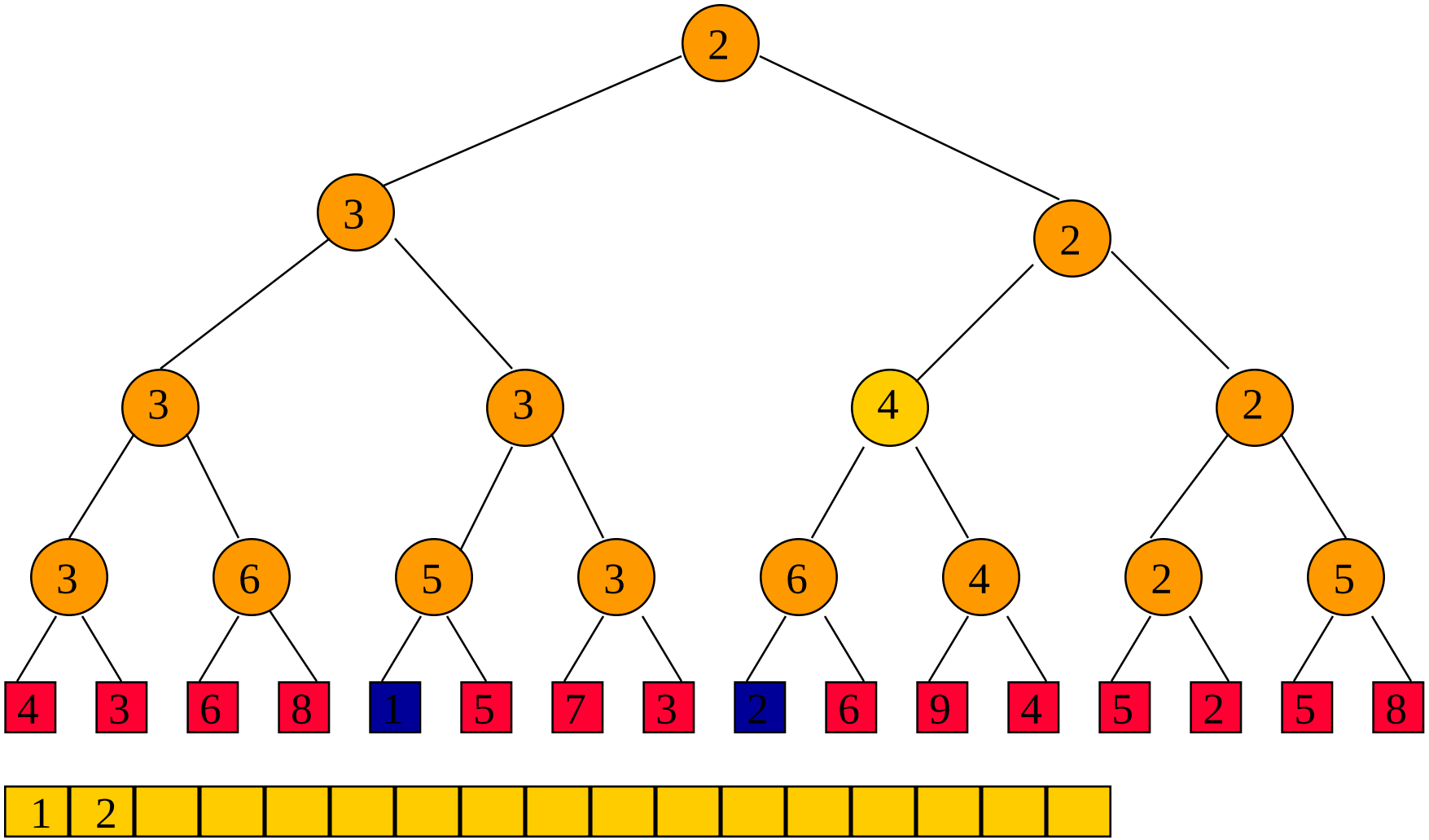
Sorted array.

Sort 16 Numbers



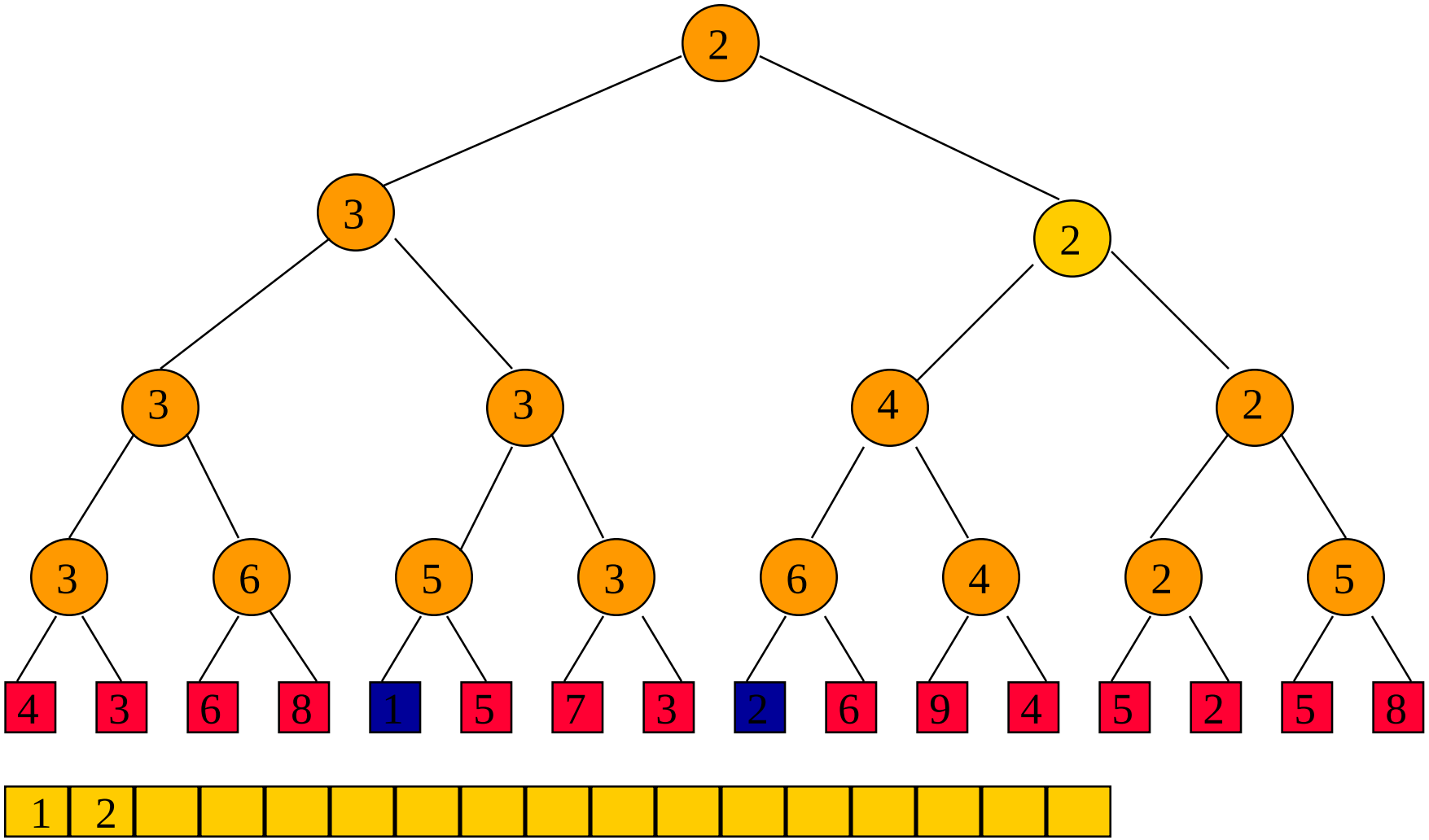
Sorted array.

Sort 16 Numbers



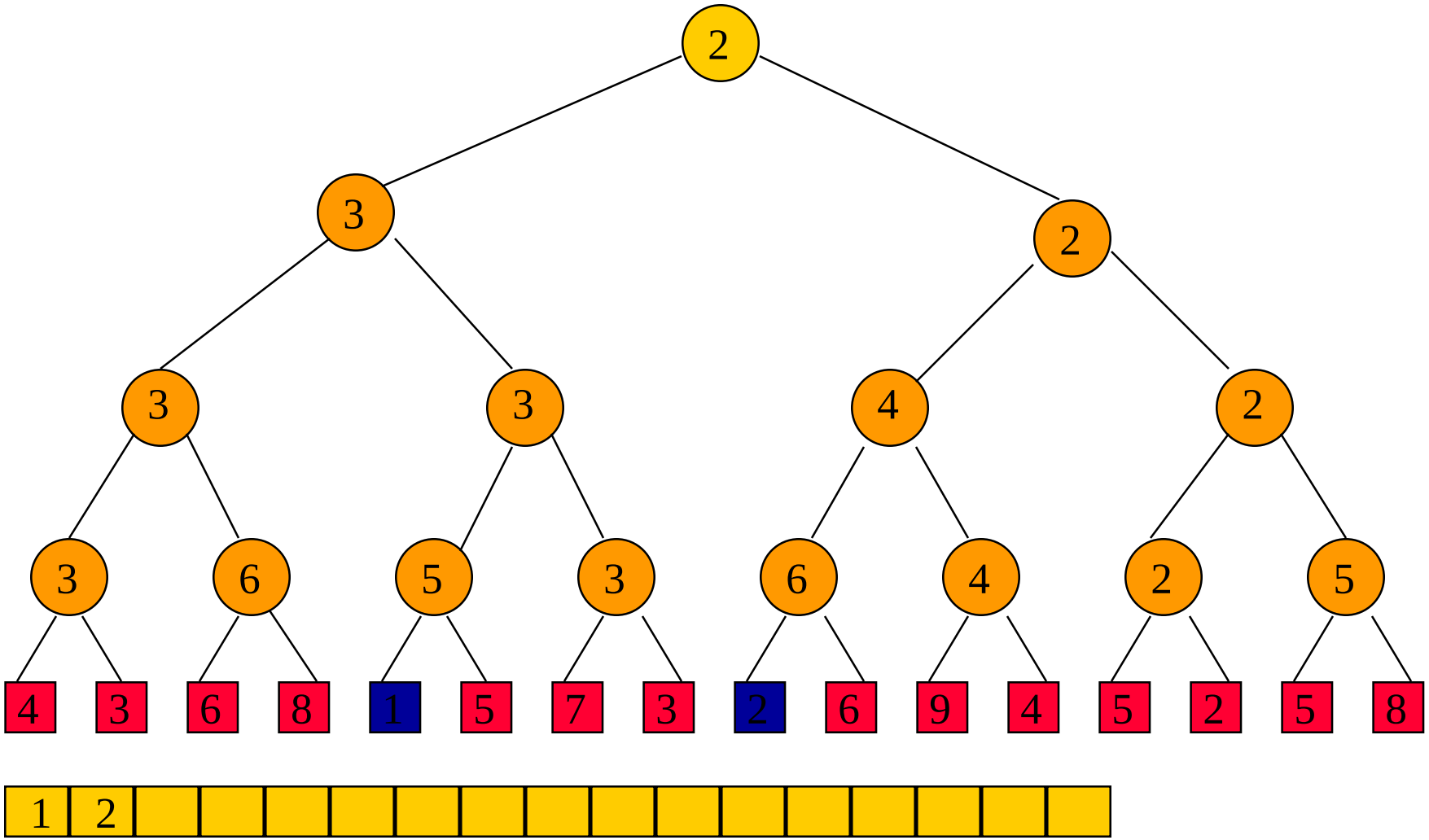
Sorted array.

Sort 16 Numbers



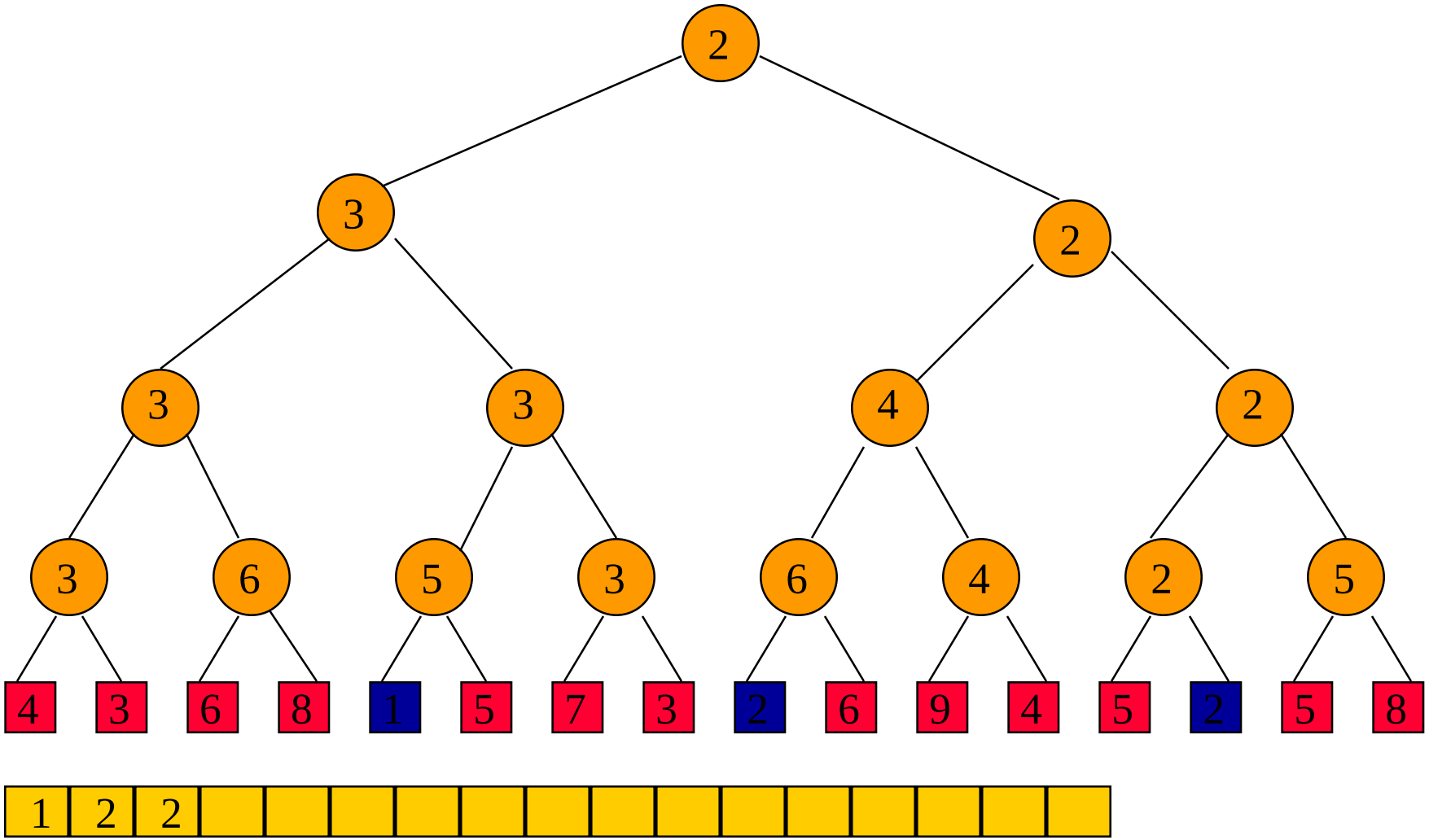
Sorted array.

Sort 16 Numbers



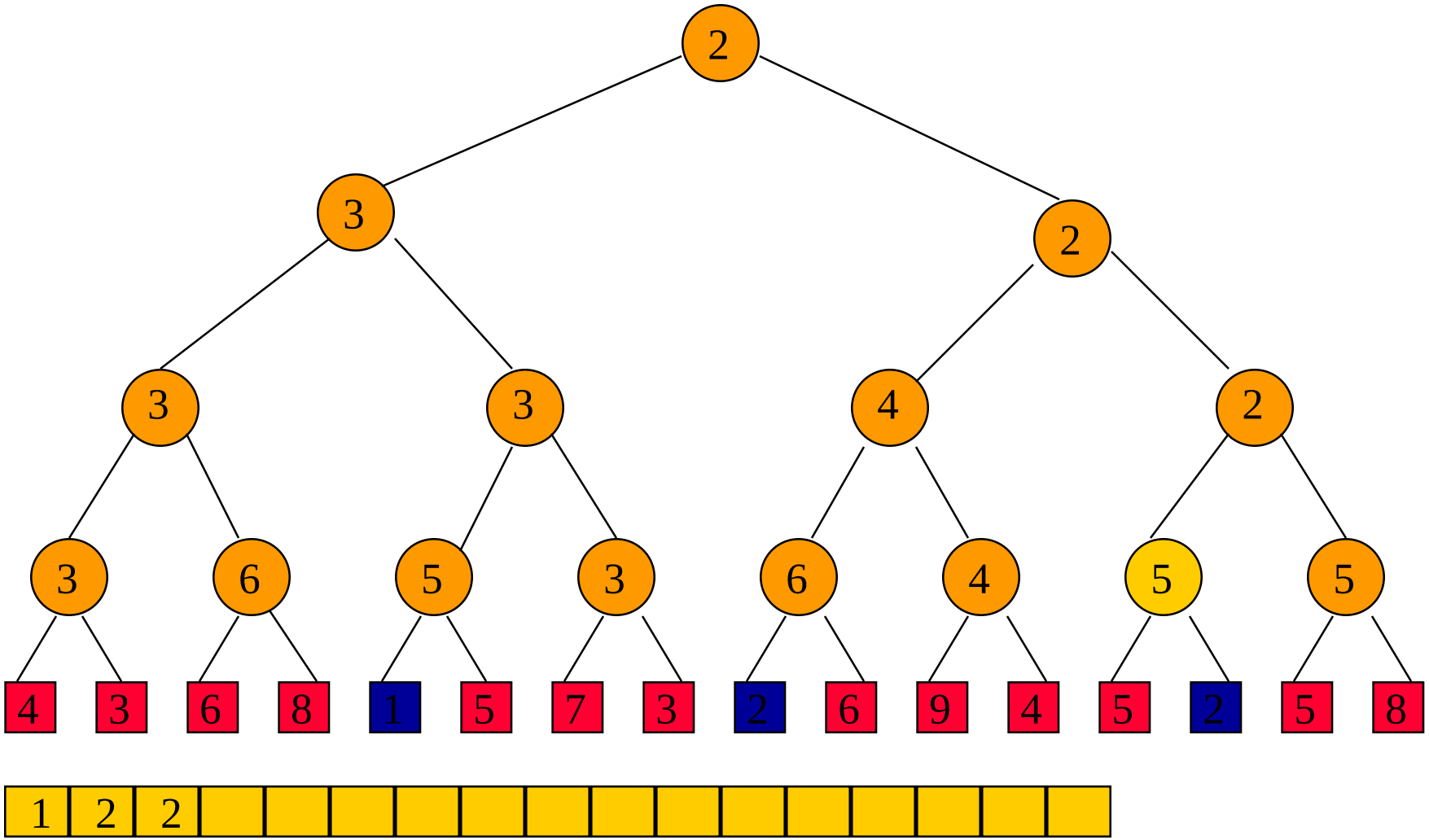
Sorted array.

Sort 16 Numbers



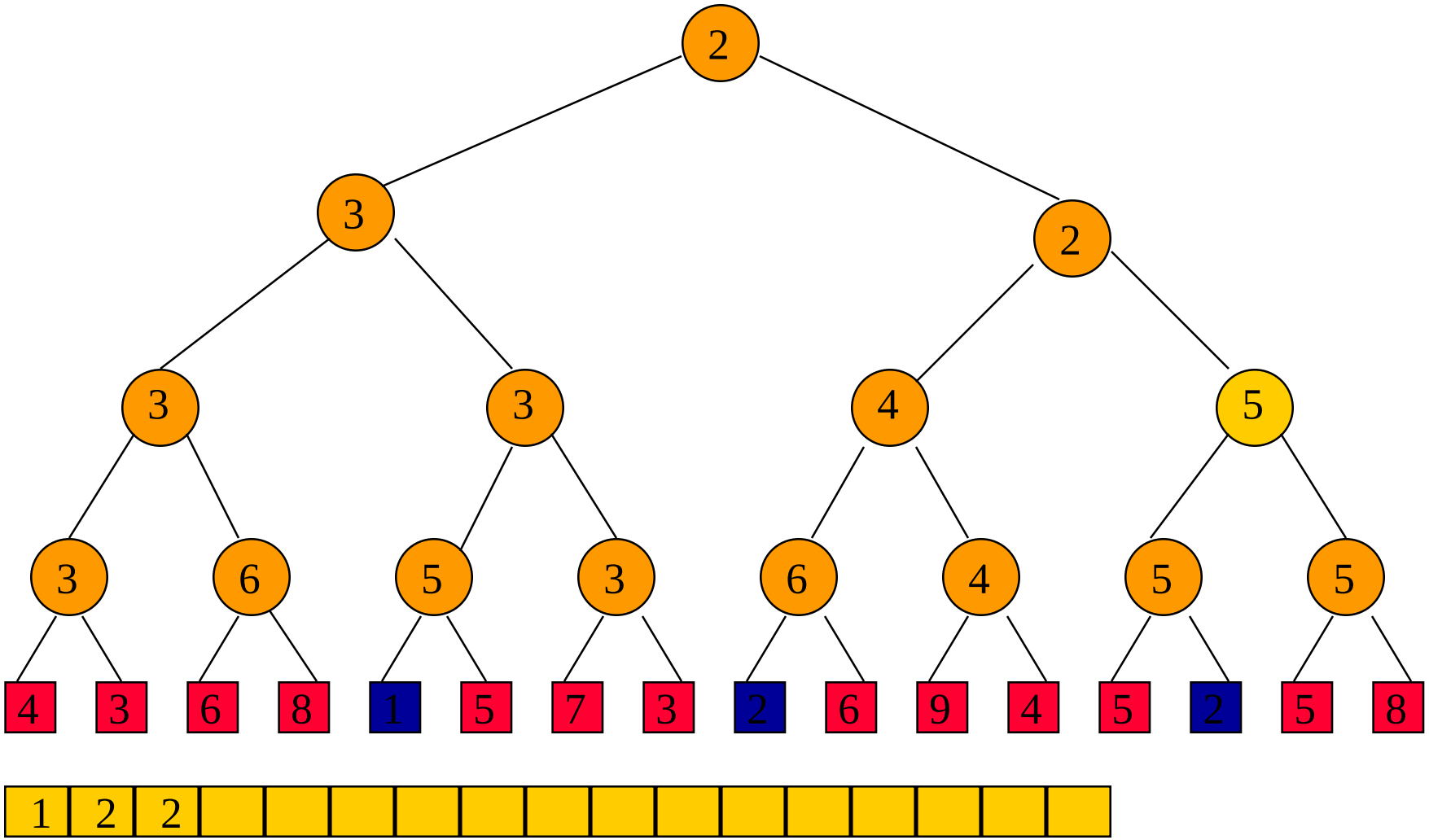
Sorted array.

Sort 16 Numbers



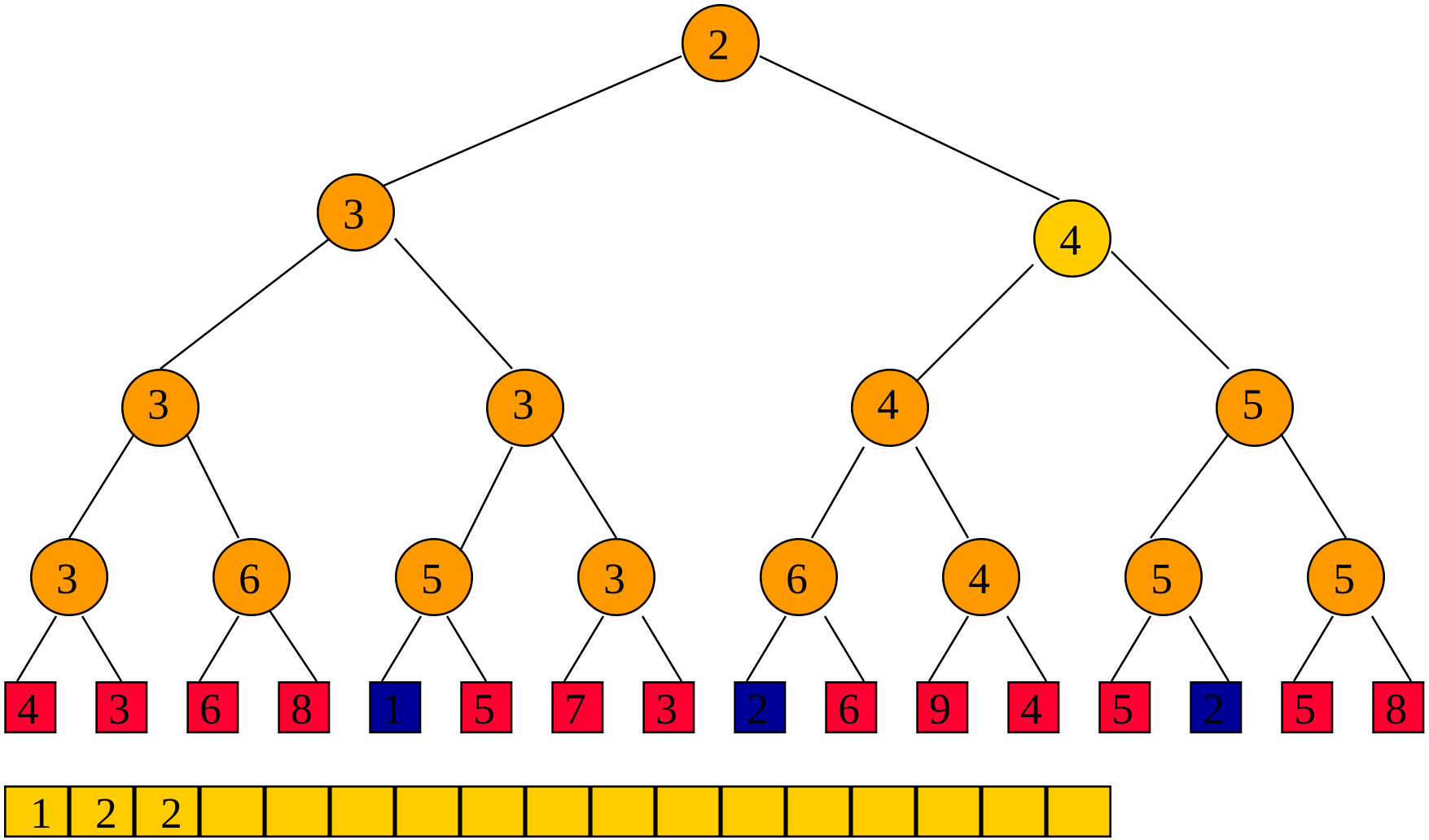
Sorted array.

Sort 16 Numbers



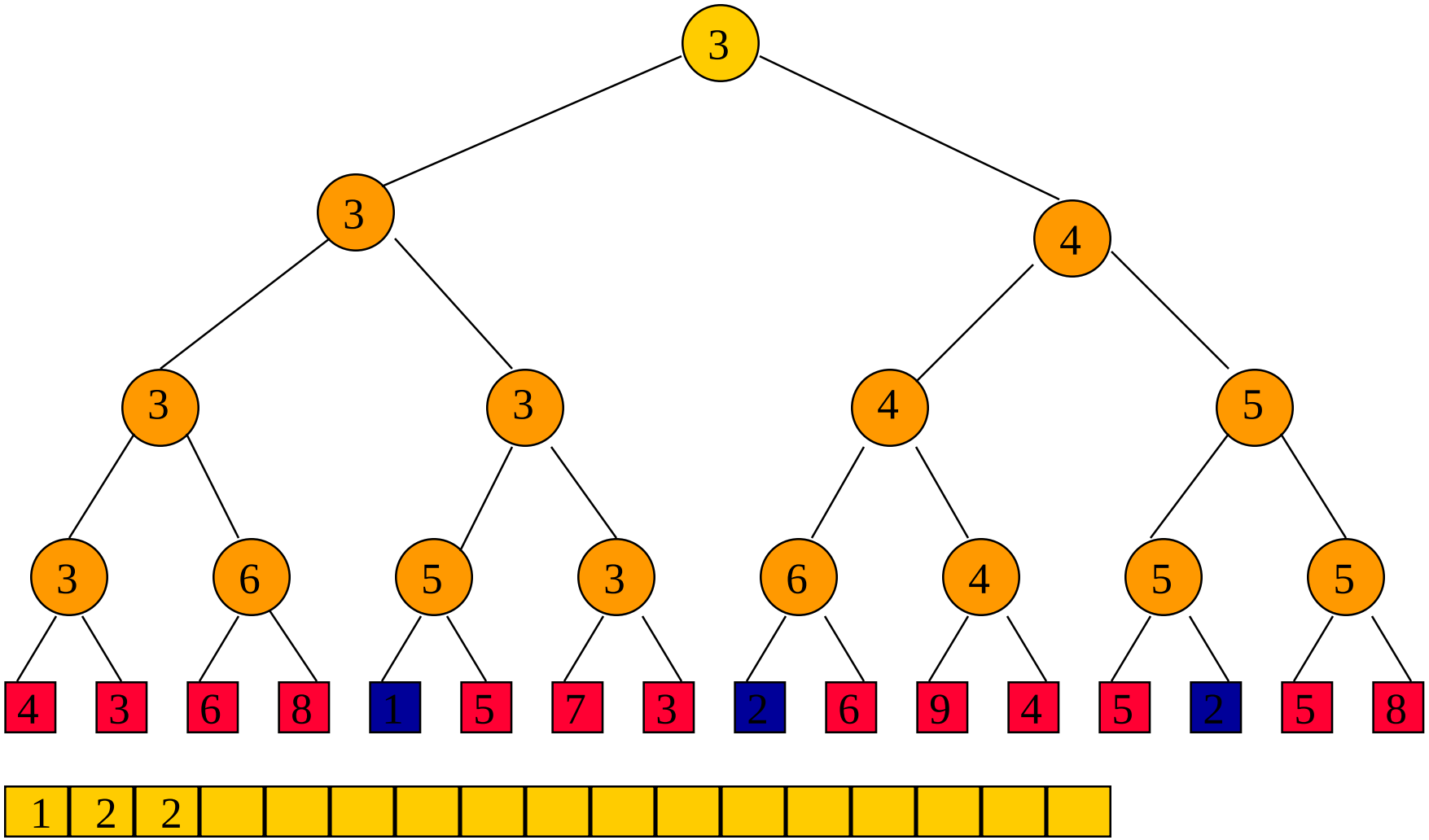
Sorted array.

Sort 16 Numbers



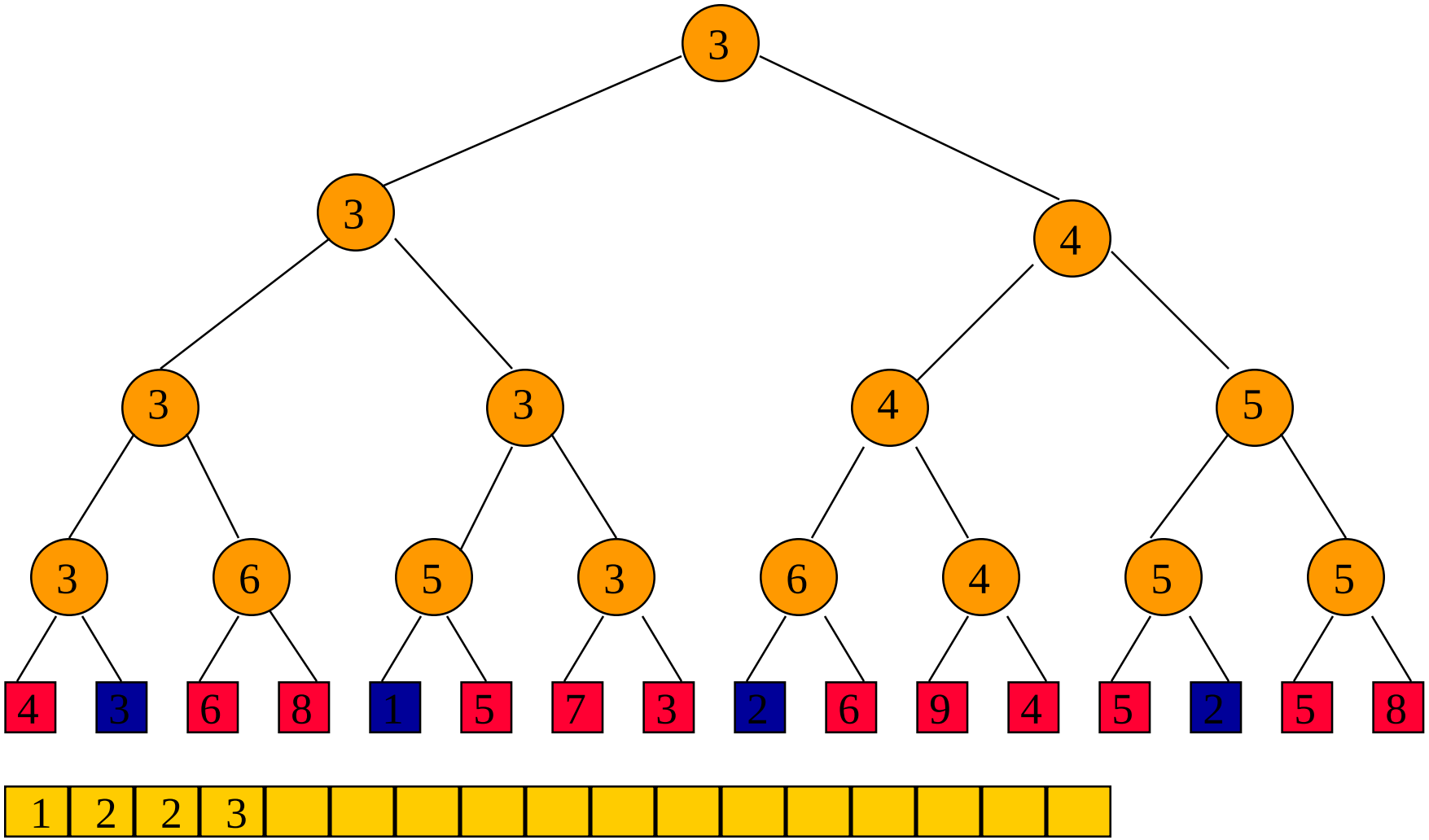
Sorted array.

Sort 16 Numbers



Sorted array.

Sort 16 Numbers



Sorted array.

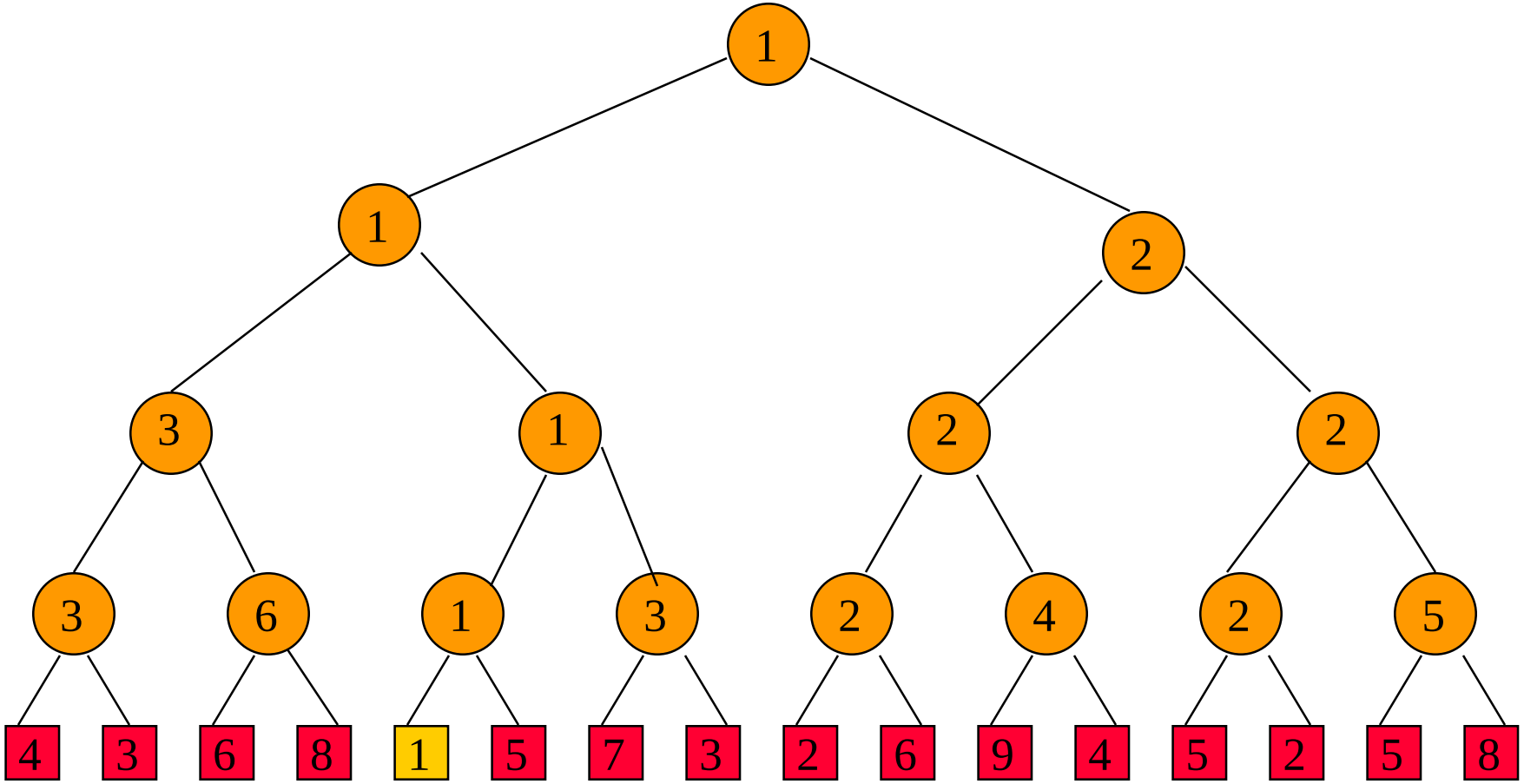
Time To Sort

- Initialize winner tree.
  - O(n) time
- Remove winner and replay.
  - O(log n) time
- Remove winner and replay n times.
  - O(n log n) time
- Total sort time is O(n log n).
- Actually Theta(n log n).

Winner Tree Operations

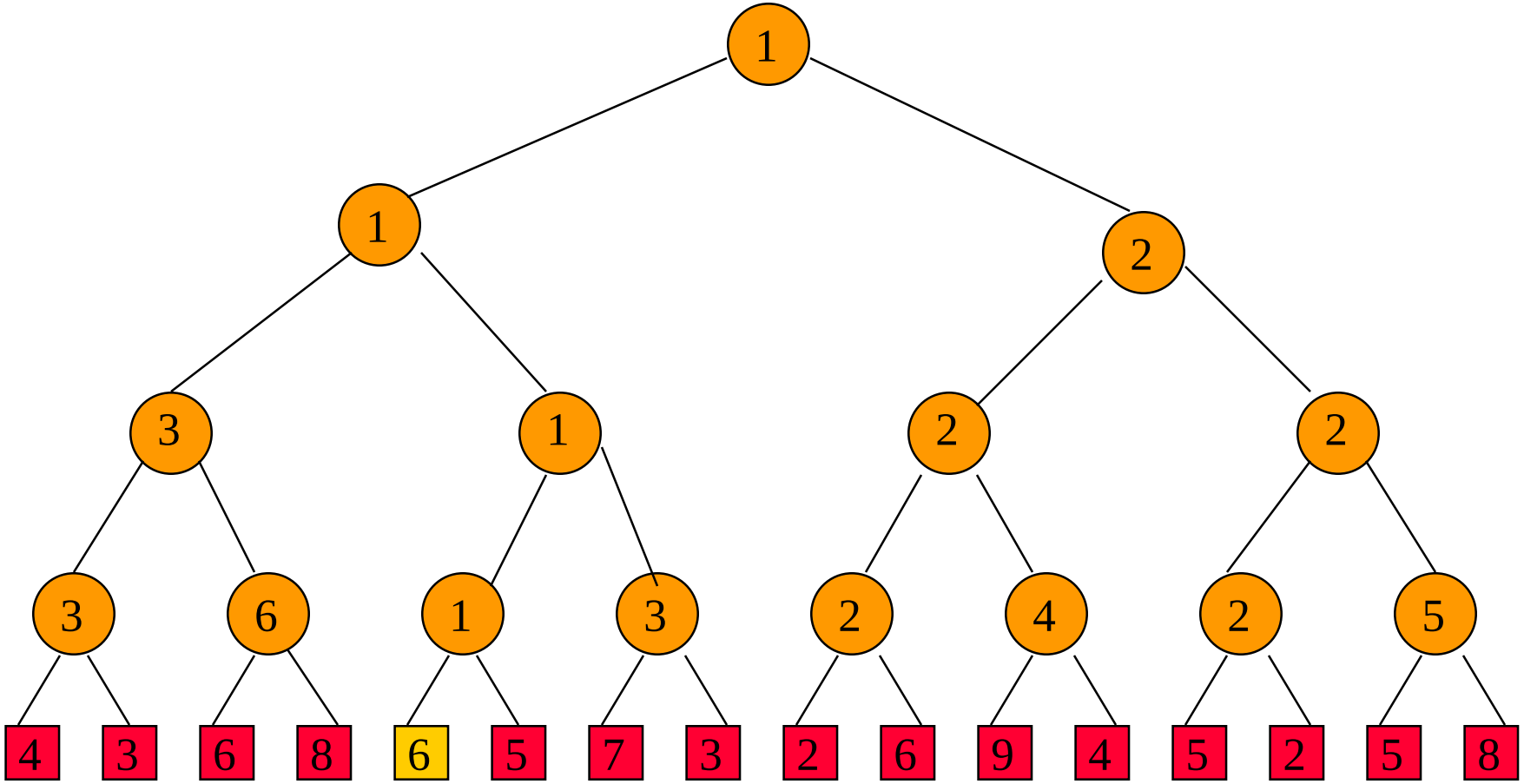
- Initialize
  - O(n) time
- Get winner
  - O(1) time
- Remove/replace winner and replay
  - O(log n) time
  - more precisely Theta(log n)

Replace Winner And Replay



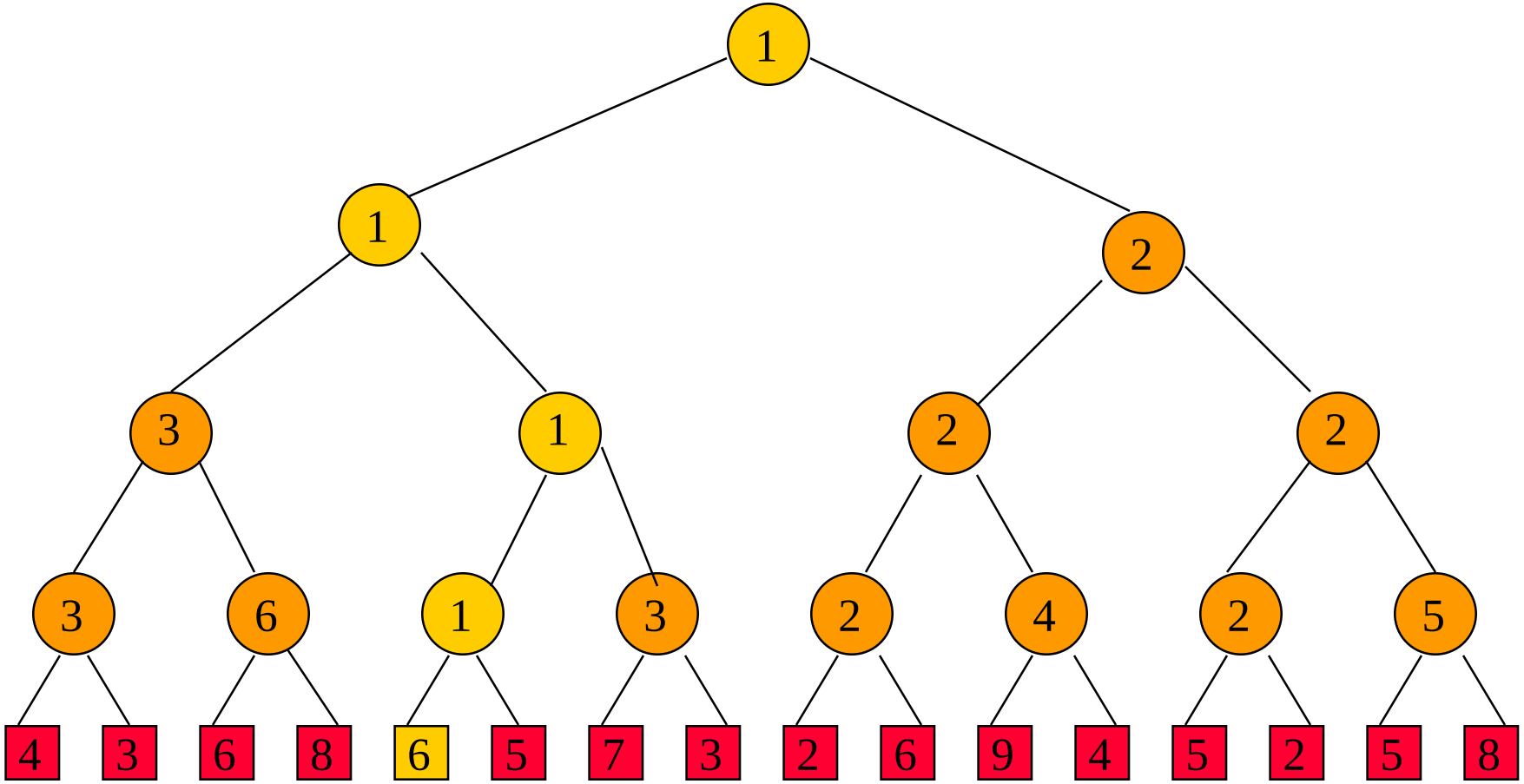
Replace winner with 6.

Replace Winner And Replay



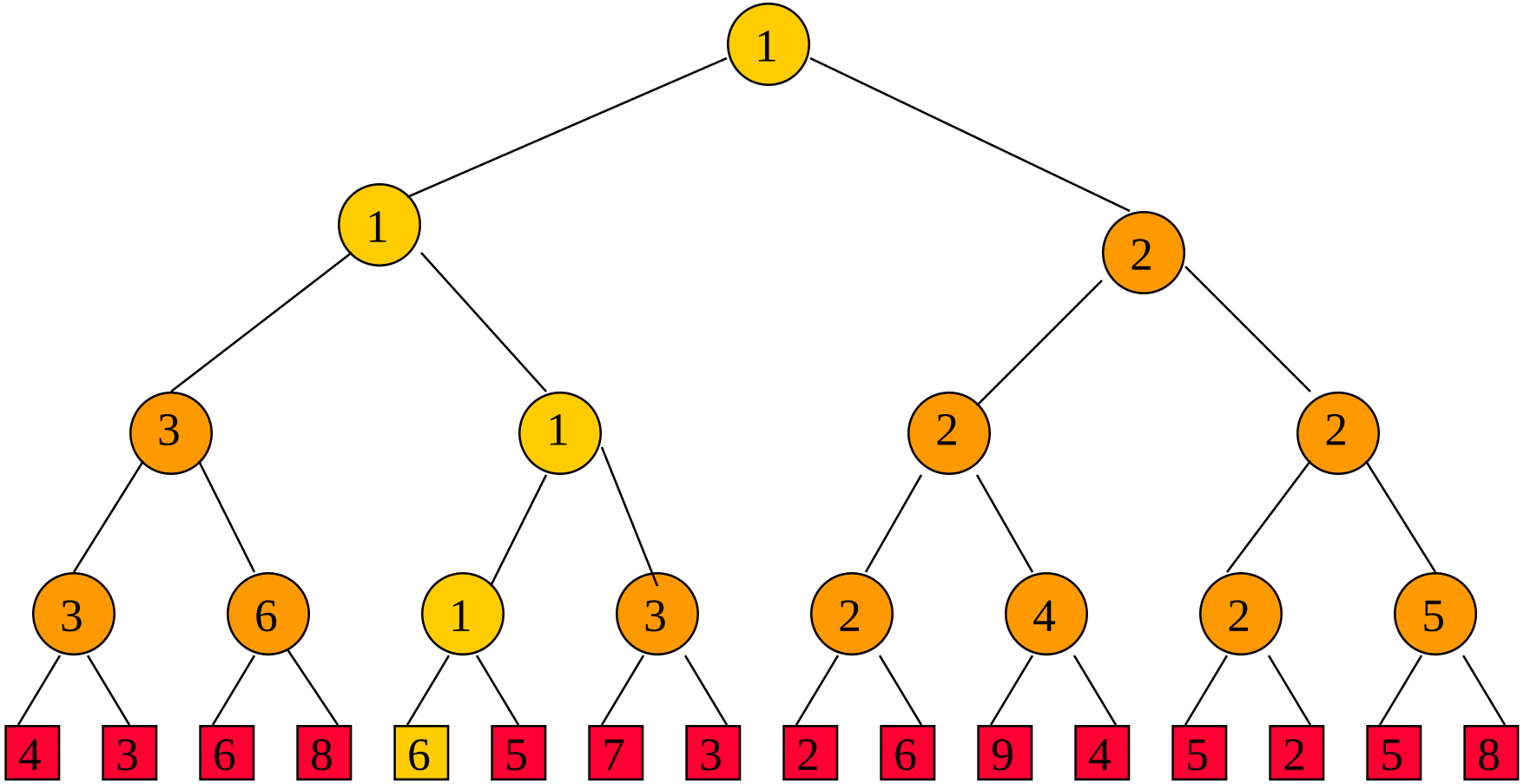
Replay matches on path to root.

Replace Winner And Replay



Replay matches on path to root.

Replace Winner And Replay

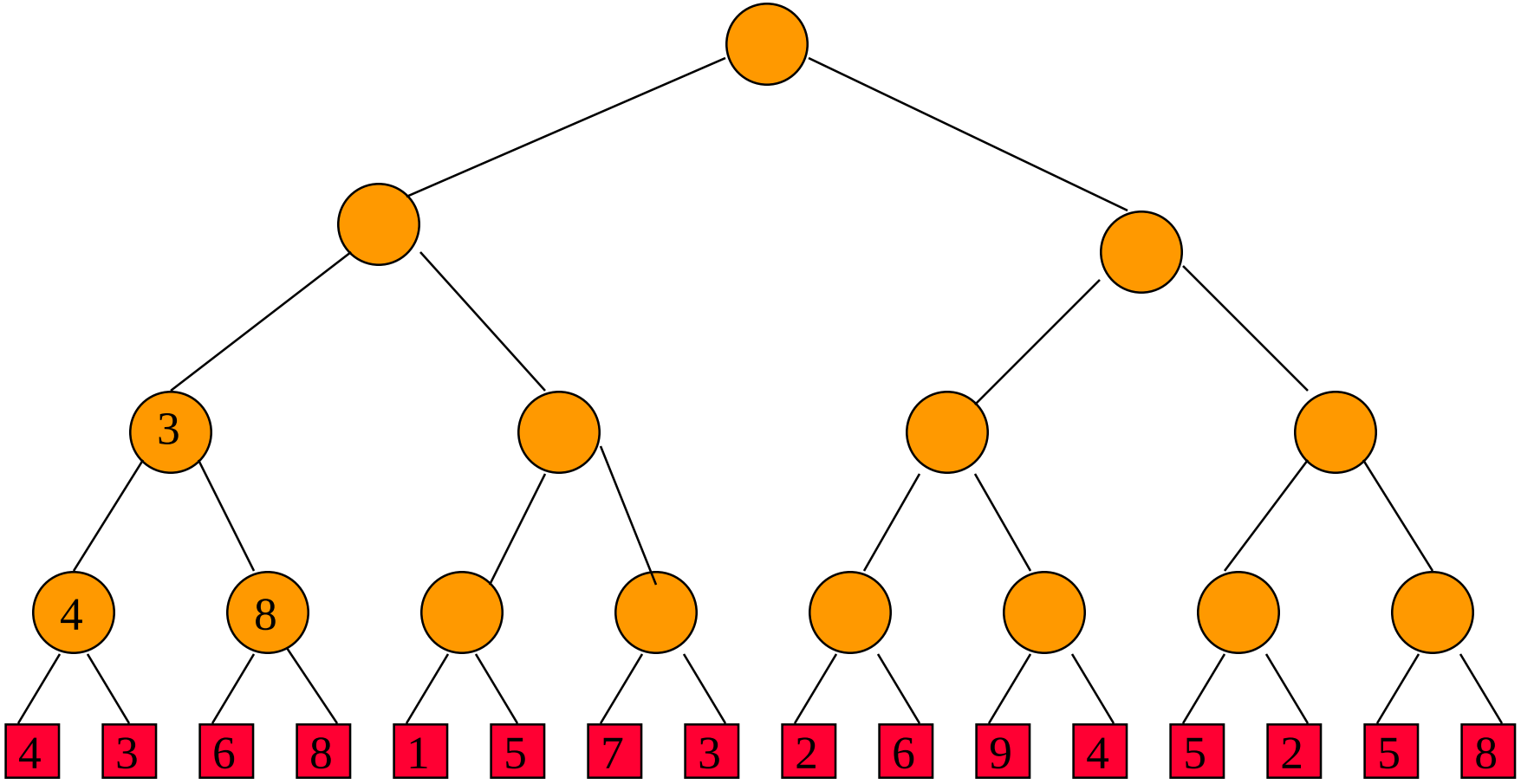


Opponent is player who lost last match played at this node.

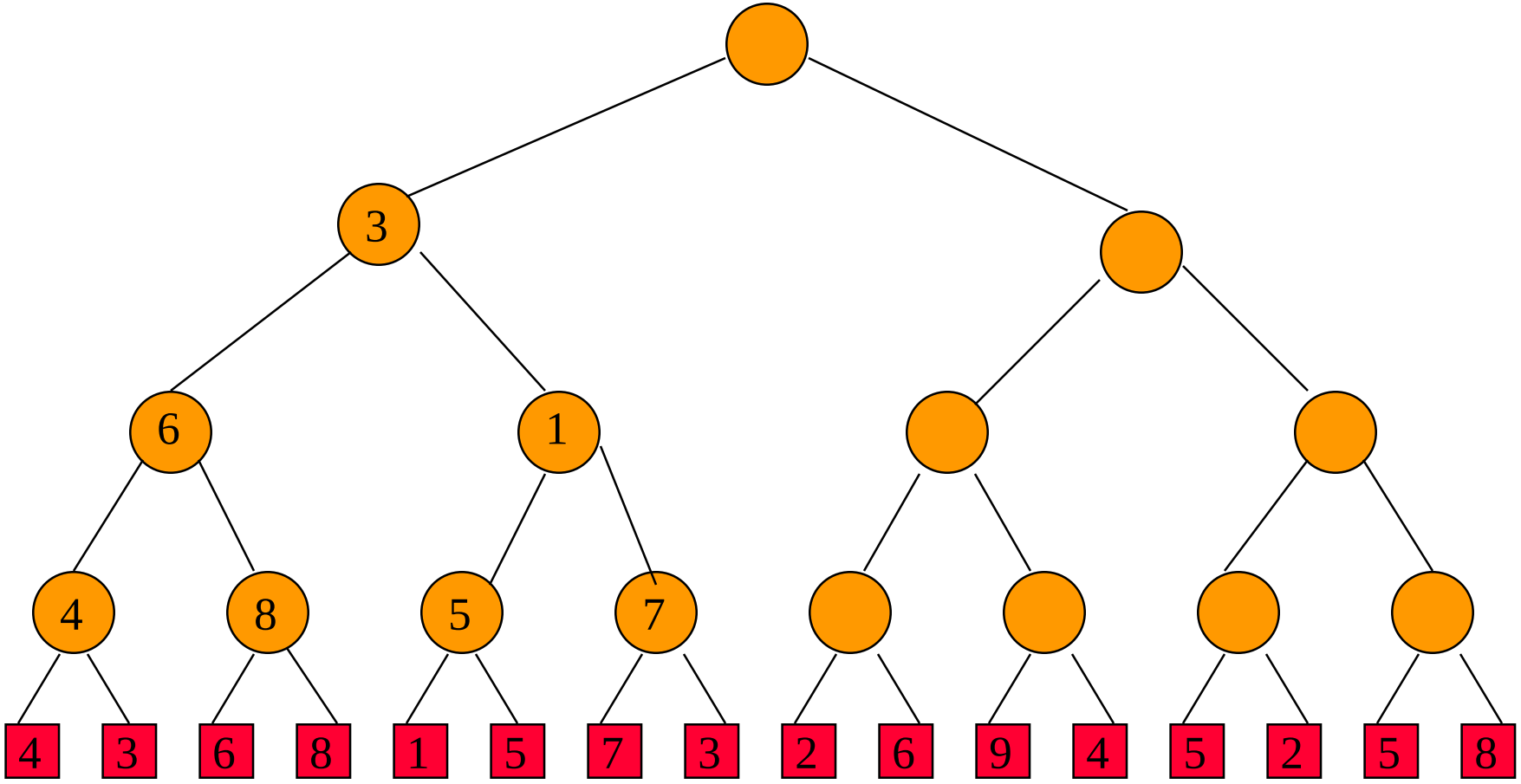
Loser Tree

Each match node stores the match loser rather than the match winner.

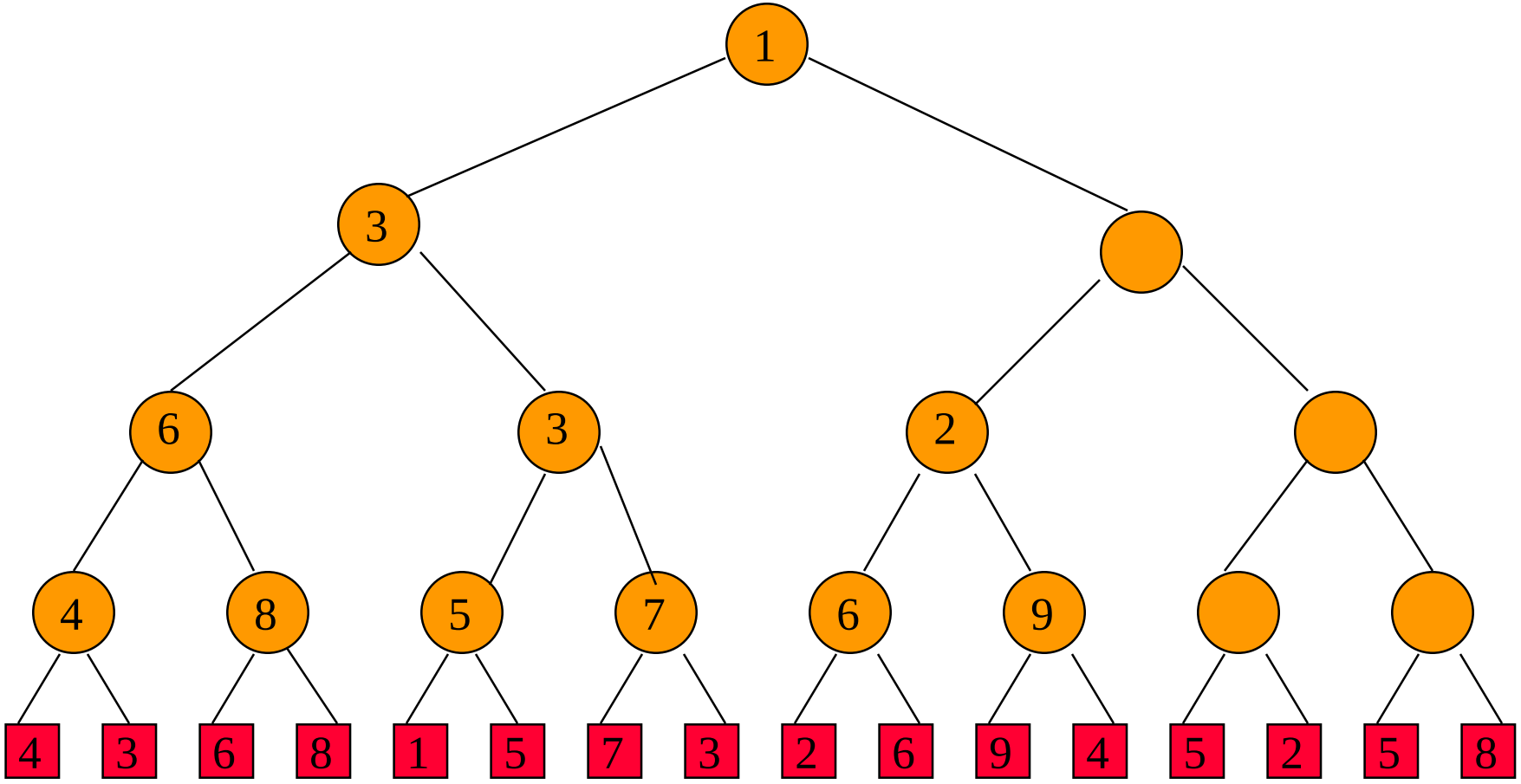
Min Loser Tree For 16 Players



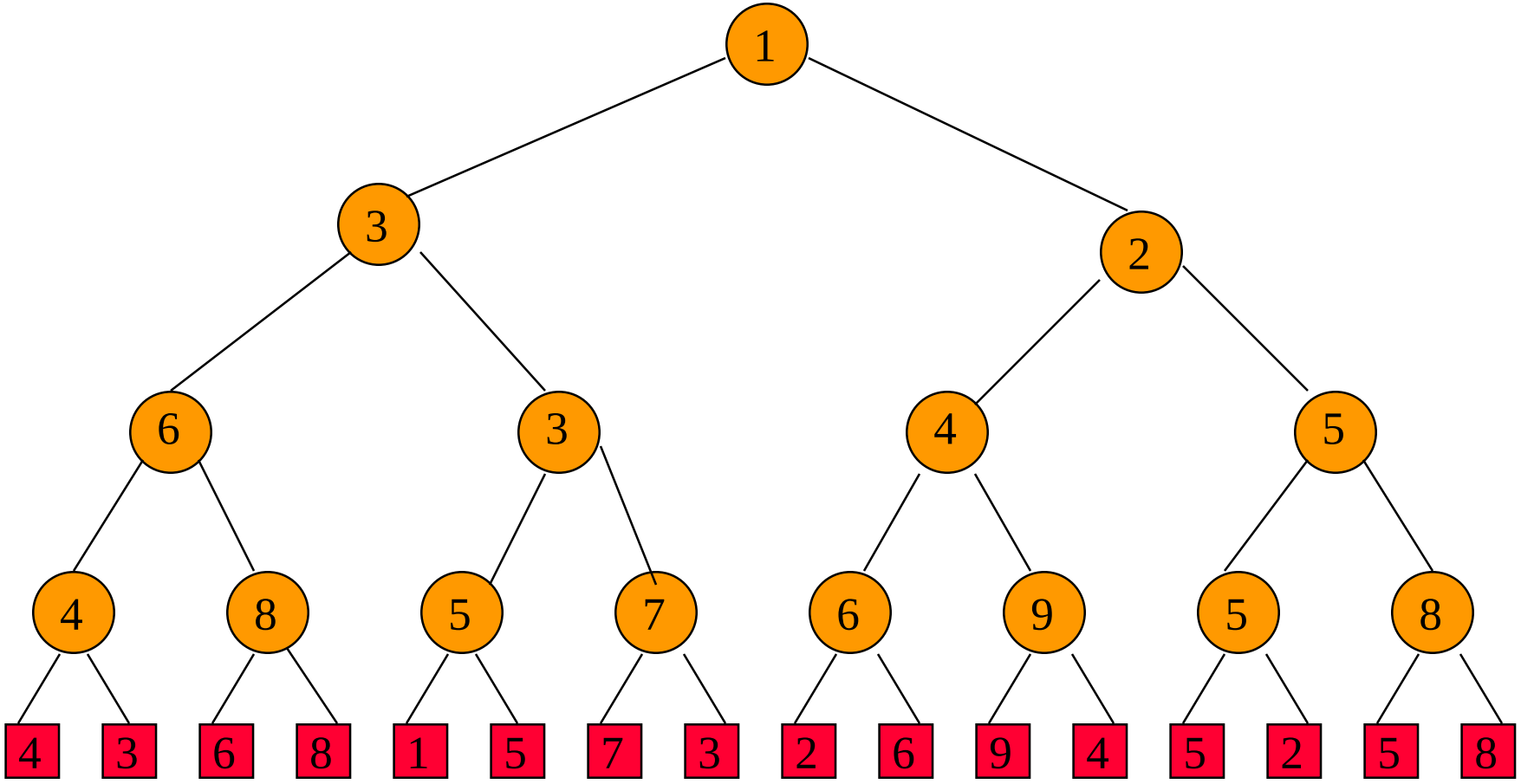
Min Loser Tree For 16 Players



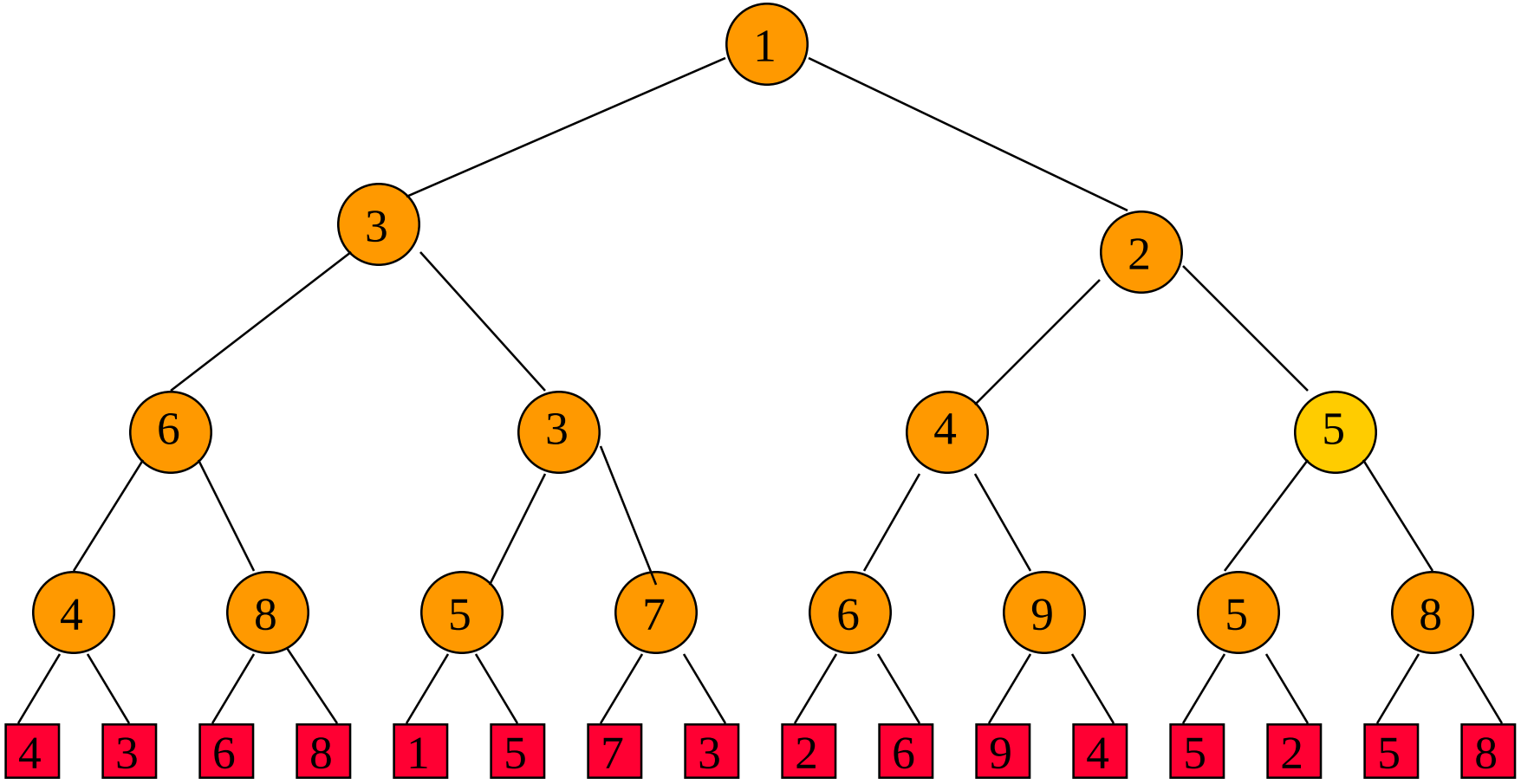
Min Loser Tree For 16 Players



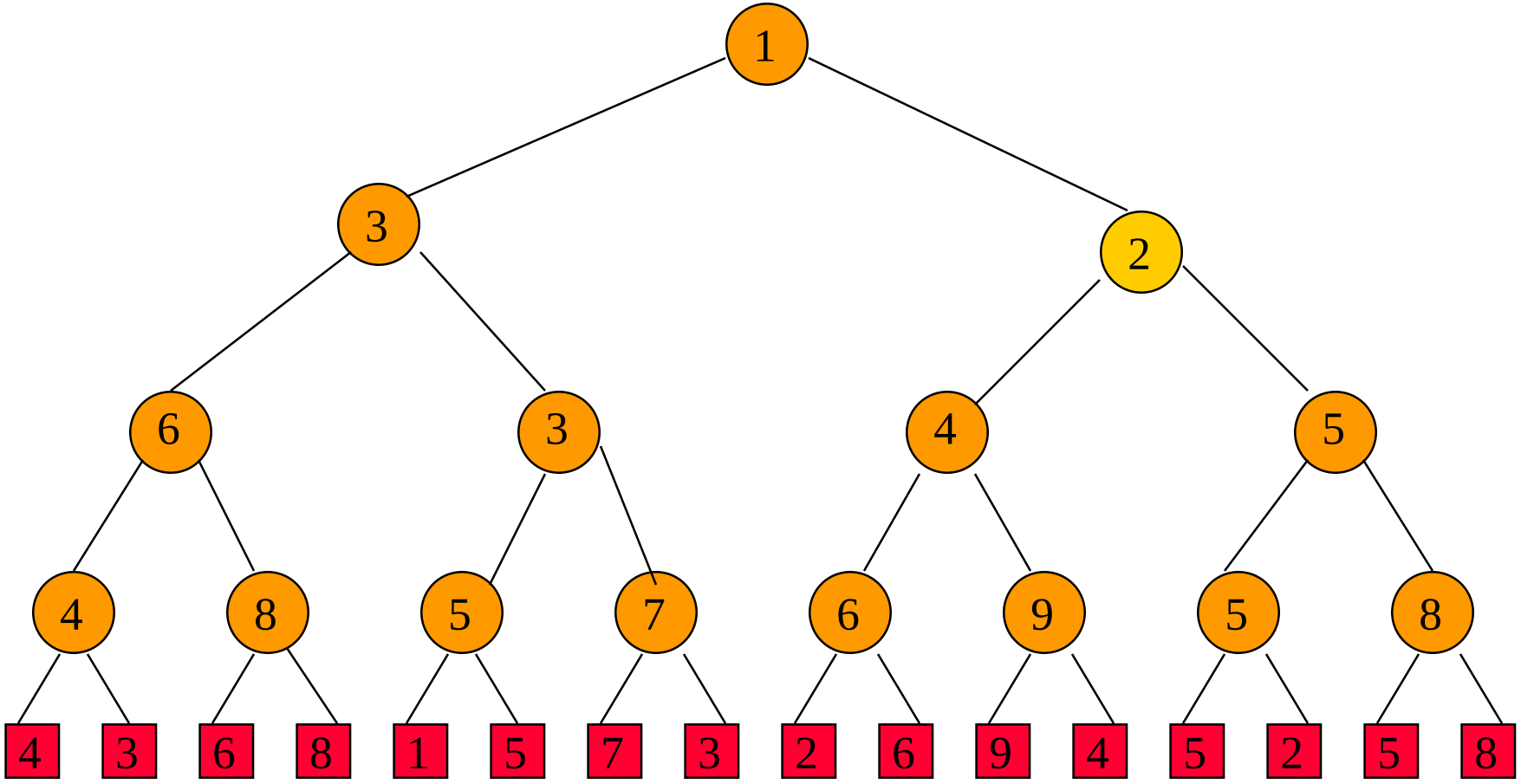
Min Loser Tree For 16 Players



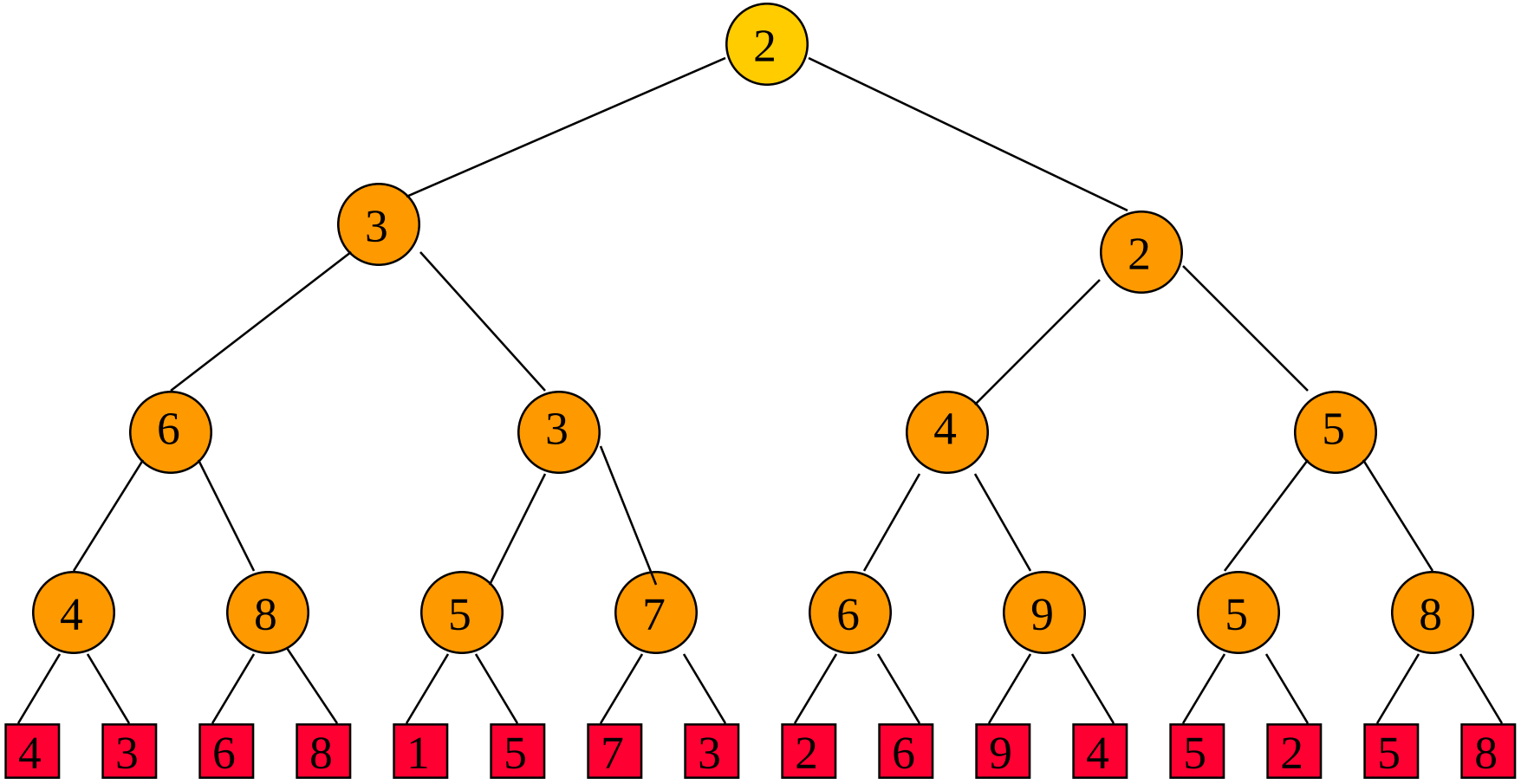
Min Loser Tree For 16 Players

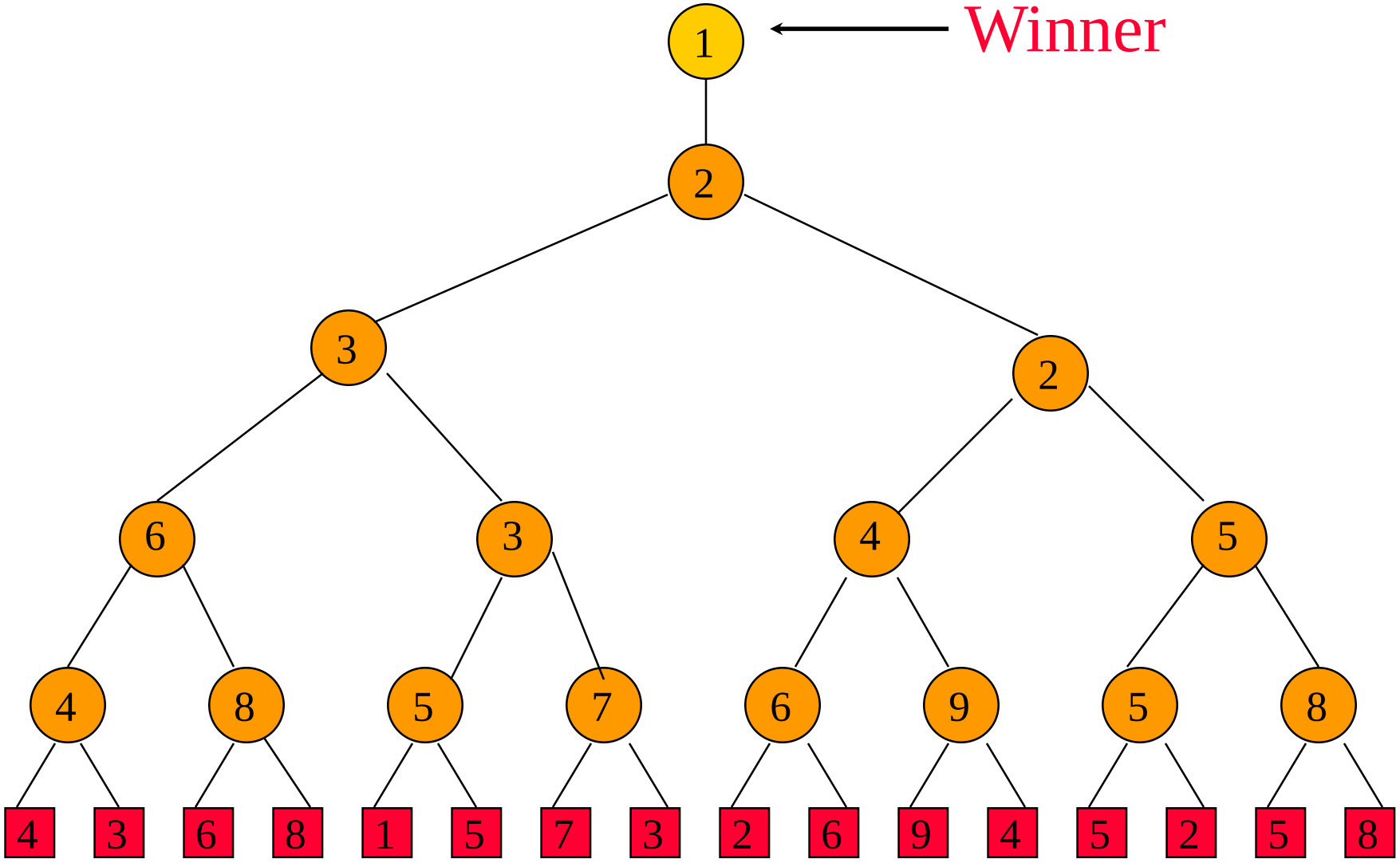


Min Loser Tree For 16 Players



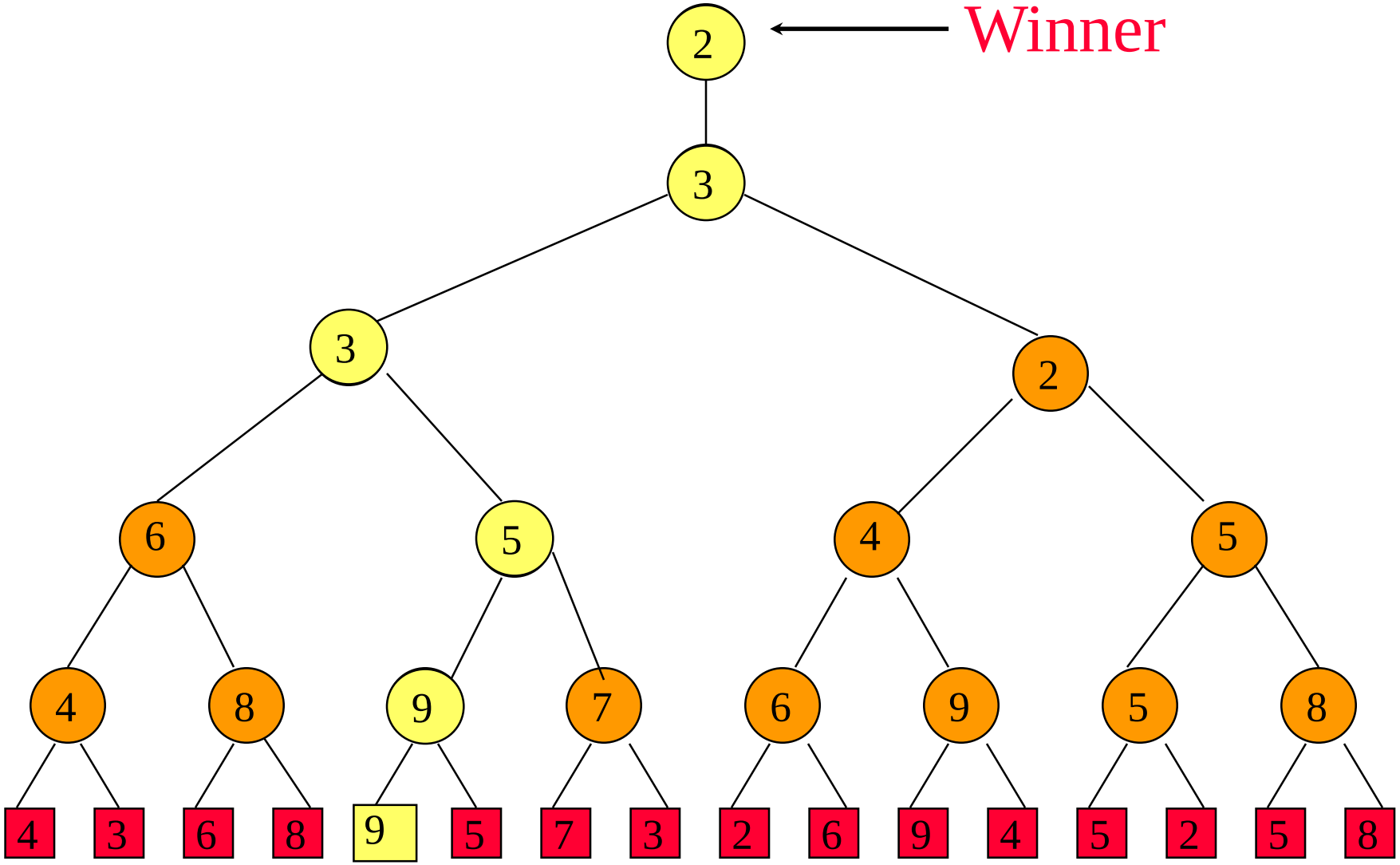
Min Loser Tree For 16 Players




Winner
1
2
3
2
6
3
4
5
4
8
5
7
6
9
5
8
4
3
6
8
1
5
7
3
2
6
9
4
5
2
5
8

Complexity Of Loser Tree Initialize

- One match at each match node.
- One store of a left child winner.
- Total time is O(n).
- More precisely Theta(n).



Replace winner with 9 and replay matches.

Complexity Of Replay

- One match at each level that has a match node.
- O(log n)
- More precisely Theta(log n).

(*not covered in class*)
More Selection Tree Applications

- k-way merging of runs during an external merge sort
- Truck loading

(*not covered in class*)
(*Section 7-4*)

An Introduction to Disjoint Sets

Disjoint Sets

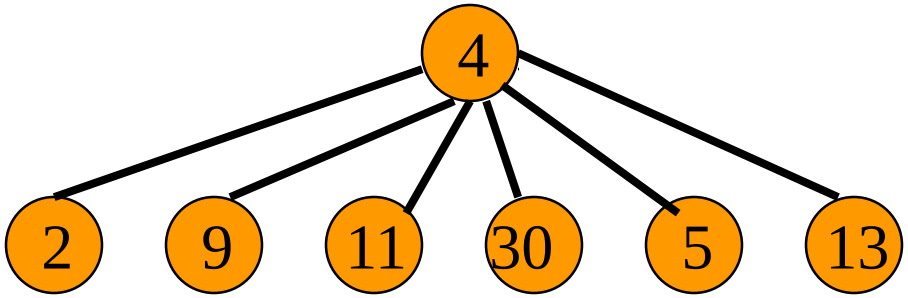
- Given a set {1, 2, …, n} of n elements.
- Initially each element is in a different set.
  - {1}, {2}, …, {n}
- An intermixed sequence of union and find operations is performed.
- A union operation combines two sets into one.
  - Each of the n elements is in exactly one set at any time.
- A find operation identifies the set that contains a particular element.

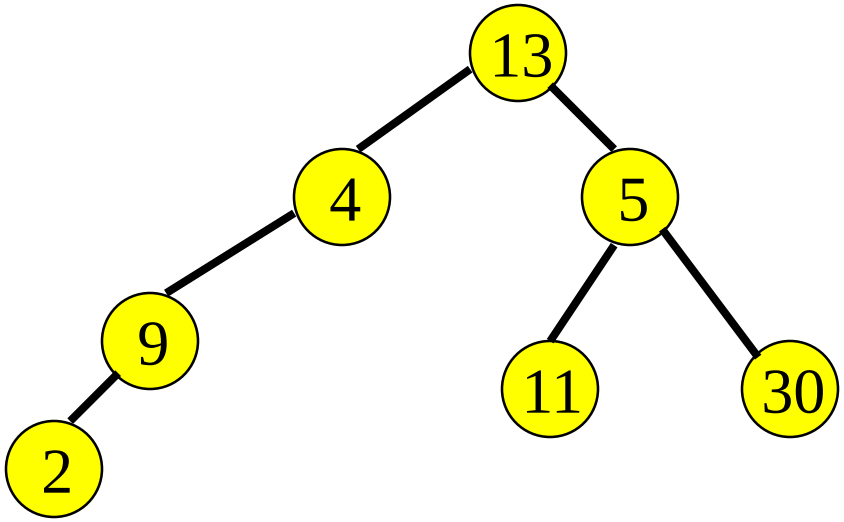
Using Arrays And Chains

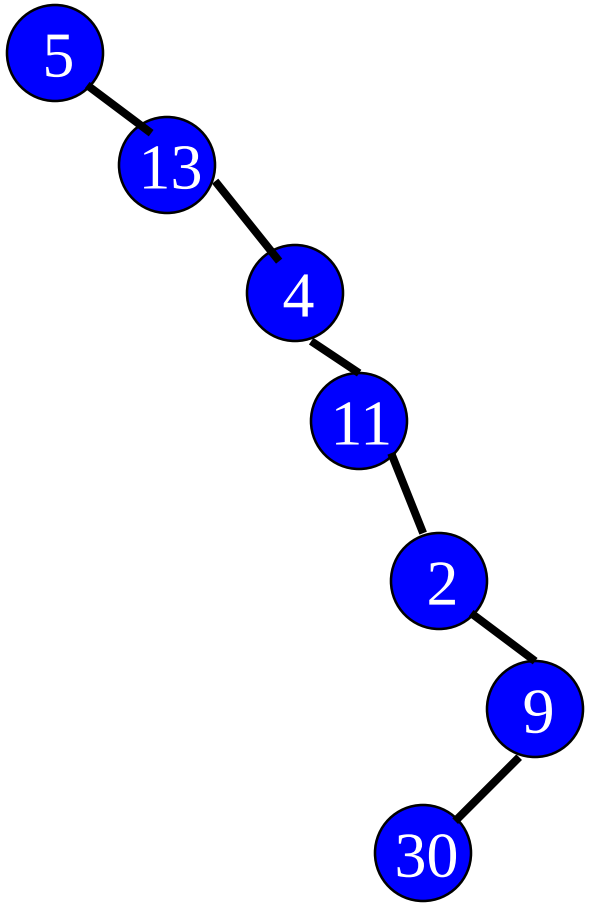
- Best time complexity using arrays and chains is $O(n + u \log u + f)$, where u and f are, respectively, the number of union and find operations that are done.
- Using a tree (not a binary tree) to represent a set, the time complexity becomes almost $O(n + f)$ (assuming at least $n/2$ union operations).

A Set As A Tree

- S = {2, 4, 5, 9, 11, 13, 30}
- Some possible tree representations:

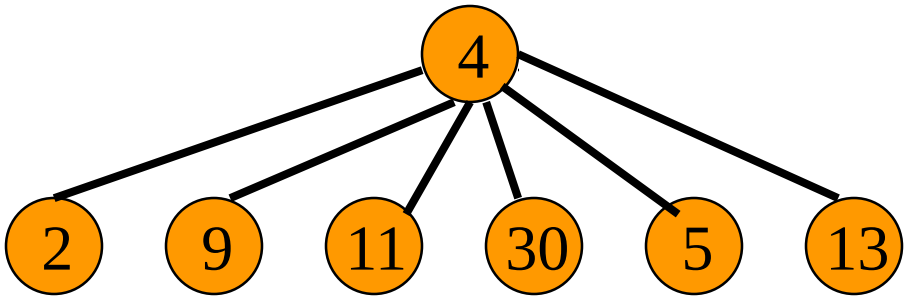






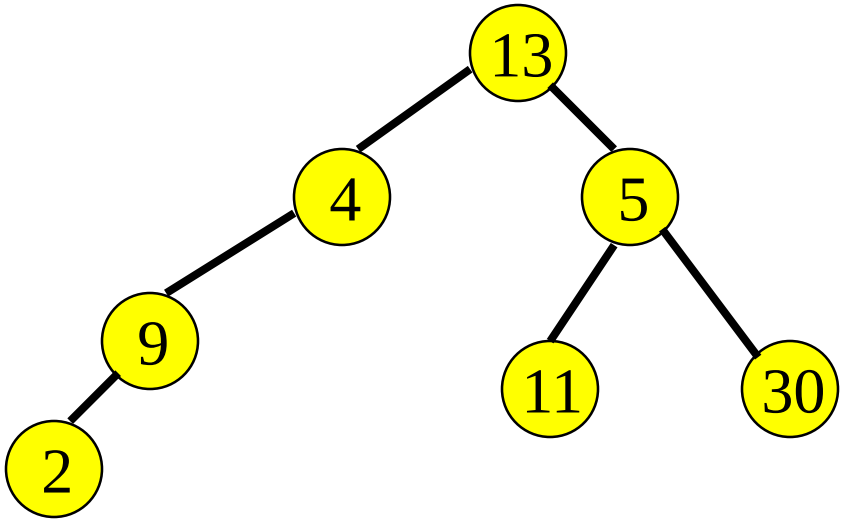
Result Of A Find Operation

- Find(i) is to identify the set that contains element i.
- In most applications of the union-find problem, the user does not provide set identifiers.
- The requirement is that Find(i) and Find(j) return the same value iff elements i and j are in the same set.



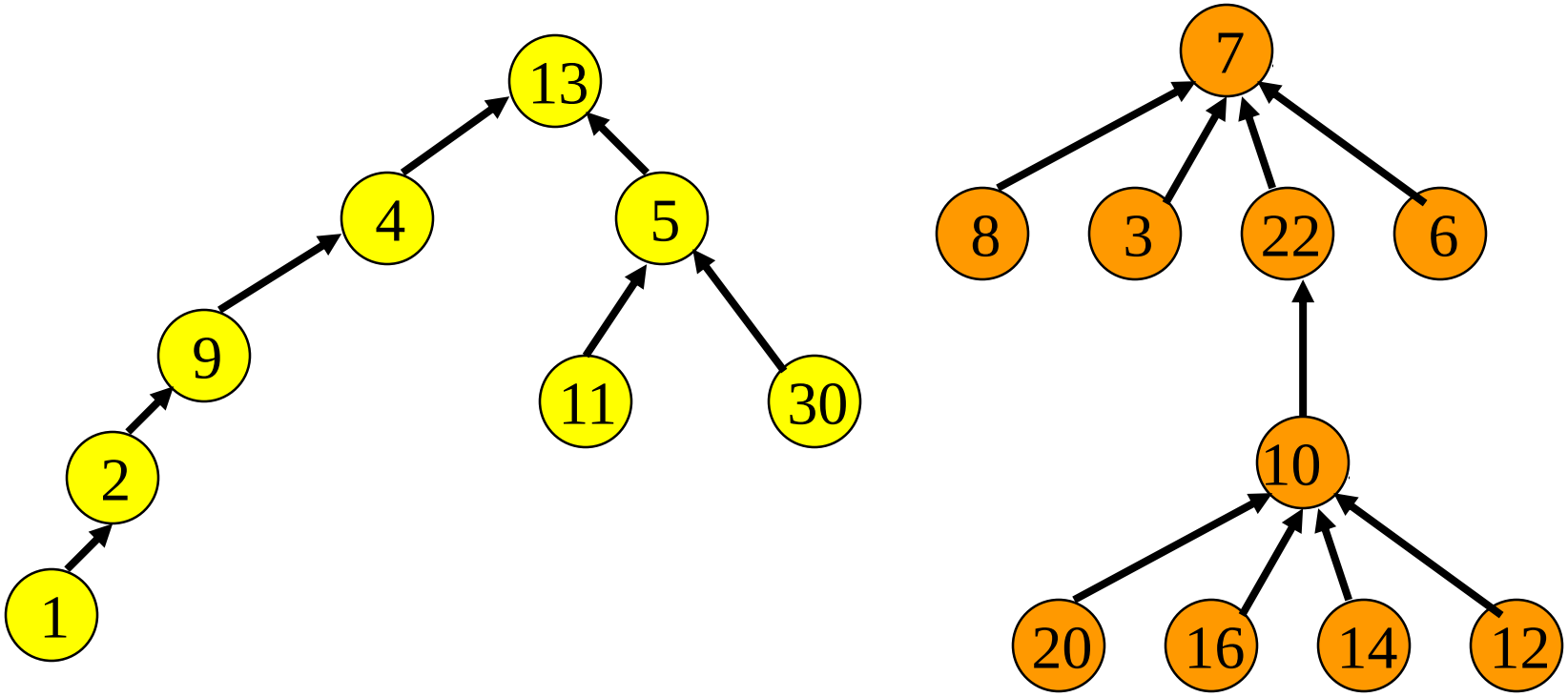
Find(i) will return the element that is in the tree root.

Strategy For Find(i)



- Start at the node that represents element i and climb up the tree until the root is reached.
- Return the element in the root.
- To climb the tree, each node must have a parent pointer.

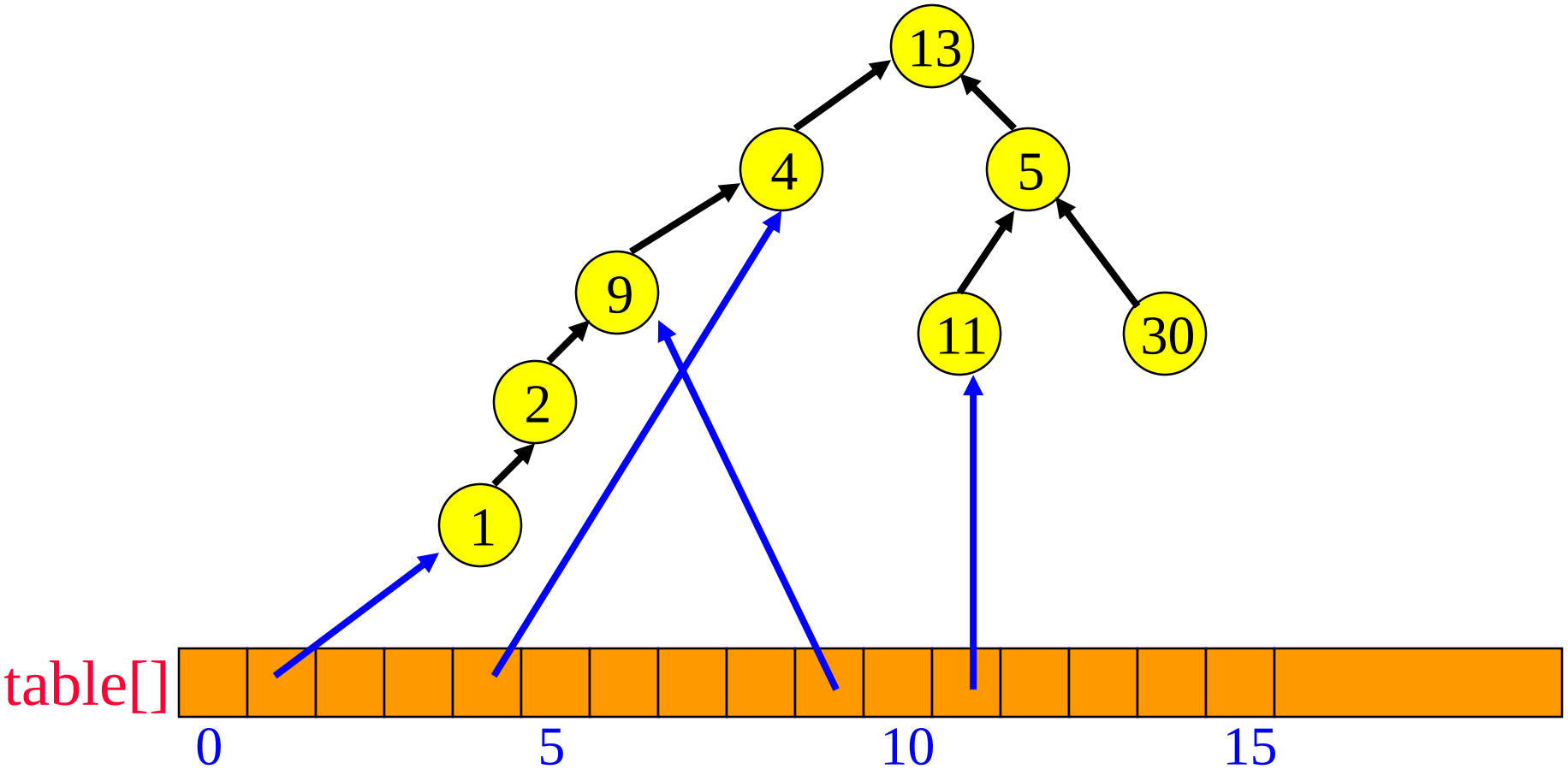
Trees With Parent Pointers



Possible Node Structure

- Use nodes that have two fields: element and parent.
  - Use an array table[] such that table[i] is a pointer to the node whose element is i.
  - To do a Find(i) operation, start at the node given by table[i] and follow parent fields until a node whose parent field is null is reached.
  - Return element in this root node.

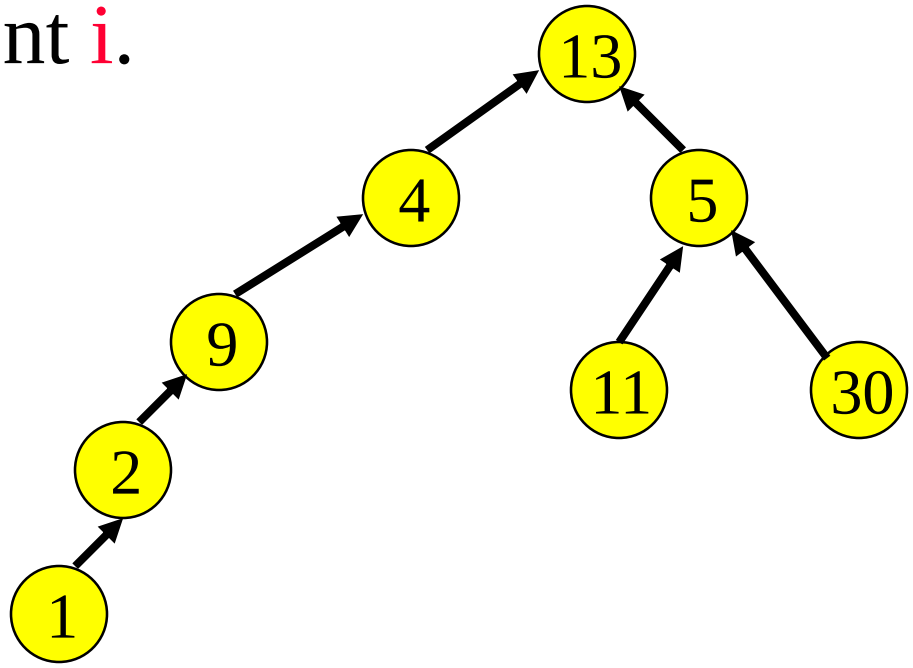
Example

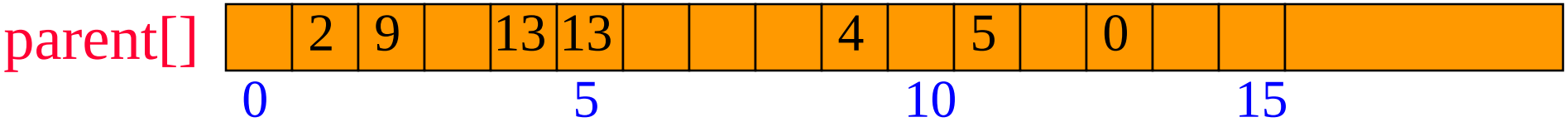


(Only some table entries are shown.)

Better Representation

- Use an integer array parent[] such that parent[i] is the element that is the parent of element i.

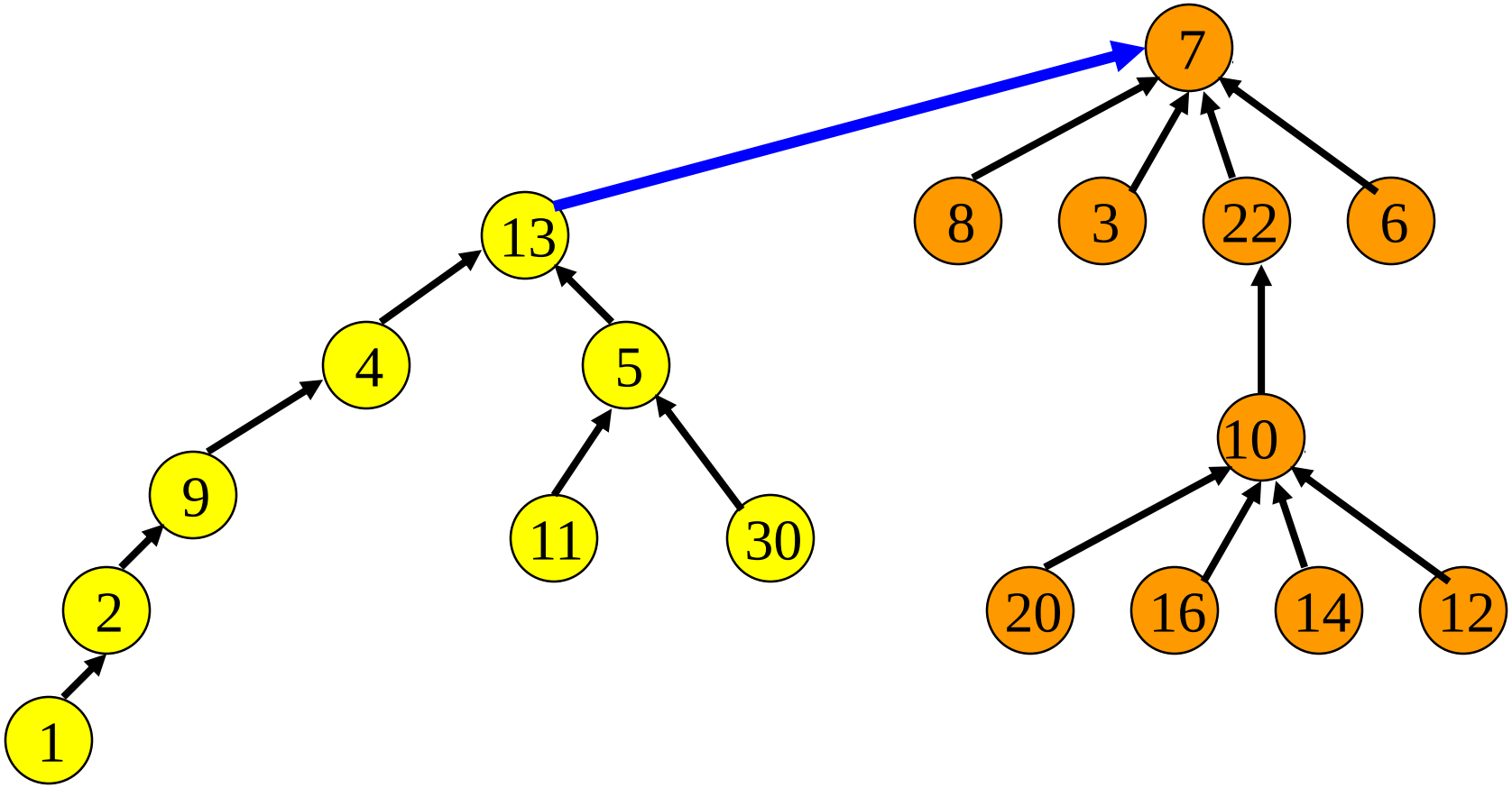




Union Operation

- Union(i,j)
  - i and j are the roots of two different trees, i != j.
- To unite the trees, make one tree a subtree of the other.
  - parent[j] = i

Union Example



- Union(7,13)

The Union Method

```
void SimpleUnion(int i, int j)
    {parent[i] = j;}
```

Time Complexity Of SimpleUnion()

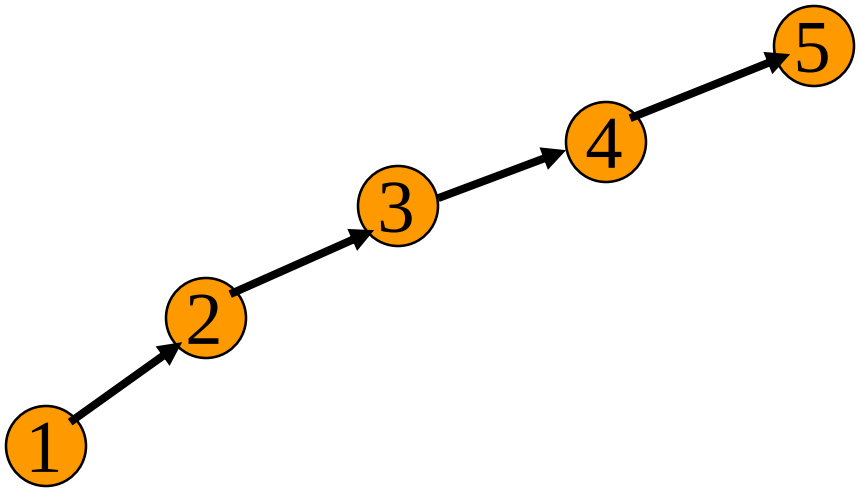
- O(1)

The Find Method

```
int SimpleFind(int i)
{
    while (parent[i] >= 0)
      i = parent[i];  // move up the tree
    return i;
}
```

Time Complexity of SimpleFind()

- Tree height may equal number of elements in tree.
  - Union(2,1), Union(3,2), Union(4,3), Union(5,4)…

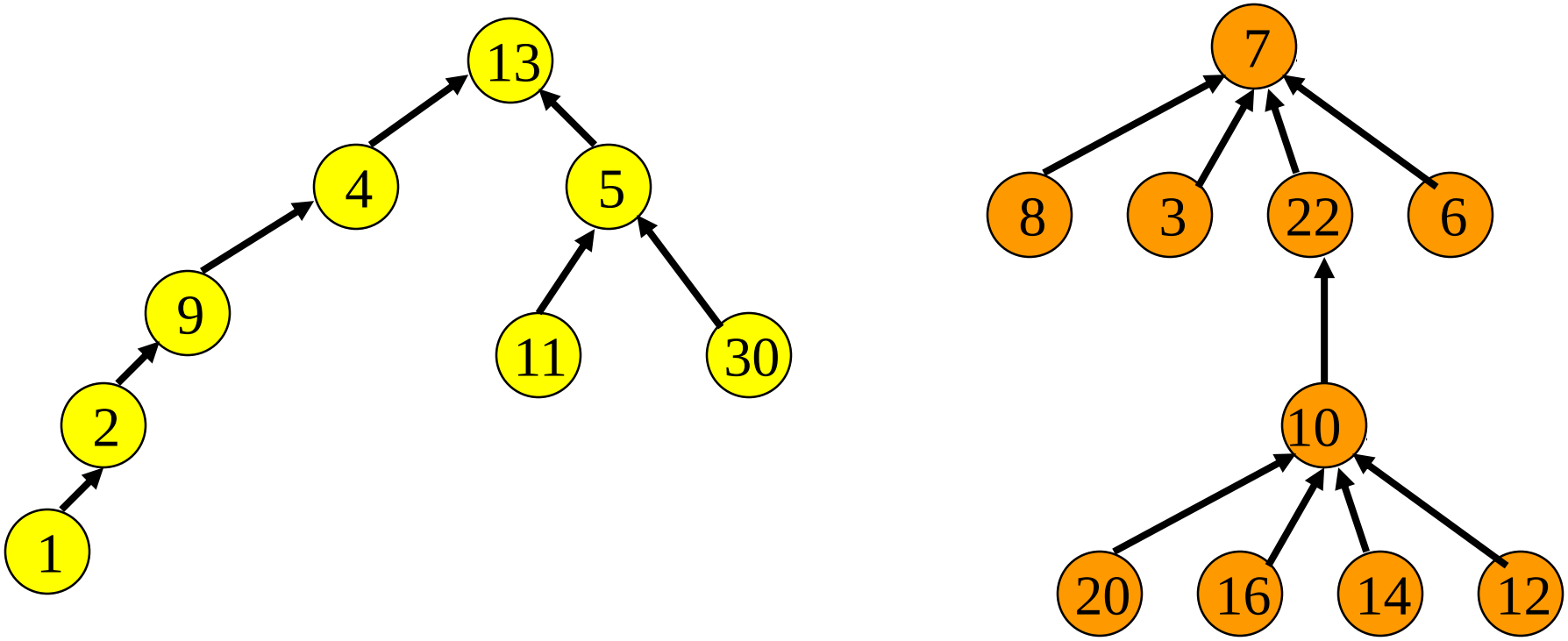


So complexity is O(u).

u Unions and f Find Operations

- $O(u + uf) = O(uf)$
- Time to initialize $parent[i] = 0$ for all i is $O(n)$.
- Total time is $O(n + uf)$.
- Worse than using a chain!
- Back to the drawing board.

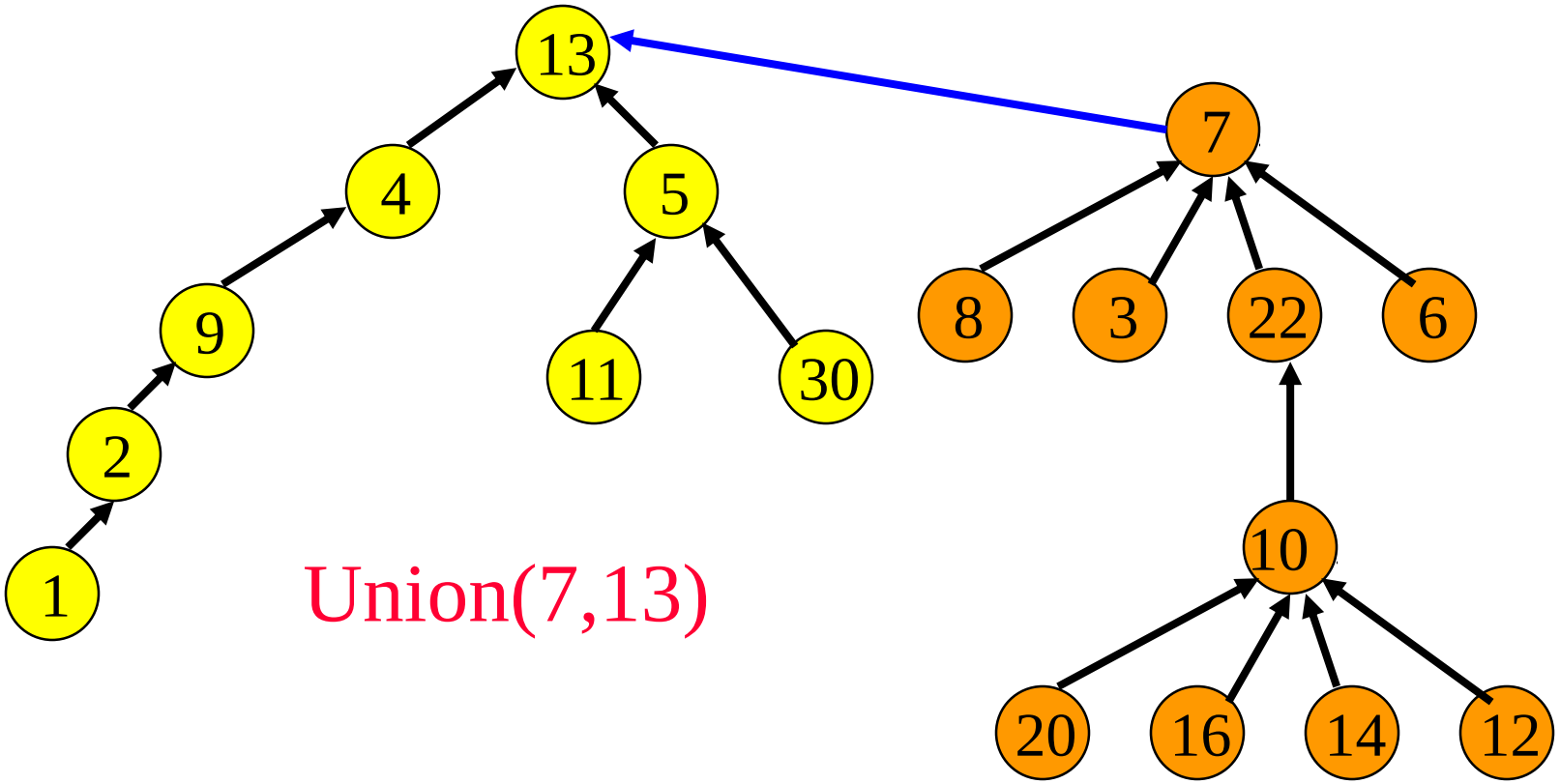
Smart Union Strategies



- Union(7,13)
- Which tree should become a subtree of the other?

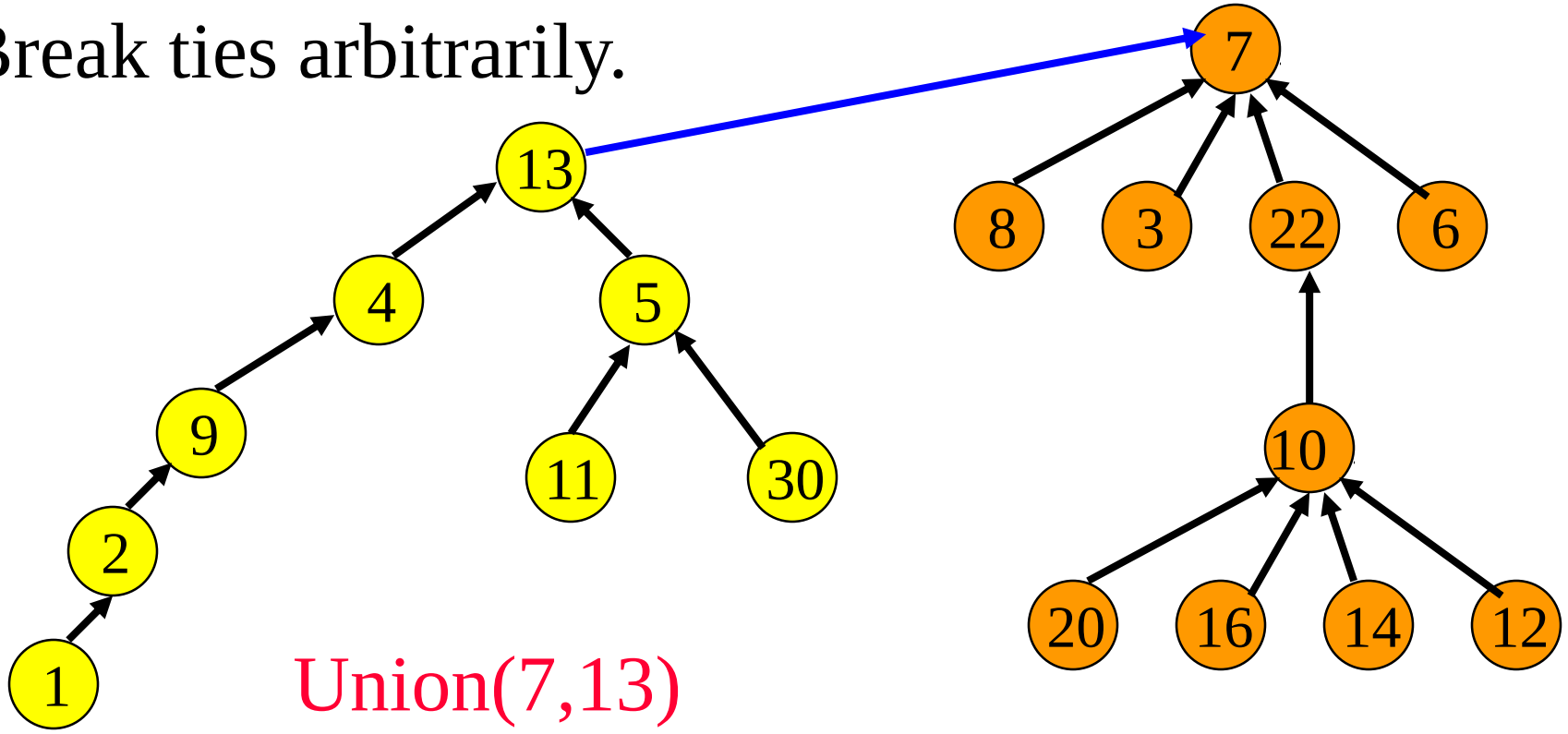
Height Rule

- Make tree with smaller height a subtree of the other tree.
- Break ties arbitrarily.



Weight Rule

- Make tree with fewer number of elements a subtree of the other tree.
- Break ties arbitrarily.



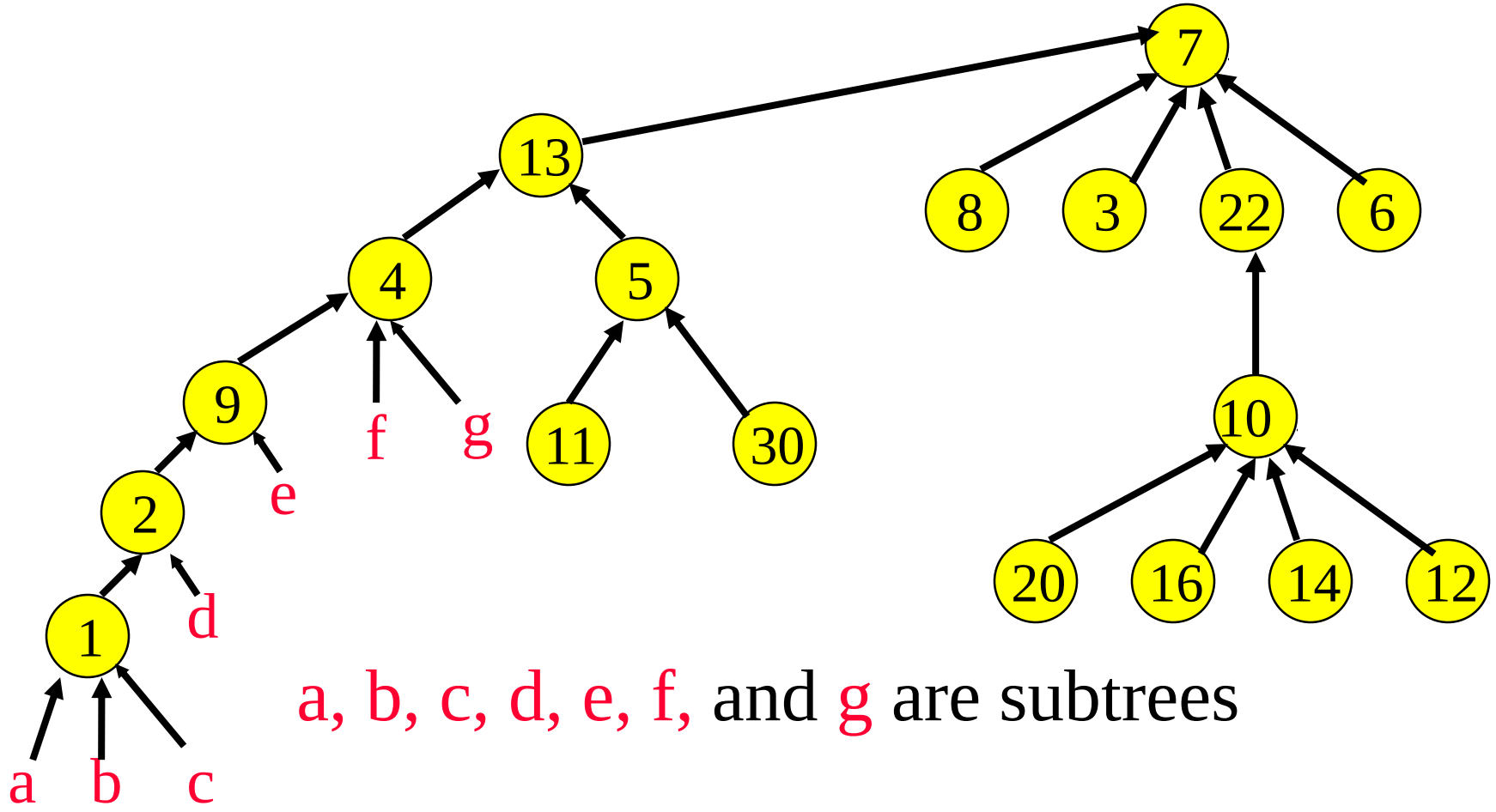
Implementation

- Root of each tree must record either its height or the number of elements in the tree.
- When a union is done using the height rule, the height increases only when two trees of equal height are united.
- When the weight rule is used, the weight of the new tree is the sum of the weights of the trees that are united.

Height Of A Tree

- Suppose we start with single element trees and perform unions using either the height or the weight rule.
- The height of a tree with p elements is at most floor $(\log_2 p) + 1$.
- Proof is by induction on p. See text.

Sprucing Up The Find Method

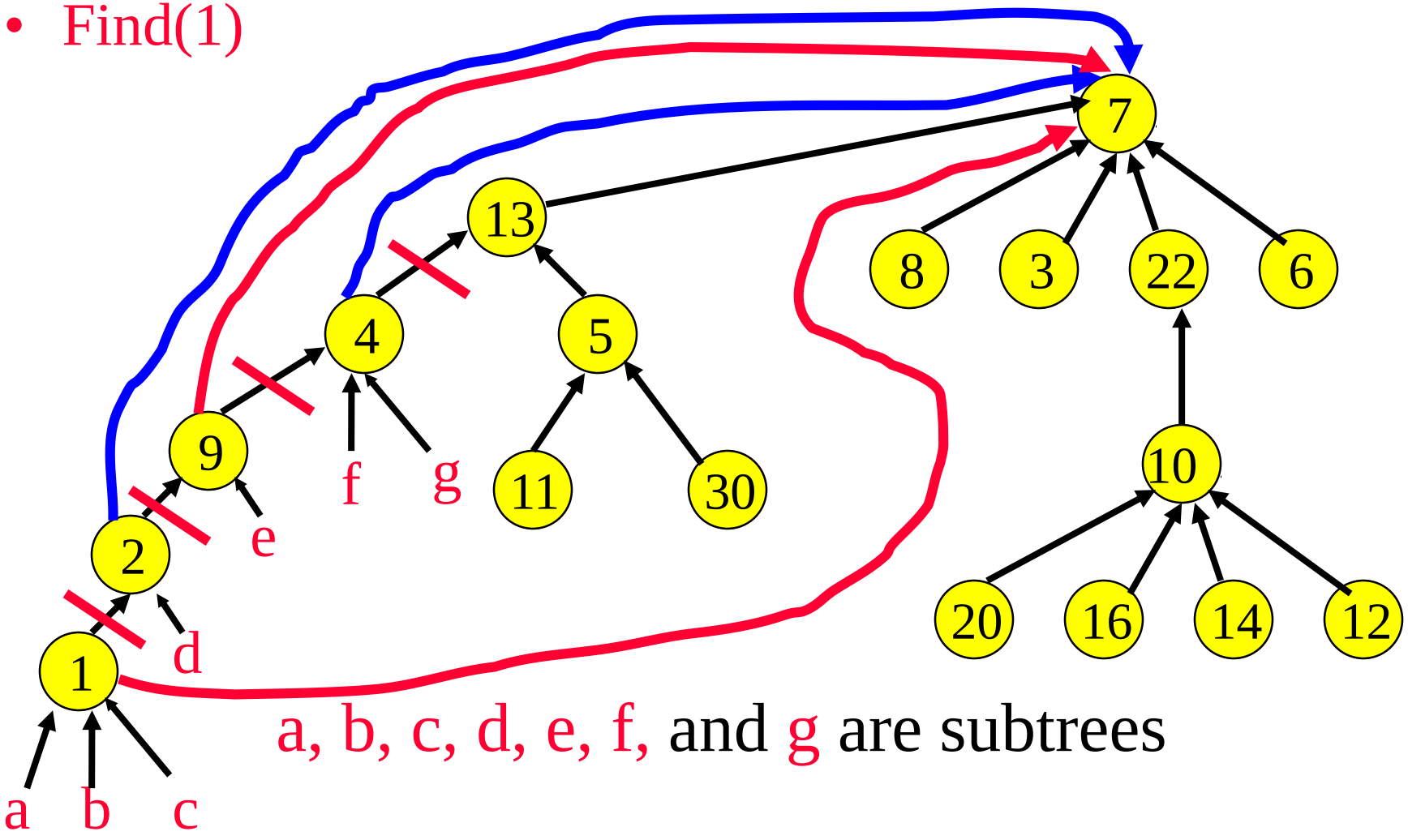


a, b, c, d, e, f, and g are subtrees

- Find(1)
- Do additional work to make future finds easier.

Path Compaction

- Make all nodes on find path point to tree root.
- Find(1)

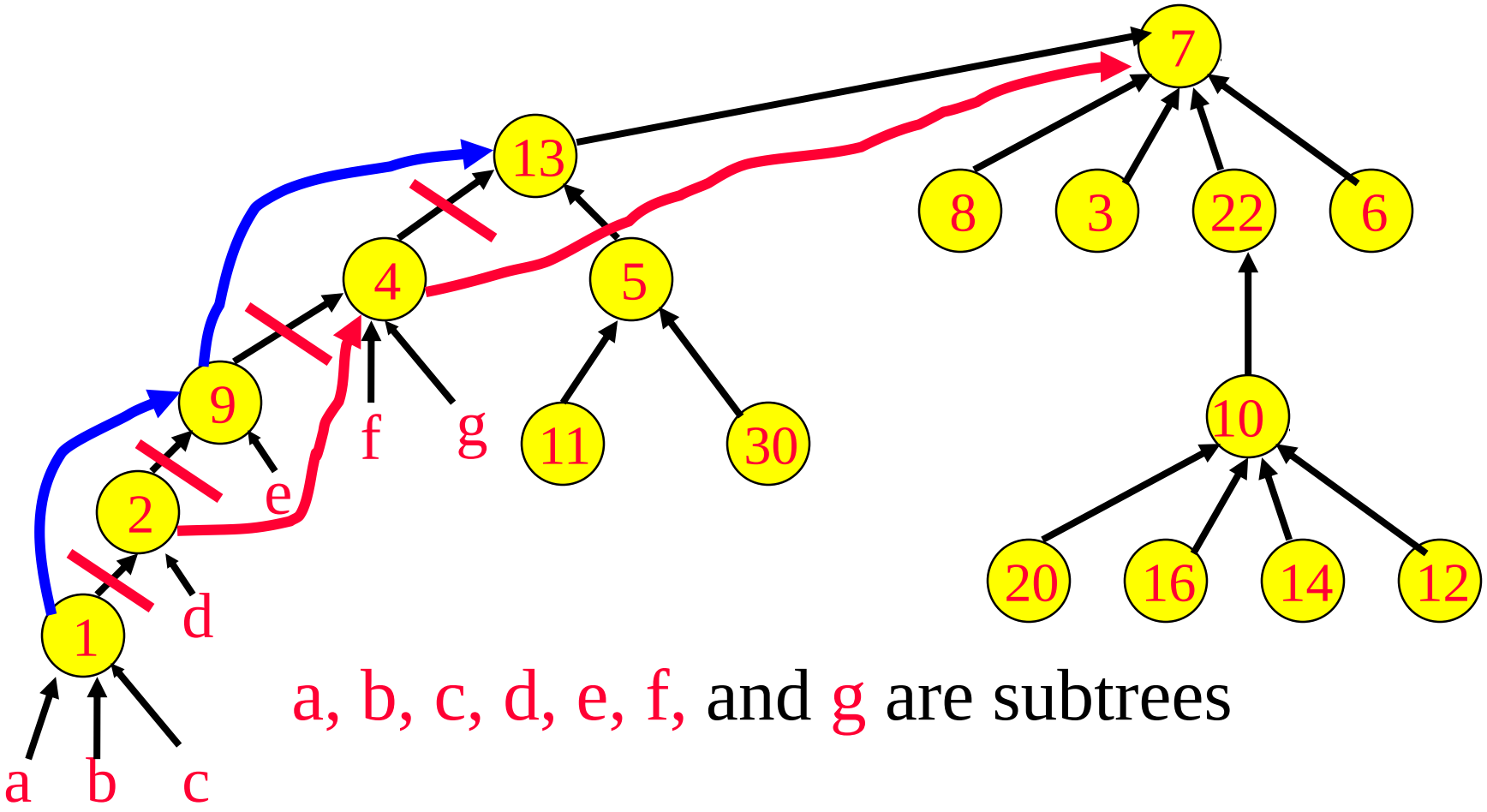


a, b, c, d, e, f, and g are subtrees

Makes two passes up the tree.

Path Splitting

- Nodes on find path point to former grandparent.
- Find(1)

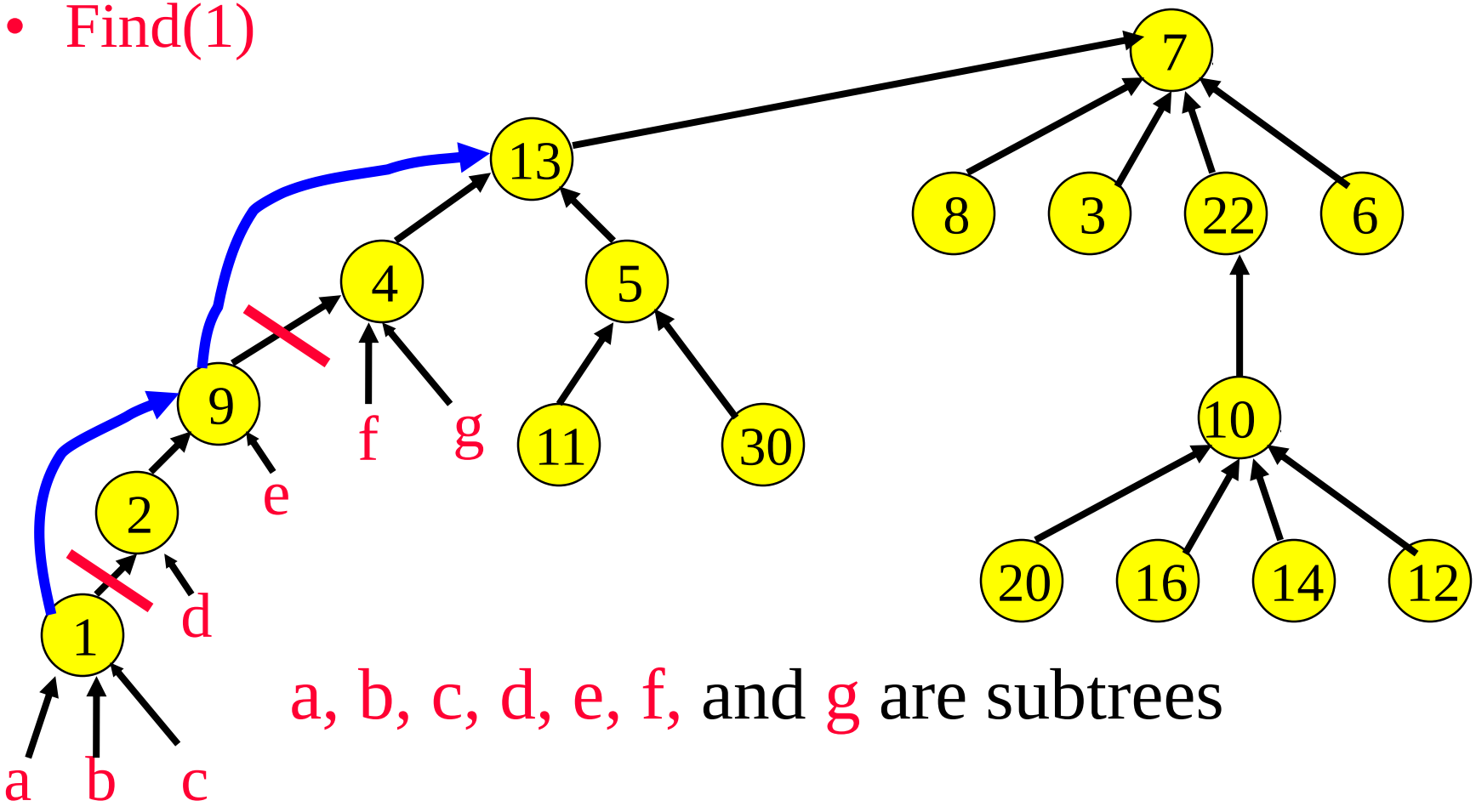


a, b, c, d, e, f, and g are subtrees

Makes only one pass up the tree.

Path Halving

- Parent pointer in every other node on find path is changed to former grandparent.
- Find(1)



a, b, c, d, e, f, and g are subtrees

Changes half as many pointers.

Time Complexity

- Ackermann's function.
  - $A(i,j) = 2^j$, $i = 1$ and $j >= 1$
  - $A(i,j) = A(i-1,2)$, $i >= 2$ and $j = 1$
  - $A(i,j) = A(i-1,A(i,j-1))$, $i, j >= 2$
- Inverse of Ackermann's function.
  - $\alpha(p,q) = \min\{z>=1 \mid A(z, p/q) > \log_2 q\}$, $p >= q >= 1$

Time Complexity

- Ackermann's function grows very rapidly as i and j are increased.
  - $A(2,4) = 2^{65,536}$
- The inverse function grows very slowly.
  - $\alpha(p,q) < 5$ until $q = 2^{A(4,1)}$
  - $A(4,1) = A(2,16) >>>> A(2,4)$
- In the analysis of the union-find problem, q is the number, n, of elements; $p = n + f$; and $u >= n/2$.
- For all practical purposes, $\alpha(p,q) < 5$.

Time Complexity

Lemma 5.6 [Tarjan and Van Leeuwen]

Let $T(f,u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u >= n/2$.

$k_1*(n + f*\alpha(f+n, n)) <= T(f,u) <= k_2*(n + f*\alpha(f+n, n))$

where k_1 and k_2 are constants.

These bounds apply when we start with singleton sets and use either the weight or height rule for unions and any one of the path compression methods for a find.