Chapter Thirteen

Advances in Heap

Section 13.1

Double-Ended Priority Queues

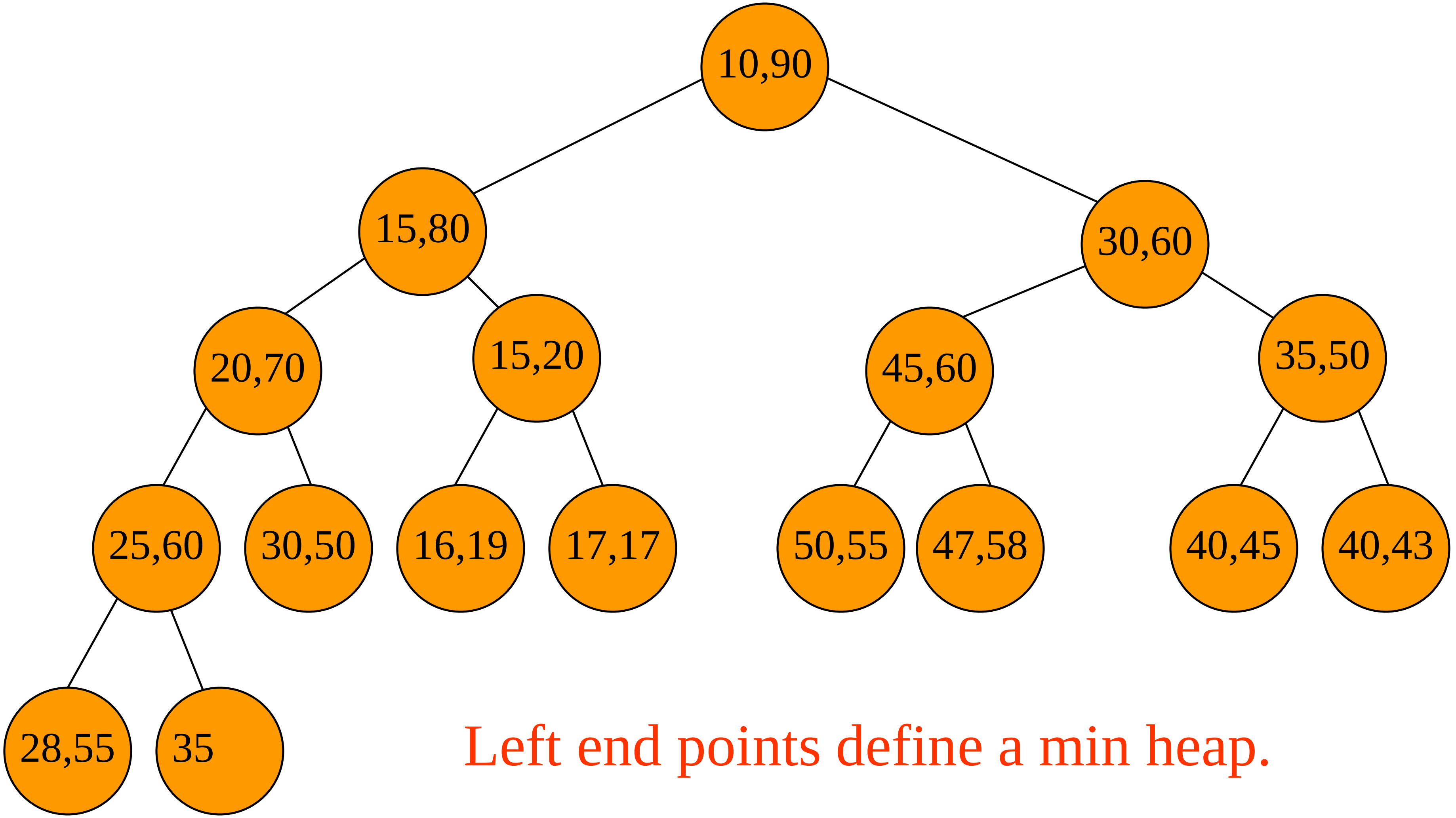
Double-Ended Priority Queues

- Primary operations
  - Insert
  - Delete Max
  - Delete Min
- Note that a single-ended priority queue supports just one of the above remove operations.

Interval Heaps

- Complete binary tree.
- Each node (except possibly last one) has 2 elements.
- Last node has 1 or 2 elements.
- Let a and b be the elements in a node P, a <= b.
- [a, b] is the interval represented by P.
- The interval represented by a node that has just one element a is [a, a].
- The interval [c, d] is contained in interval [a, b] iff a <= c <= d <= b.
- In an interval heap each node's (except for root) interval is contained in that of its parent.

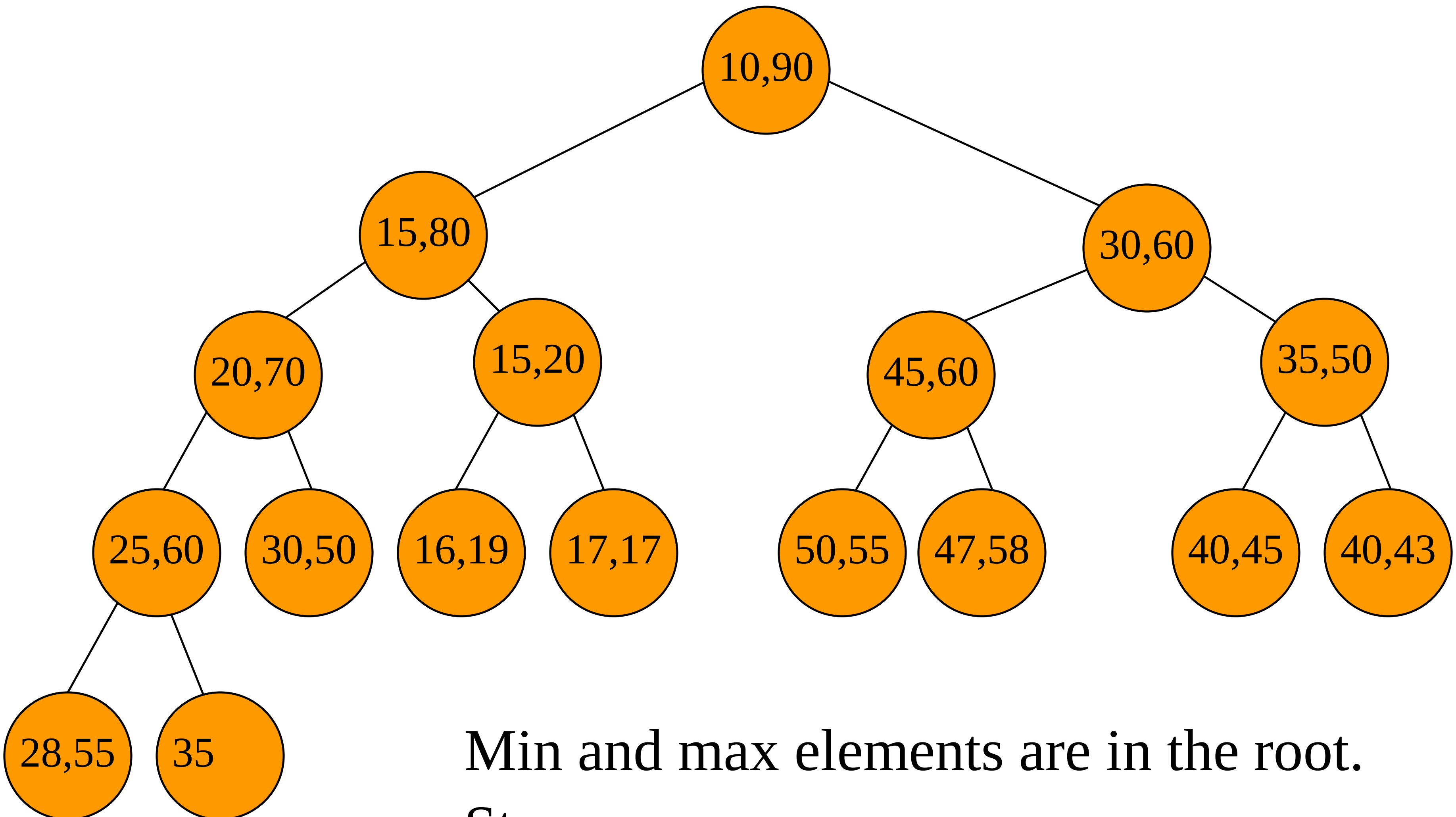
Example Interval Heap



Left end points define a min heap.

Right end points define a max heap.

Example Interval Heap

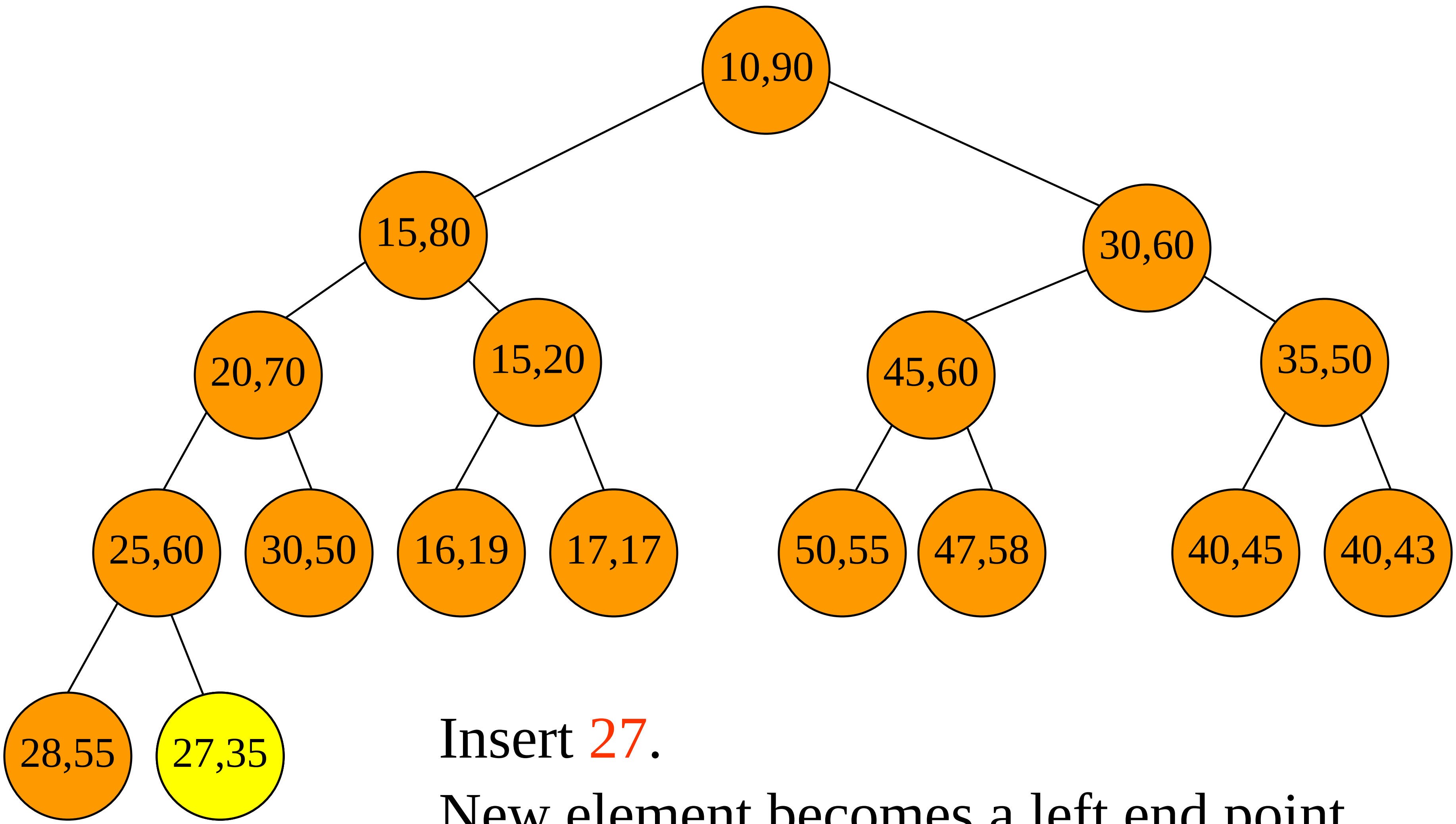


Min and max elements are in the root.

Store as an array.

Height is ~$\log_2 n$.

Insert An Element

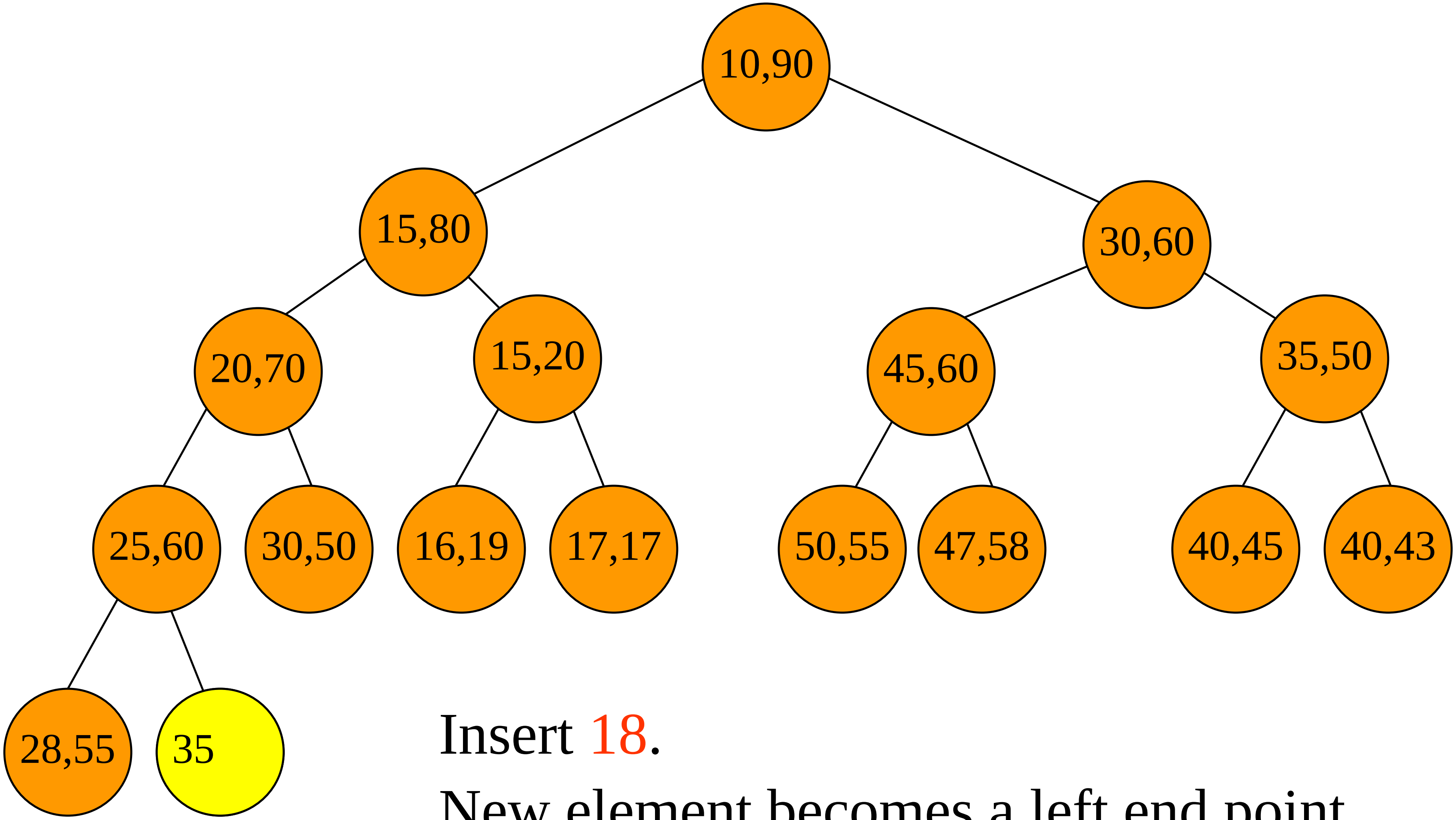


Insert 27.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

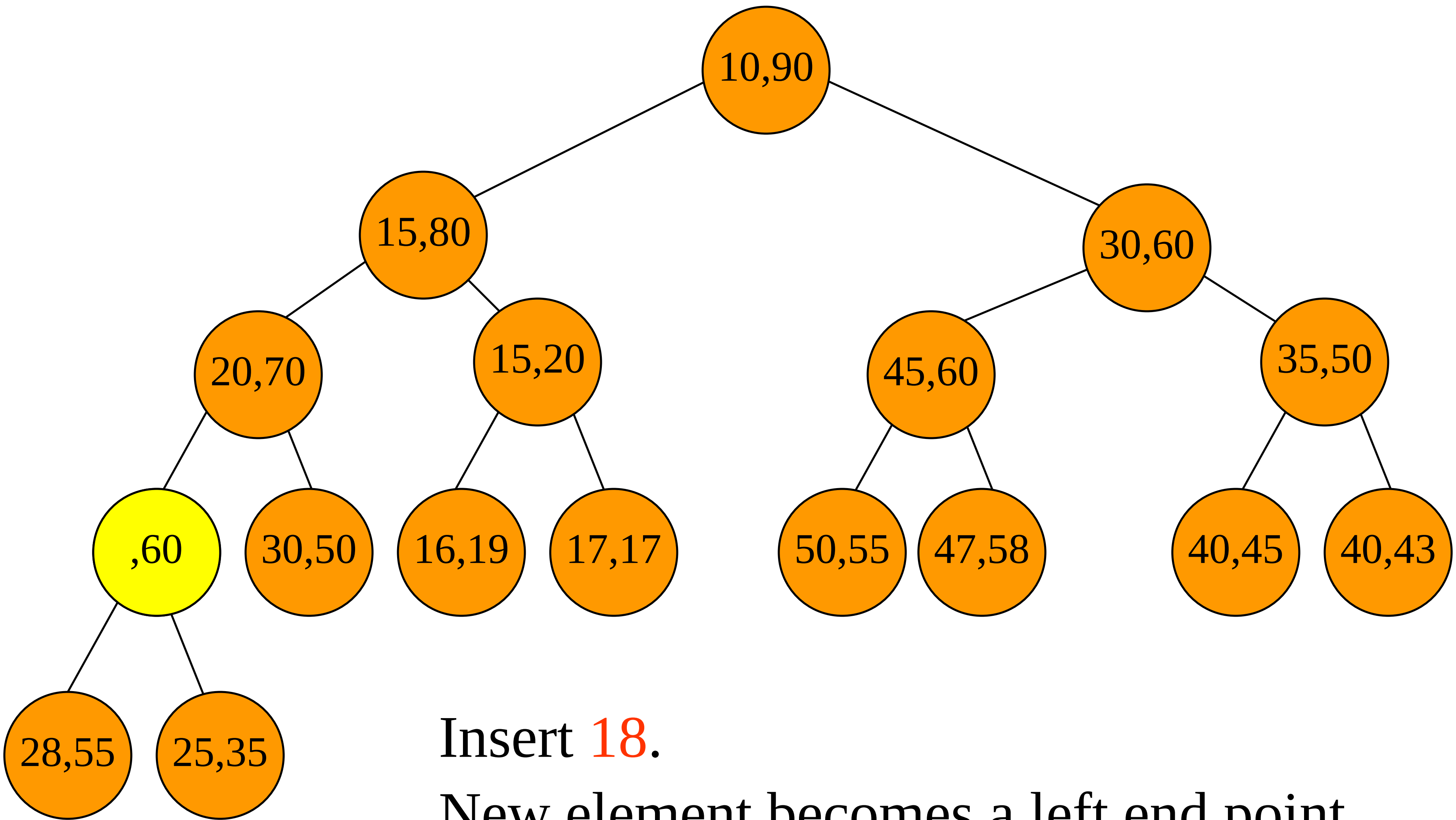


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

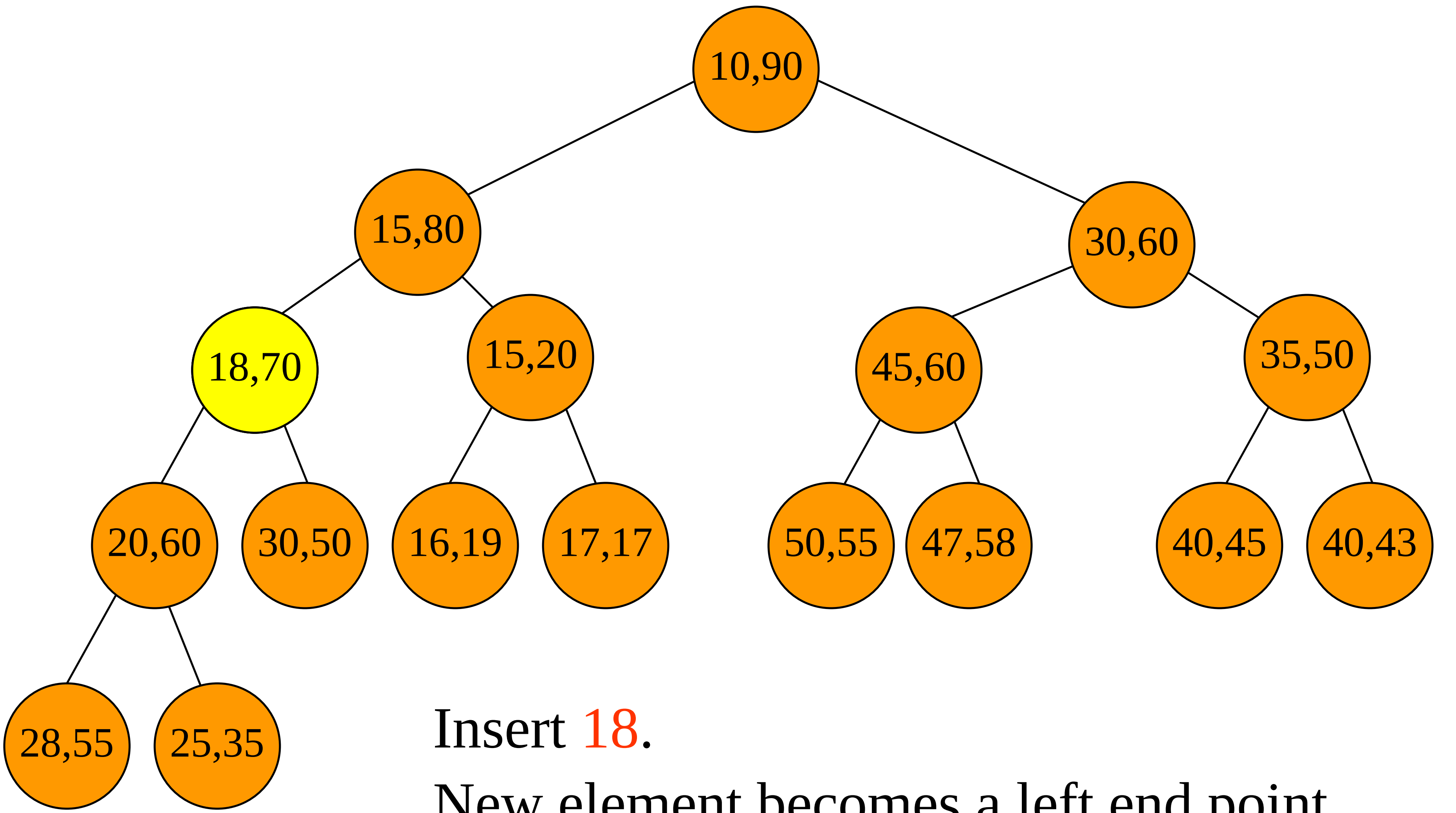


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

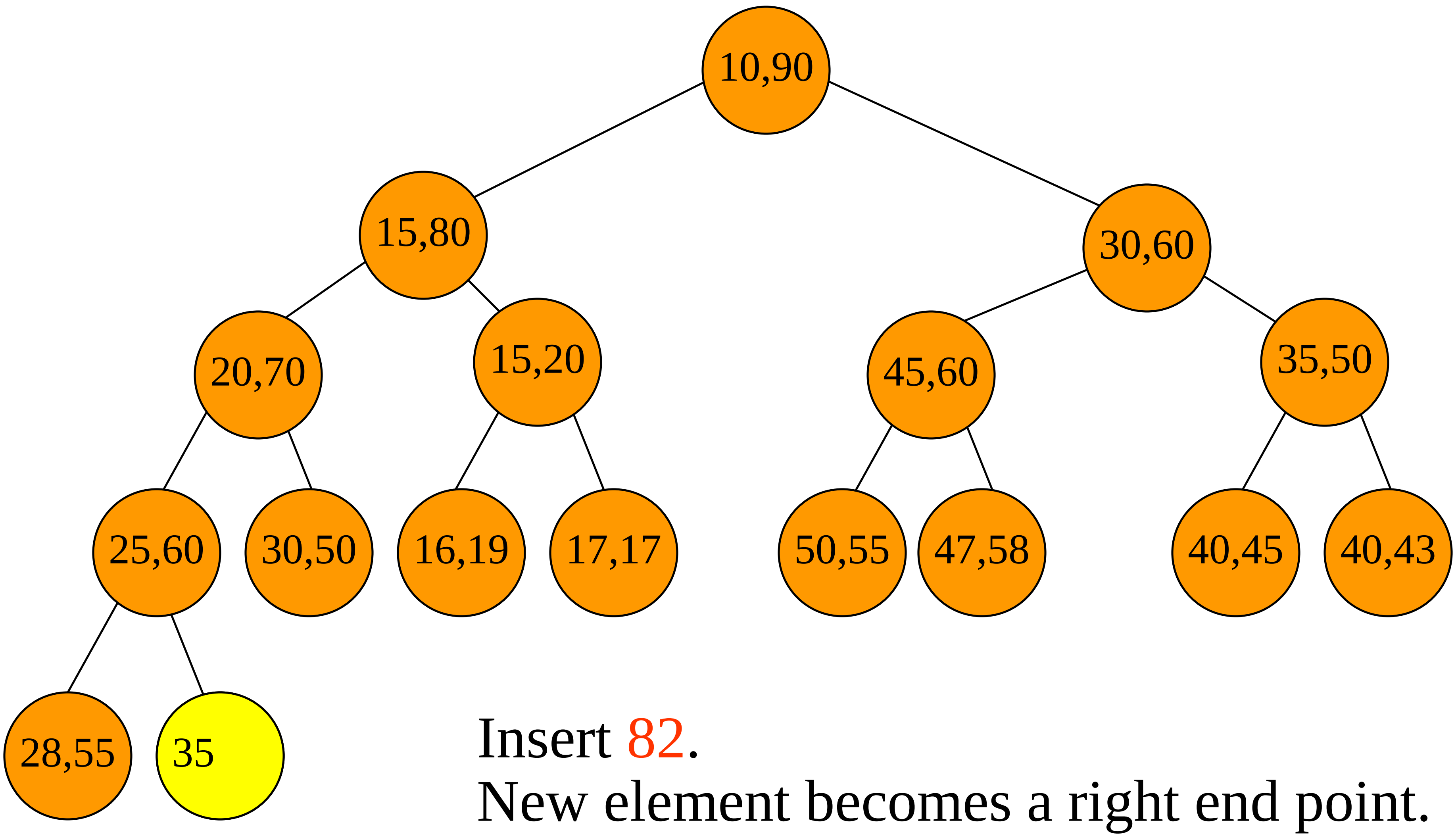


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

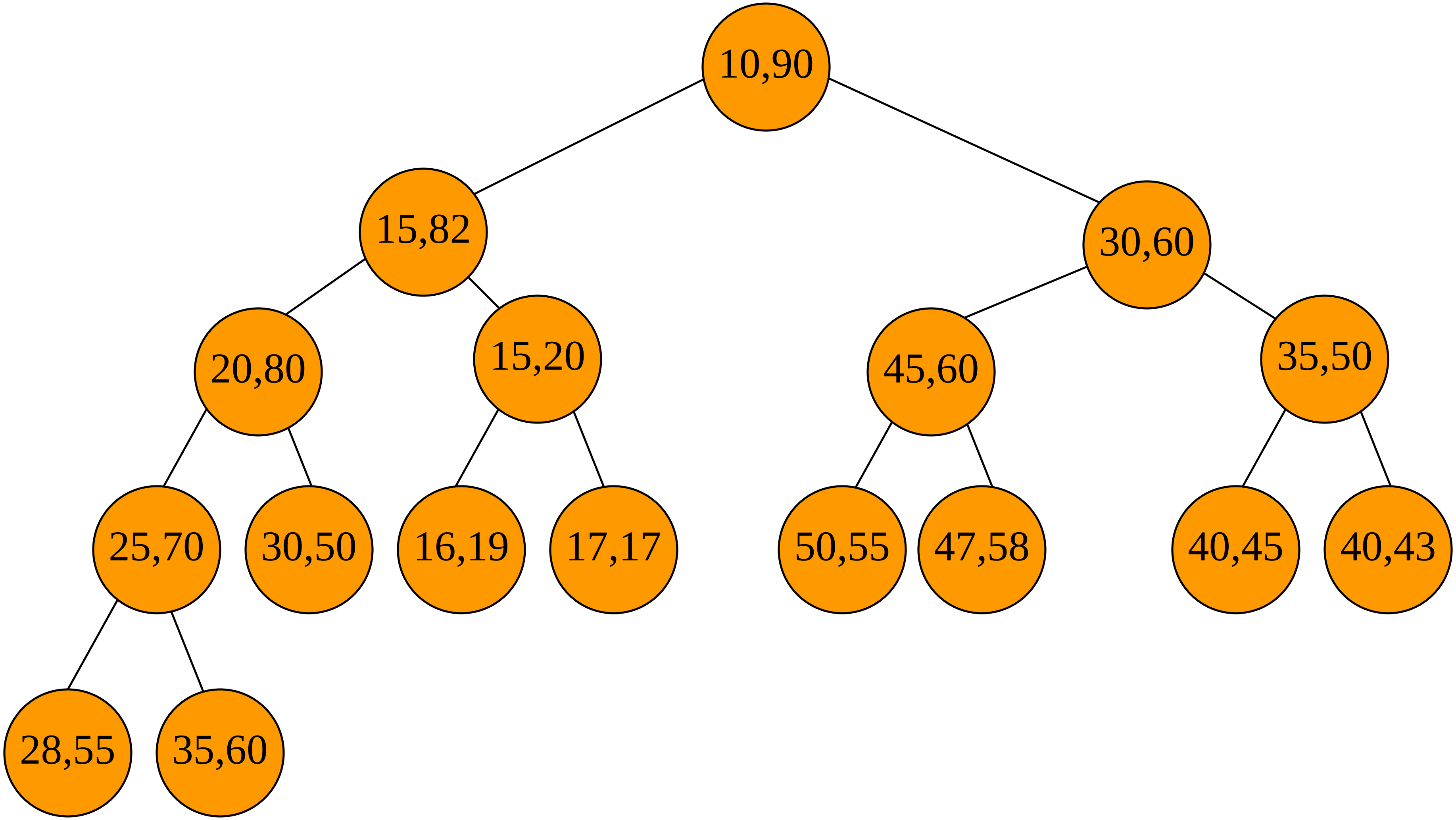
Yet Another Insert



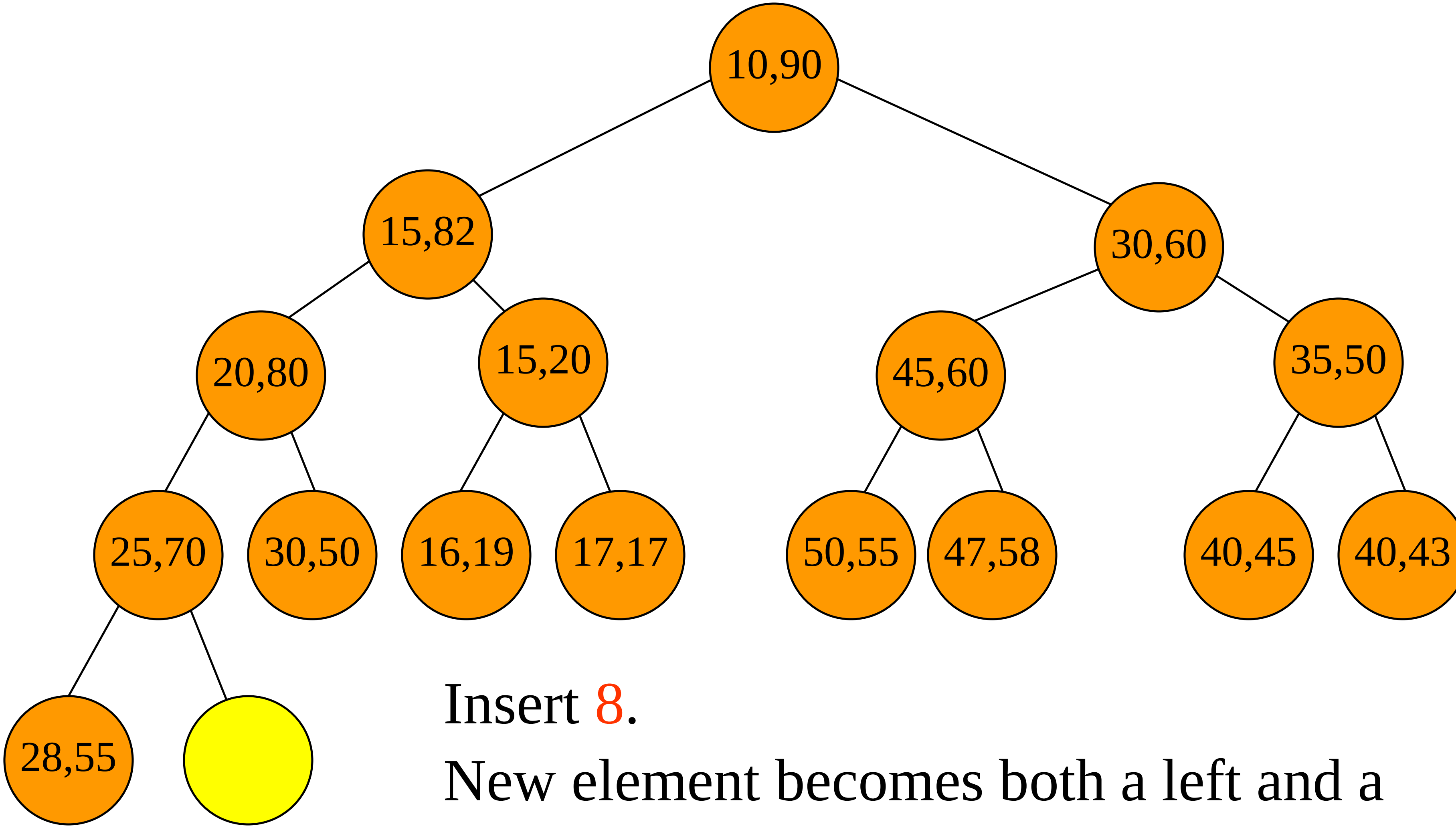
Insert 82.
New element becomes a right end point.

Insert new element into max heap.

After 82 Inserted



One More Insert Example

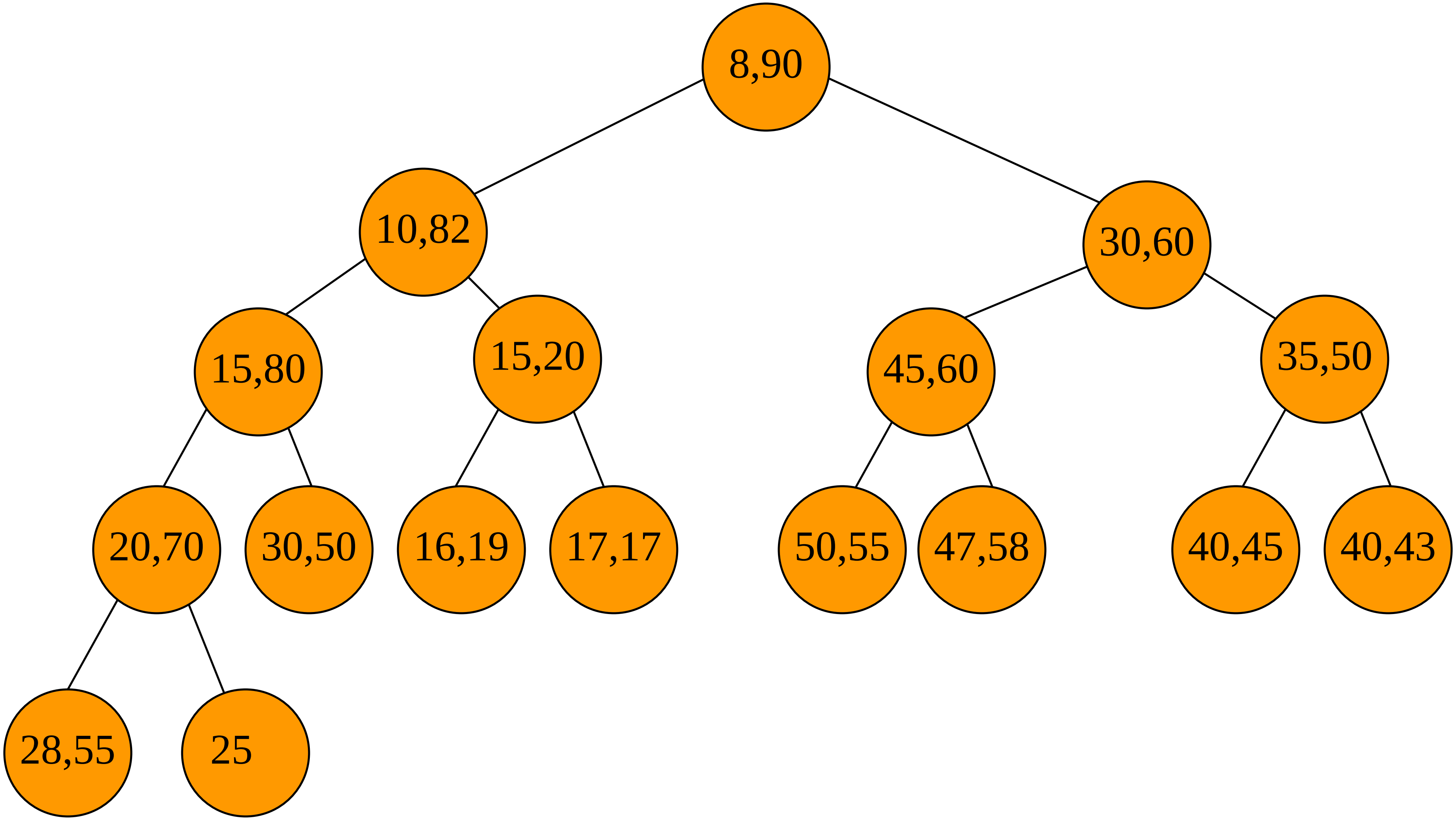


Insert 8.

New element becomes both a left and a right end point.

Insert new element into min heap.

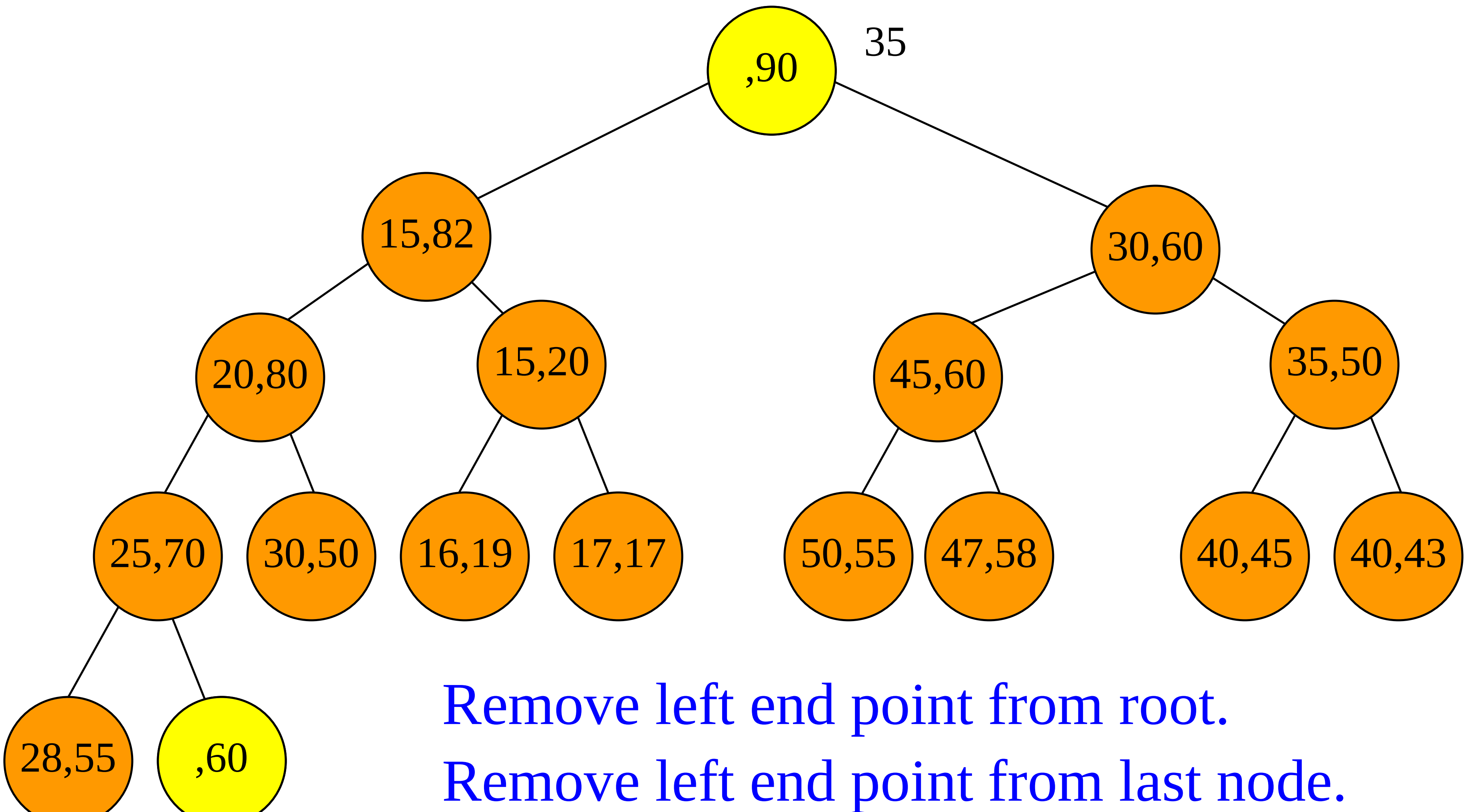
After 8 Is Inserted



Remove Min Element

- n = 0 => fail.
- n = 1 => heap becomes empty.
- n = 2 => only one node, take out left end point.
- n > 2 => not as simple.

Remove Min Element Example



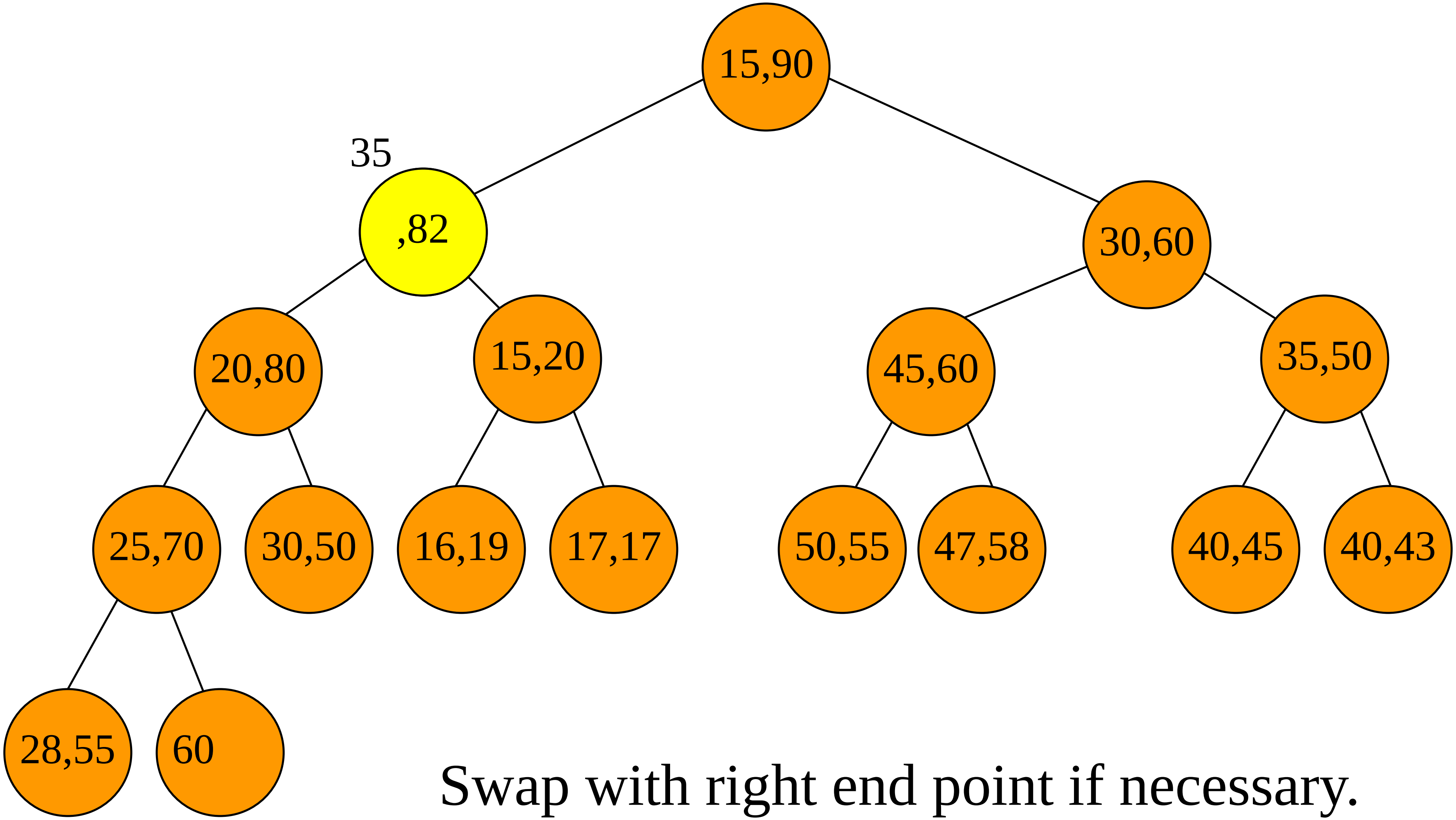
Remove left end point from root.

Remove left end point from last node.

Delete last node if now empty.

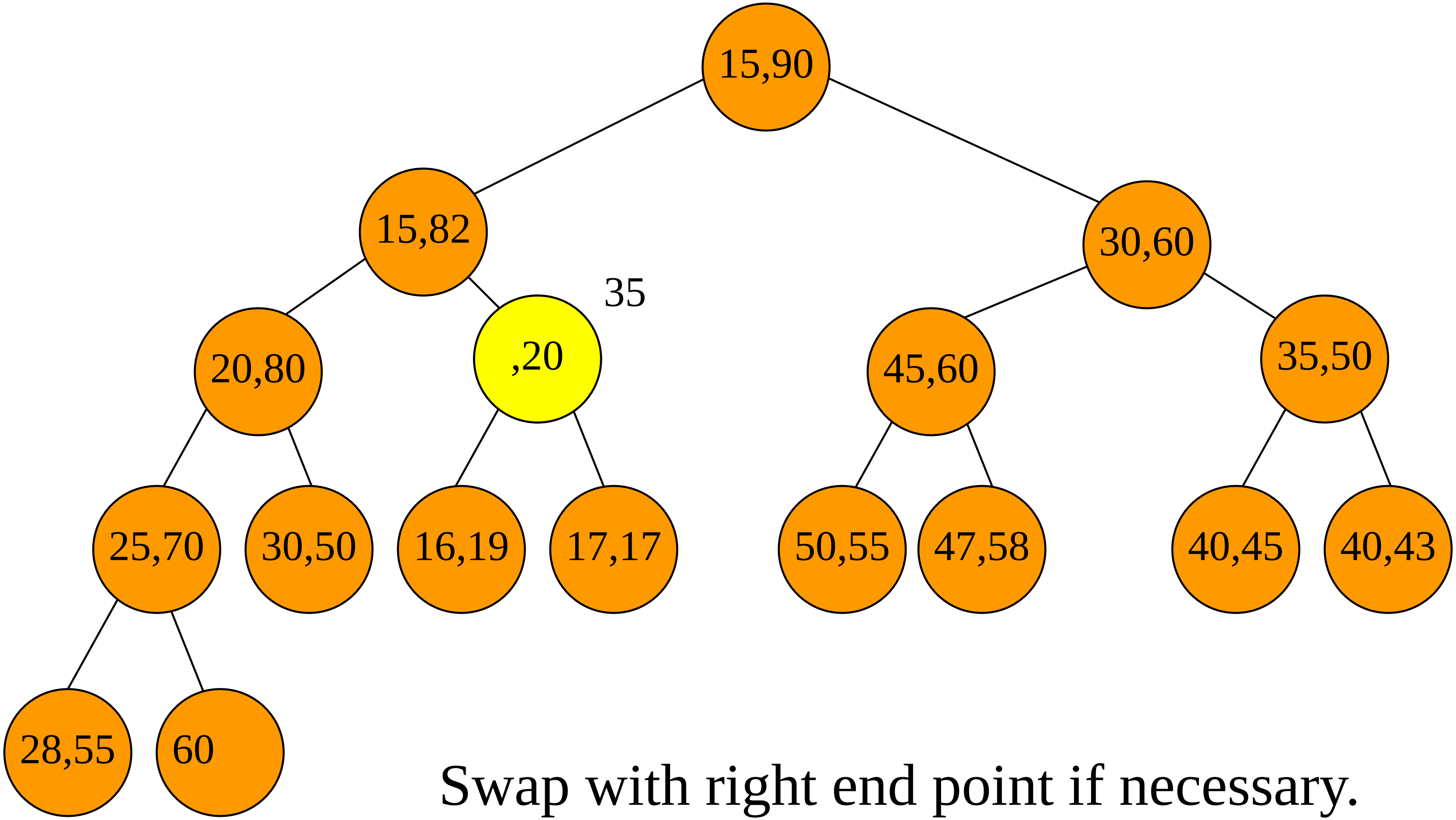
Reinsert into min heap, begin at root.

Remove Min Element Example



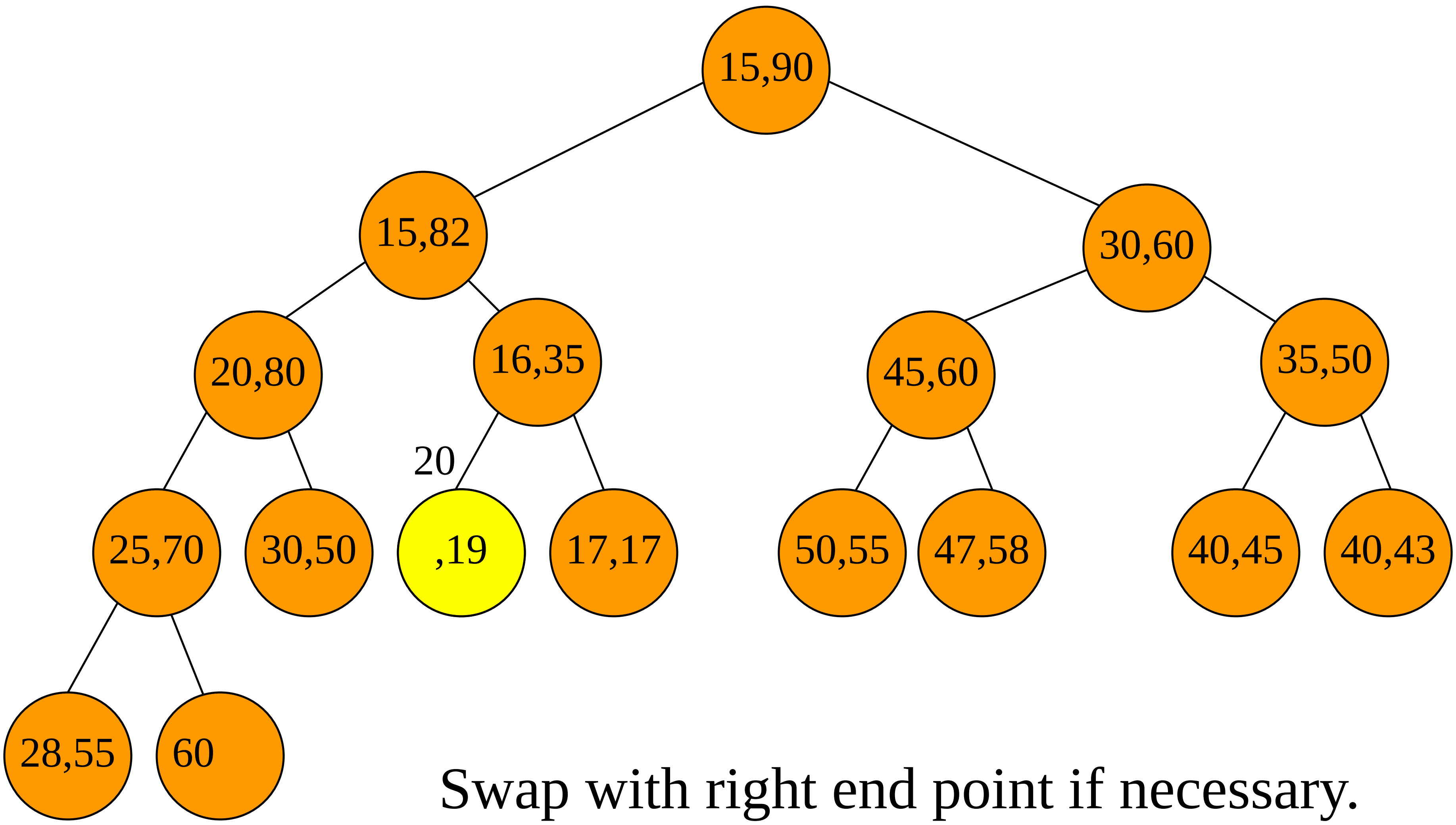
Swap with right end point if necessary.

Remove Min Element Example



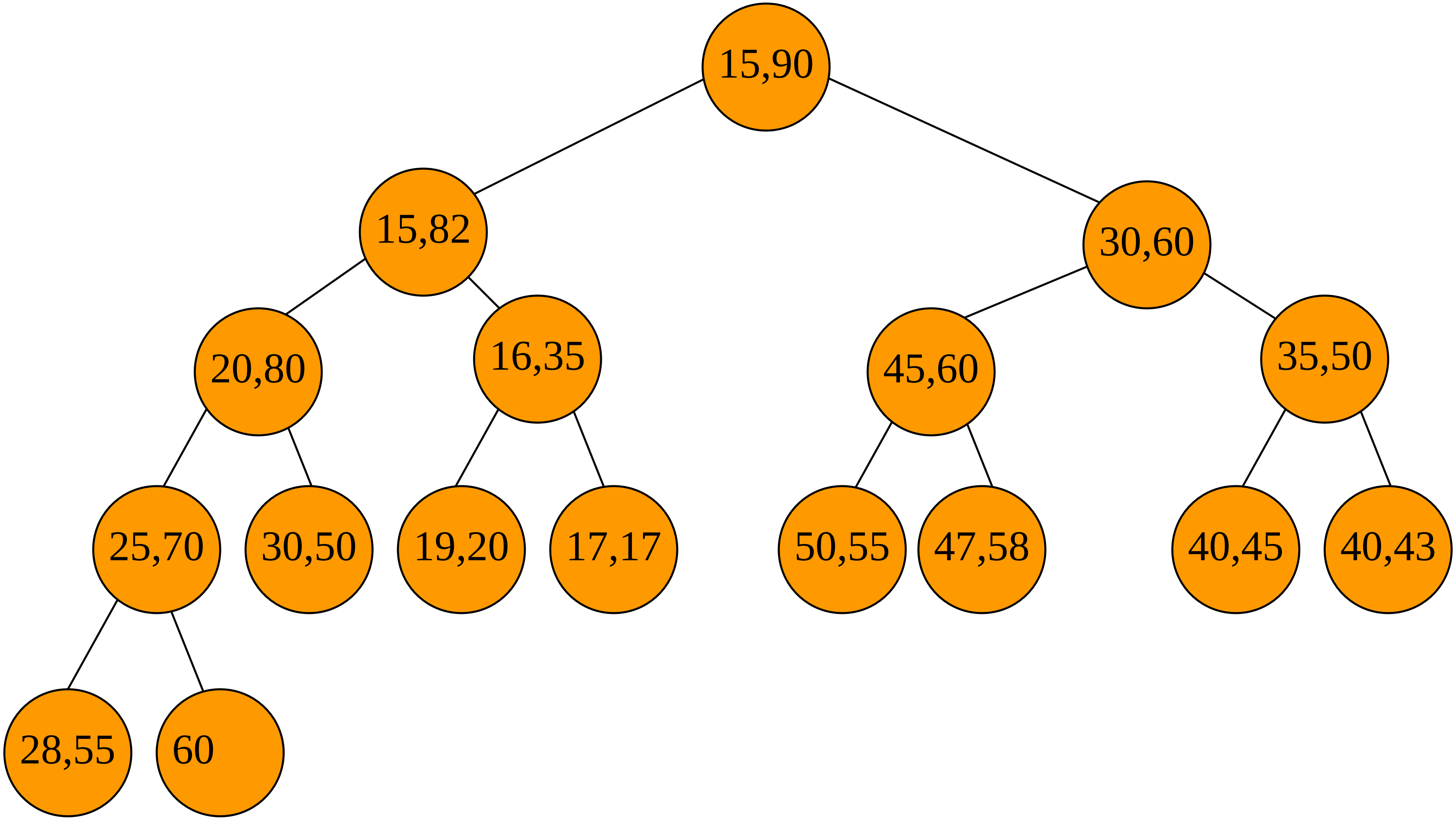
Swap with right end point if necessary.

Remove Min Element Example

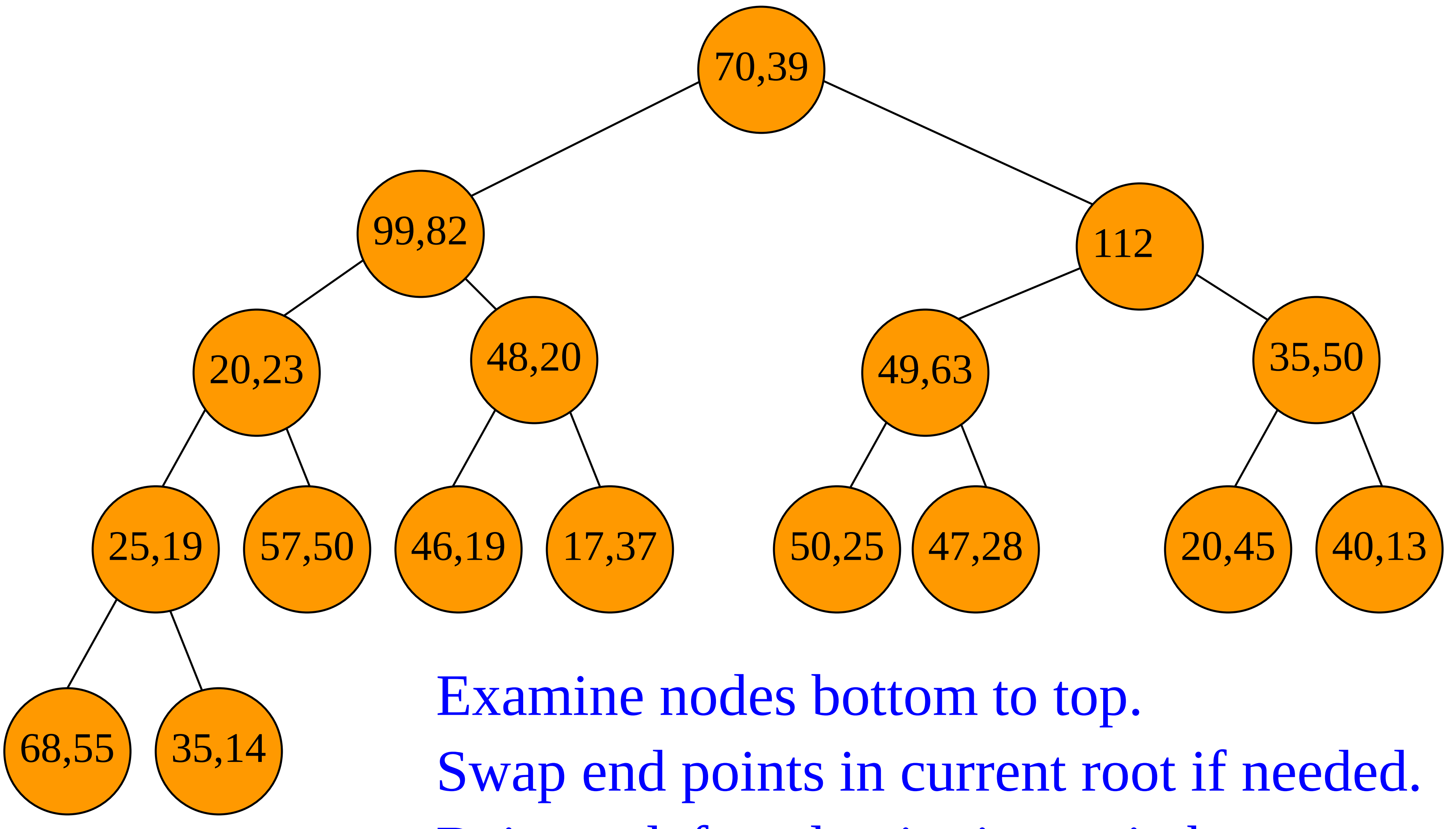


Swap with right end point if necessary.

Remove Min Element Example



Initialize



Examine nodes bottom to top.

Swap end points in current root if needed.

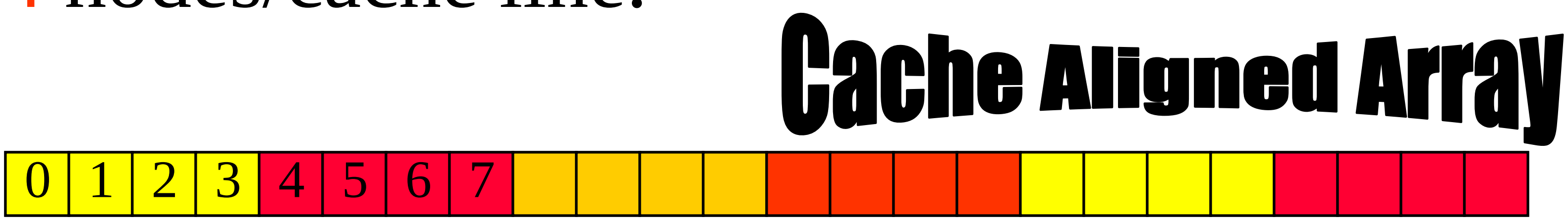
Reinsert left end point into min heap.

Reinsert right end point into max heap.

Cache Optimization

- Heap operations.
  - Uniformly distributed keys.
  - Insert percolates 1.6 levels up the heap on average.
  - Remove min (max) height – 1 levels down the heap.
- Optimize cache utilization for remove min (max).

- L1 cache line is 32 bytes.
- L1 cache is 16KB.
- Heap node size is 8 bytes (1 8-byte element).
- 4 nodes/cache line.



- A remove min (max) has ~h L1 cache misses on average.
  - Root and its children are in the same cache line.
  - $\sim\log_2 n$ cache misses.
  - Only half of each cache line is used (except root’s).

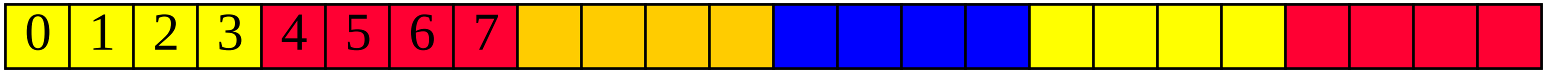
d-ary Heap

- Complete n node tree whose degree is d.
- Min (max) tree.
- Number nodes in breadth-first manner with root being numbered 1.
- Parent(i) = ceil((i – 1)/d).
- Children are $d*(i-1) + 2$, …, $\min\{d*i + 1, n\}$.
- Height is $\log_d n$.
- Height of 4-ary heap is half that of 2-ary heap.

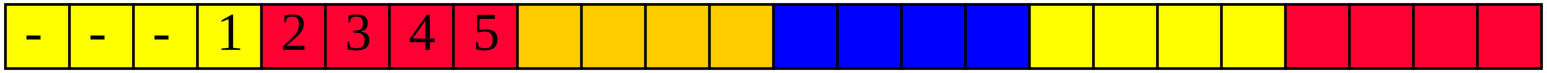
- Worst-case insert moves up half as many levels as when $d = 2$.
  - Average remains at about 1.6 levels.
- Remove-min operations now do 4 compares per level rather than 2 (determine smallest child and see if this child is smaller than element being relocated).
  - But, number of levels is half.
  - Other operations associated with remove min are halved (move small element up, loop iterations, etc.)

4-Heap Cache Utilization

- Standard mapping into cache-aligned array.



- Siblings are in 2 cache lines.
  - ~$\log_2 n$ cache misses for average remove min (max).
- Shift 4-heap by 2 slots.



- Siblings are in same cache line.
  - ~$\log_4 n$ cache misses for average remove min (max).

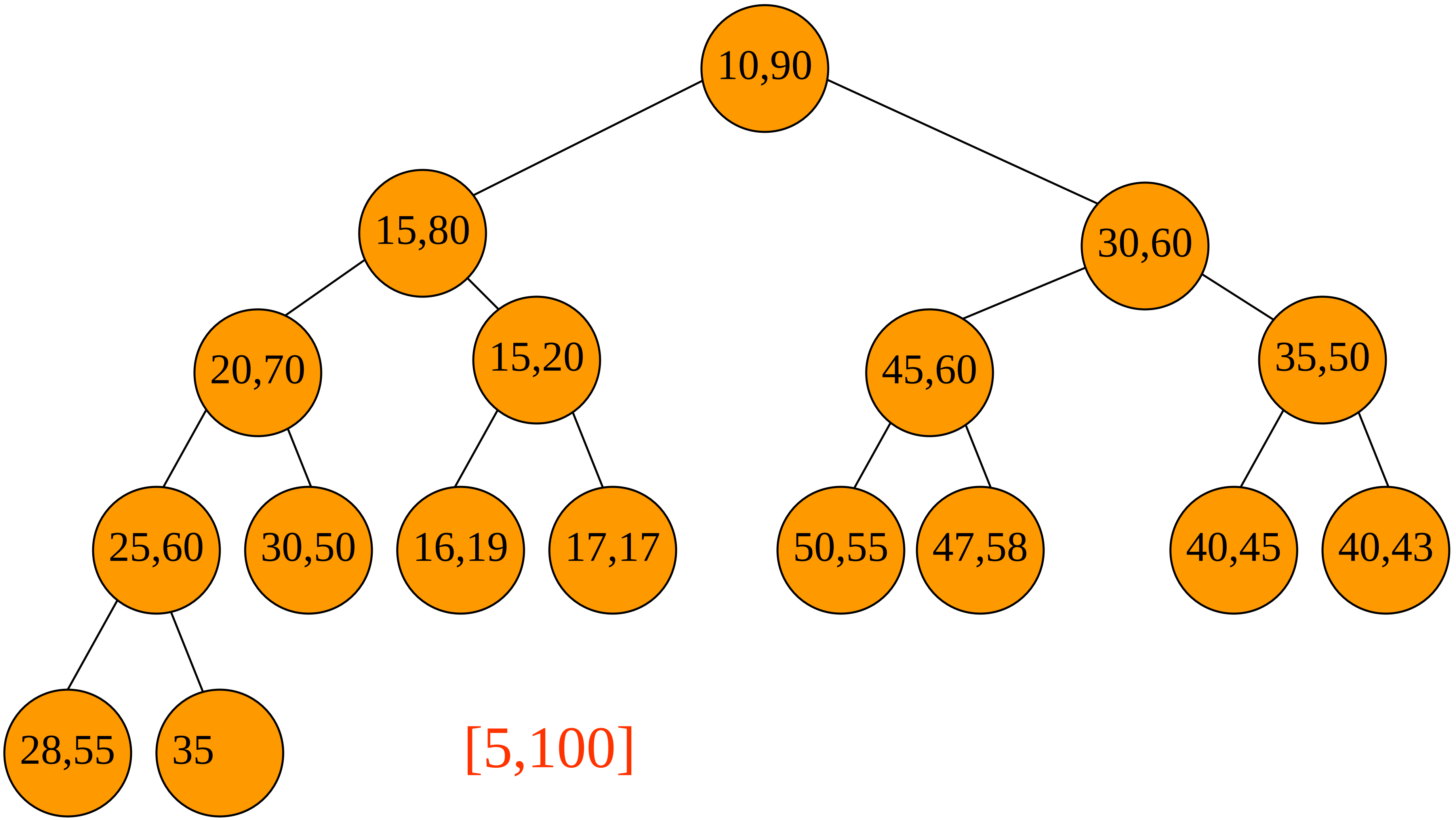
d-ary Heap Performance

- Speedup of about 1.5 to 1.8 when sorting 1 million elements using heapsort and cache-aligned 4-heap vs. 2-heap that begins at array position 0.
- Cache-aligned 4-heap generally performs as well as, or better, than other d-heaps.
- Use degree 4 complete tree for interval heaps instead of degree 2.

Application Of Interval Heaps

- Complementary range search problem.
  - Collection of 1D points (numbers).
  - Insert a point.
    - O(log n)
  - Remove a point given its location in the structure.
    - O(log n)
  - Report all points not in the range [a,b], a <= b.
    - O(k), where k is the number of points not in the range.

Example



[5,100]

[2,65]

Section 13.2

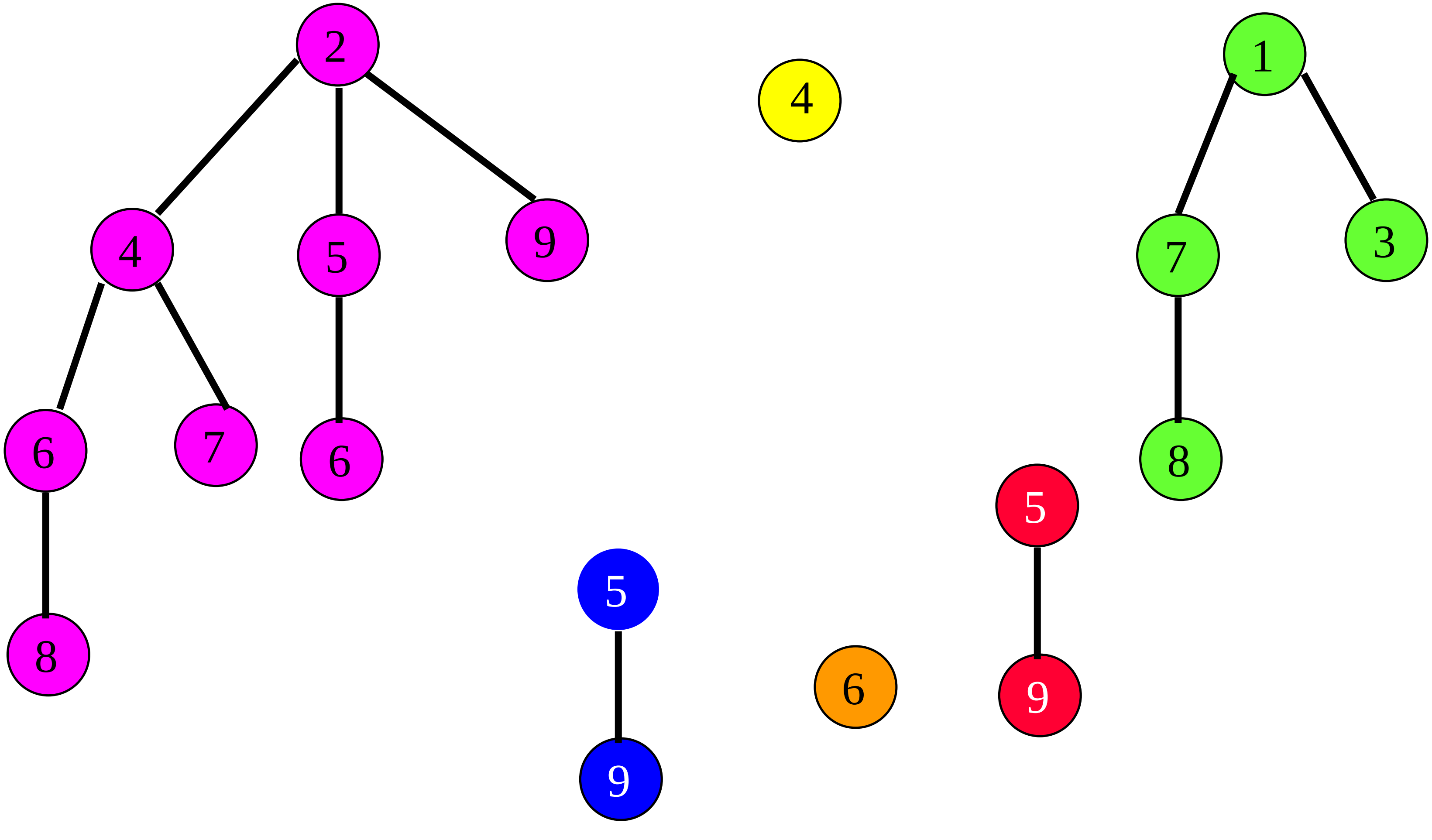
Binomial Heap

Binomial Heaps

	Leftist trees	Binomial heaps	
		Actual	Amortized
Insert	O(log n)	O(1)	O(1)
Delete min (or max)	O(log n)	O(n)	O(log n)
Meld	O(log n)	O(1)	O(1)

Min Binomial Heap

- Collection of min trees.

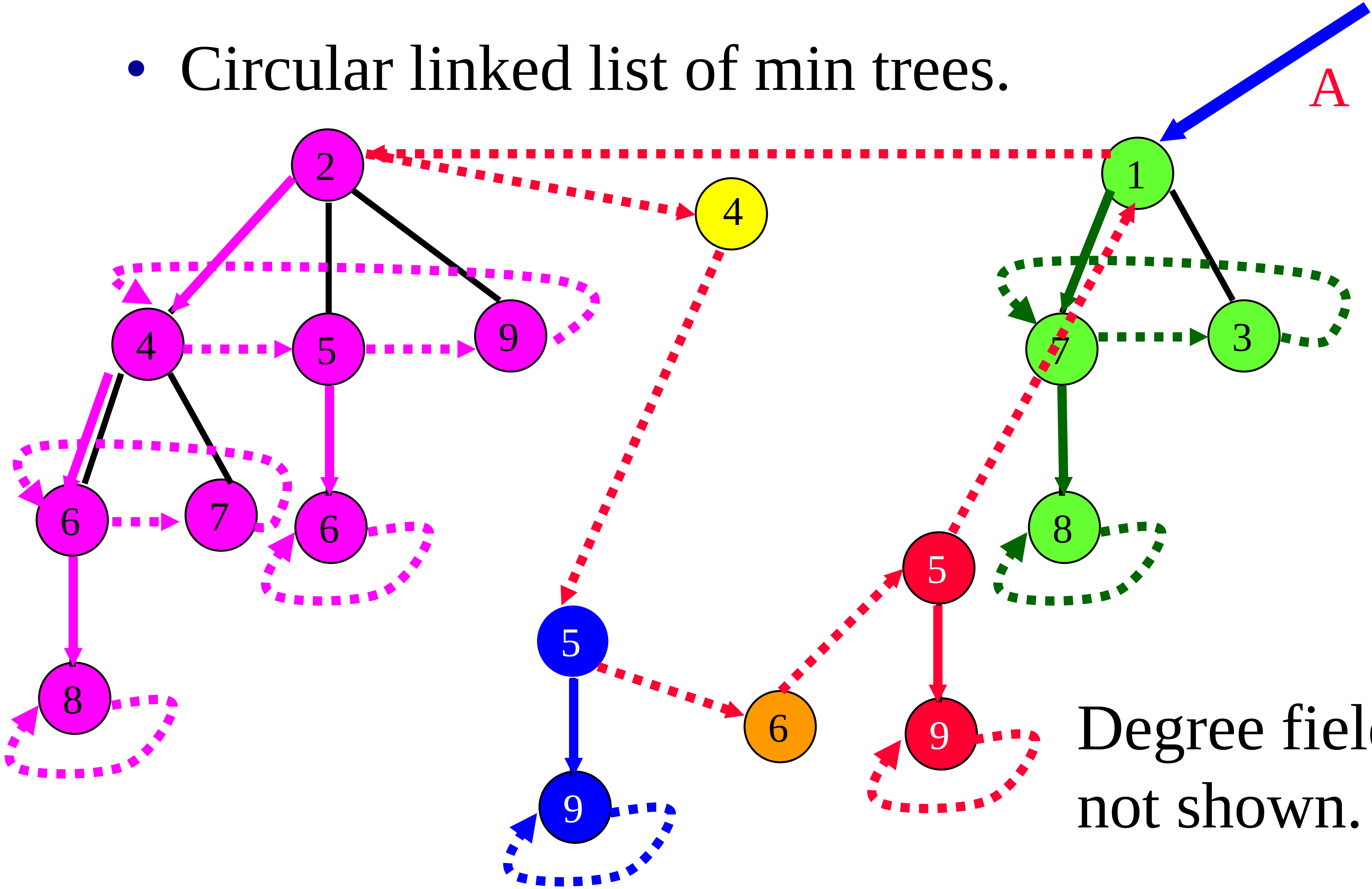


Node Structure

- Degree
  - Number of children.
- Child
  - Pointer to one of the node's children.
  - Null iff node has no child.
- Sibling
  - Used for circular linked list of siblings.
- Data

Binomial Heap Representation

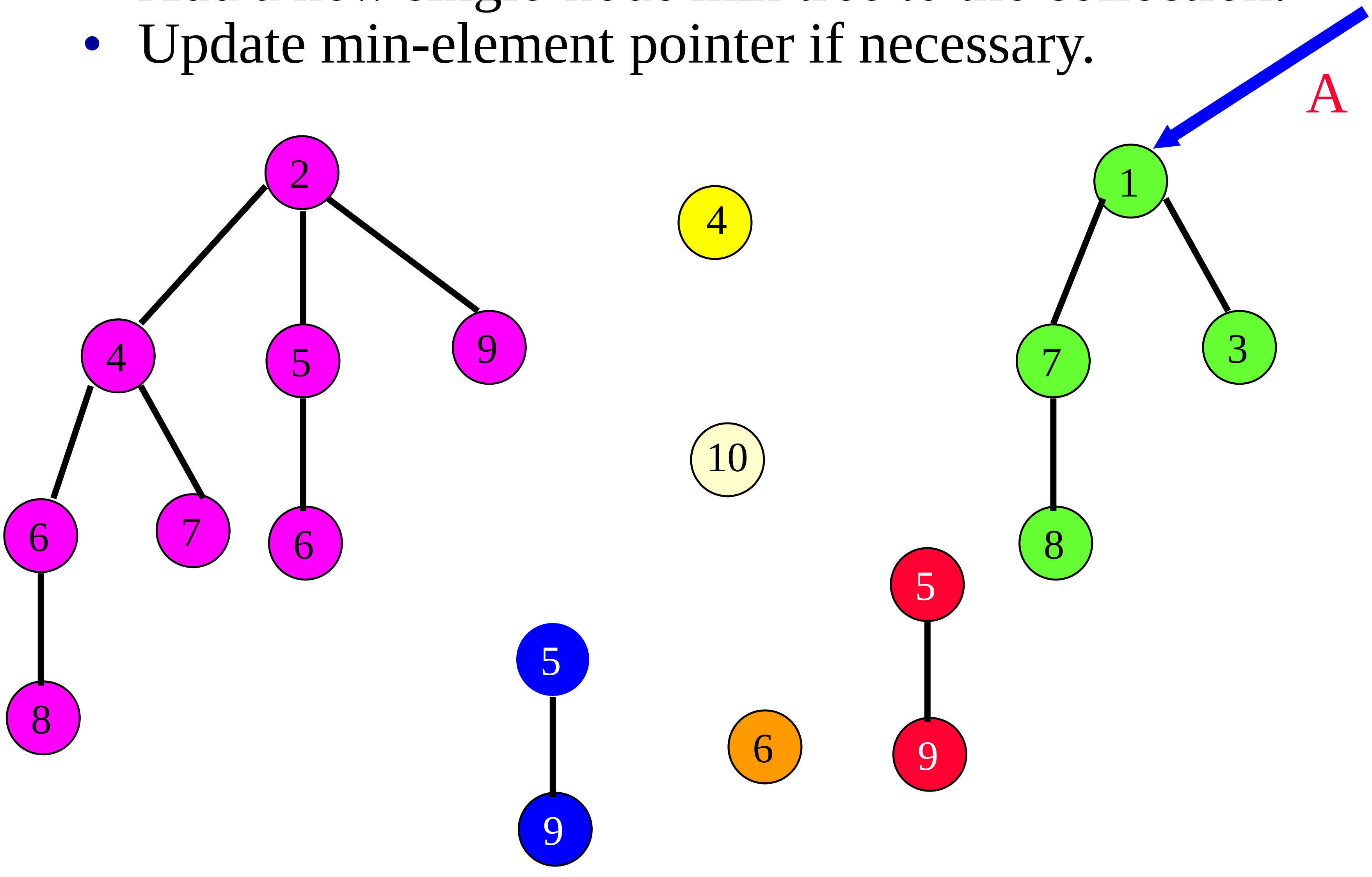
- Circular linked list of min trees.



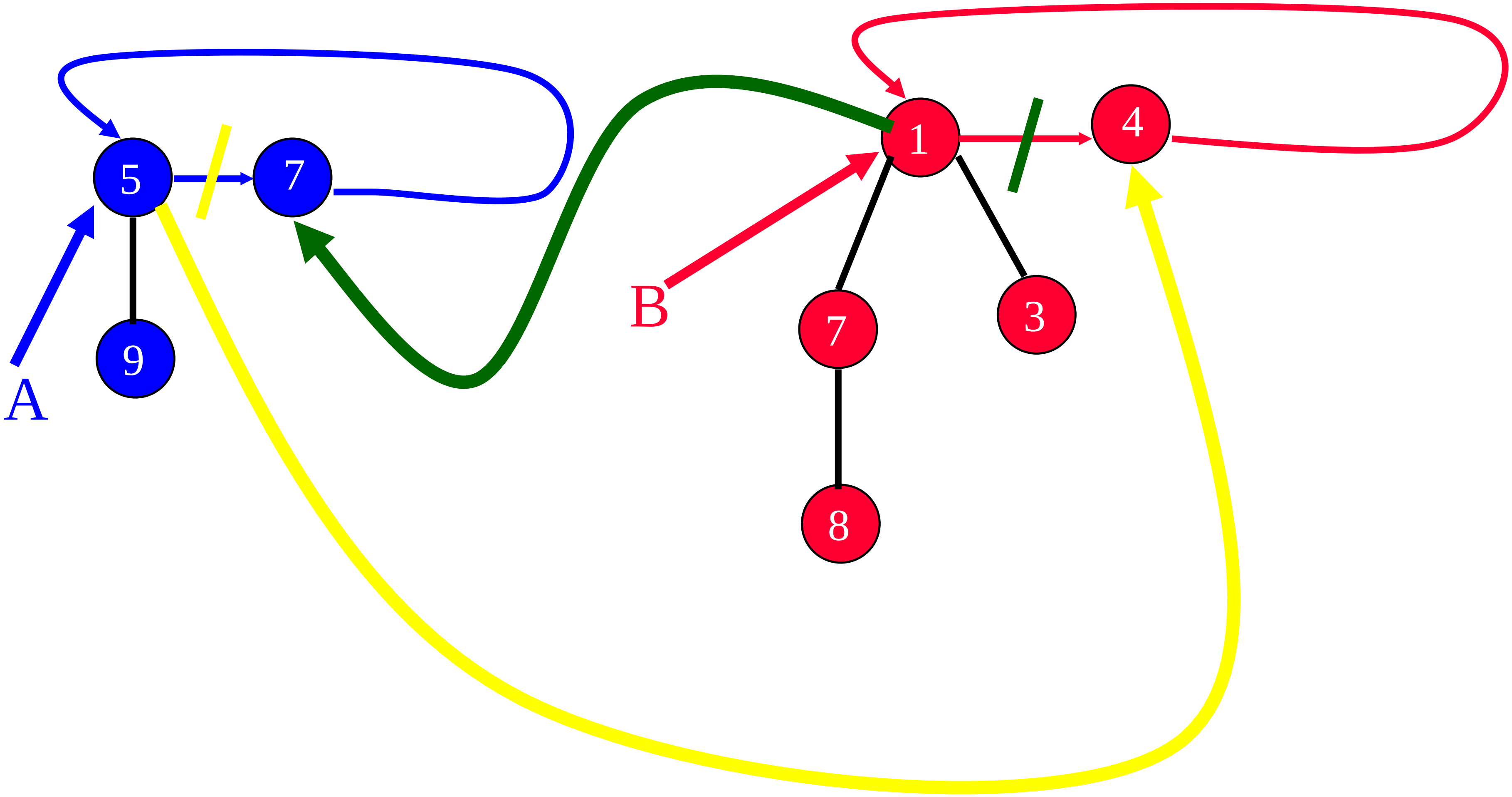
Degree fields not shown.

Insert 10

- Add a new single-node min tree to the collection.
- Update min-element pointer if necessary.

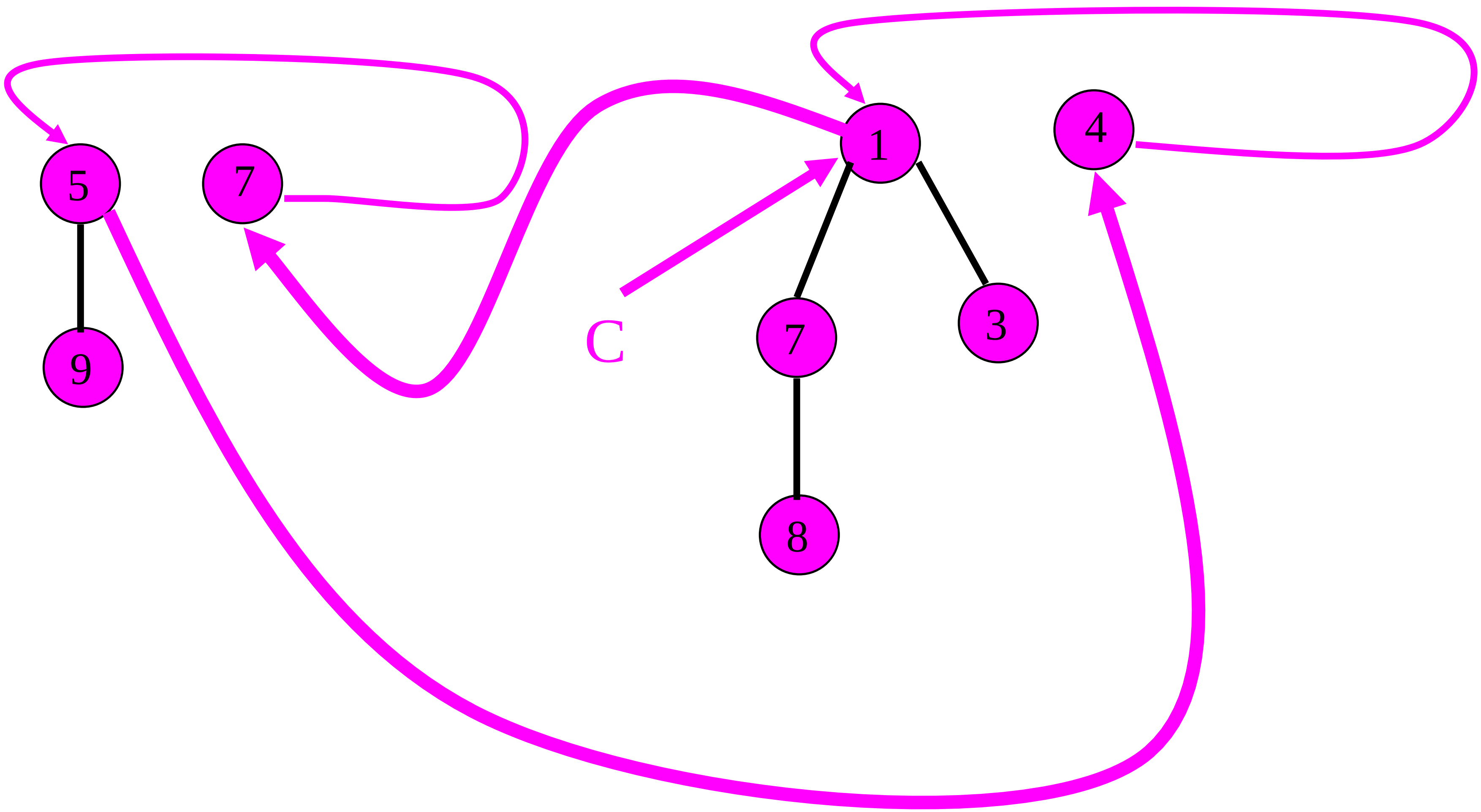


Meld



- Combine the 2 top-level circular lists.
- Set min-element pointer.

Meld

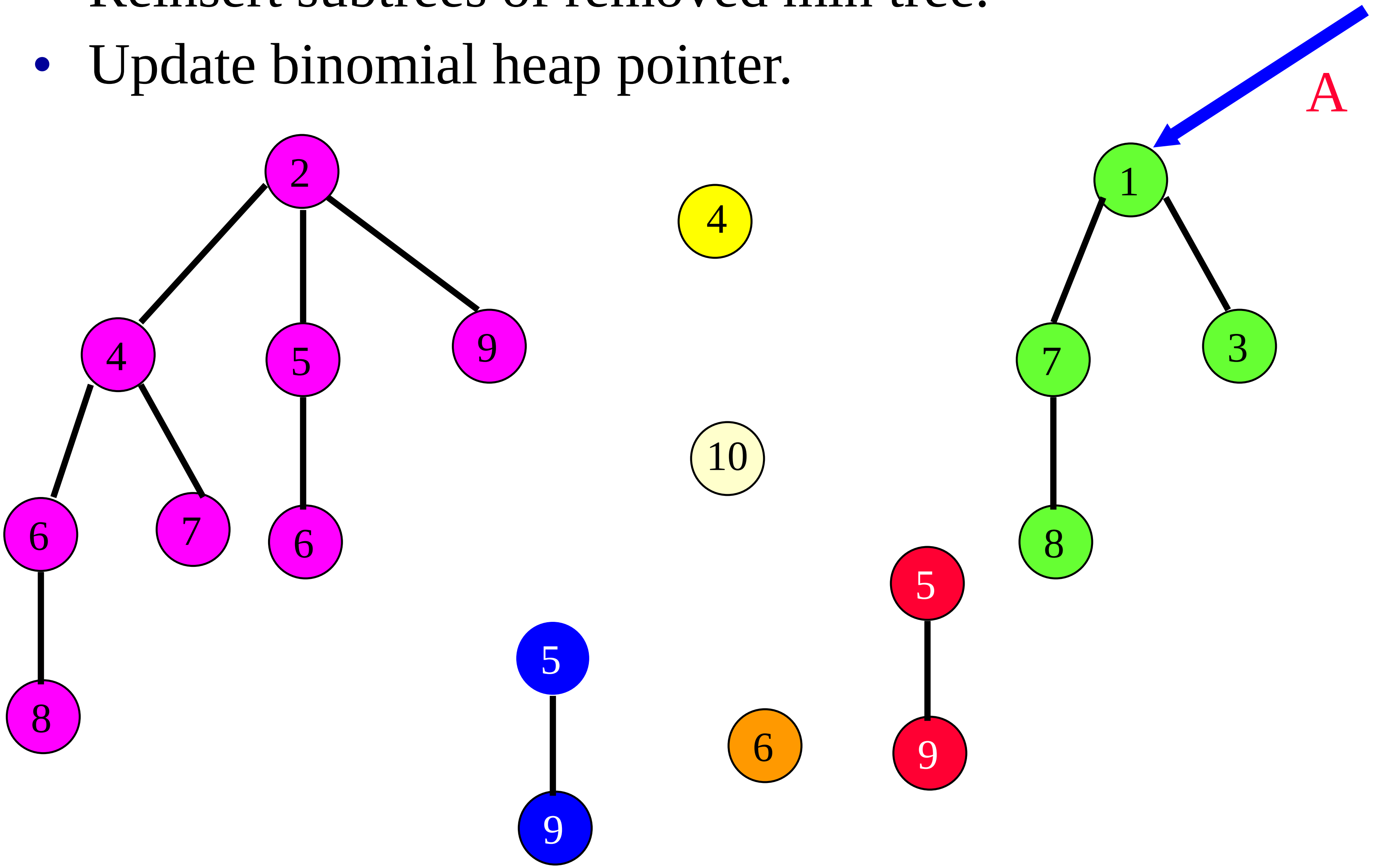


Delete Min

- Empty binomial heap => fail.

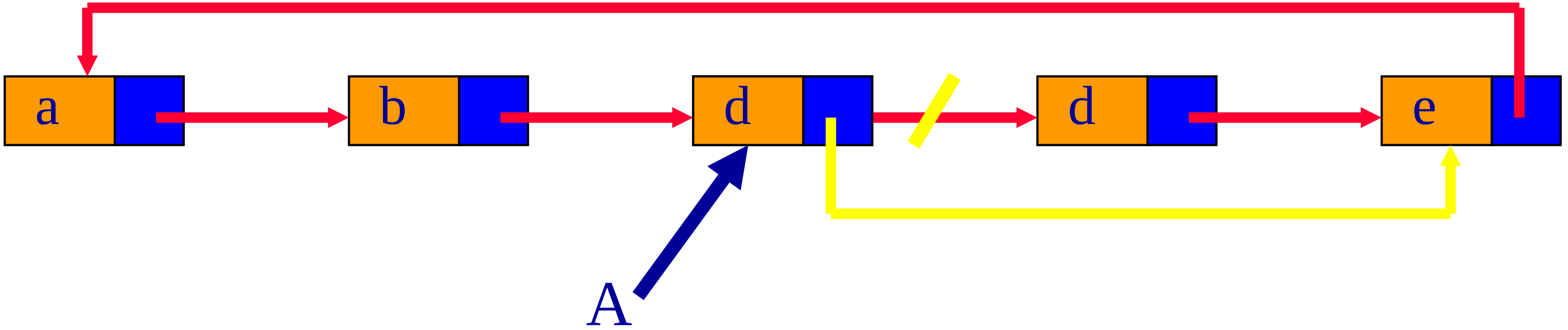
Nonempty Binomial Heap

- Remove a min tree.
- Reinsert subtrees of removed min tree.
- Update binomial heap pointer.



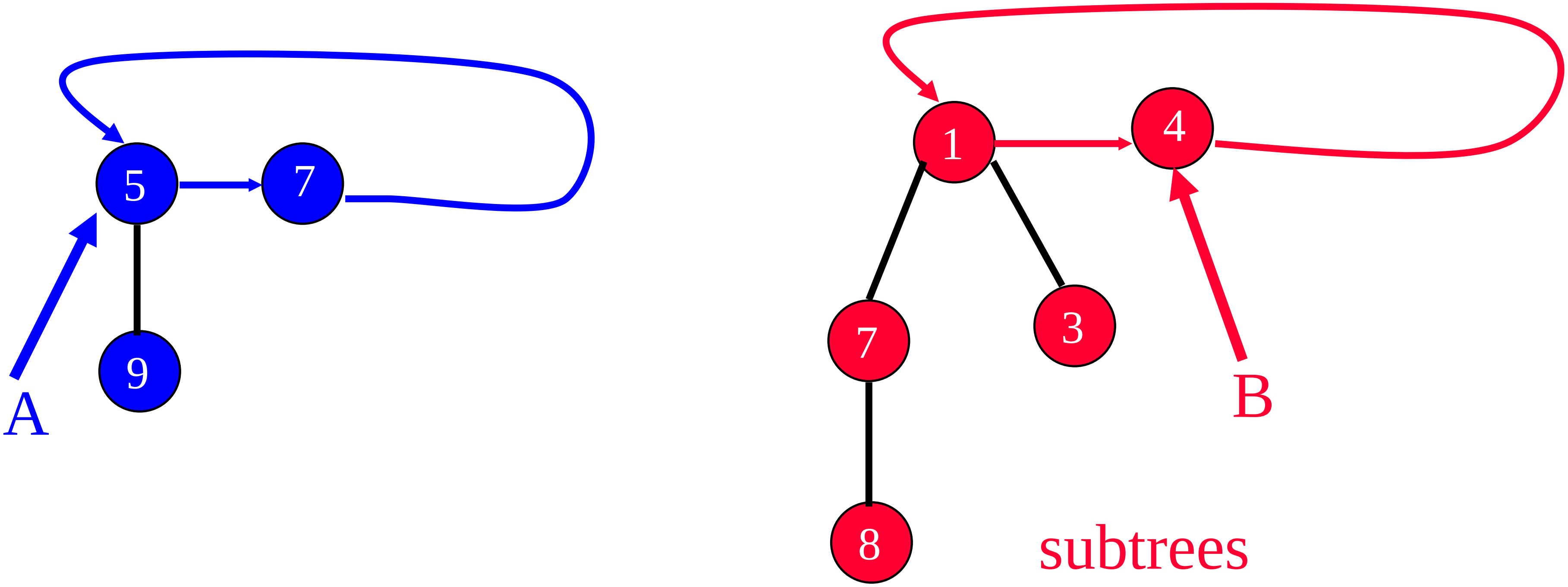
Remove Min Tree

- Same as remove a node from a circular list.



- No next node => empty after remove.
- Otherwise, copy next-node data and remove next node.

Reinsert Subtrees



- Combine the 2 top-level circular lists.
  - Same as in meld operation.

Update Binomial Heap Pointer

- Must examine roots of all min trees to determine the min value.

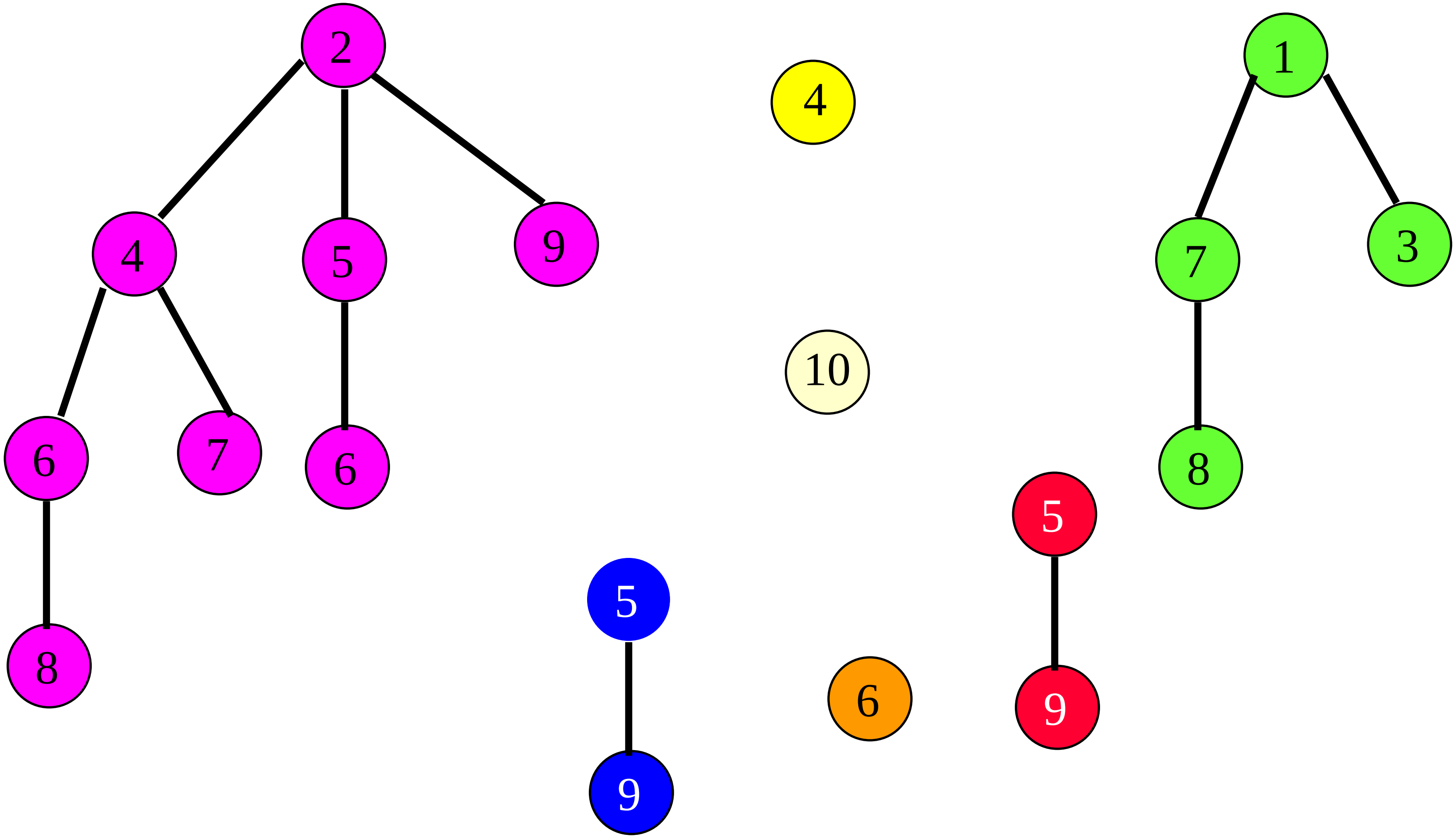
Complexity Of Delete Min

- Remove a min tree.
  - O(1).
- Reinsert subtrees.
  - O(1).
- Update binomial heap pointer.
  - O(s), where s is the number of min trees in final top-level circular list.
  - s = O(n).
- Overall complexity of remove min is O(n).

Enhanced Delete Min

- During reinsert of subtrees, pairwise combine min trees whose roots have equal degree.

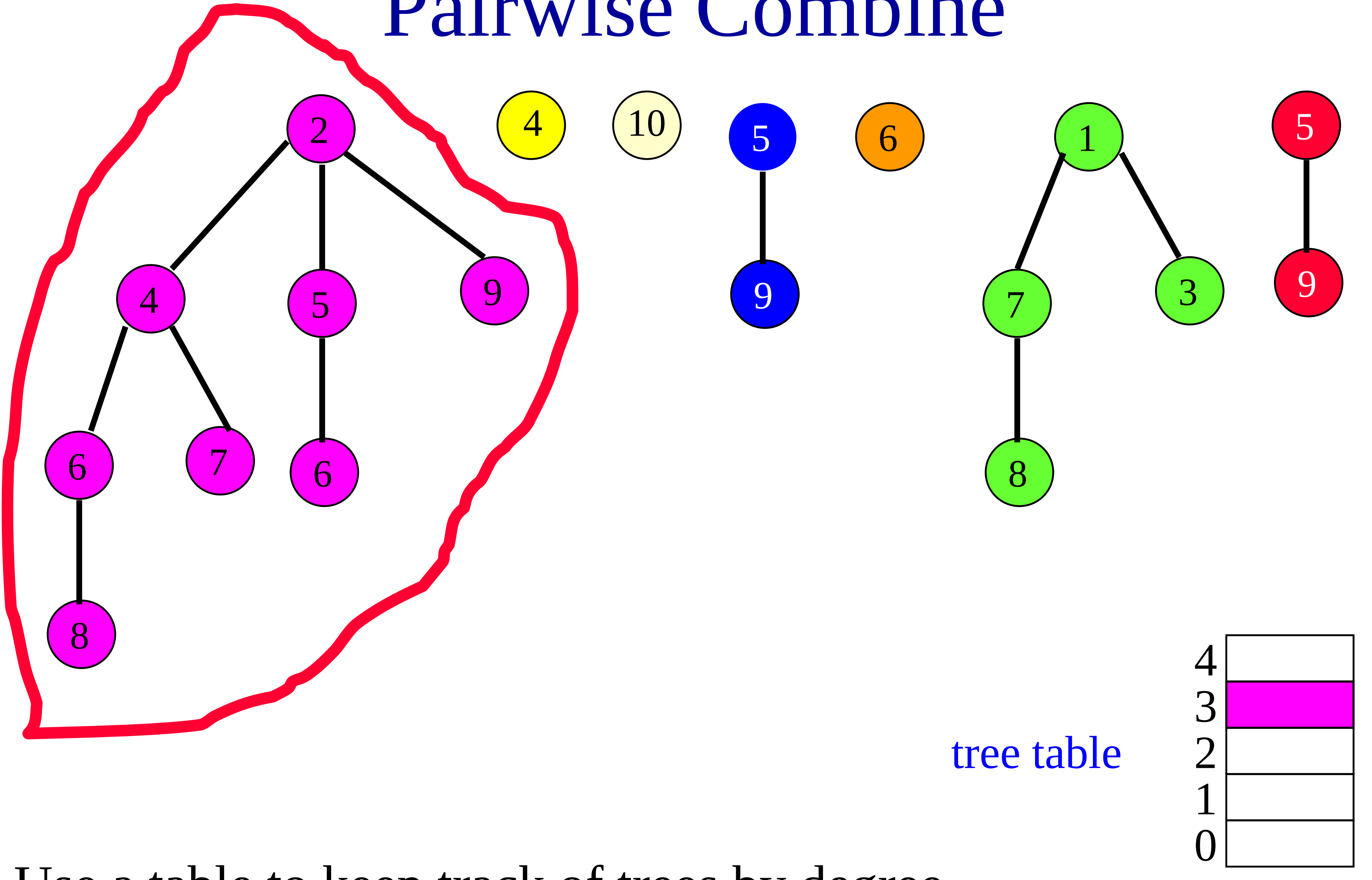
Pairwise Combine



Examine the s = 7 trees in some order.

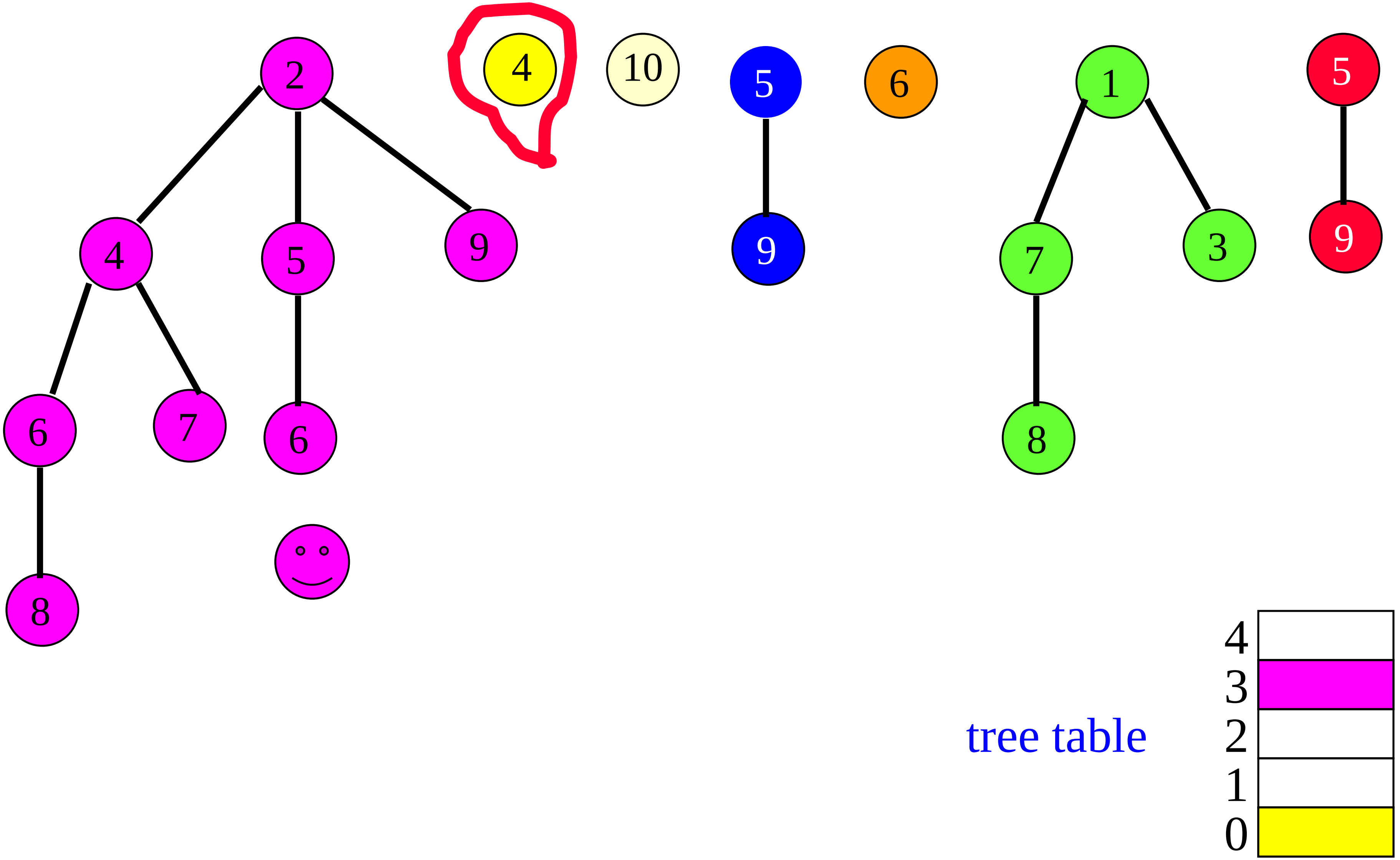
Determined by the 2 top-level circular lists.

Pairwise Combine

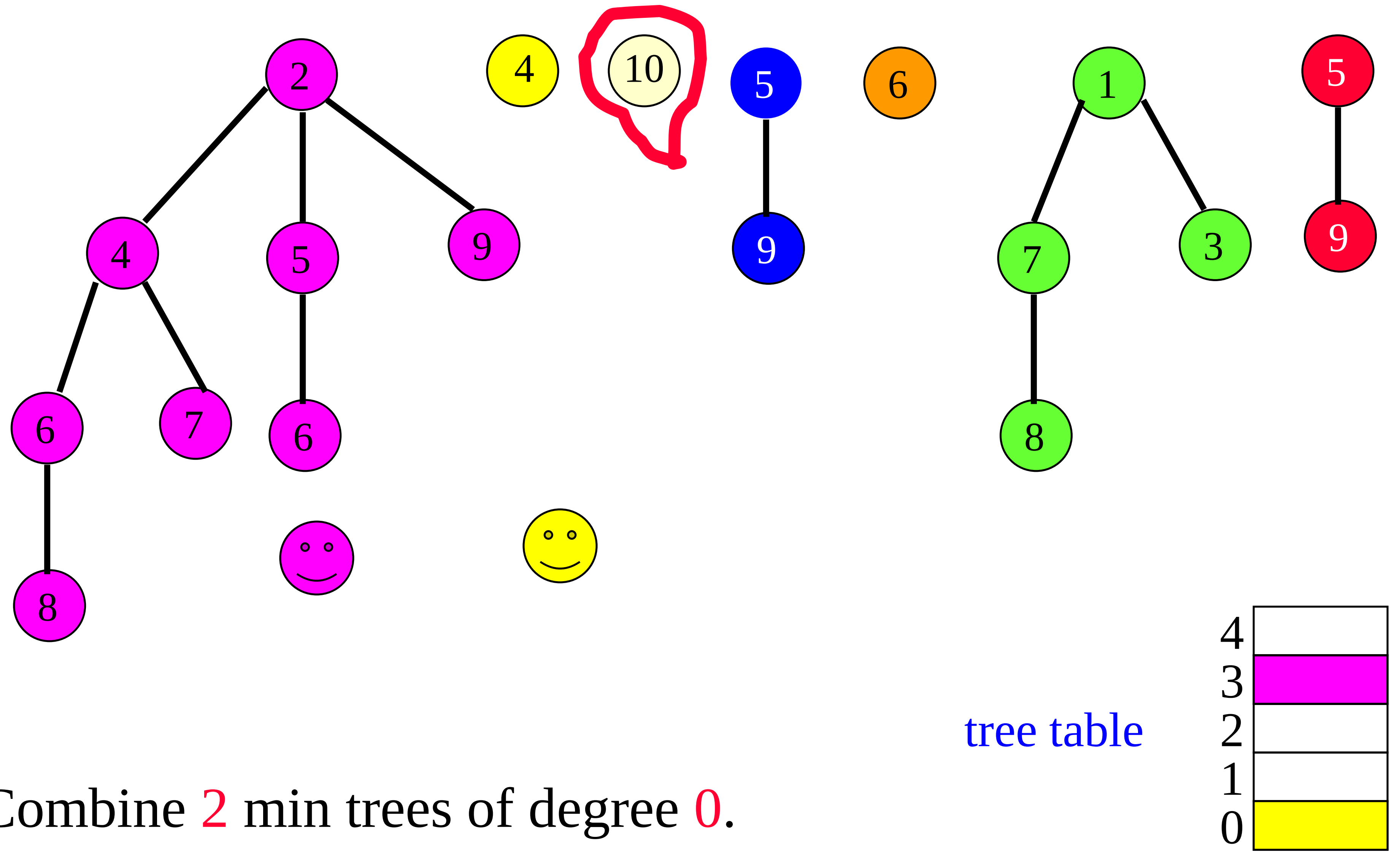


Use a table to keep track of trees by degree.

Pairwise Combine



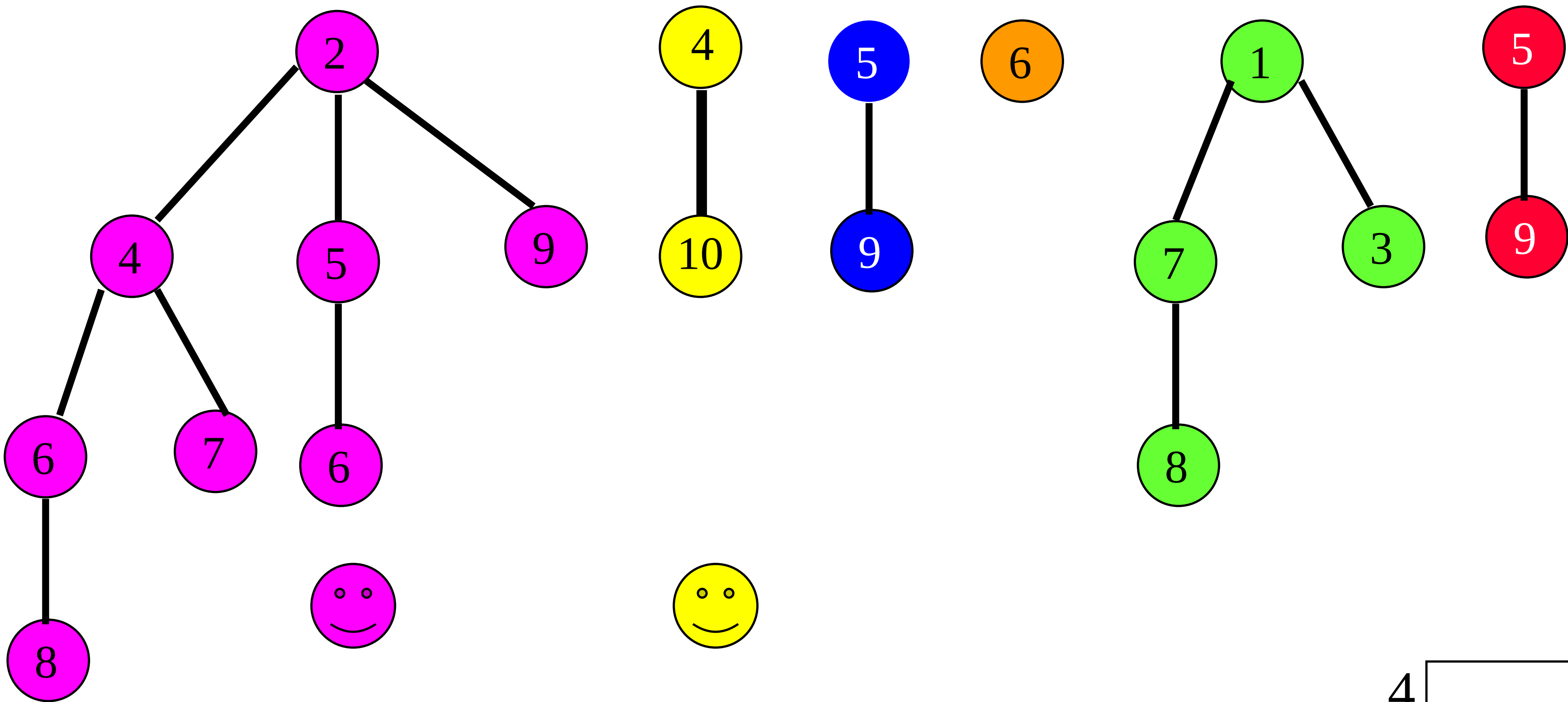
Pairwise Combine

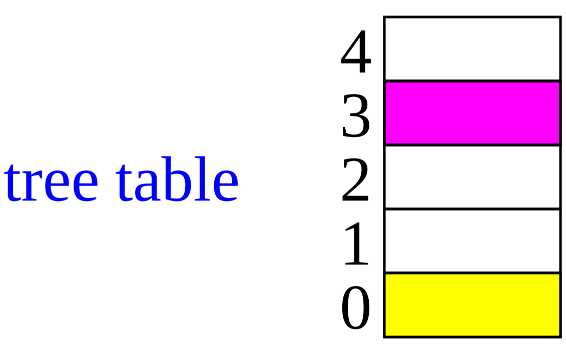


Combine 2 min trees of degree 0.

Make the one with larger root a subtree of other.

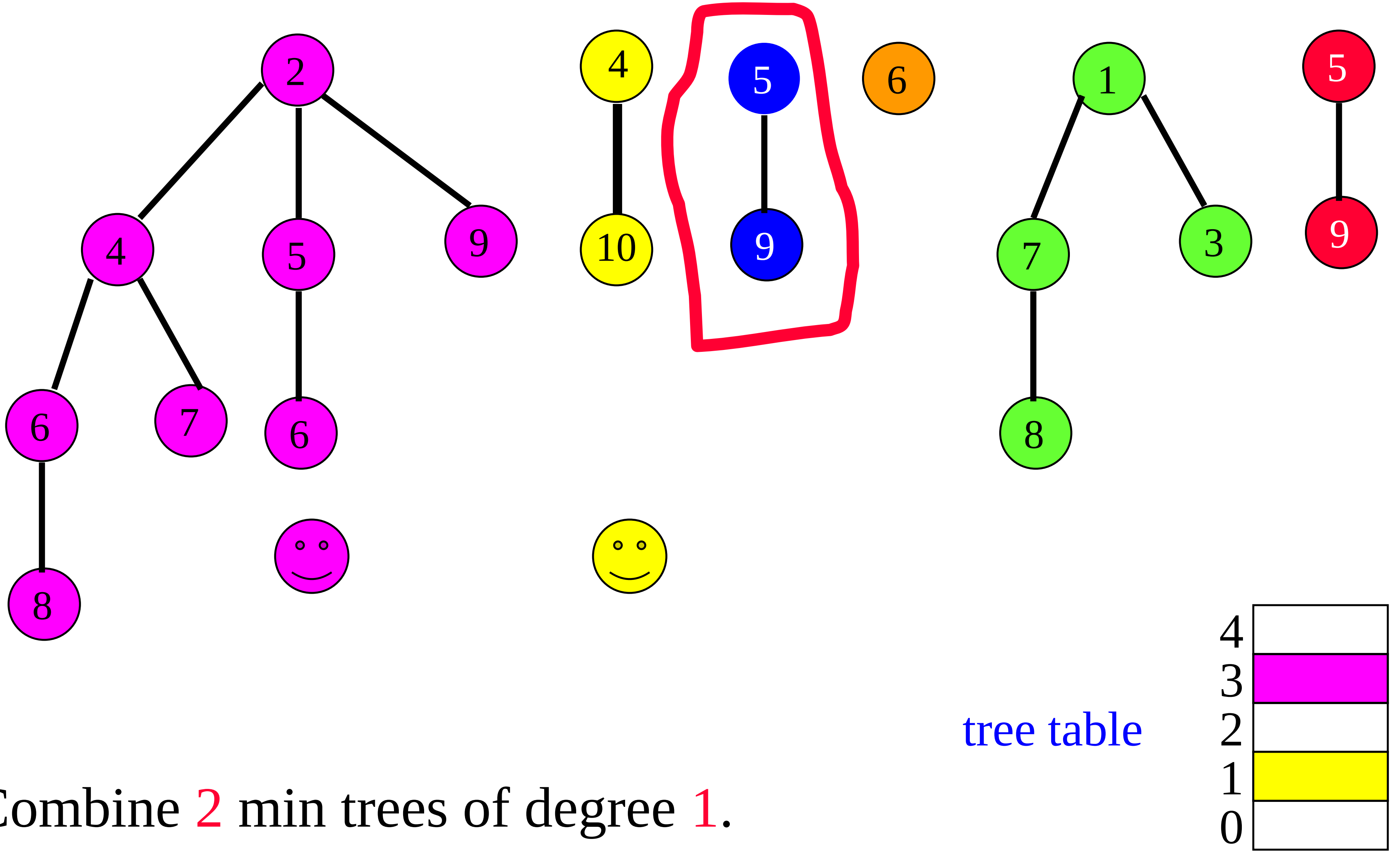
Pairwise Combine





Update tree table.

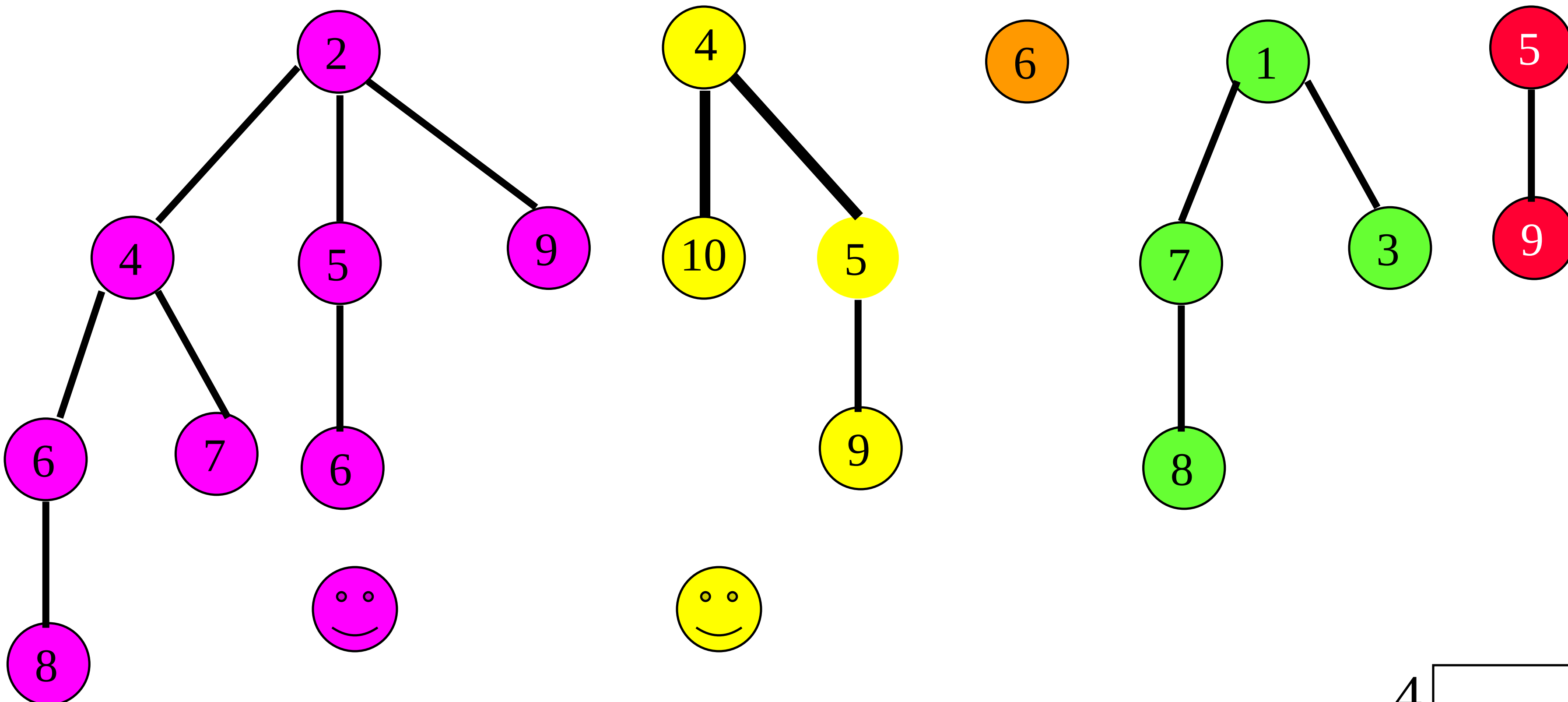
Pairwise Combine

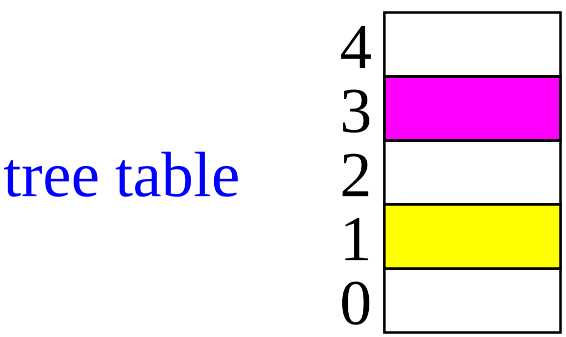


Combine 2 min trees of degree 1.

Make the one with larger root a subtree of other.

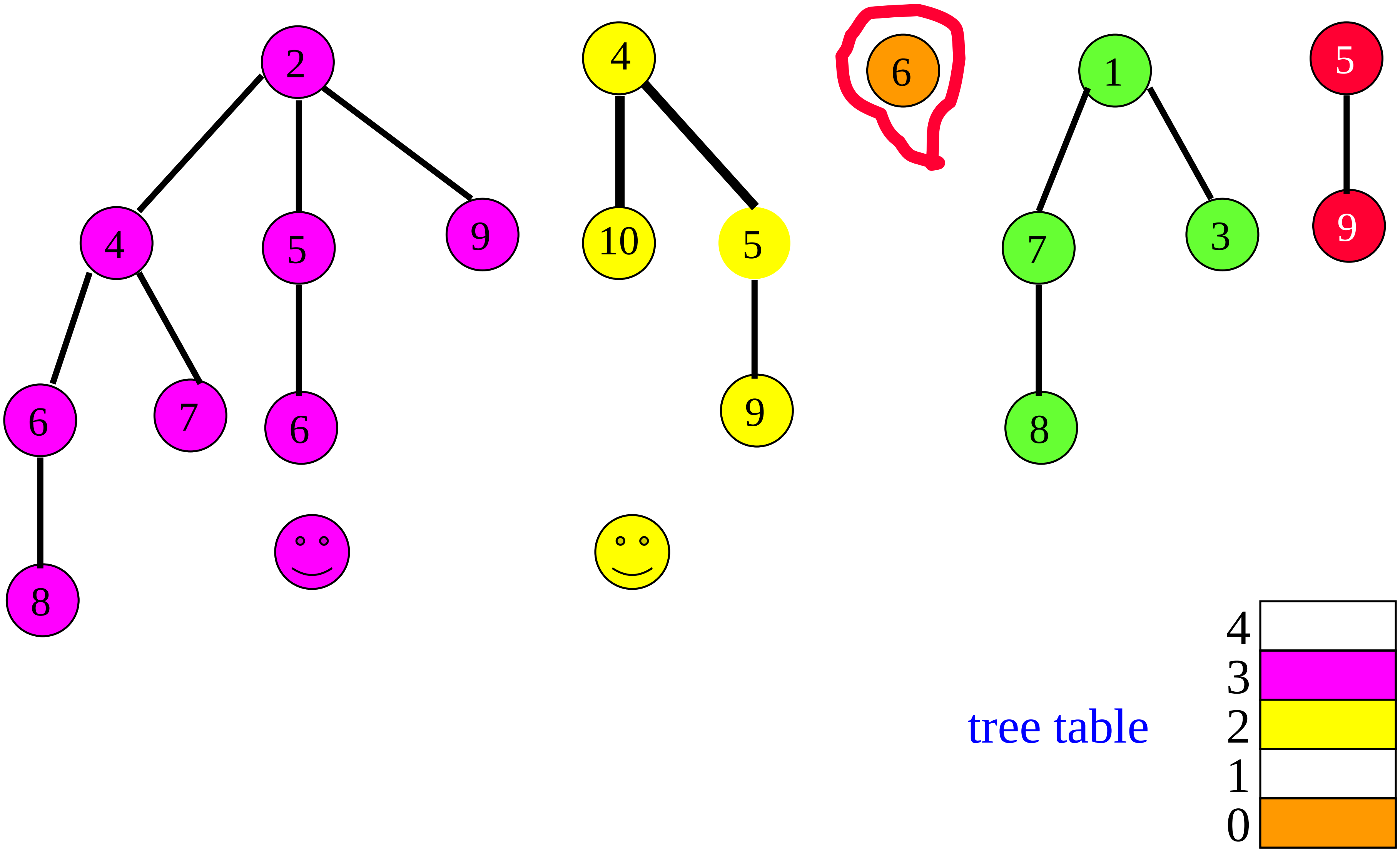
Pairwise Combine



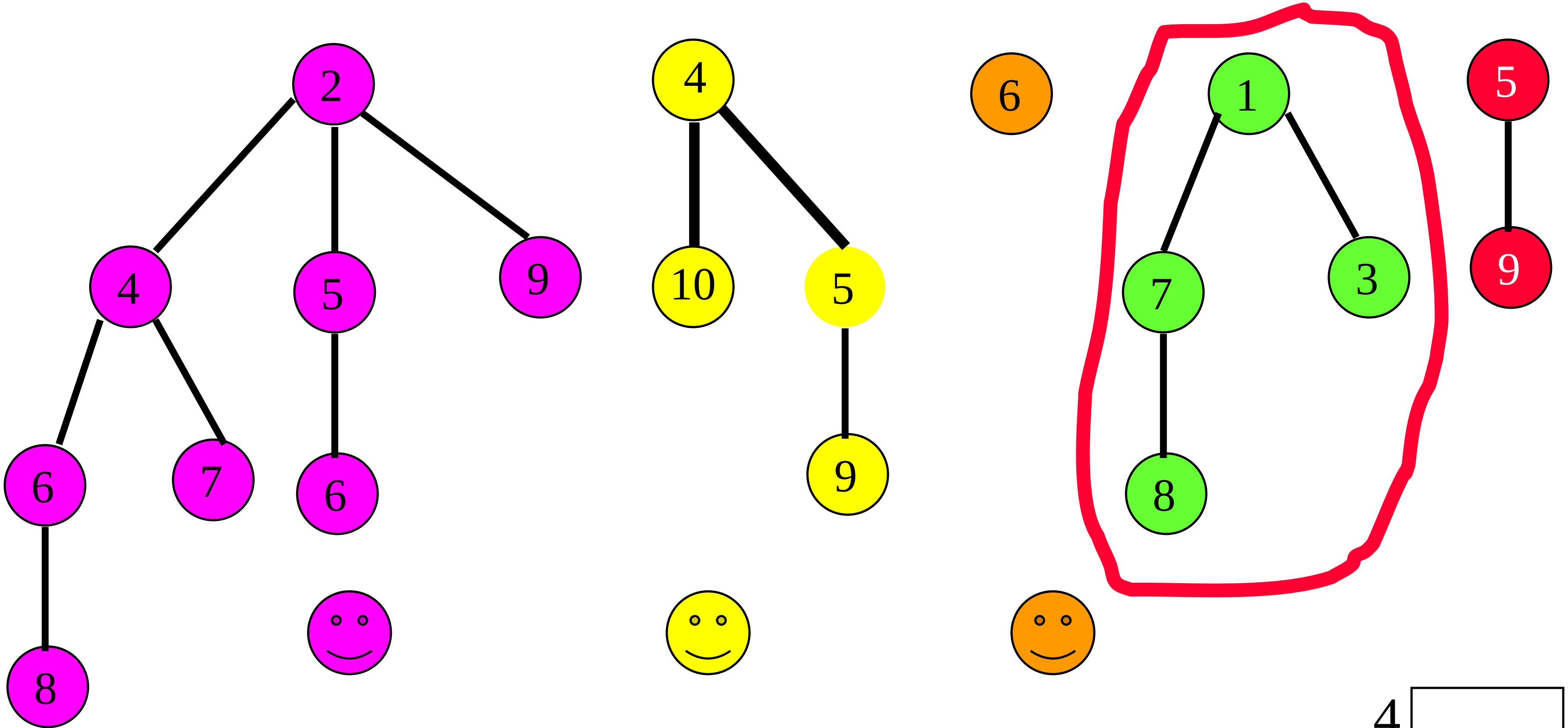


Update tree table.

Pairwise Combine



Pairwise Combine



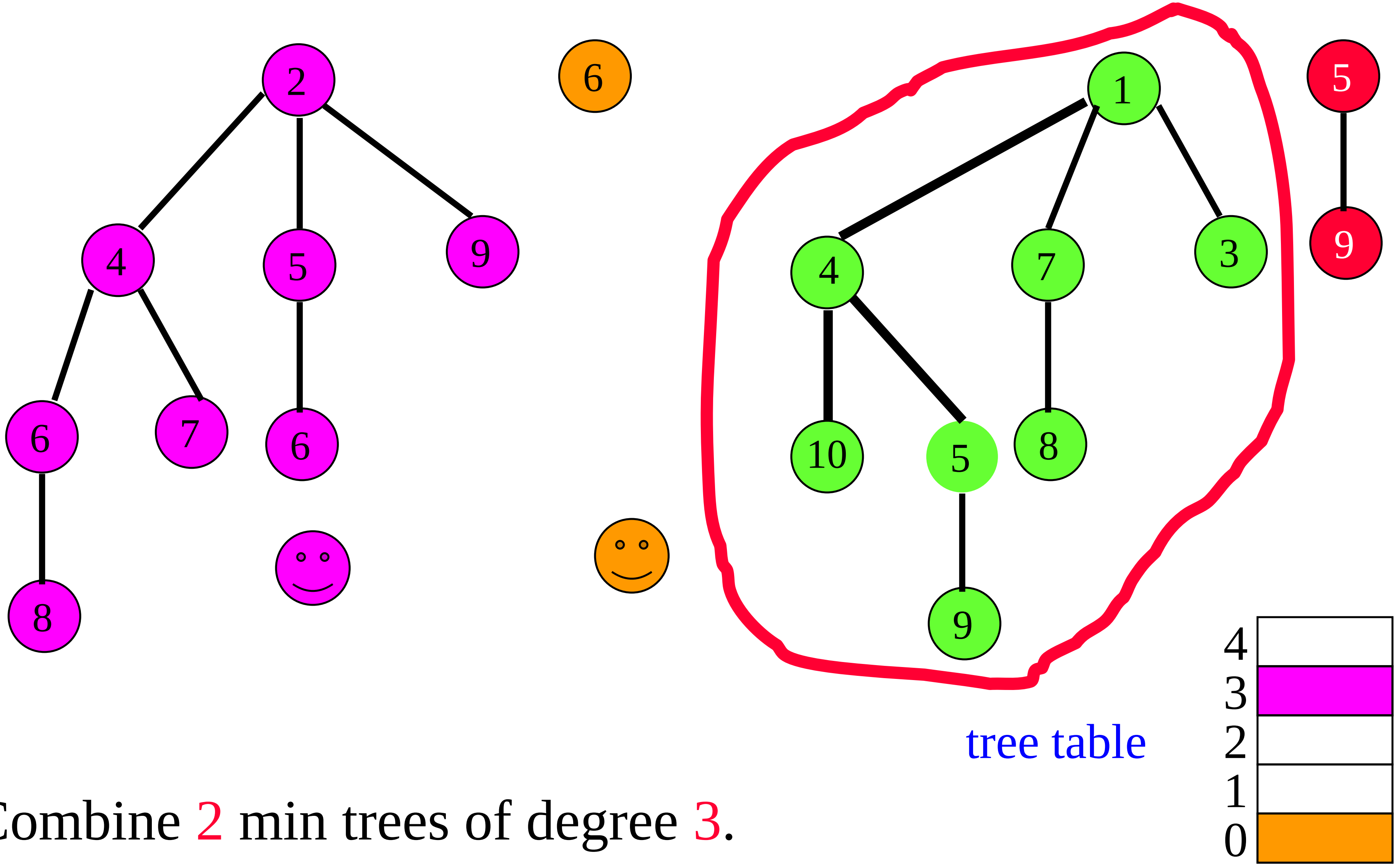
tree table

4	
3	
2	
1	
0	

Combine 2 min trees of degree 2.

Make the one with larger root a subtree of other.

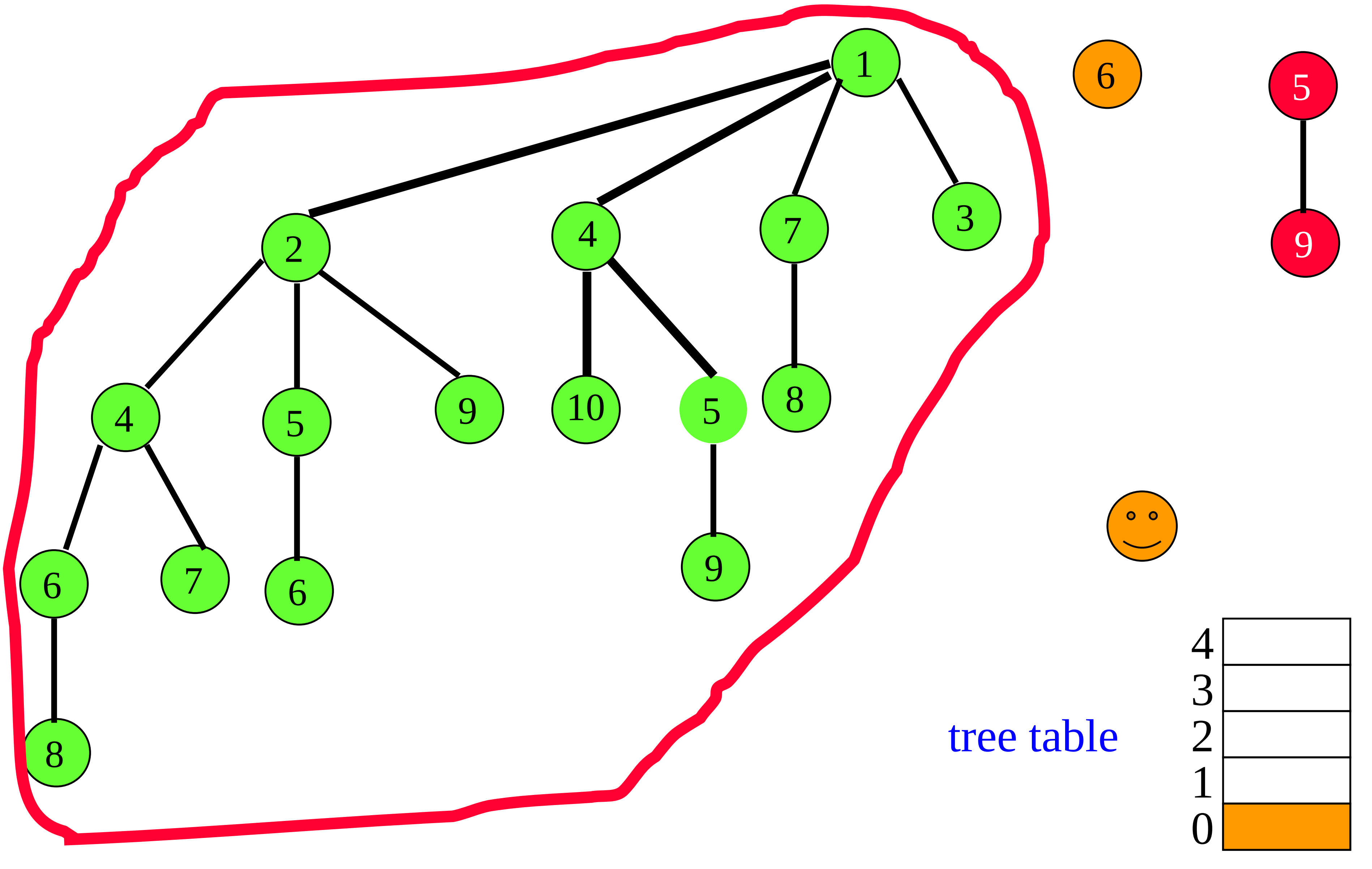
Pairwise Combine



Combine 2 min trees of degree 3.

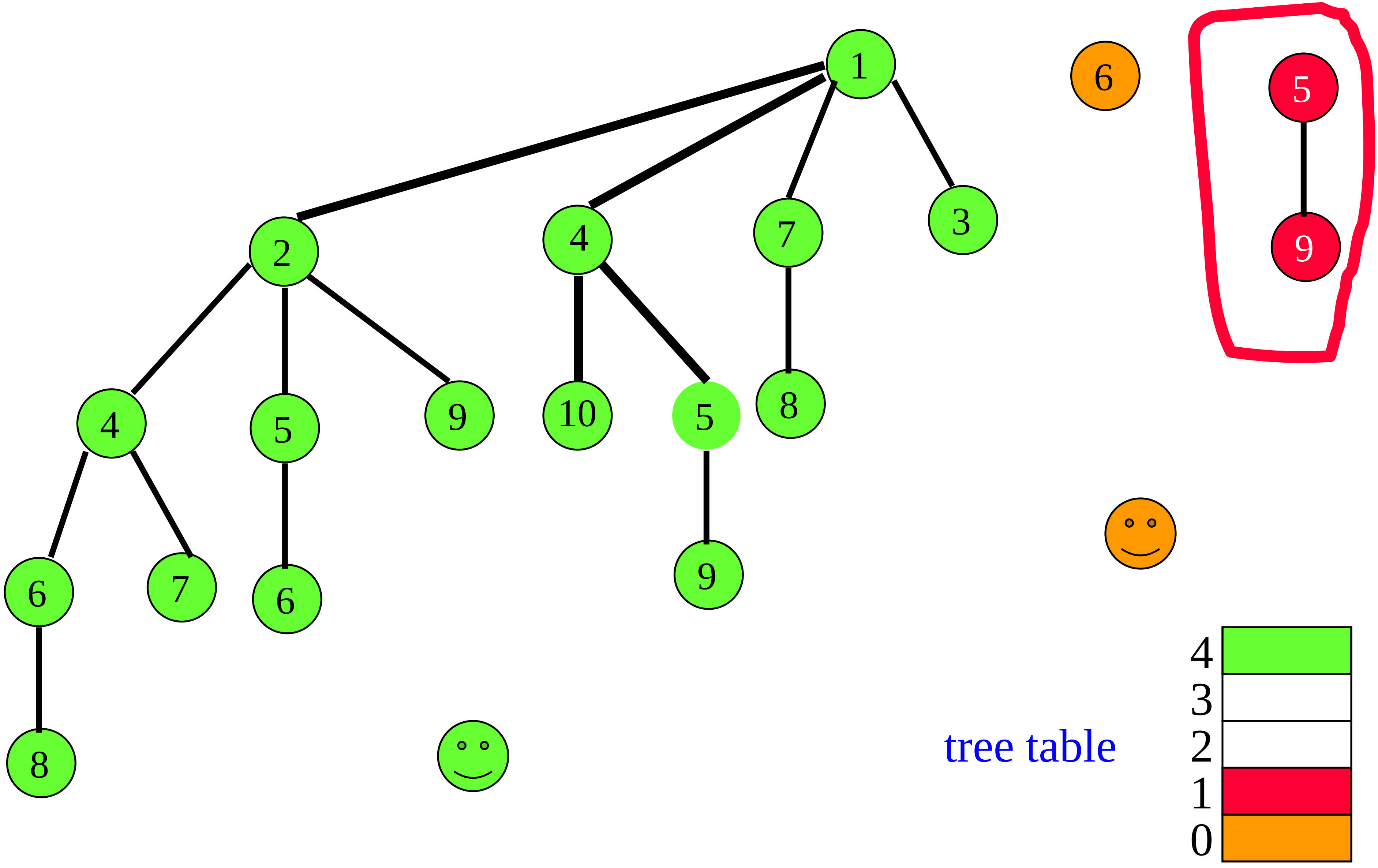
Make the one with larger root a subtree of other.

Pairwise Combine



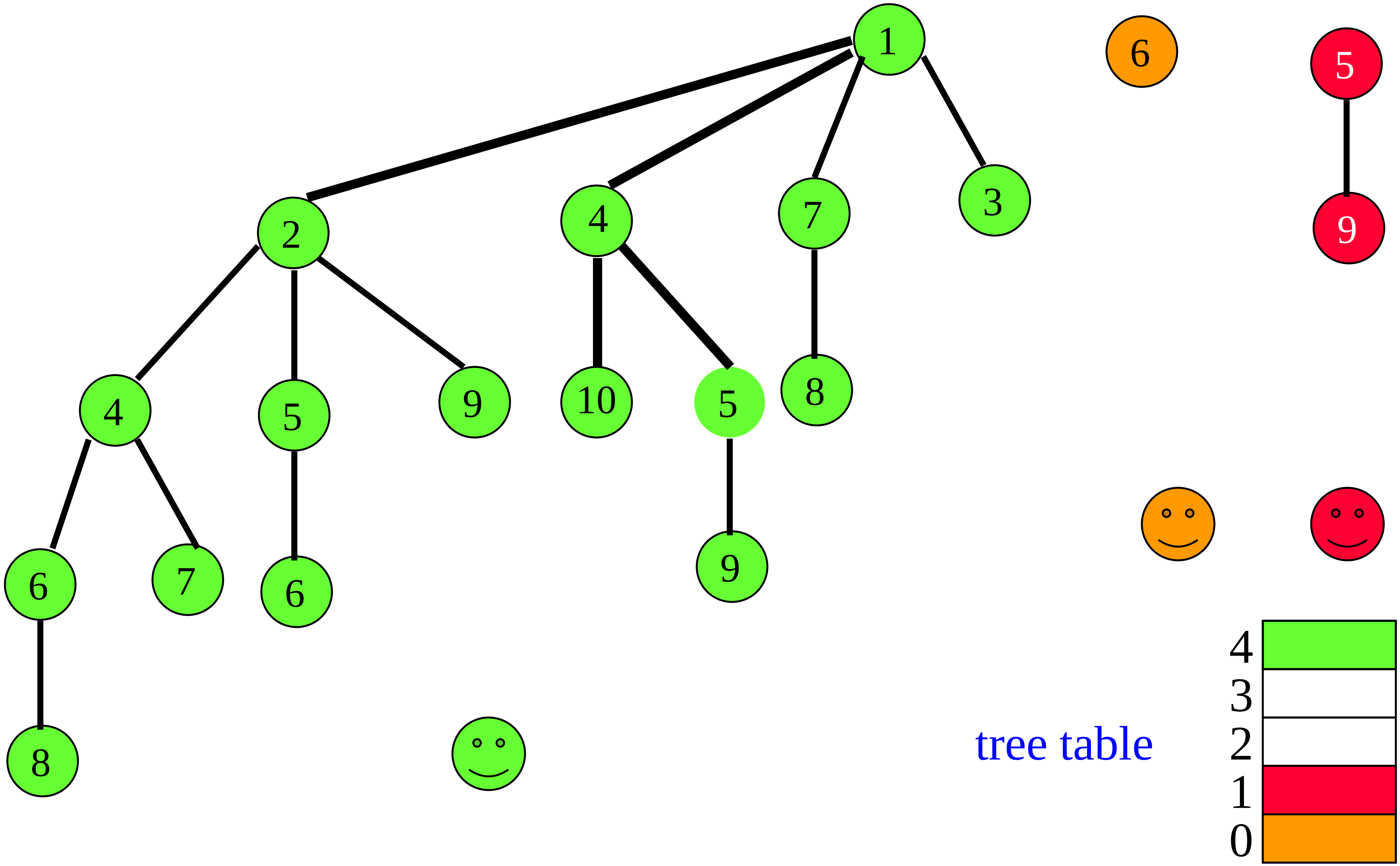
Update tree table.

Pairwise Combine



tree table

Pairwise Combine



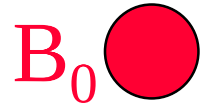
Create circular list of remaining trees.

Complexity Of Delete Min

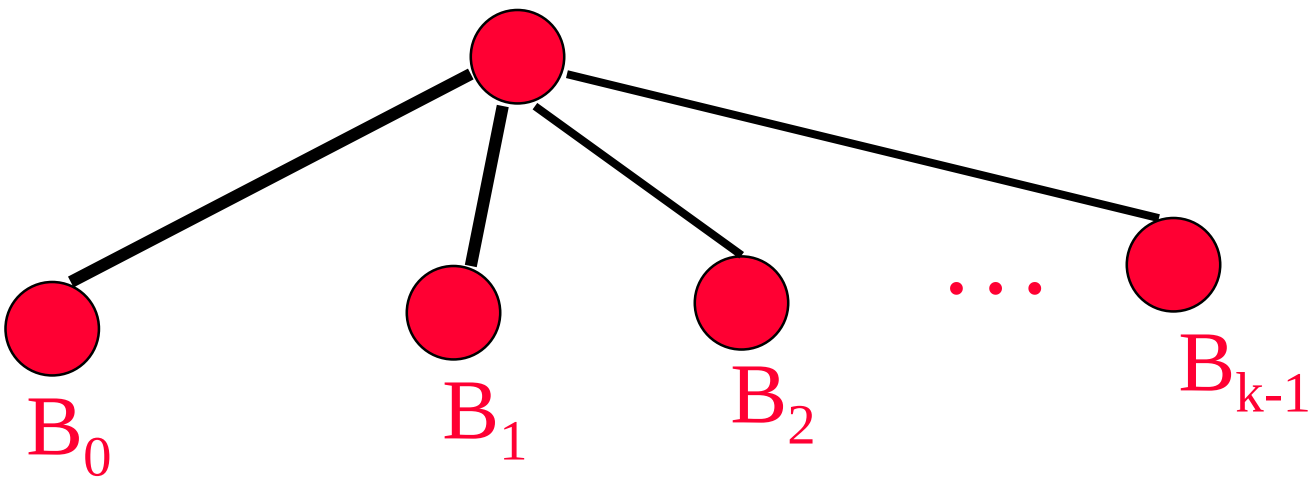
- Create and initialize tree table.
  - O(MaxDegree).
  - Done once only.
- Examine s min trees and pairwise combine.
  - O(s).
- Collect remaining trees from tree table, reset table entries to null, and set binomial heap pointer.
  - O(MaxDegree).
- Overall complexity of remove min.
  - O(MaxDegree + s).

Binomial Trees

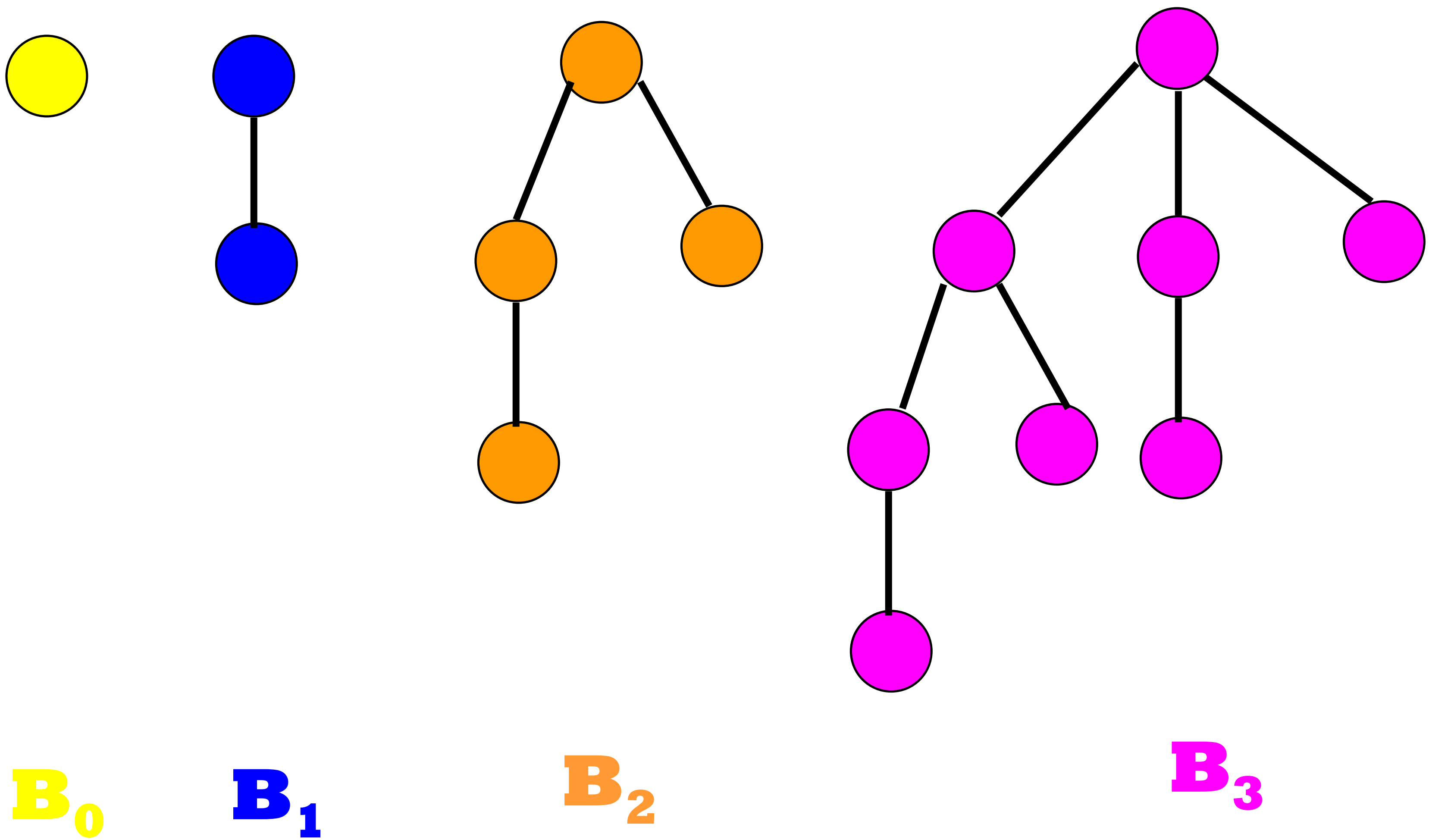
- B_k is degree k binomial tree.



- B_k , $k > 0$, is:

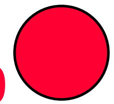


Examples

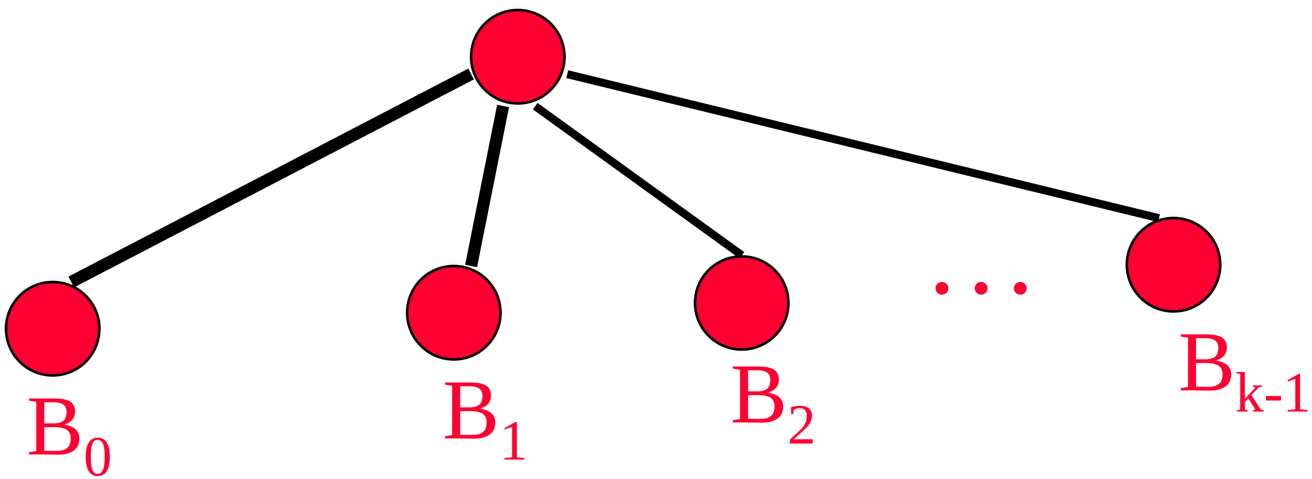


Number Of Nodes In B_k

- N_k = number of nodes in B_k.

B_0  $N_0 = 1$

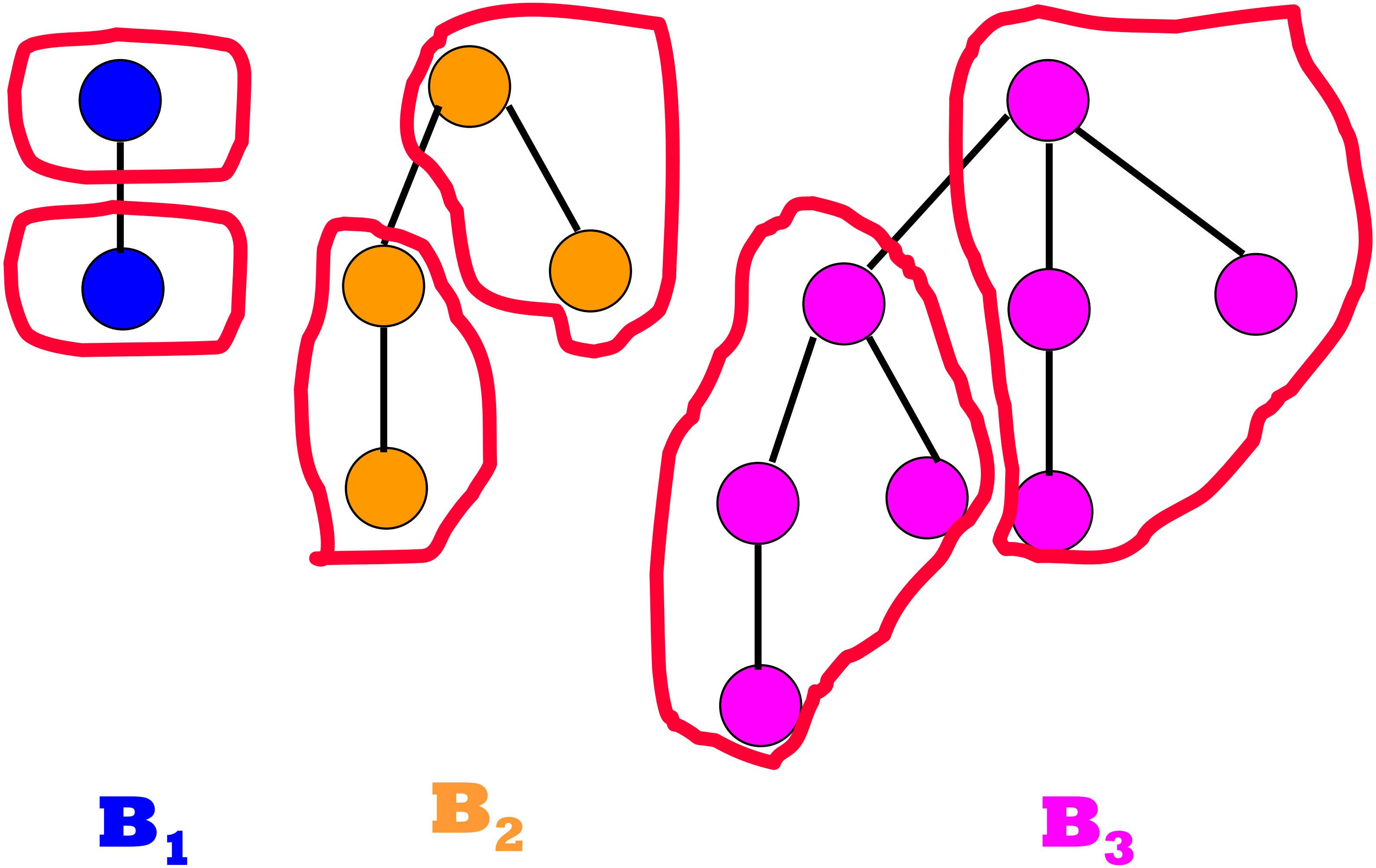
- B_k , $k > 0$, is:



- $N_k = N_0 + N_1 + N_2 + \ldots + N_{k-1} + 1$
  $= 2^k$.

Equivalent Definition

- B_k , $k > 0$, is two B_{k-1}s.
- One of these is a subtree of the other.



N_k And MaxDegree

- $N_0 = 1$
- $N_k = 2N_{k-1}$

  $= 2^k$.
- If we start with zero elements and perform operations as described, then all trees in all binomial heaps are binomial trees.
- So, $\text{MaxDegree} = O(\log n)$.

Analysis Of Binomial Heaps

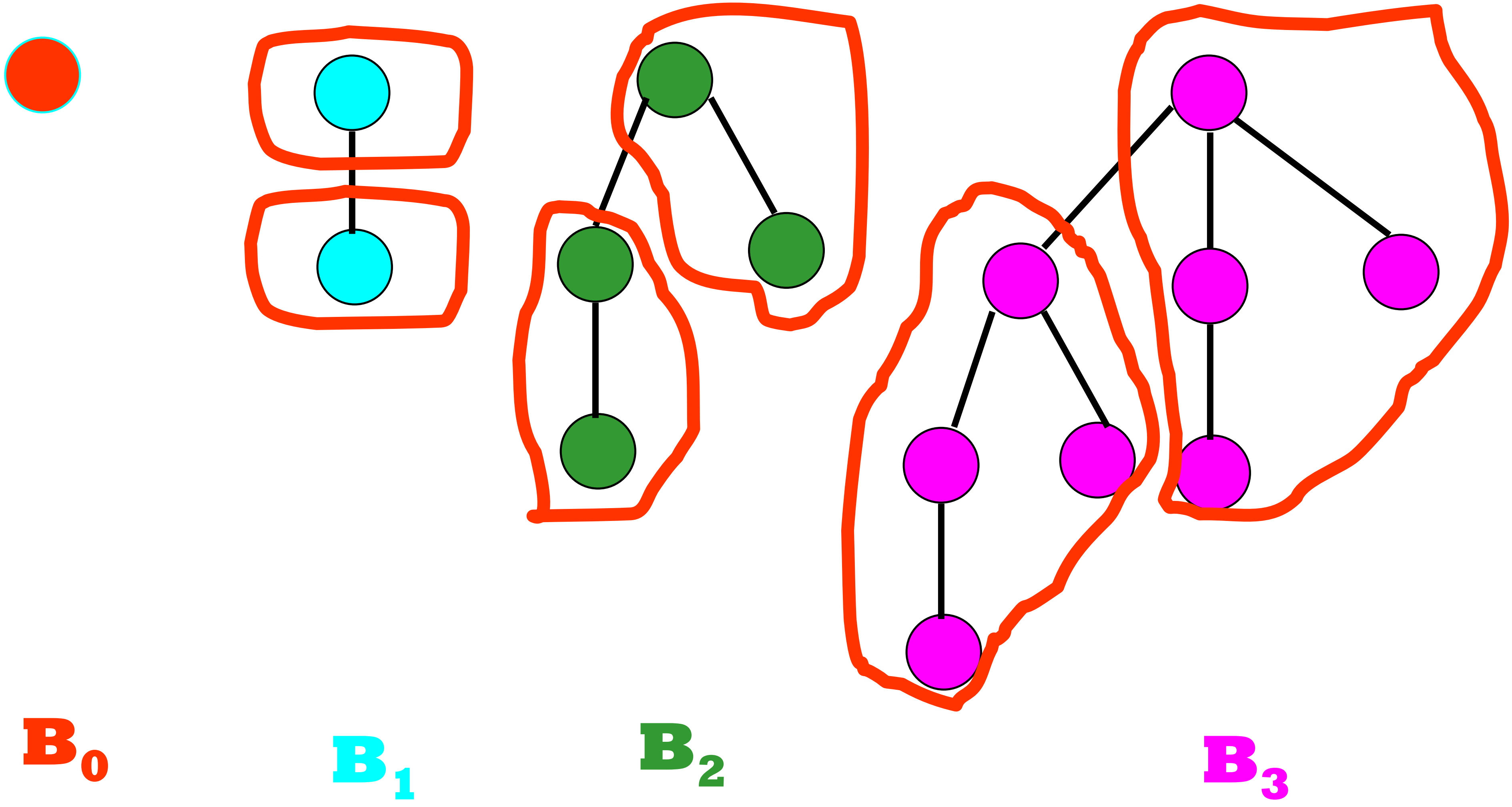
	Leftist trees	Binomial heaps	
		Actual	Amortized
Insert	O(log n)	O(1)	O(1)
Delete min (or max)	O(log n)	O(n)	O(log n)
Meld	O(log n)	O(1)	O(1)

Operations

- Insert
  - Add a new min tree to top-level circular list.
- Meld
  - Combine two circular lists.
- Delete min
  - Pairwise combine min trees whose roots have equal degree.
  - O(MaxDegree + s), where s is number of min trees following removal of min element but before pairwise combining.

Binomial Trees

- B_k , $k > 0$, is two B_{k-1}s.
- One of these is a subtree of the other.



All Trees In Binomial Heap Are Binomial Trees

- Insert creates a B_0.
- Meld does not create new trees.
- Pairwise combine takes two trees of equal degree and makes one a subtree of the other.
- Let n be the number of operations performed.
  - Number of inserts is at most n.
  - No binomial tree has more than n elements.
  - MaxDegree $<= \log_2 n$.
  - Complexity of remove min is $O(\log n + s) = O(n)$.

Aggregate Method

- Get a good bound on the cost of every sequence of operations and divide by the number of operations.
- Results in same amortized cost for each operation, regardless of operation type.
- Can’t use this method, because we want to show a different amortized cost for remove mins than for inserts and melds.

Aggregate Method – Alternative

- Get a good bound on the cost of every sequence of delete mins and divide by the number of delete mins.
- Consider the sequence insert, insert, …, insert, delete min.
  - The cost of the delete min is $O(n)$, where n is the number of operations in the sequence.
  - So, amortized cost of a delete min is $O(n/1) = O(n)$.

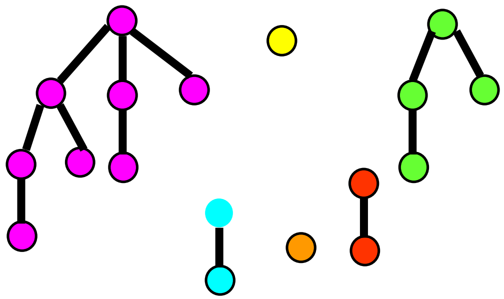
Accounting Method

- Guess the amortized cost.
  - Insert => 2.
  - Meld => 1.
  - Delete min => $3\log_2 n$.
- Show that $P(i) – P(0) >= 0$ for all i.

Potential Function

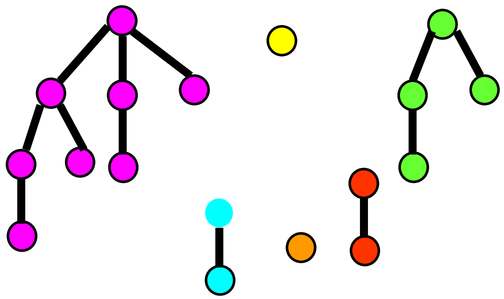
- $P(i) = amortizedCost(i) – actualCost(i) + P(i – 1)$
- $P(i) – P(0)$ is the amount by which the first i operations have been over charged.
- We shall use a credit scheme to show $P(i) – P(0) >= 0$ for all i.
- $P(i) =$ number of credits after operation i.
- Initially number of credits is 0.
- $P(0) = 0$.

Insert



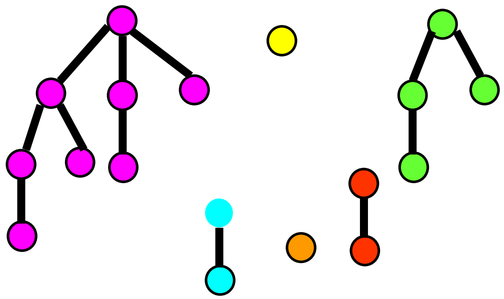
- Guessed amortized cost = 2.
- Use 1 unit to pay for the actual cost of the insert.
- Keep the remaining 1 unit as a credit for a future delete min operation.
- Keep this credit with the min tree that is created by the insert operation.
- Potential increases by 1, because there is an overcharge of 1.

Meld



- Guessed amortized cost = 1.
- Use 1 unit to pay for the actual cost of the meld.
- Potential is unchanged, because actual and amortized costs are the same.

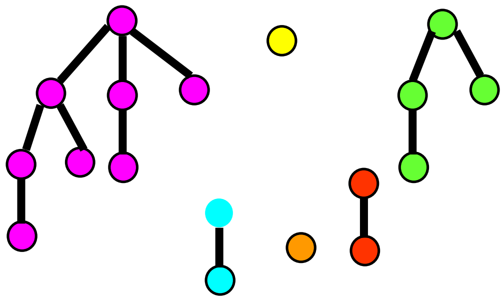
Delete Min



- Let MinTrees be the set of min trees in the binomial heap just before delete min.
- Let u be the degree of min tree whose root is removed.
- Let s be the number of min trees in binomial heap just before pairwise combining.
  - s = #MinTrees + u – 1
- Actual cost of delete min is <= MaxDegree + s

  <= $2\log_2 n - 1$ + #MinTrees.

Delete Min



- Guessed amortized cost $= 3\log_2 n$.
- Actual cost $<= 2\log_2 n - 1 + \#MinTrees$.
- Allocation of amortized cost.
  - Use $2\log_2 n - 1$ to pay part of actual cost.
  - Keep remaining $\log_2 n + 1$ as a credit to pay part of the actual cost of a future delete min operation.
  - Put 1 unit of credit on each of the at most $\log_2 n + 1$ min trees left behind by the delete min operation.
  - Discard the remaining credits (if any).

Paying Actual Cost Of A Delete Min

- Actual cost $<= 2\log_2 n - 1 + \#MinTrees$
- How is it paid for?
  - $2\log_2 n - 1$ comes from amortized cost of this delete min operation.
  - #MinTrees comes from the min trees themselves, at the rate of 1 unit per min tree.
  - Potential remains nonnegative, because there are enough credits to pay the balance of the actual cost.

Potential Method

- Guess a suitable potential function for which $P(i) - P(0) >= 0$ for all i.
- Derive amortized cost of ith operation using
  $\Delta P = P(i) - P(i-1)$
  $= \text{amortized cost} - \text{actual cost}$
- amortized cost = actual cost + ΔP

Potential Function

- $P(i) = \Sigma\#MinTrees(j)$
  - #MinTrees(j) is #MinTrees for binomial heap j.
  - When binomial heaps A and B are melded, A and B are no longer included in the sum.
- $P(0) = 0$
- $P(i) >= 0$ for all i.
- ith operation is an insert.
  - Actual cost of insert = 1
  - $\Delta P = P(i) - P(i-1) = 1$
  - Amortized cost of insert = actual cost + ΔP

    = 2

ith Operation Is A Meld

- Actual cost of meld = 1
- $P(i) = \Sigma\#MinTrees(j)$
- $\Delta P = P(i) - P(i-1) = 0$
- Amortized cost of meld = actual cost + ΔP

  = 1

ith Operation Is A Delete Min

- old => value just before the delete min
- new => value just after the delete min.
- $\#MinTrees^{old}(j)$ => value of #MinTrees in jth binomial heap just before this delete min.
- Assume delete min is done in kth binomial heap.

ith Operation Is A Delete Min

- Actual cost of delete min from binomial heap k

  $<= 2\log_2 n - 1 + \#MinTrees^{old}(k)$

- $\Delta P = P(i) - P(i - 1)$

  $= \Sigma[\#MinTrees^{new}(j) - \#MinTrees^{old}(j)]$

  $= \#MinTrees^{new}(k) - \#MinTrees^{old}(k)$.

- Amortized cost of delete min = actual cost + ΔP

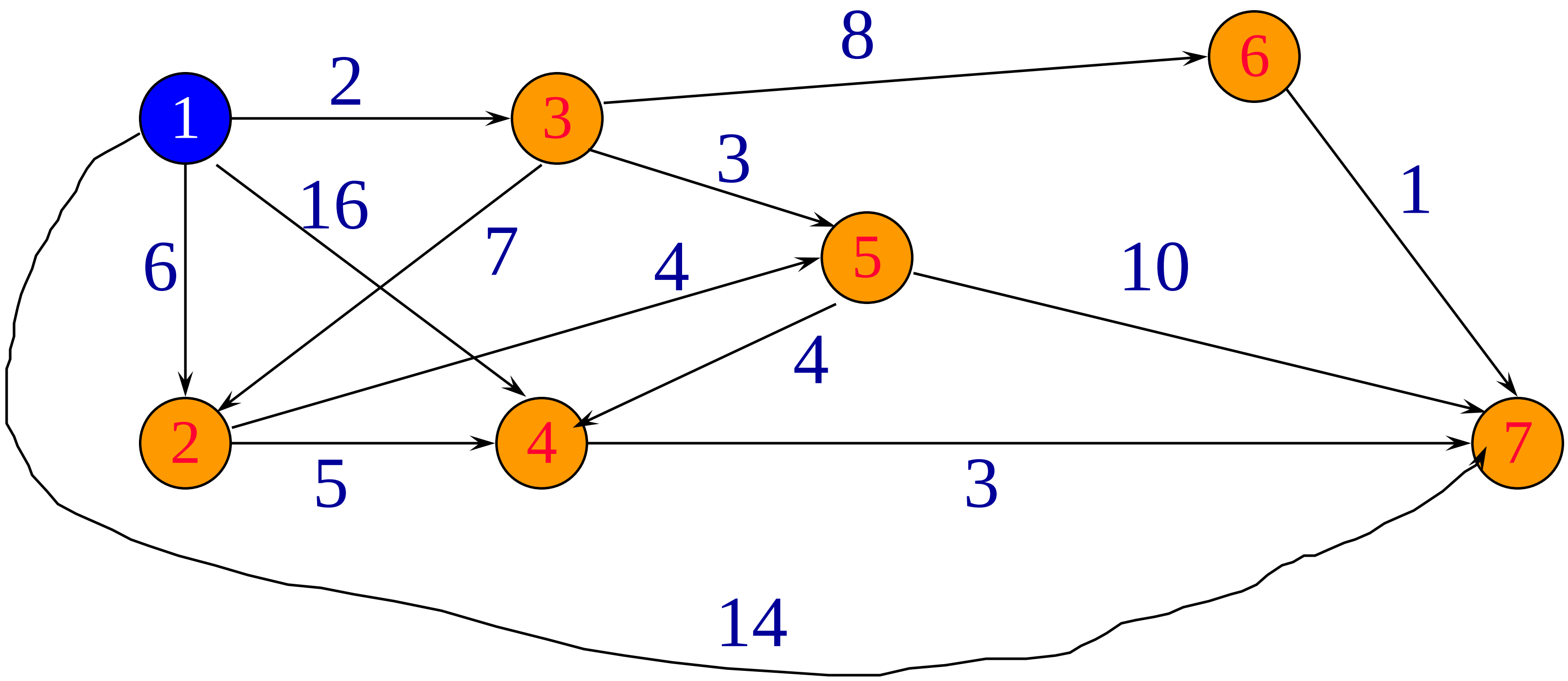
  $<= 2\log_2 n - 1 + \#MinTrees^{new}(k)$

  $<= 3\log_2 n$.

Fibonacci Heaps, referenced

	Actual	Amortized
Insert	O(1)	O(1)
Delete min (or max)	O(n)	O(log n)
Meld	O(1)	O(1)
Delete	O(n)	O(log n)
Decrease key (or increase)	O(n)	O(1)

Single Source All Destinations Shortest Paths



Greedy Single Source All Destinations

- Known as Dijkstra’s algorithm.
- Let d(i) be the length of a shortest one edge extension of an already generated shortest path, the one edge extension ends at vertex i.
- The next shortest path is to an as yet unreached vertex for which the d() value is least.
- After the next shortest path is generated, some d() values are updated (decreased).

Operations On d()

- Remove min.
  - Done O(n) times, where n is the number of vertices in the graph.
- Decrease d().
  - Done O(e) times, where e is the number of edges in the graph.
- Array.
  - $O(n^2)$ overall complexity.
- Min heap.
  - $O(n\log n + e\log n)$ overall complexity.
- Fibonacci heap.
  - $O(n\log n + e)$ overall complexity.

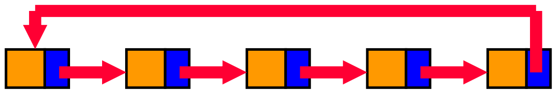
Prim's Min-Cost Spanning Tree Algorithm

- Array.
  - $O(n^2)$ overall complexity.
- Min heap.
  - $O(n\log n + e\log n)$ overall complexity.
- Fibonacci heap.
  - $O(n\log n + e)$ overall complexity.

Min Fibonacci Heap

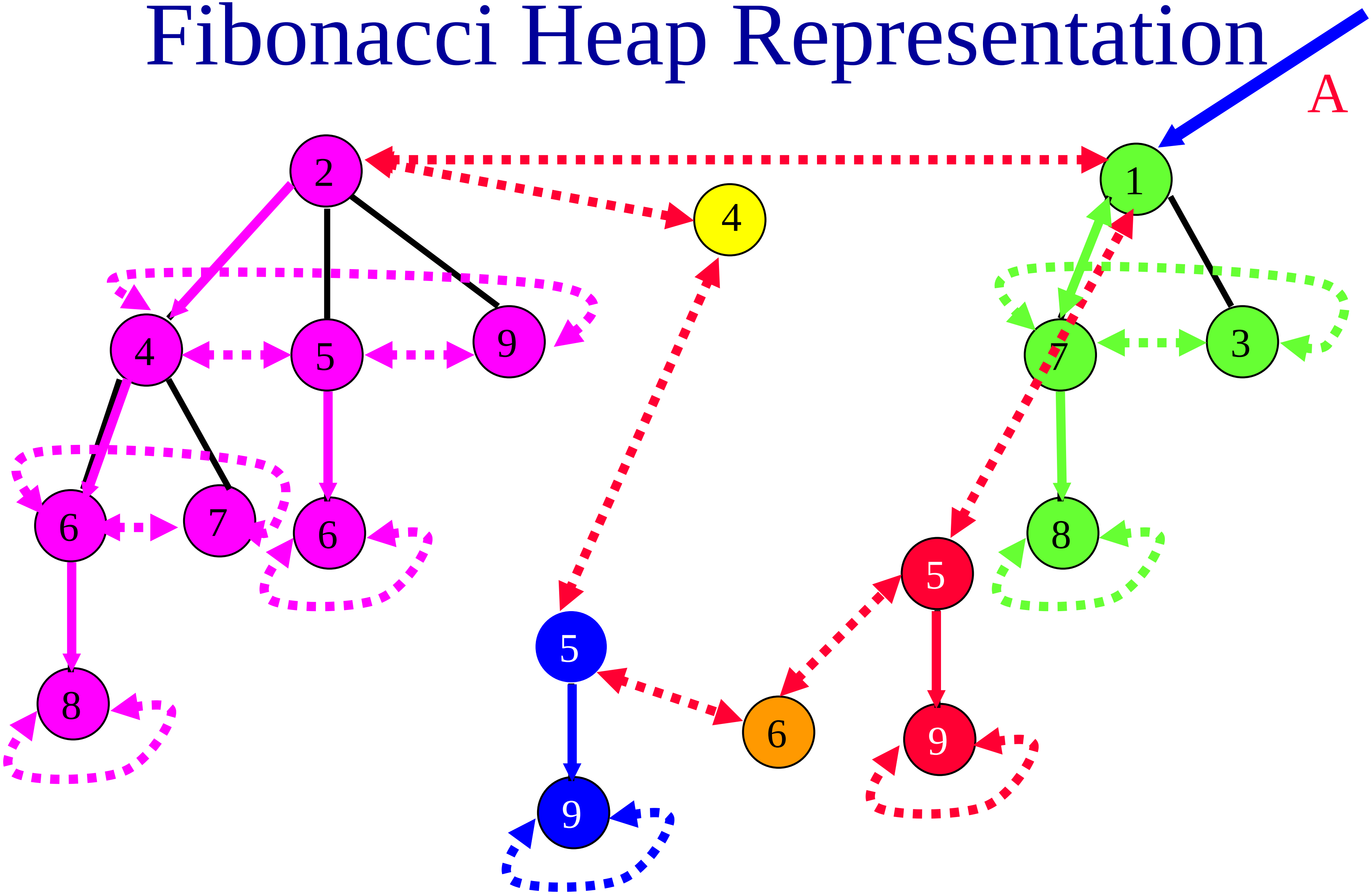
- Collection of min trees.
- The min trees need not be Binomial trees.

Node Structure



- Degree, Child, Data
- Left and Right Sibling
  - Used for circular doubly linked list of siblings.
- Parent
  - Pointer to parent node.
- ChildCut
  - True if node has lost a child since it became a child of its current parent.
  - Set to false by remove min, which is the only operation that makes one node a child of another.
  - Undefined for a root node.

Fibonacci Heap Representation

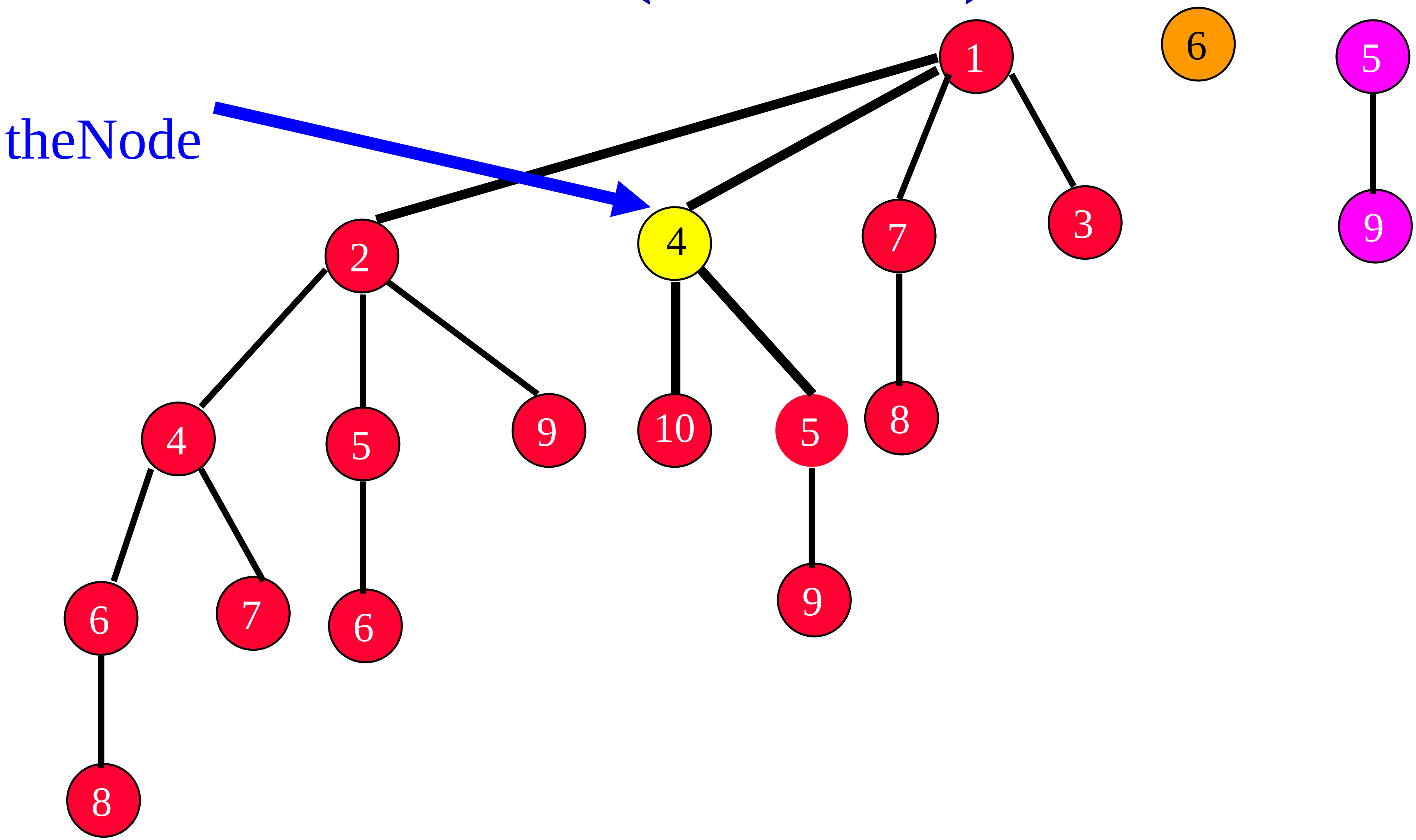


- Parent and ChildCut fields not shown.

Delete(theNode)

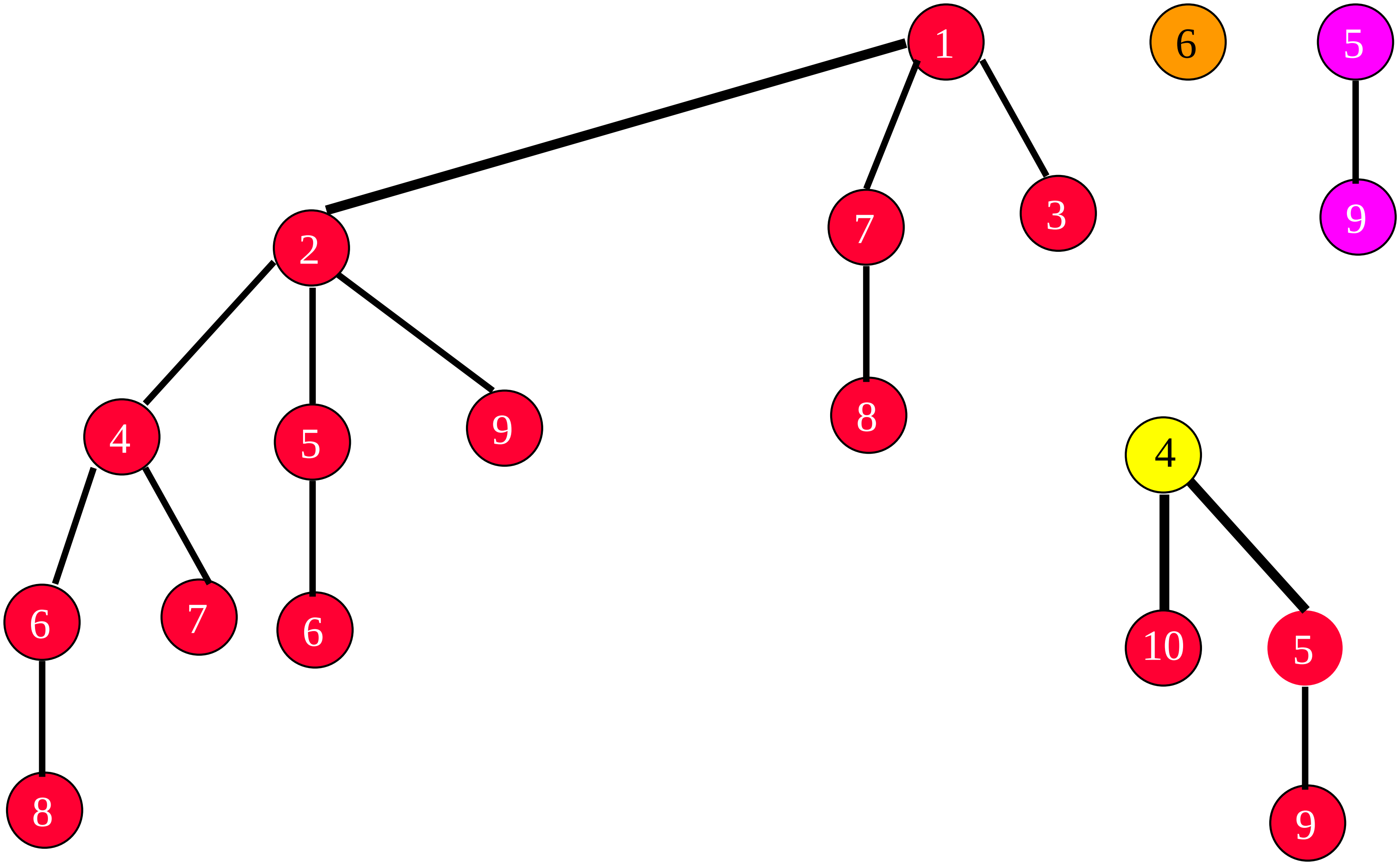
- theNode points to the Fibonacci heap node that contains the element that is to be deleted.
- theNode points to min element => do a delete min.
  - In this case, complexity is the same as that for delete min.

Delete(theNode)



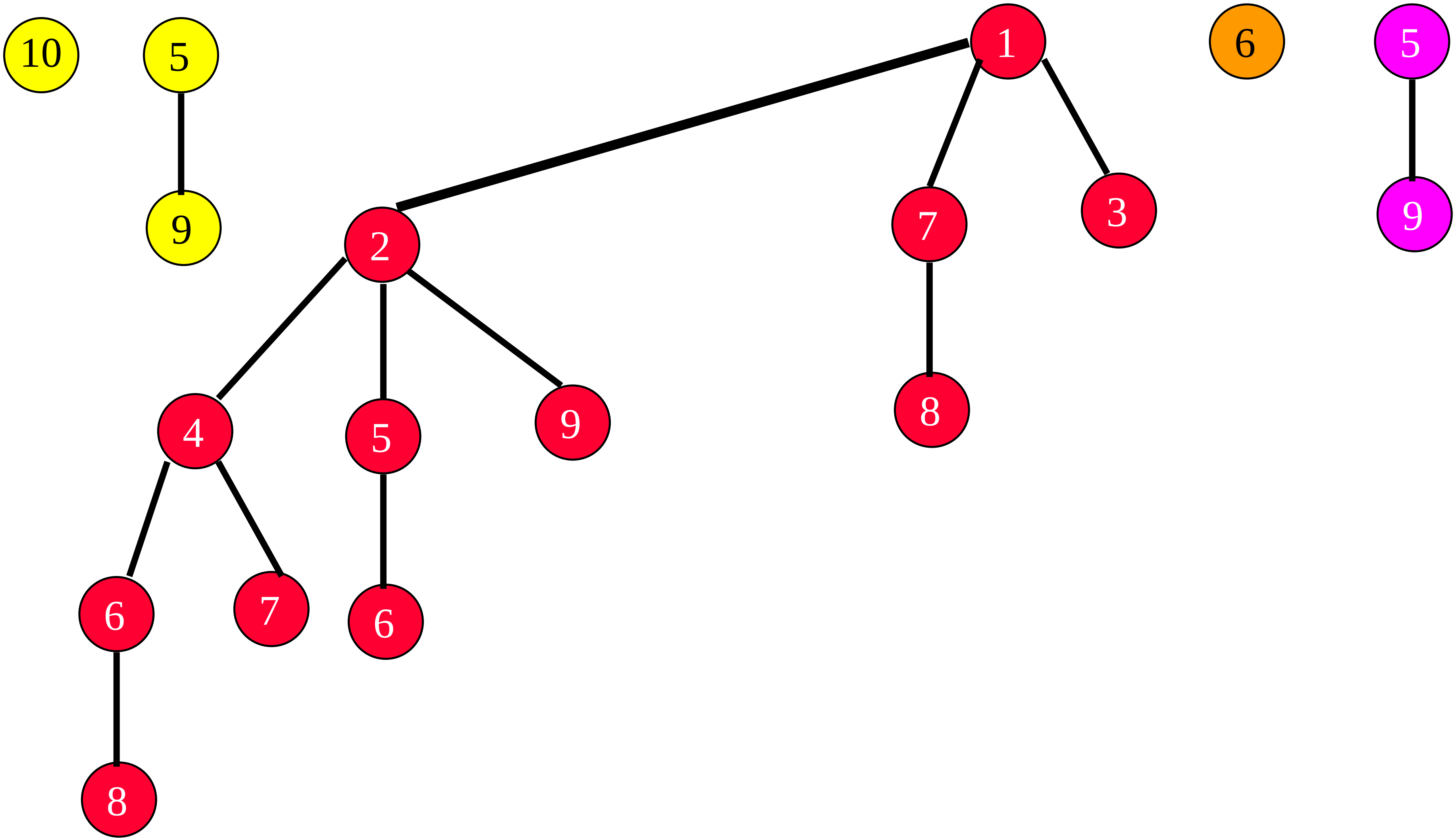
Remove theNode from its doubly linked sibling list.

Delete(theNode)

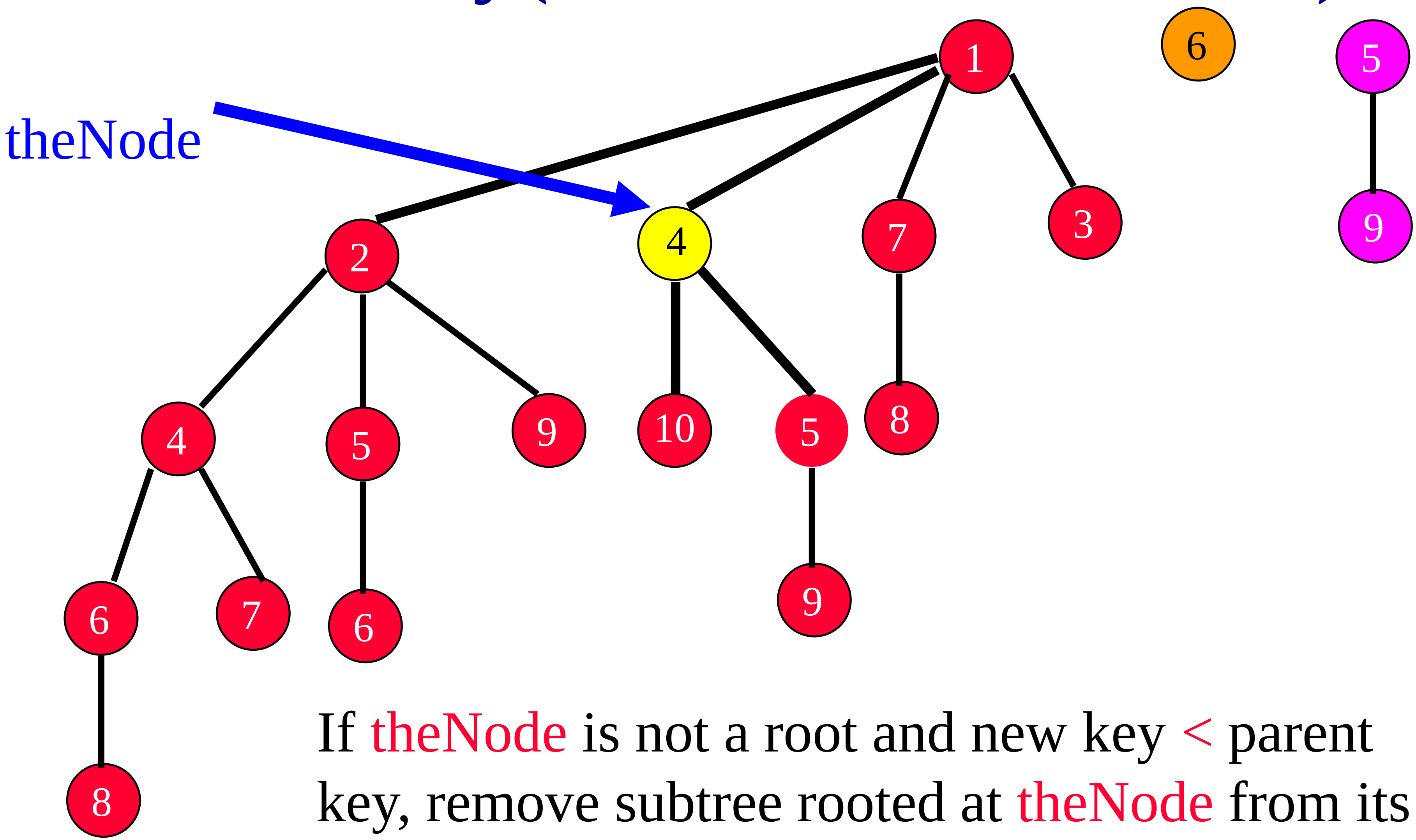


Combine top-level list and children of theNode.

Delete(theNode)



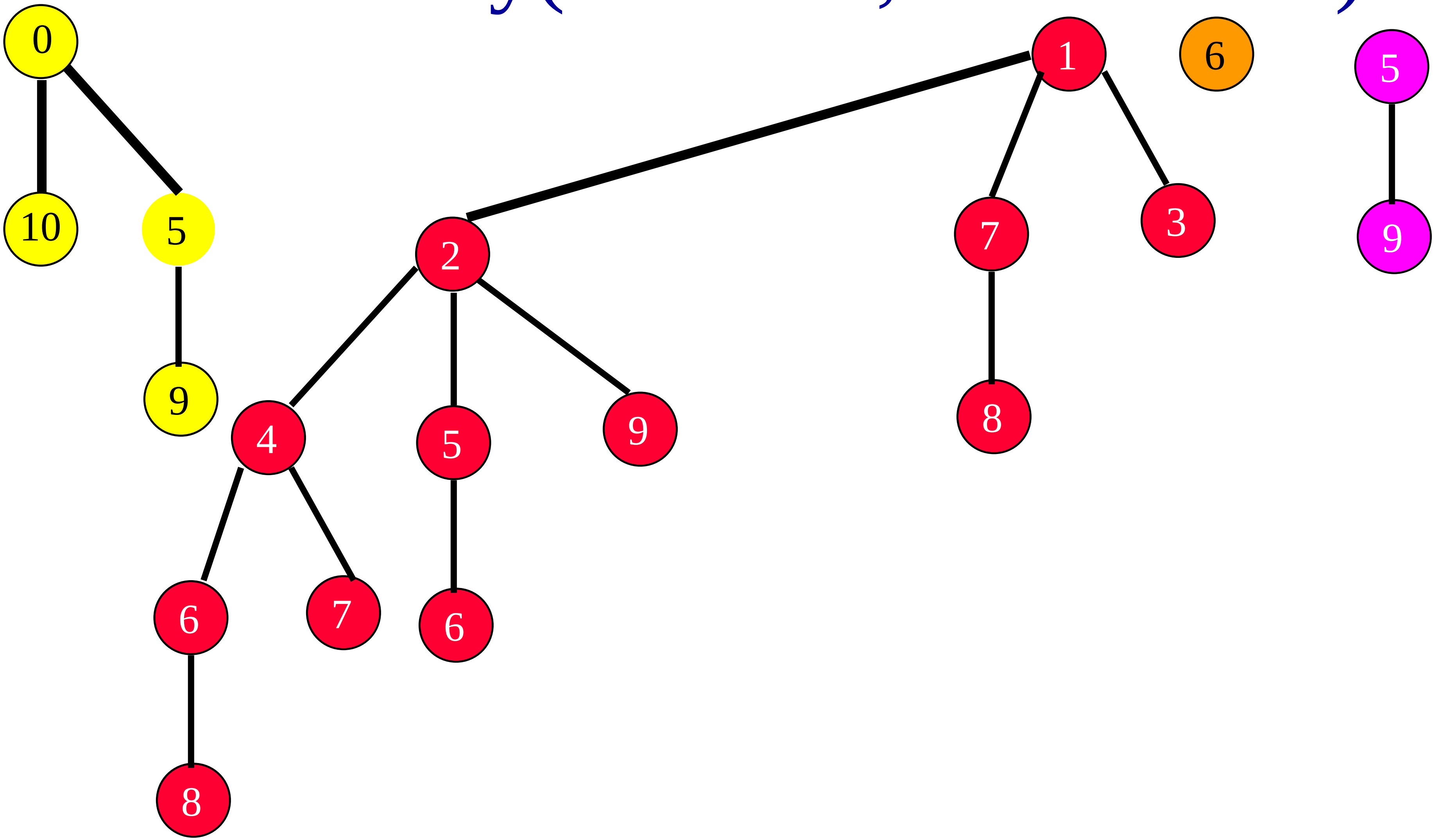
DecreaseKey(theNode, theAmount)



If theNode is not a root and new key < parent key, remove subtree rooted at theNode from its doubly linked sibling list.

Insert into top-level list.

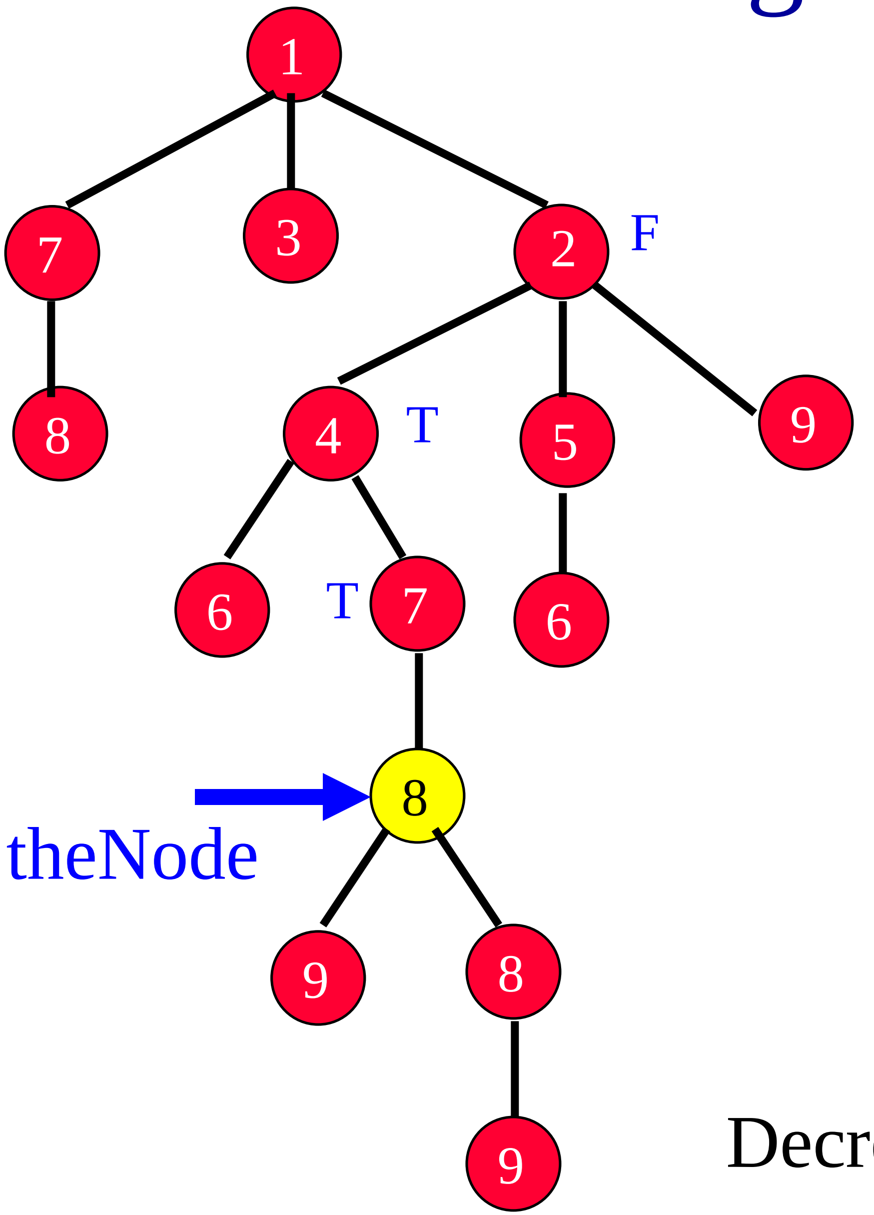
DecreaseKey(theNode, theAmount)



Cascading Cut

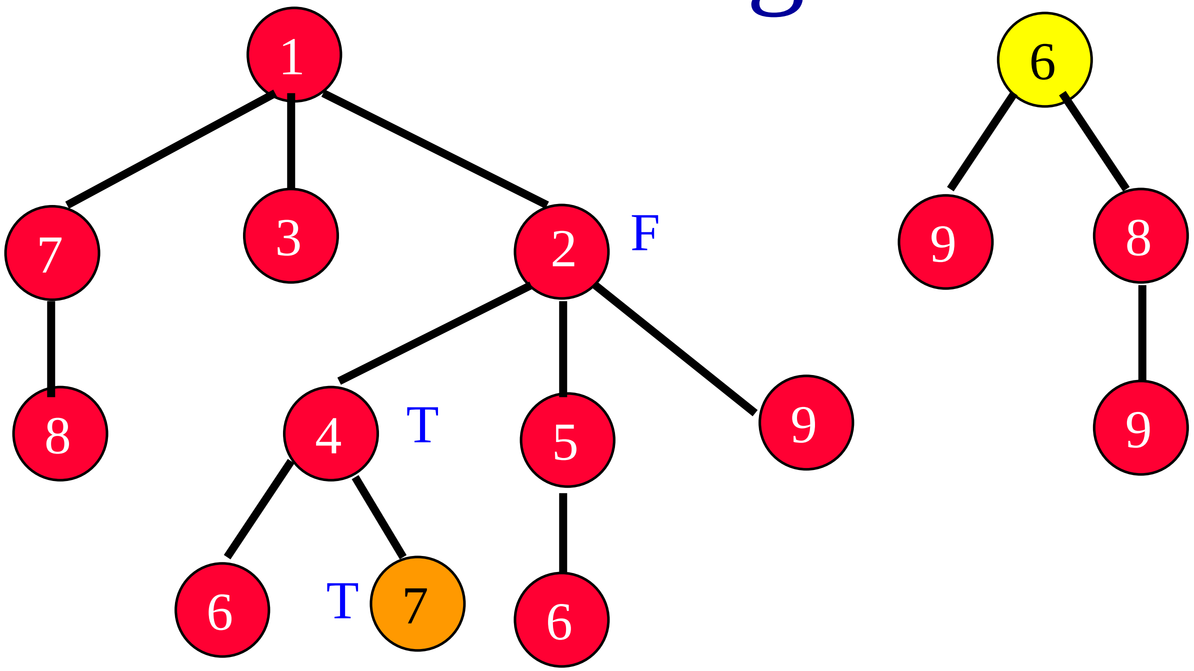
- When theNode is cut out of its sibling list in a remove or decrease key operation, follow path from parent of theNode to the root.
- Encountered nodes (other than root) with ChildCut = true are cut from their sibling lists and inserted into top-level list.
- Stop at first node with ChildCut = false.
- For this node, set ChildCut = true.

Cascading Cut Example

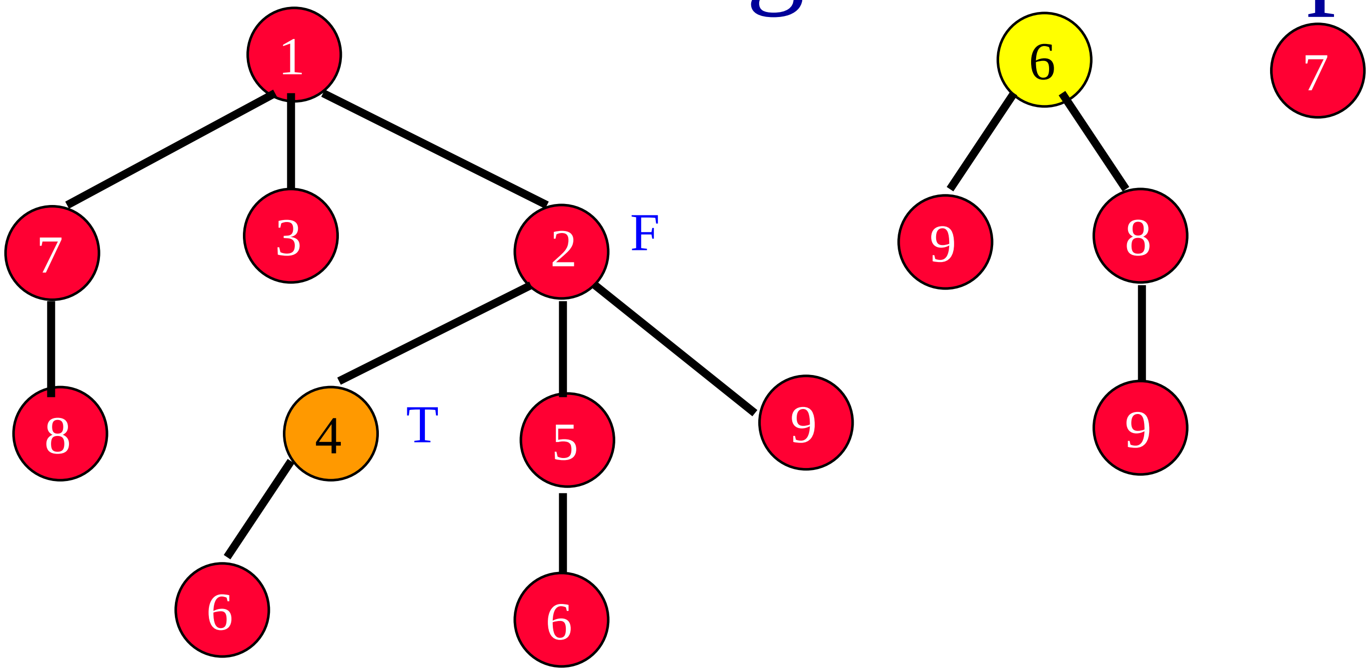


Decrease key by 2.

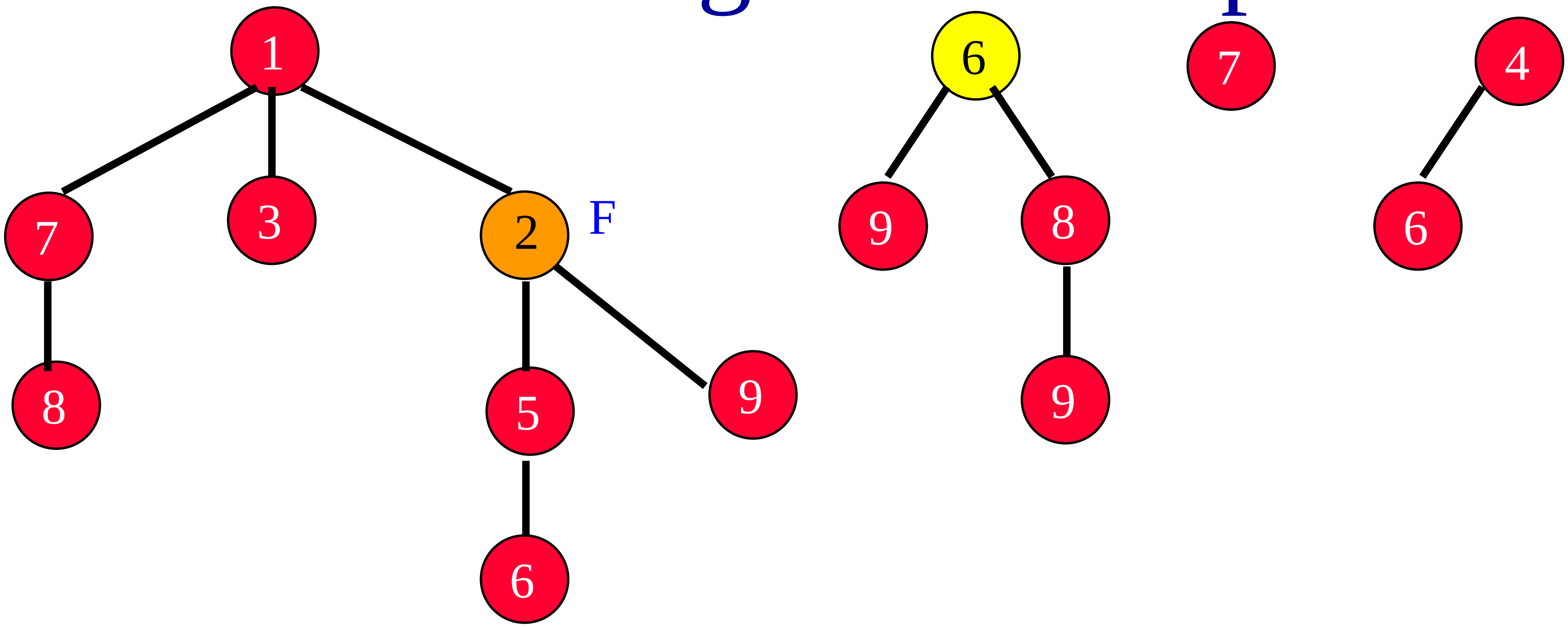
Cascading Cut Example



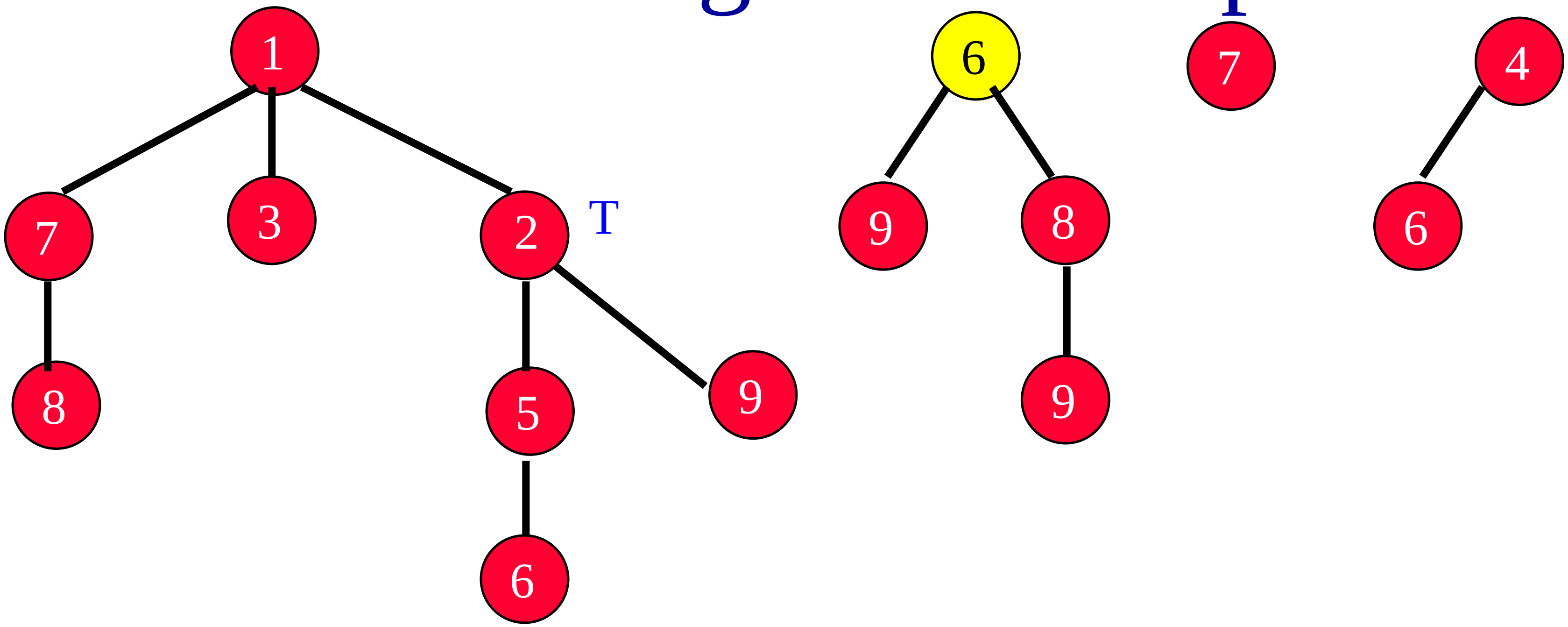
Cascading Cut Example



Cascading Cut Example



Cascading Cut Example



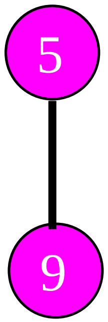
Actual complexity of cascading cut is O(h) = O(n).

Analysis Of Fibonacci Heaps

	Actual	Amortized
Insert	O(1)	O(1)
Delete min (or max)	O(n)	O(log n)
Meld	O(1)	O(1)
Delete	O(n)	O(log n)
Decrease key (or increase)	O(n)	O(1)

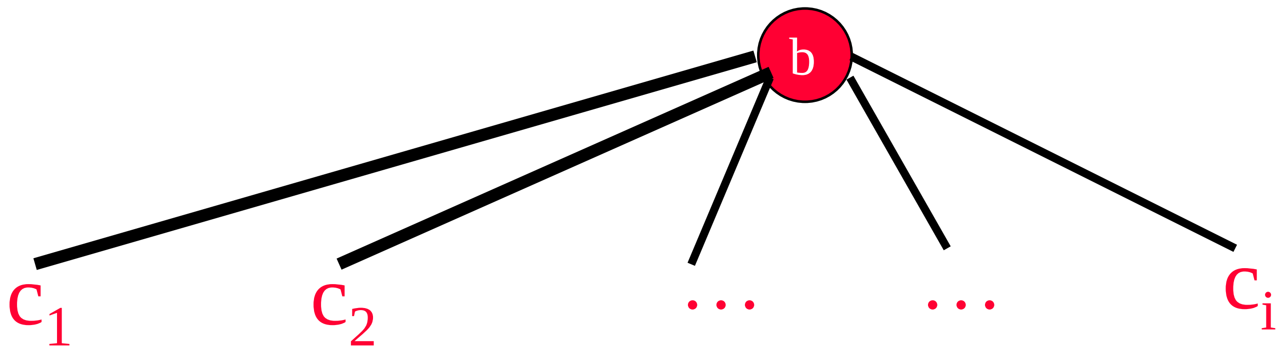
(**MaxDegree not included in this year**)

- Let N_i = min # of nodes in any min (sub)tree whose root has i children.
- $N_0 = 1$.



- $N_1 = 2$.

N_i , i > 1

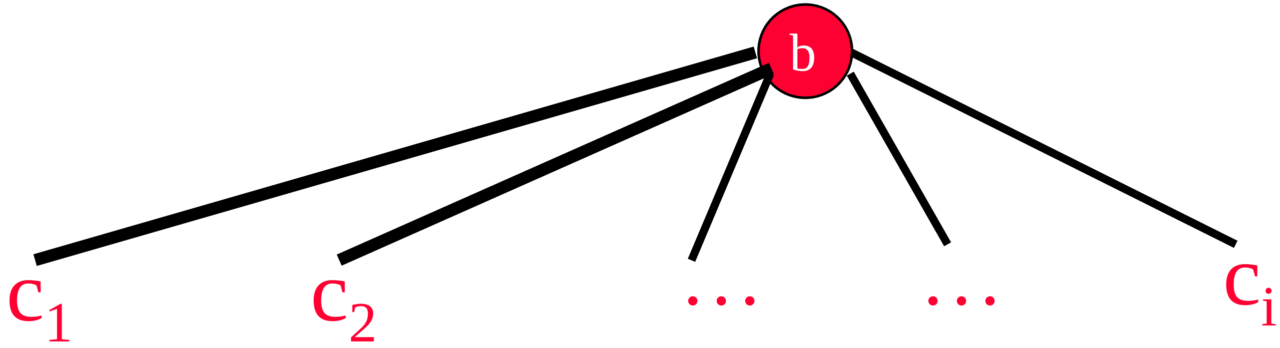


- Children of b are labeled in the order in which they became children of b.
  - c_1 became a child before c_2 did, and so on.
- So, when c_k became a child of b, degree(b) >= k –1.
- degree(c_k) at the time when c_k became a child of b

  = degree(b) at the time when c_k became a child of b

  >= k – 1.

N_i , i > 1



- So, current degree(c_k) >= max{0, k – 2}.
- So, $N_i = N_0 + (\Sigma_{0<=q<=i-2} N_q) + 1$

$$= (\Sigma_{0<=q<=i-2} N_q) + 2.$$

Fibonacci Numbers

- $F_0 = 0$.
- $F_1 = 1$.
- $F_i = F_{i-1} + F_{i-2}$, $i > 1$

  $= (\Sigma_{0<=q<=i-2} F_q) + 1$, $i > 1$.

- $N_0 = 1$.
- $N_1 = 2$.
- $N_i = (\Sigma_{0<=q<=i-2} N_q) + 2$, $i > 1$.

- $N_i = F_{i+2} \sim ((1 + \mathrm{sqrt}(5))/2)^i$, $i >= 0$.

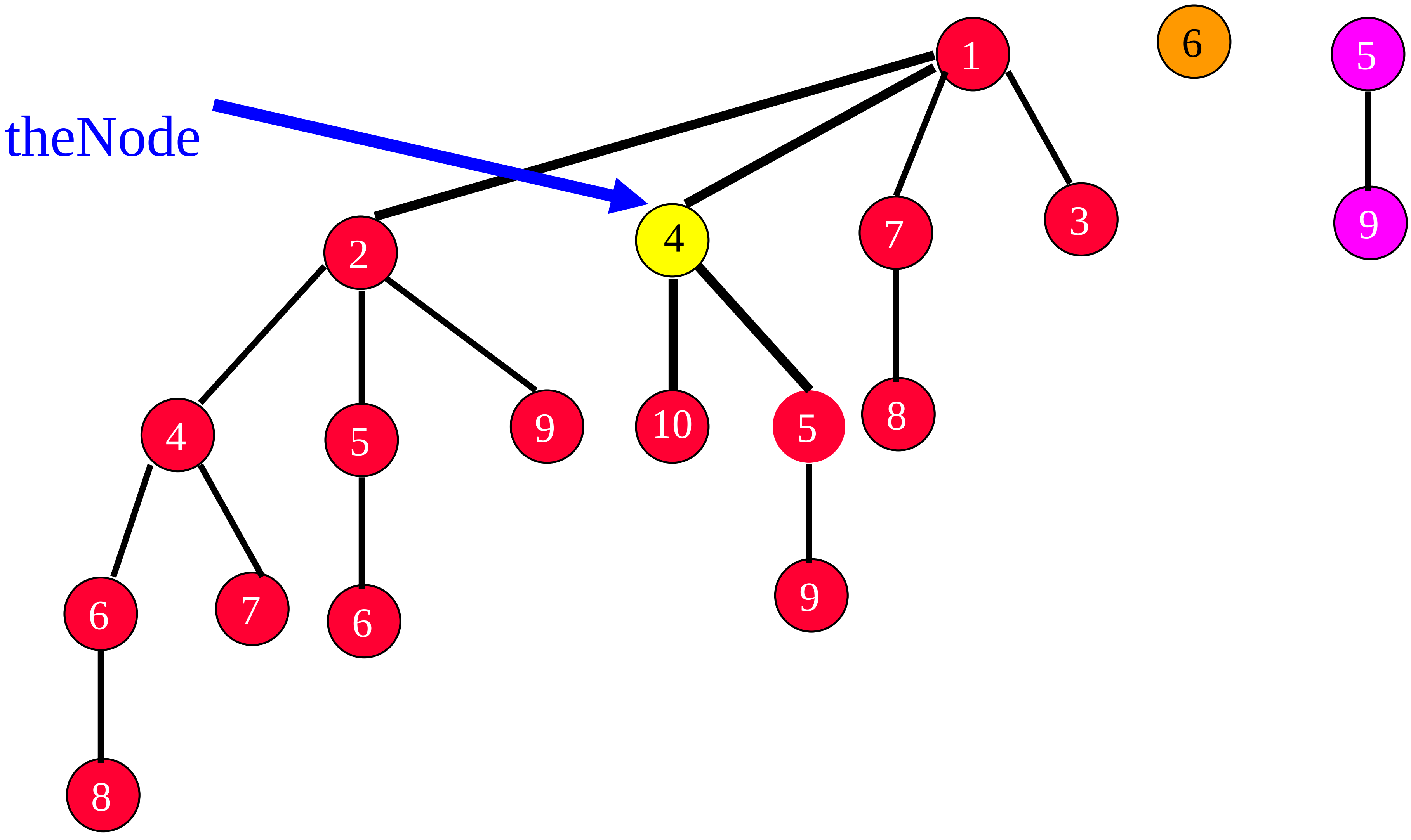
MaxDegree

- MaxDegree <= $\log_{\phi} n$, where $\phi = (1 + \mathrm{sqrt}(5))/2$.

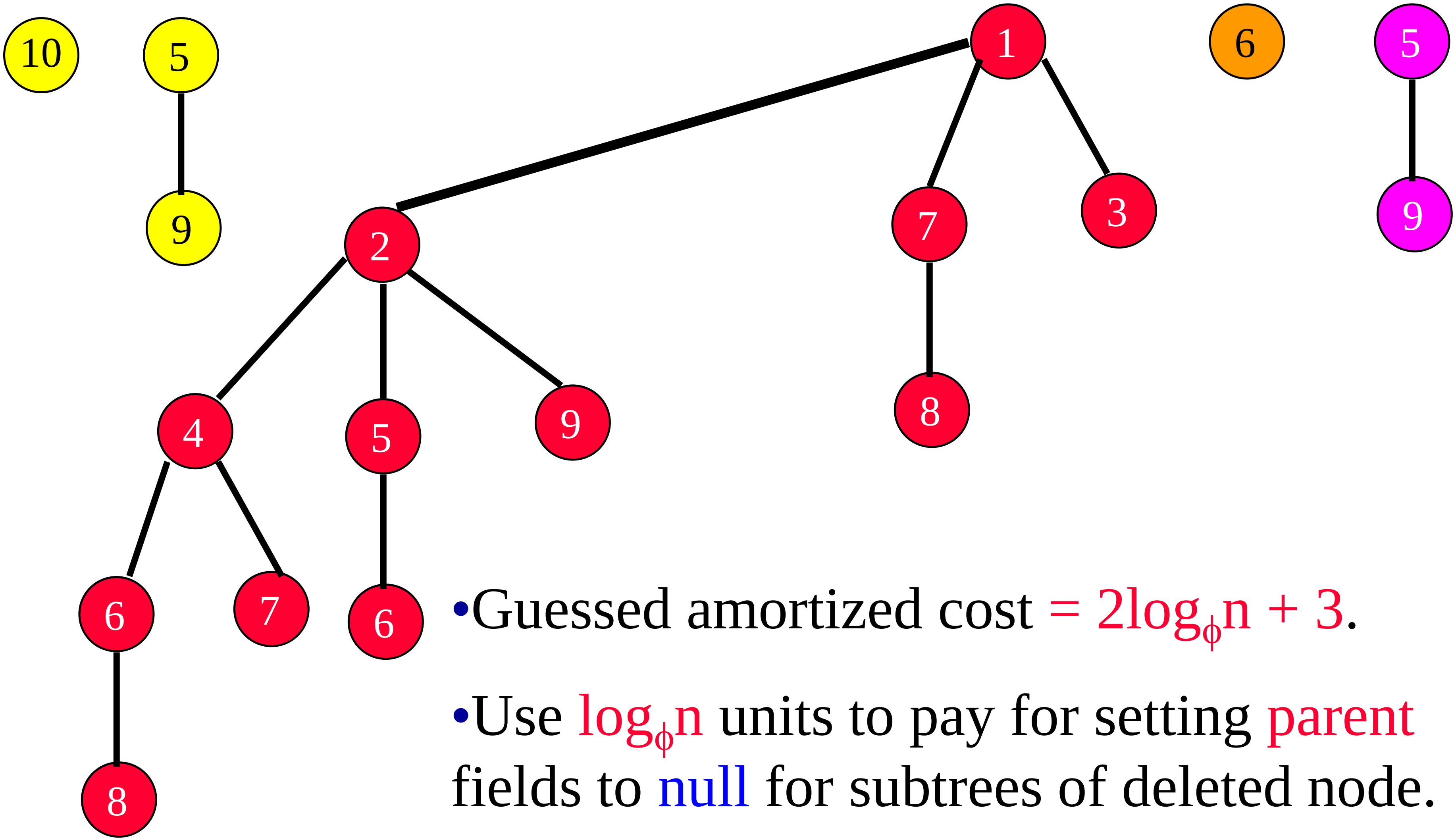
Accounting Method

- Insert.
  - Guessed amortized cost = 2.
  - Use 1 unit to pay for the actual cost of the insert and keep the remaining 1 unit as a credit for a future remove min operation.
  - Keep this credit with the min tree that is created by the insert operation.
- Meld.
  - Guessed amortized cost = 1.
  - Use 1 unit to pay for the actual cost of the meld.

Delete Nonmin Element

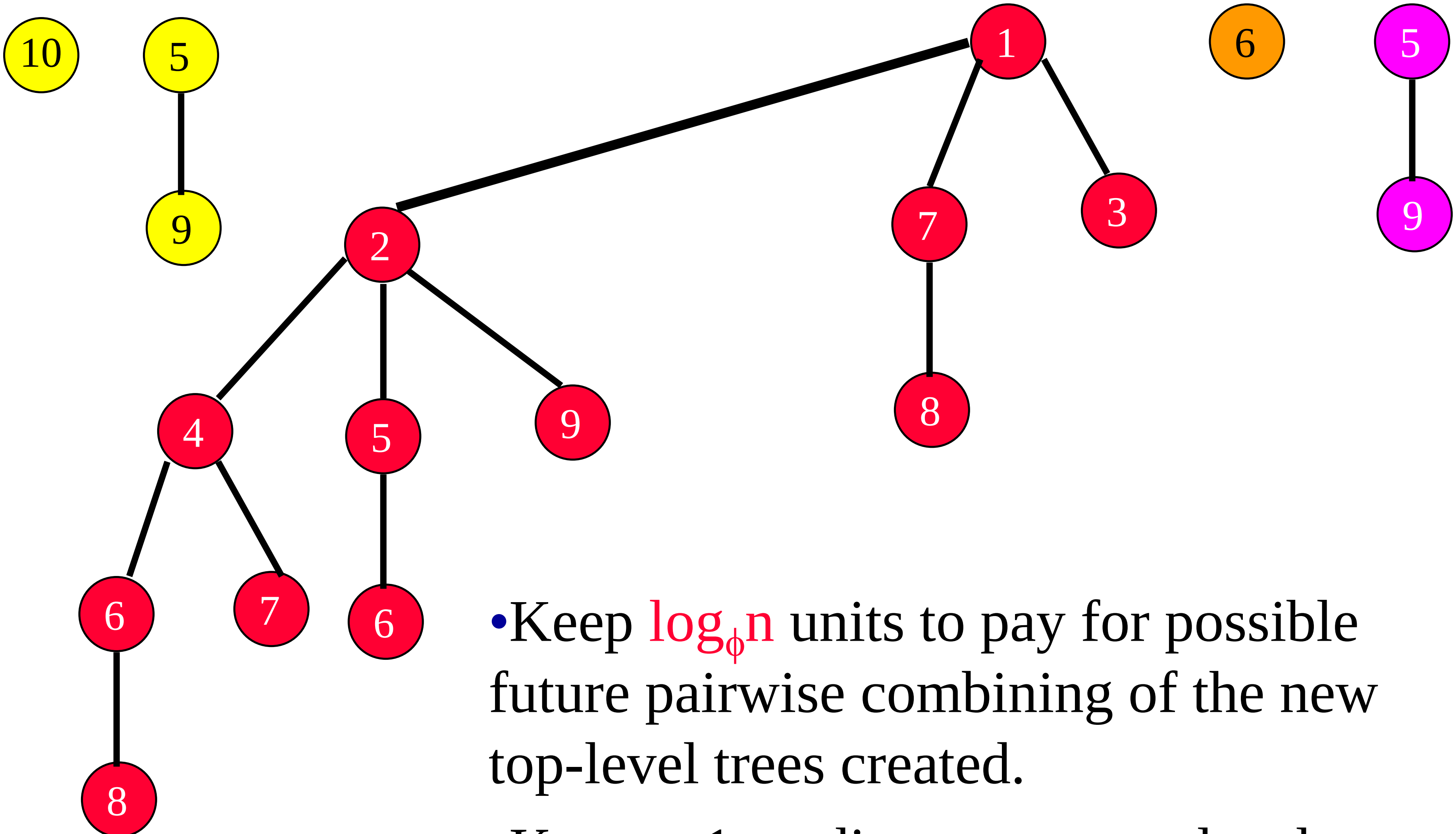


Delete Nonmin Element



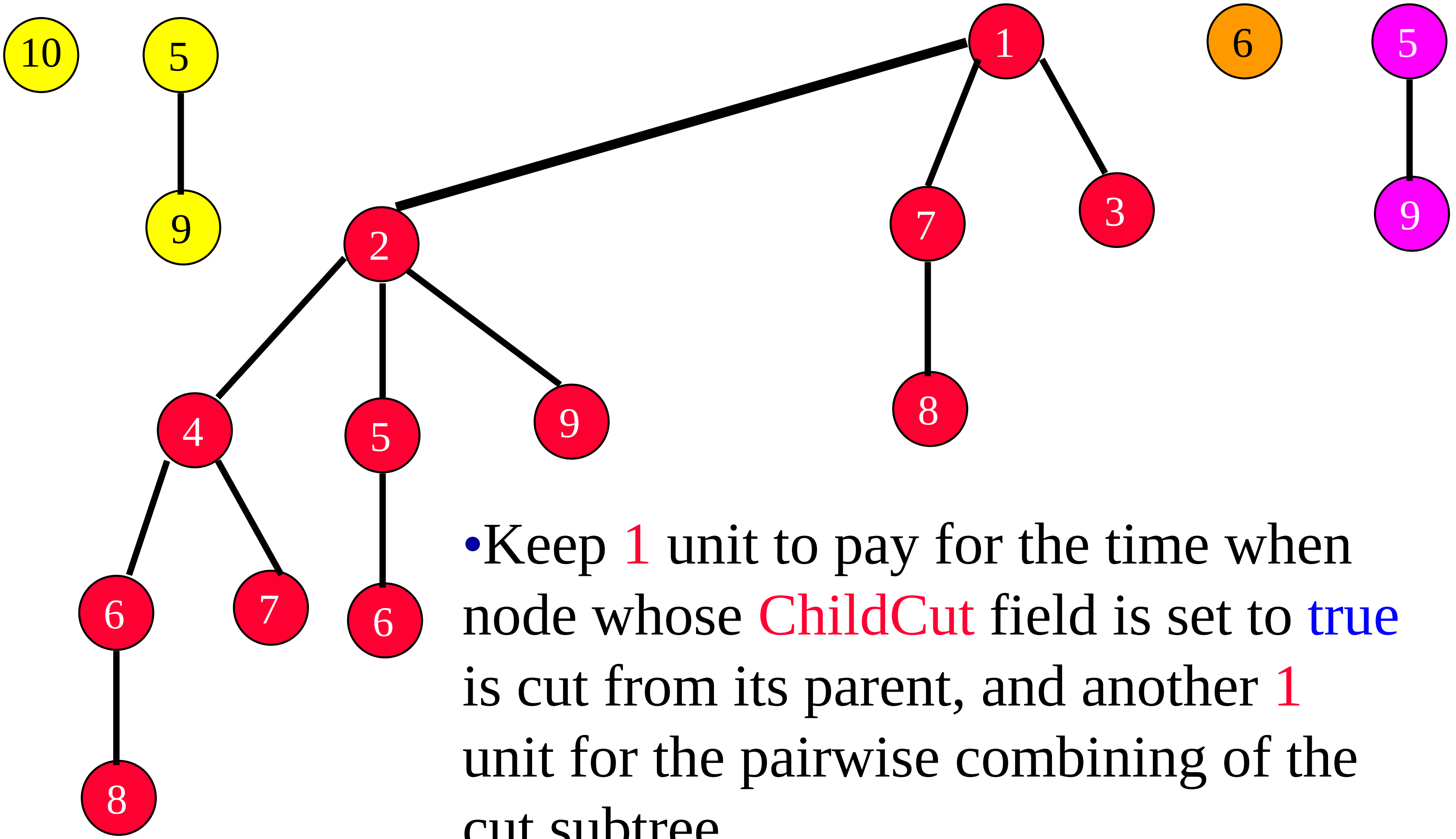
- Guessed amortized cost $= 2\log_\phi n + 3$.
- Use $\log_\phi n$ units to pay for setting parent fields to null for subtrees of deleted node.
- Use 1 unit to pay for remaining work not related to cascading cut.

Delete Nonmin Element



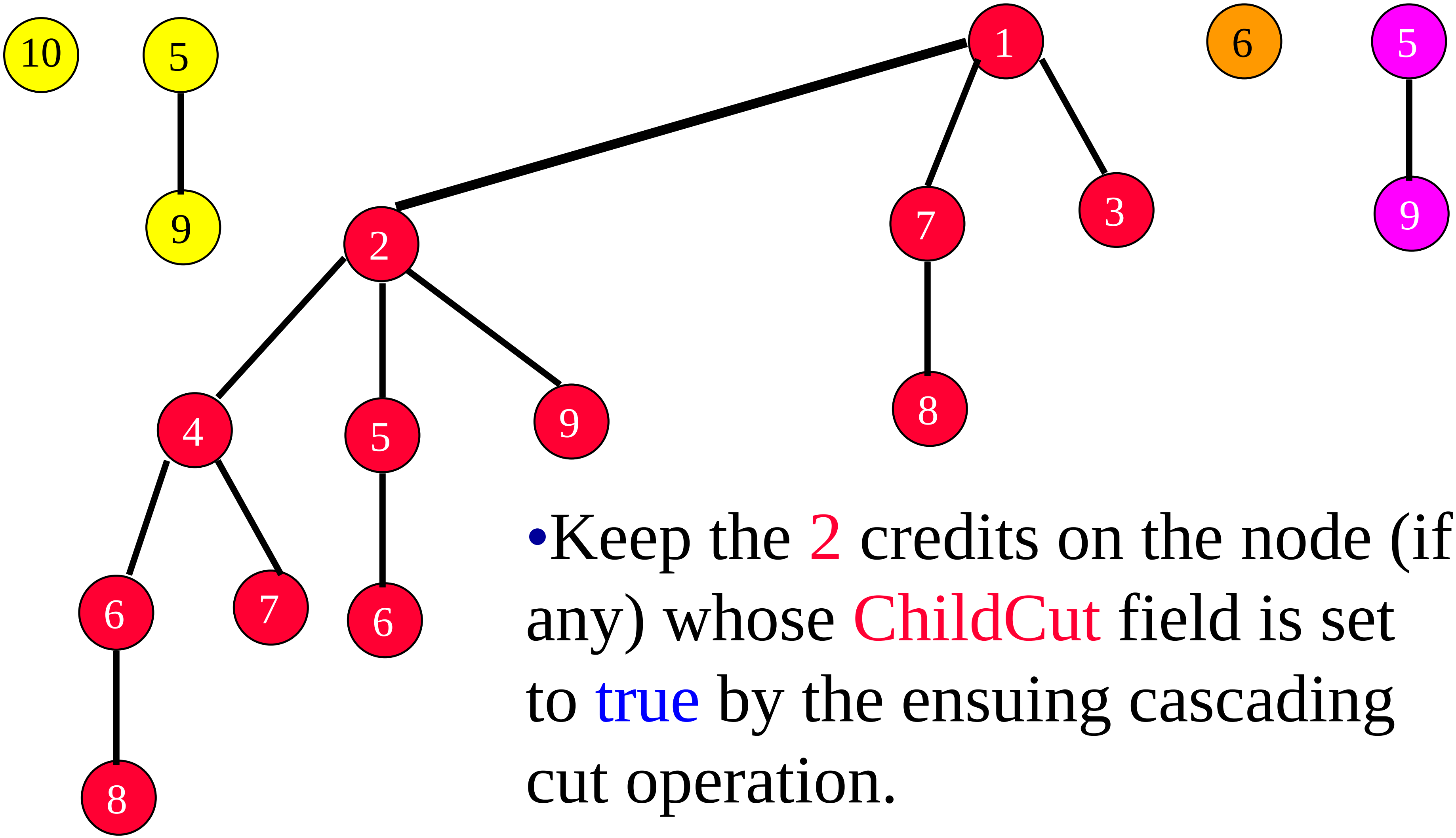
- Keep $\log_{\phi} n$ units to pay for possible future pairwise combining of the new top-level trees created.
- Kept as 1 credit per new top-level tree.
- Discard excess credits (if any).

Delete Nonmin Element



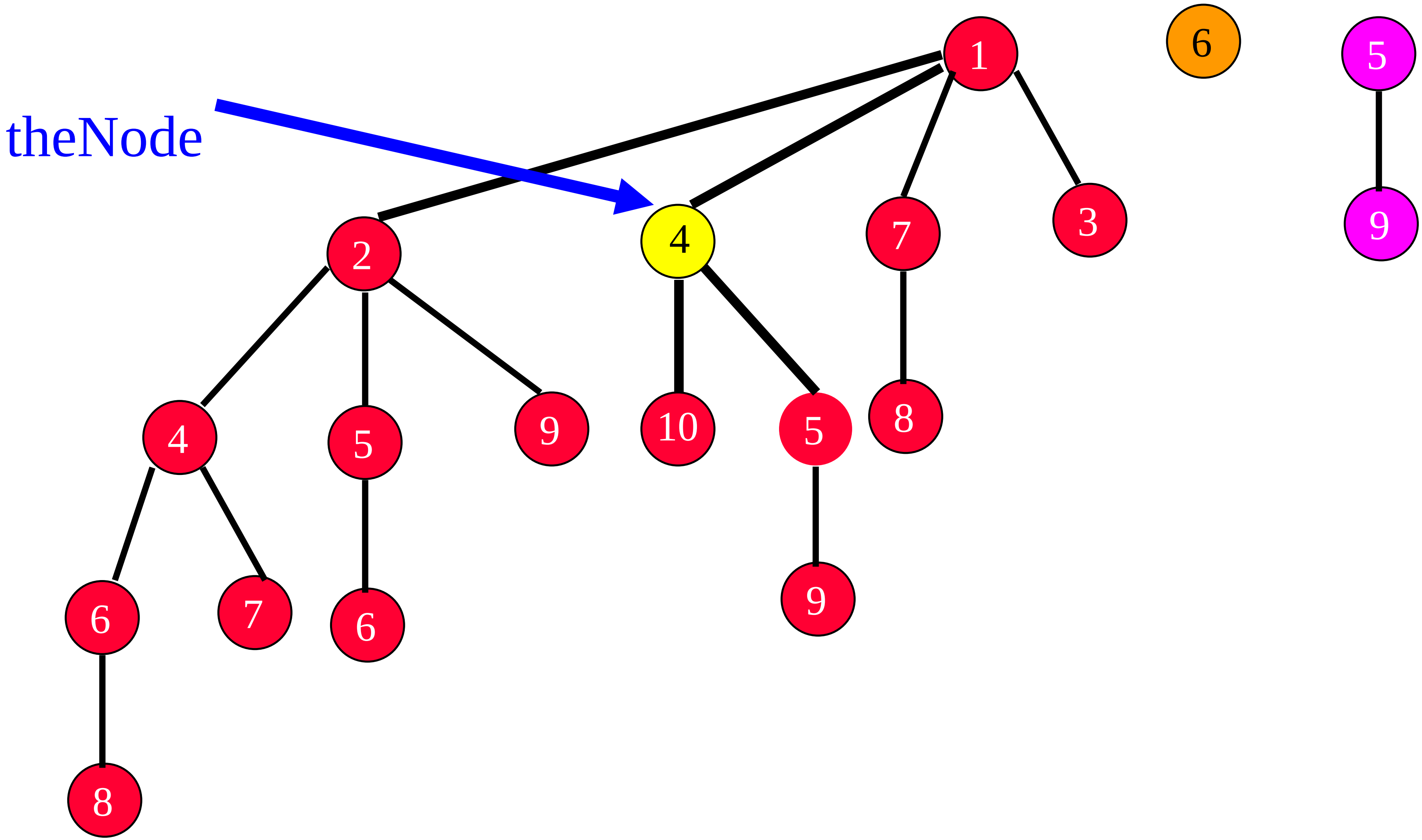
- Keep 1 unit to pay for the time when node whose ChildCut field is set to true is cut from its parent, and another 1 unit for the pairwise combining of the cut subtree.

Delete Nonmin Element

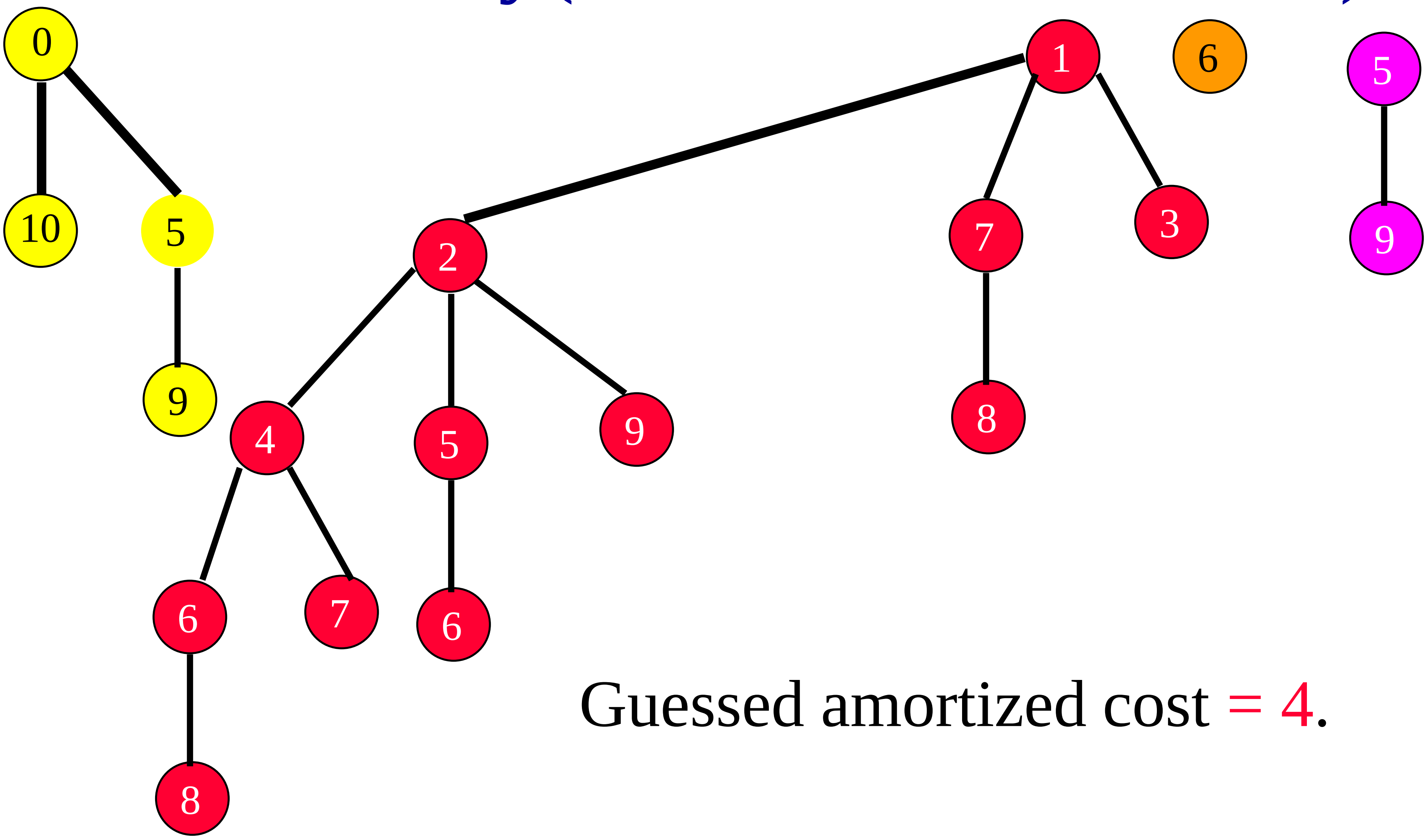


- Keep the 2 credits on the node (if any) whose ChildCut field is set to true by the ensuing cascading cut operation.
  - If there is no such node, discard the credits.

DecreaseKey(theNode, theAmount)

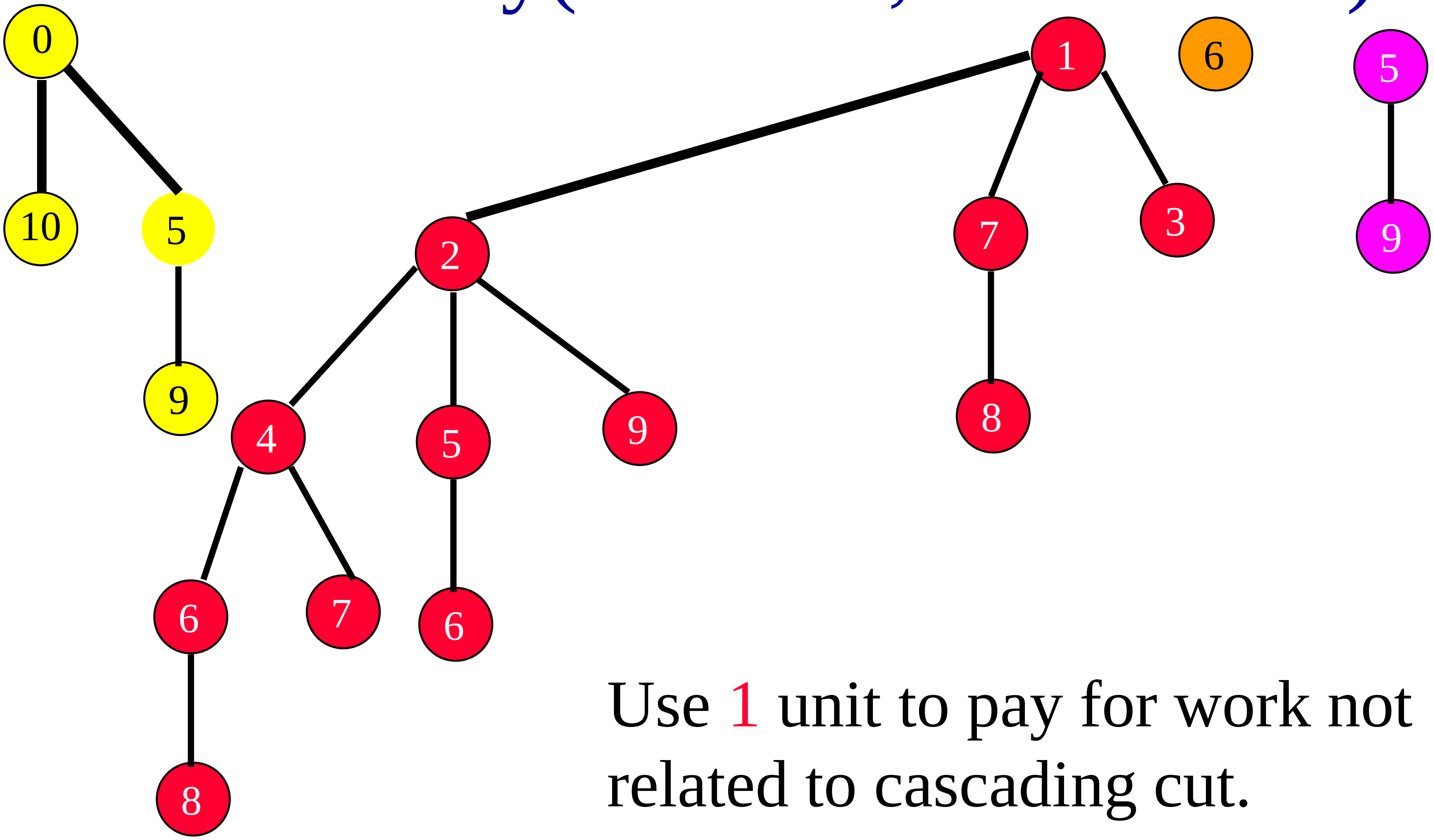


DecreaseKey(theNode, theAmount)



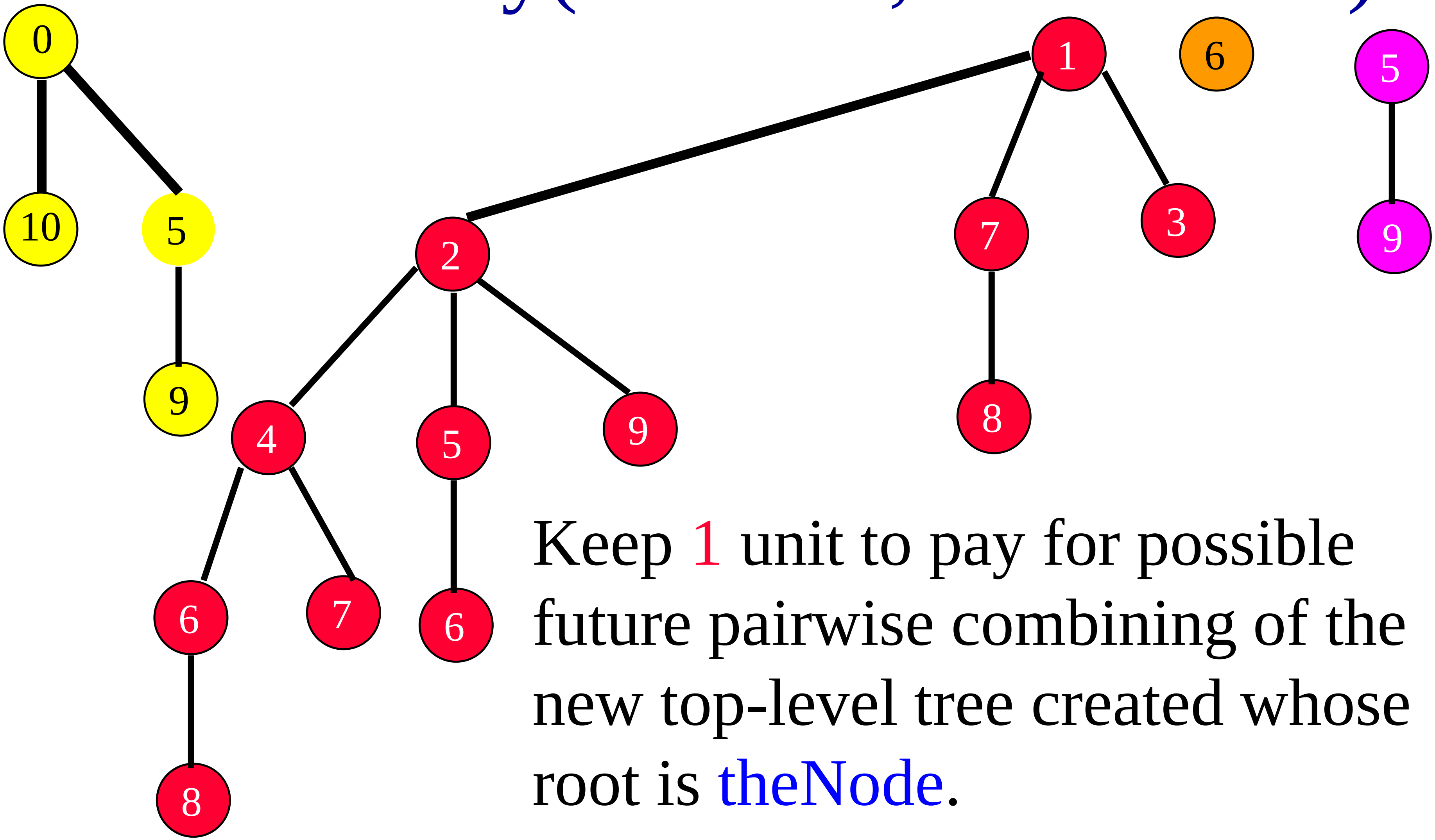
Guessed amortized cost = 4.

DecreaseKey(theNode, theAmount)



Use 1 unit to pay for work not related to cascading cut.

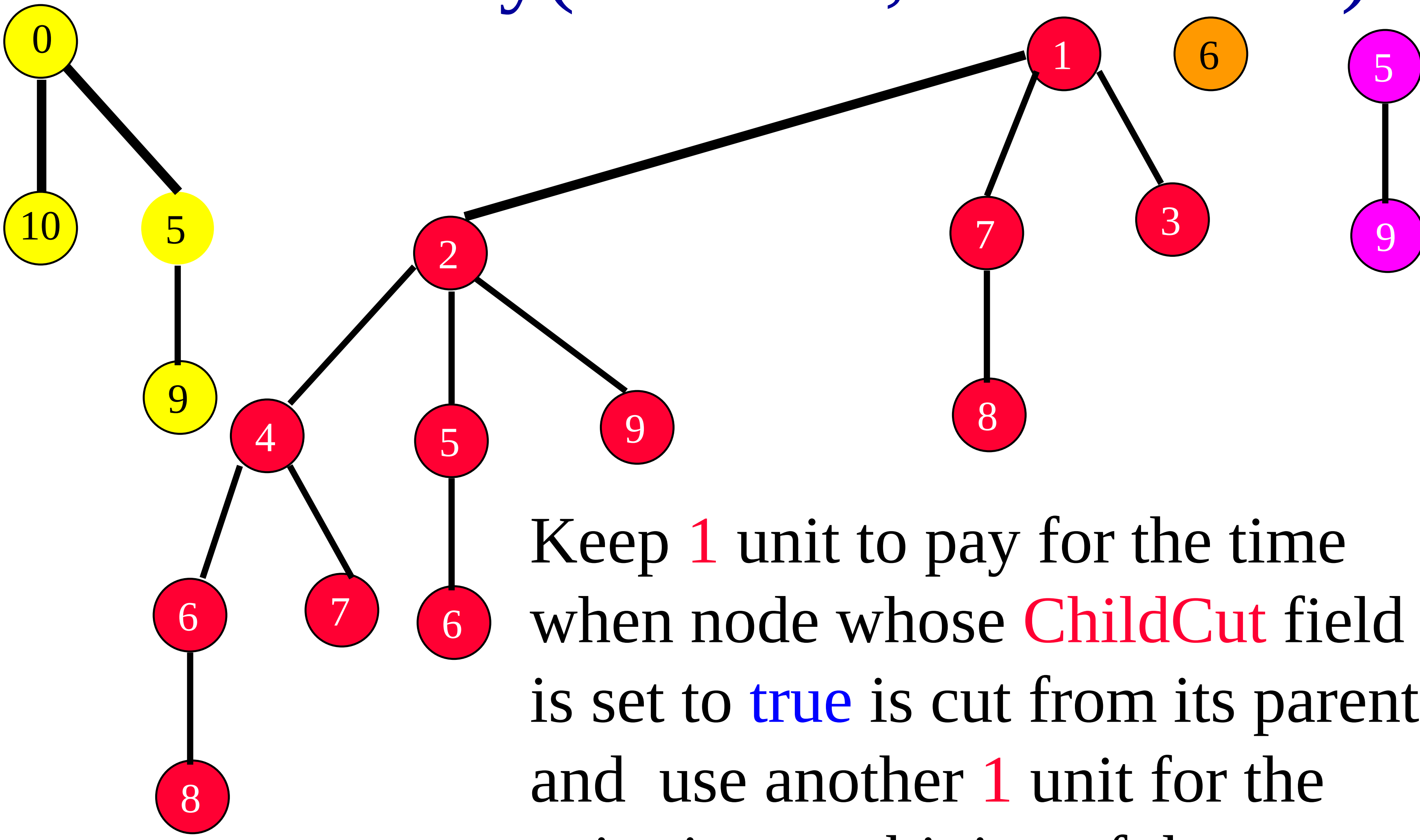
DecreaseKey(theNode, theAmount)



Keep 1 unit to pay for possible future pairwise combining of the new top-level tree created whose root is theNode.

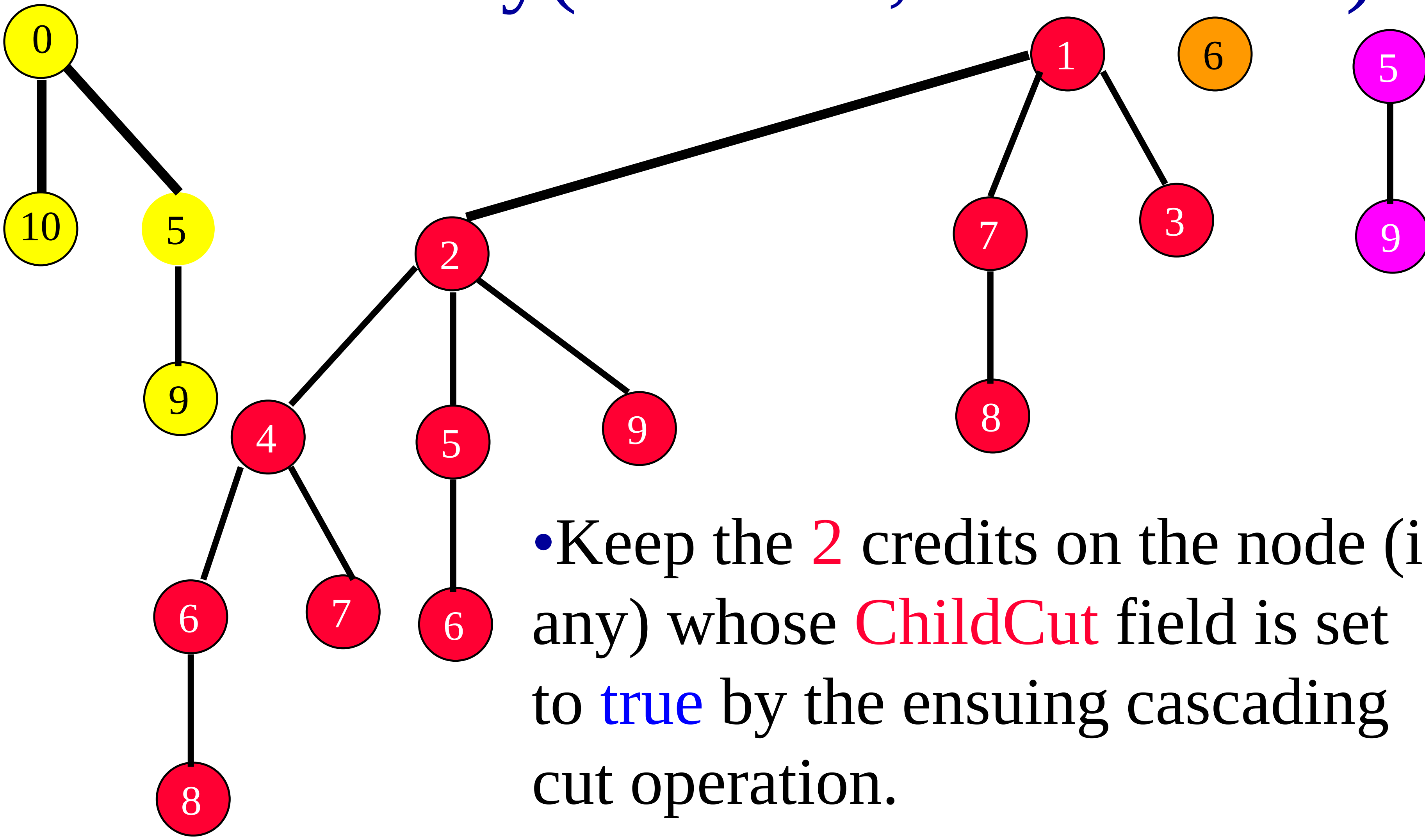
Kept as credit on theNode.

DecreaseKey(theNode, theAmount)



Keep 1 unit to pay for the time when node whose ChildCut field is set to true is cut from its parent, and use another 1 unit for the pairwise combining of the cut subtree.

DecreaseKey(theNode, theAmount)



- Keep the 2 credits on the node (if any) whose ChildCut field is set to true by the ensuing cascading cut operation.
  - If there is no such node, discard the credits.

Delete Min

- Guessed amortized cost $= 3\log_\phi n$.
- Actual cost $<= 2\log_\phi n - 1 + \#MinTrees$.
- Allocation of amortized cost.
  - Use $2\log_\phi n - 1$ to pay part of actual cost.
  - Keep remaining $\log_\phi n + 1$ as a credit to pay part of the actual cost of a future delete min operation.
  - Put 1 unit of credit on each of the at most $\log_\phi n + 1$ min trees left behind by the delete min operation.
  - Discard the remaining credits (if any).

Paying Actual Cost Of A Delete Min

- Actual cost $<= 2\log_\phi n - 1 + \#MinTrees$
- How is it paid for?
  - $2\log_\phi n - 1$ is paid for from the amortized cost of the delete min.
  - #MinTrees is paid by the 1 unit credit on each of the min trees in the Fibonacci heap just prior to the delete min operation.

Who Pays For Cascading Cut?

- Only nodes with ChildCut = true are cut during a cascading cut.
- The actual cost to cut a node is 1.
- This cost is paid from the 2 units of credit on the node whose ChildCut field is true. The remaining unit of credit is kept with the min tree that has been cut and now becomes a top-level tree.

Potential Method

- $P(i) = \Sigma[\#MinTrees(j) + 2*\#NodesWithTrueChildCut(j)]$
  - #MinTrees(j) is #MinTrees for Fibonacci heap j.
  - When Fibonacci heaps A and B are melded, A and B are no longer included in the sum.
- $P(0) = 0$
- $P(i) >= 0$ for all i.

Pairing Heaps

	Fibonacci	Pairing
Insert	O(1)	O(1)
Delete min (or max)	O(n)	O(n)
Meld	O(1)	O(1)
Delete	O(n)	O(n)
Decrease key (or increase)	O(n)	O(1)

Actual Complexity

Pairing Heaps

	Fibonacci	Pairing
Insert	O(1)	O(log n)
Delete min (or max)	O(log n)	O(log n)
Meld	O(1)	O(log n)
Delete	O(log n)	O(log n)
Decrease key (or increase)	O(1)	O(log n)

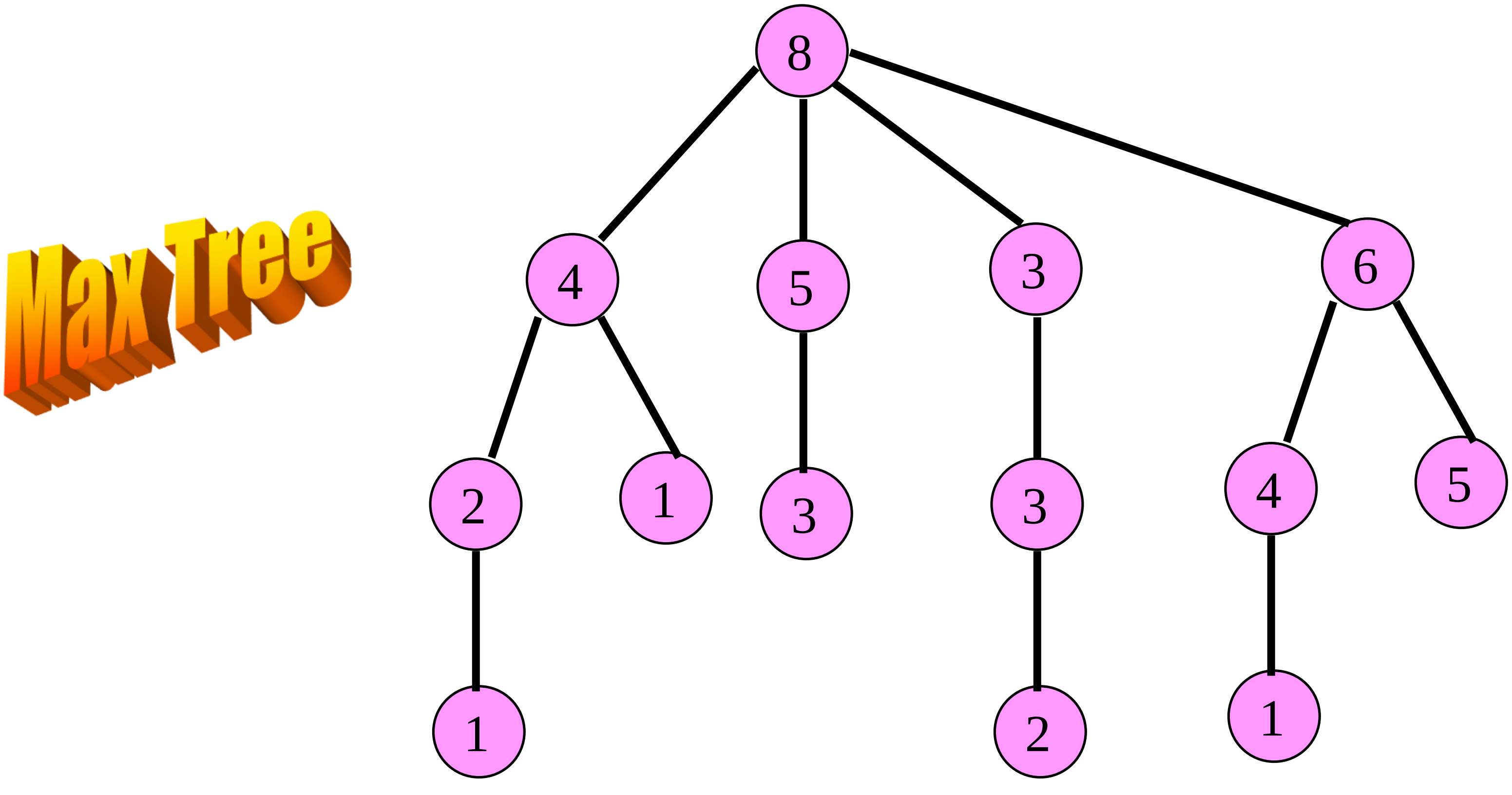
Amortized Complexity

Pairing Heaps

- Experimental results suggest that pairing heaps are actually faster than Fibonacci heaps.
  - Simpler to implement.
  - Smaller runtime overheads.
  - Less space per node.

Definition

- A min (max) pairing heap is a min (max) tree in which operations are done in a specified manner.

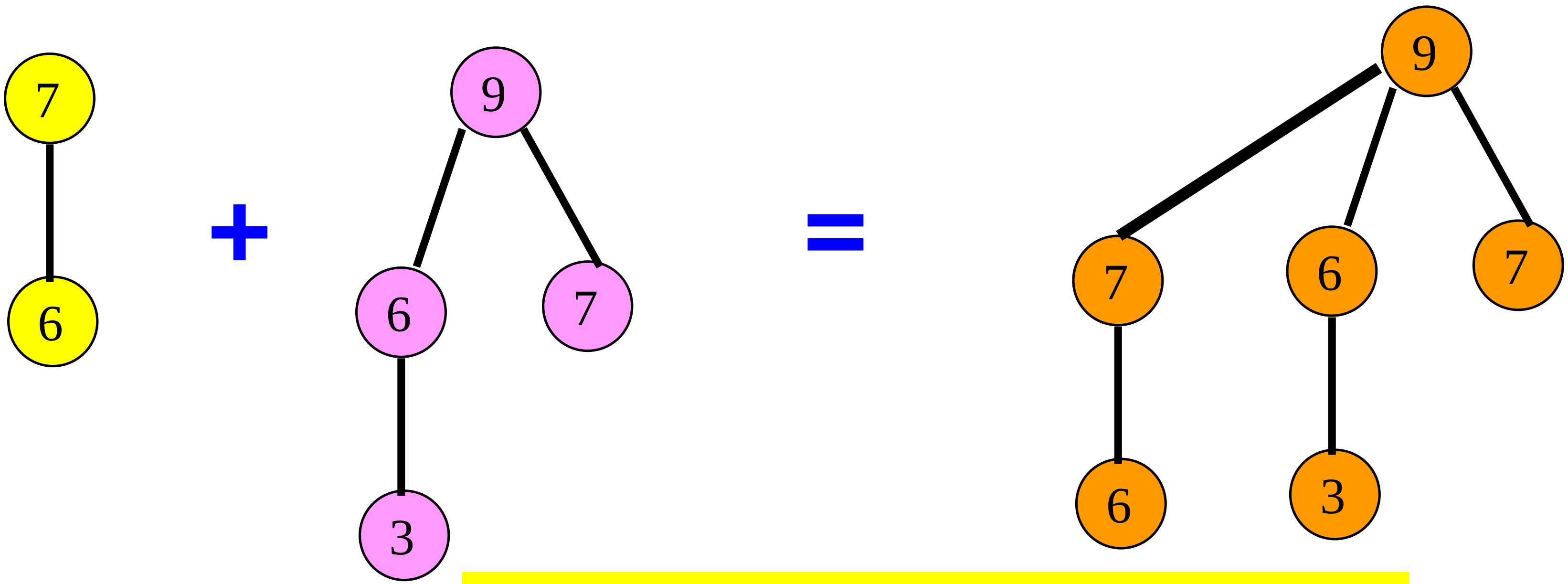


Node Structure

- Child
  - Pointer to first node of children list.
- Left and Right Sibling
  - Used for doubly linked linked list (not circular) of siblings.
  - Left pointer of first node is to parent.
  - x is first node in list iff x.left.child = x.
- Data
- Note: No Parent, Degree, or ChildCut fields.

Meld – Max Pairing Heap

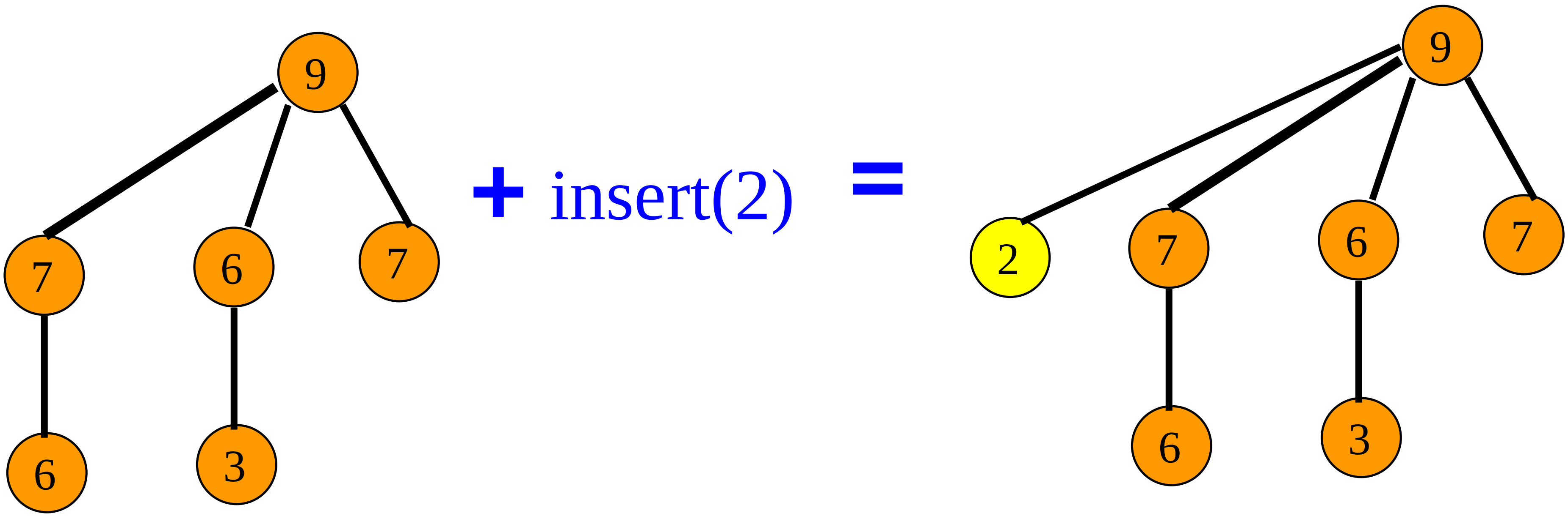
- Compare-Link Operation
  - Compare roots.
  - Tree with smaller root becomes leftmost subtree.



- Actual cost = O(1).

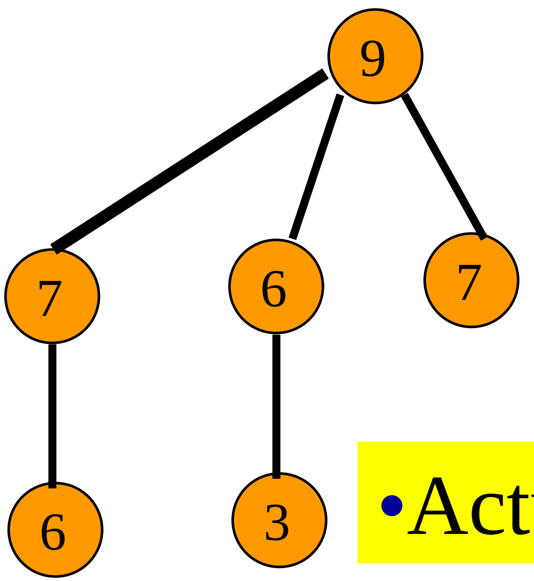
Insert

- Create 1-element max tree with new item and meld with existing max pairing heap.

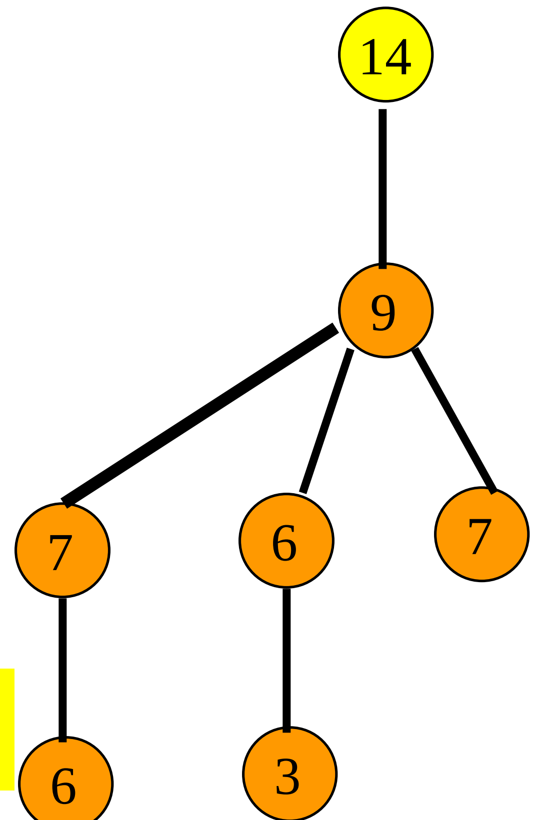


Insert

- Create 1-element max tree with new item and meld with existing max pairing heap.



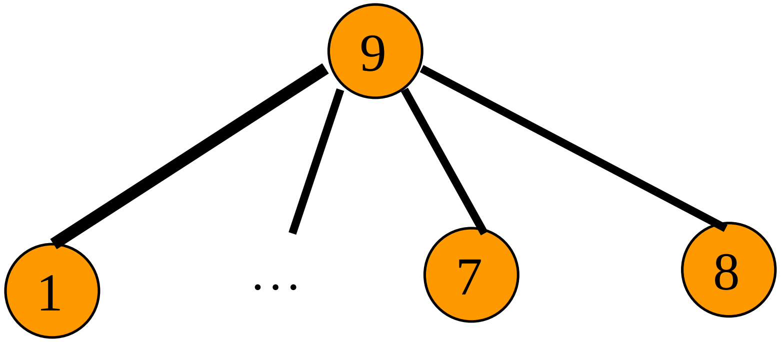
+ insert(14) **=**



- Actual cost = O(1).

Worst-Case Degree

- Insert 9, 8, 7, …, 1, in this order.



•Worst-case degree = n –1.

Worst-Case Height

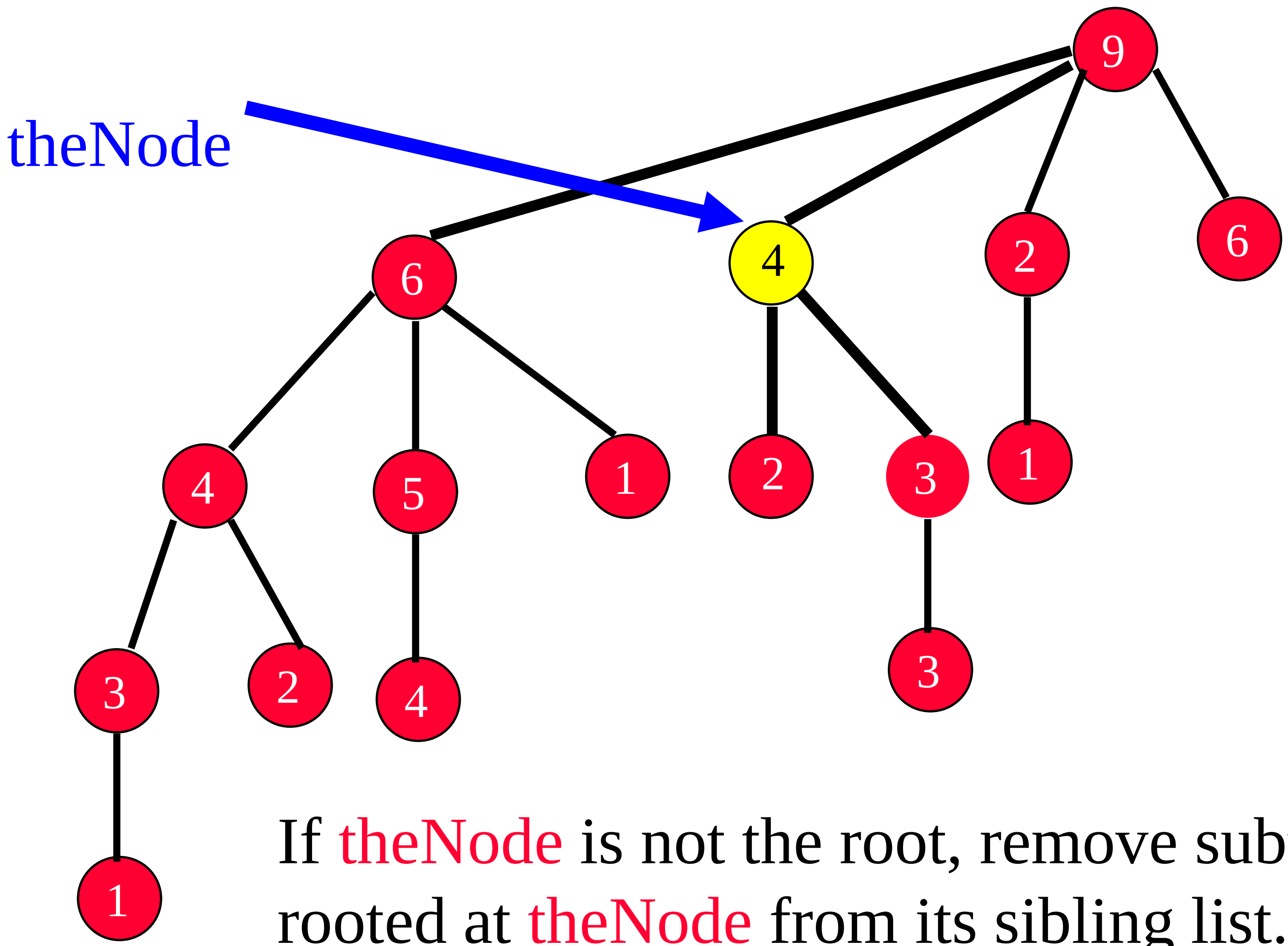
- Insert 1, 2, 3, …, n, in this order.

- Worst-case height = n.

IncreaseKey(theNode, theAmount)

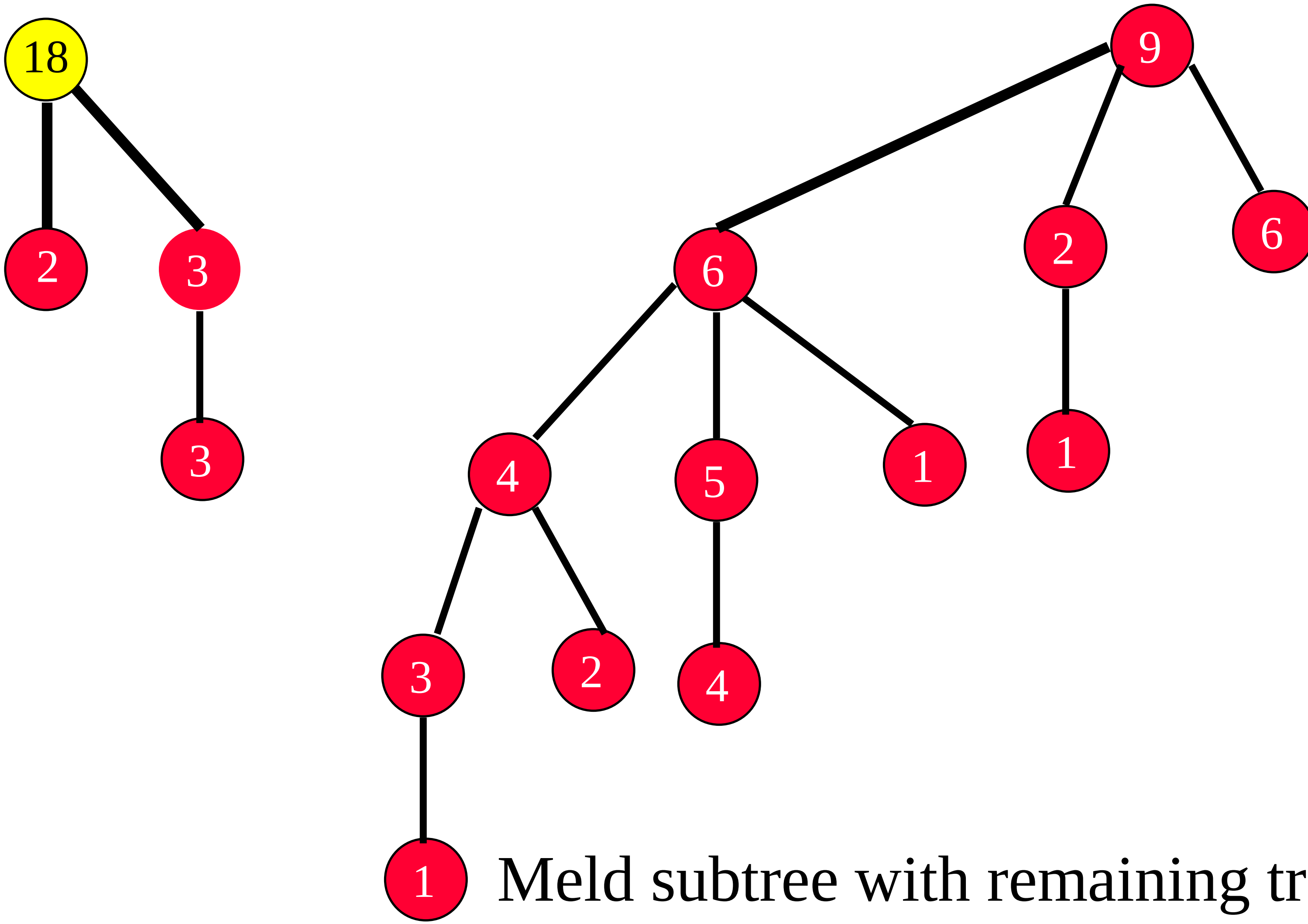
- Since nodes do not have parent fields, we cannot easily check whether the key in theNode becomes larger than that in its parent.
- So, detach theNode from sibling doubly-linked list and meld.

IncreaseKey(theNode, theAmount)



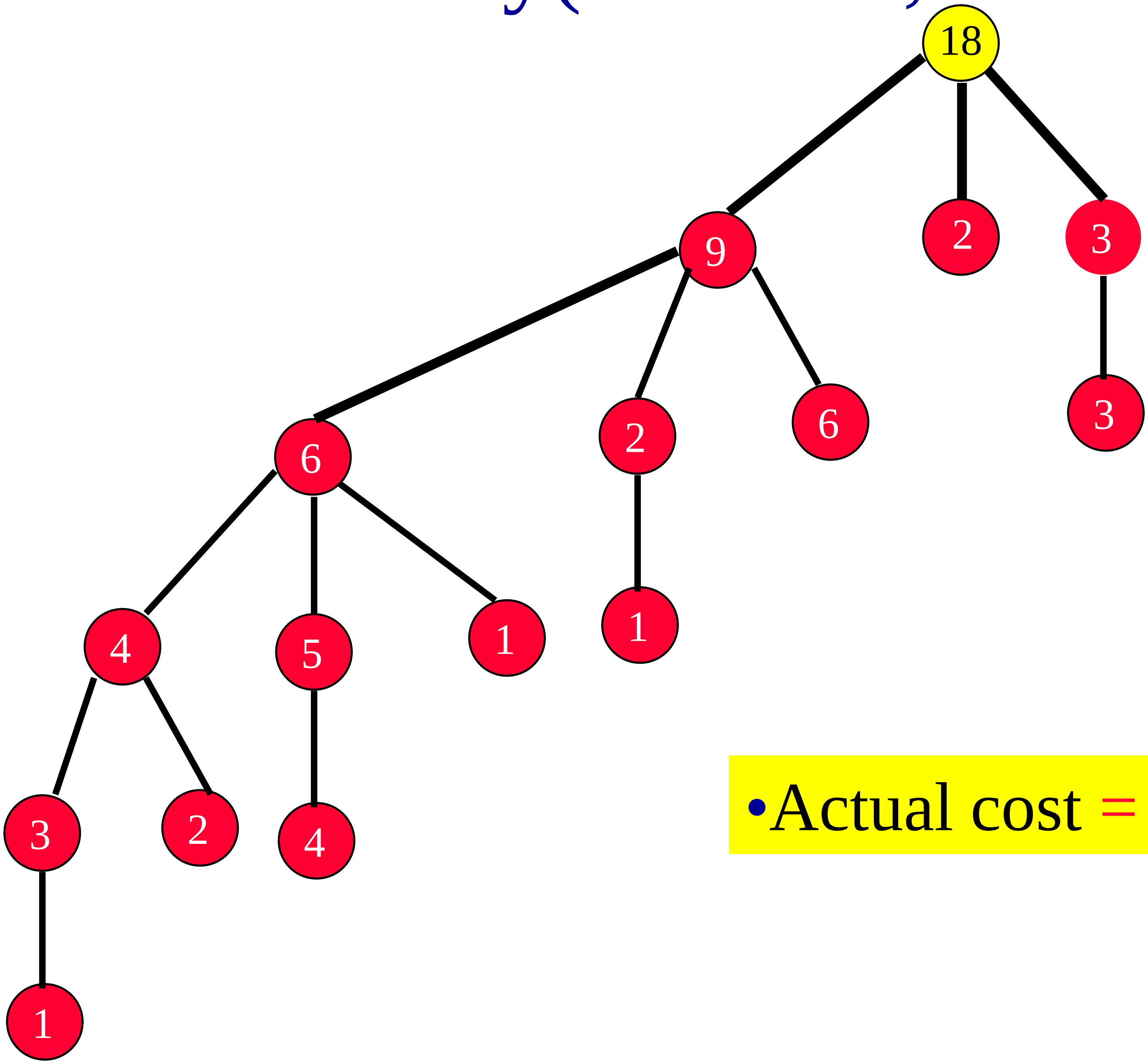
If theNode is not the root, remove subtree rooted at theNode from its sibling list.

IncreaseKey(theNode, theAmount)



Meld subtree with remaining tree.

IncreaseKey(theNode, theAmount)



- Actual cost = O(1).

Delete Max

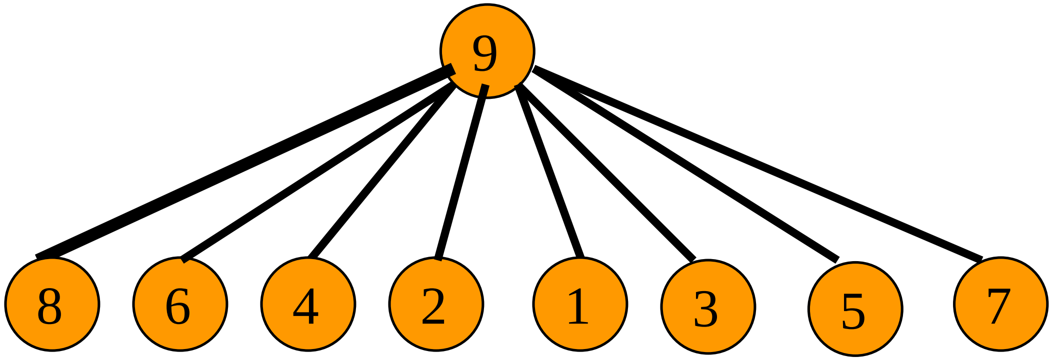
- If empty => fail.
- Otherwise, remove tree root and meld subtrees into a single max tree.
- How to meld subtrees?
  - Good way => O(log n) amortized complexity for remove max.
  - Bad way => O(n) amortized complexity.

Bad Way To Meld Subtrees

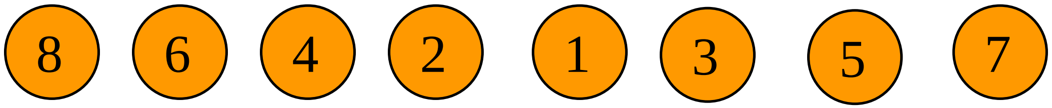
- currentTree = first subtree.
- for (each of the remaining trees)

  currentTree = compareLink(currentTree,
                                        nextTree);

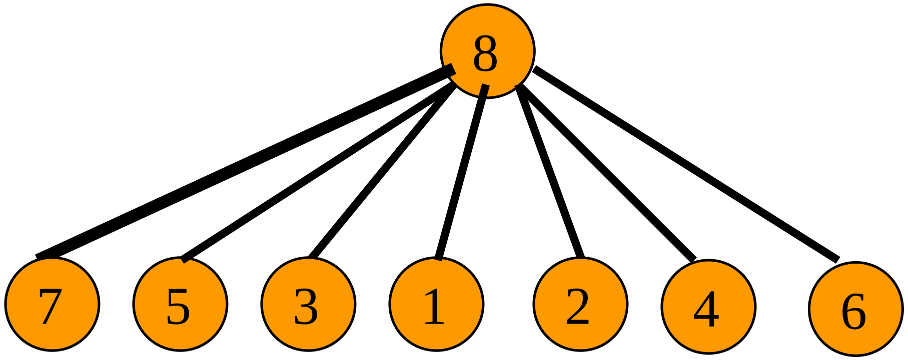
Example



- Delete max.



- Meld into one tree.



Example

- Actual cost of insert is 1.
- Actual cost of delete max is degree of root.
- n/2 inserts (9, 7, 5, 3, 1, 2, 4, 6, 8) followed by n/2 delete maxs.
    - Cost of inserts is n/2.
    - Cost of delete maxs is $1 + 2 + \ldots + n/2 - 1 = \Theta(n^2)$.
    - If amortized cost of an insert is $O(1)$, amortized cost of a delete max must be $\Theta(n)$.

Good Ways To Meld Subtrees

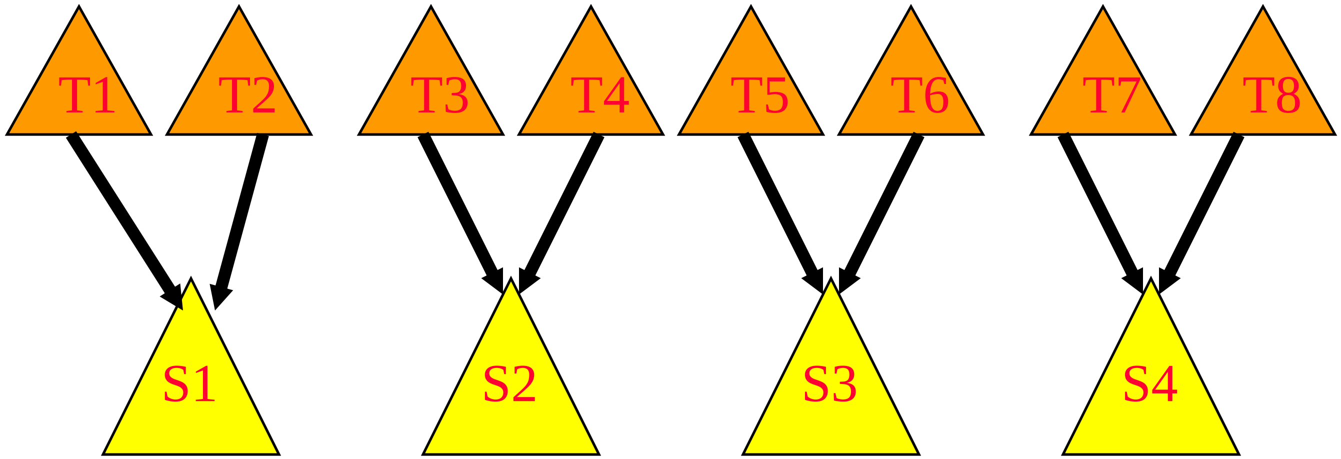
- Two-pass scheme.
- Multipass scheme.
- Both have same asymptotic complexity.
- Two-pass scheme gives better observed performance.

Two-Pass Scheme

- Pass 1.
  - Examine subtrees from left to right.
  - Meld pairs of subtrees, reducing the number of subtrees to half the original number.
  - If # subtrees was odd, meld remaining original subtree with last newly generated subtree.
- Pass 2.
  - Start with rightmost subtree of Pass 1. Call this the working tree.
  - Meld remaining subtrees, one at a time, from right to left, into the working tree.

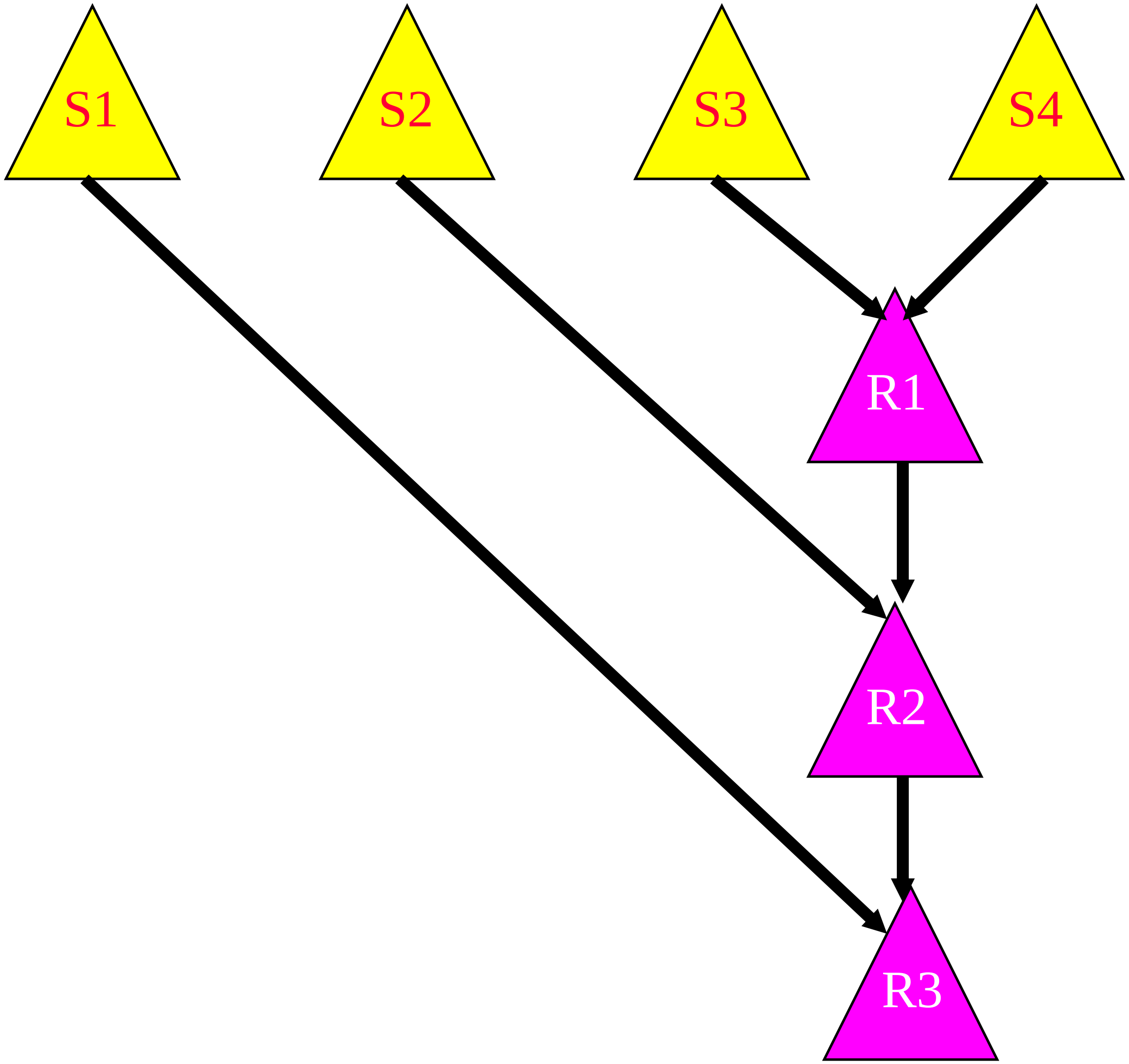
Two-Pass Scheme – Example

Pass 1



Two-Pass Scheme – Example

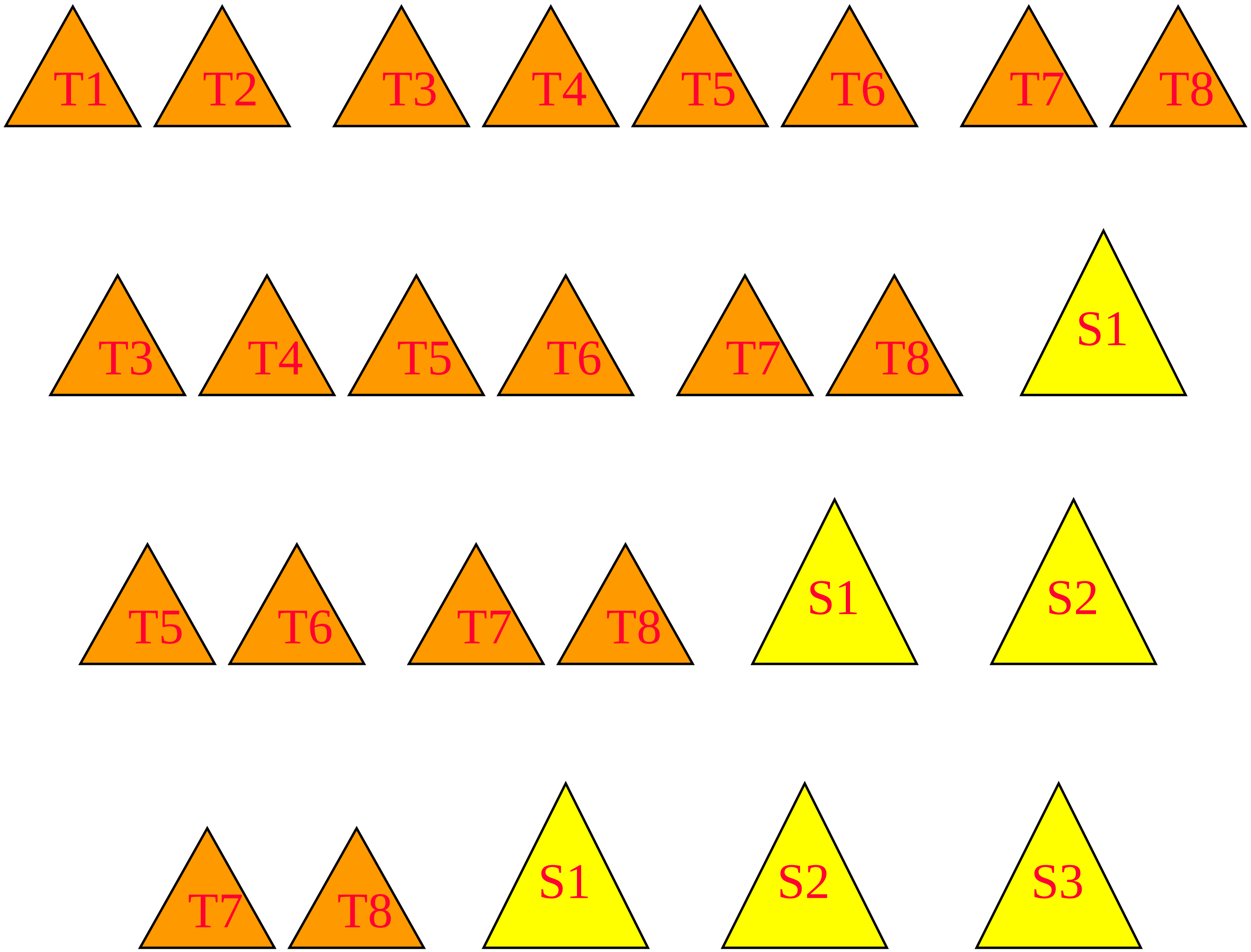
Pass 2



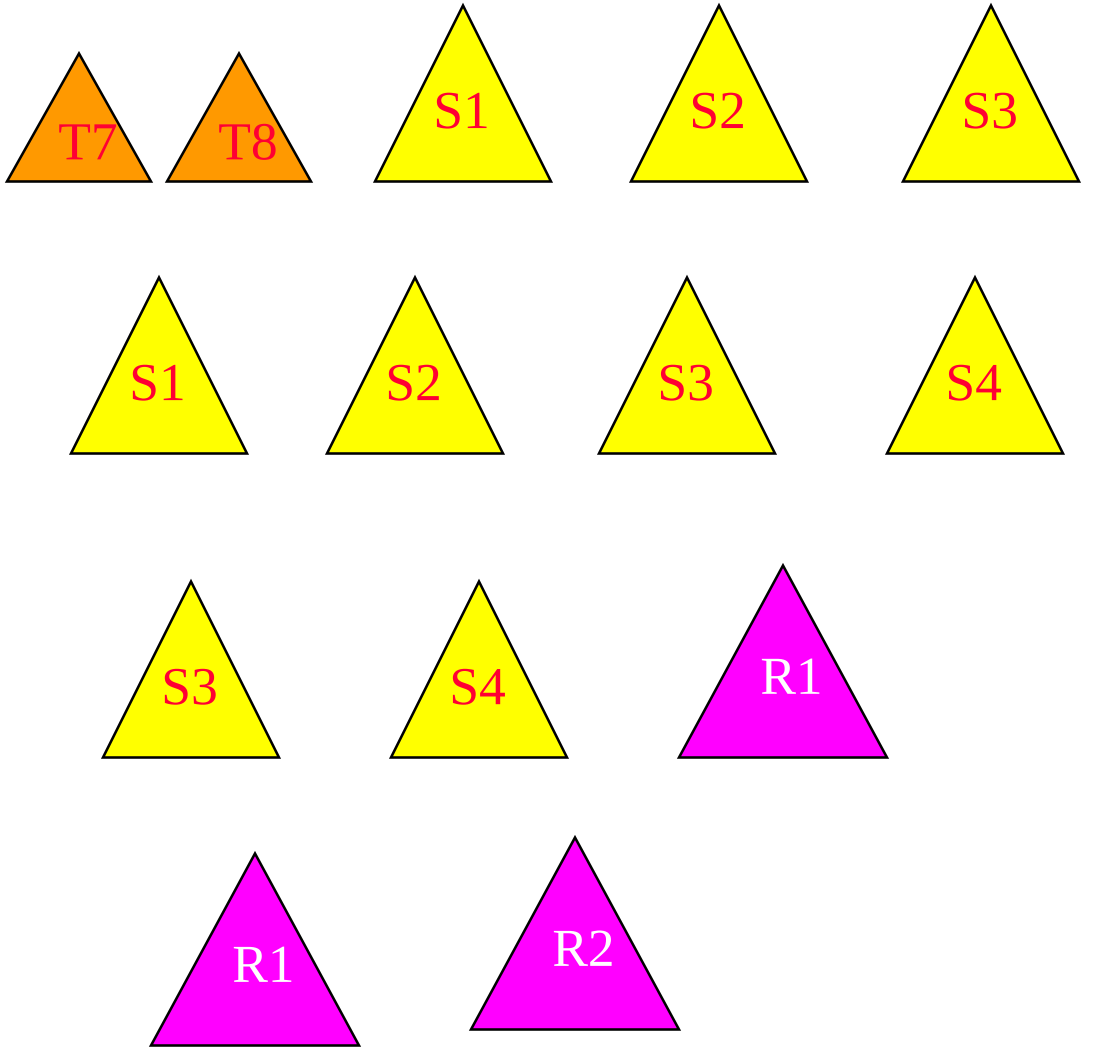
Multipass Scheme

- Place the subtrees into a FIFO queue.
- Repeat until 1 tree remains.
  - Remove 2 subtrees from the queue.
  - Meld them.
  - Put the resulting tree onto the queue.

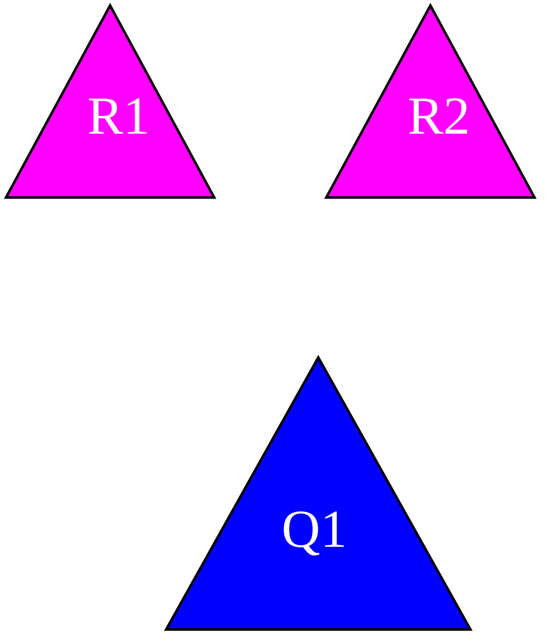
Multipass Scheme – Example



Multipass Scheme--Example



Multipass Scheme--Example

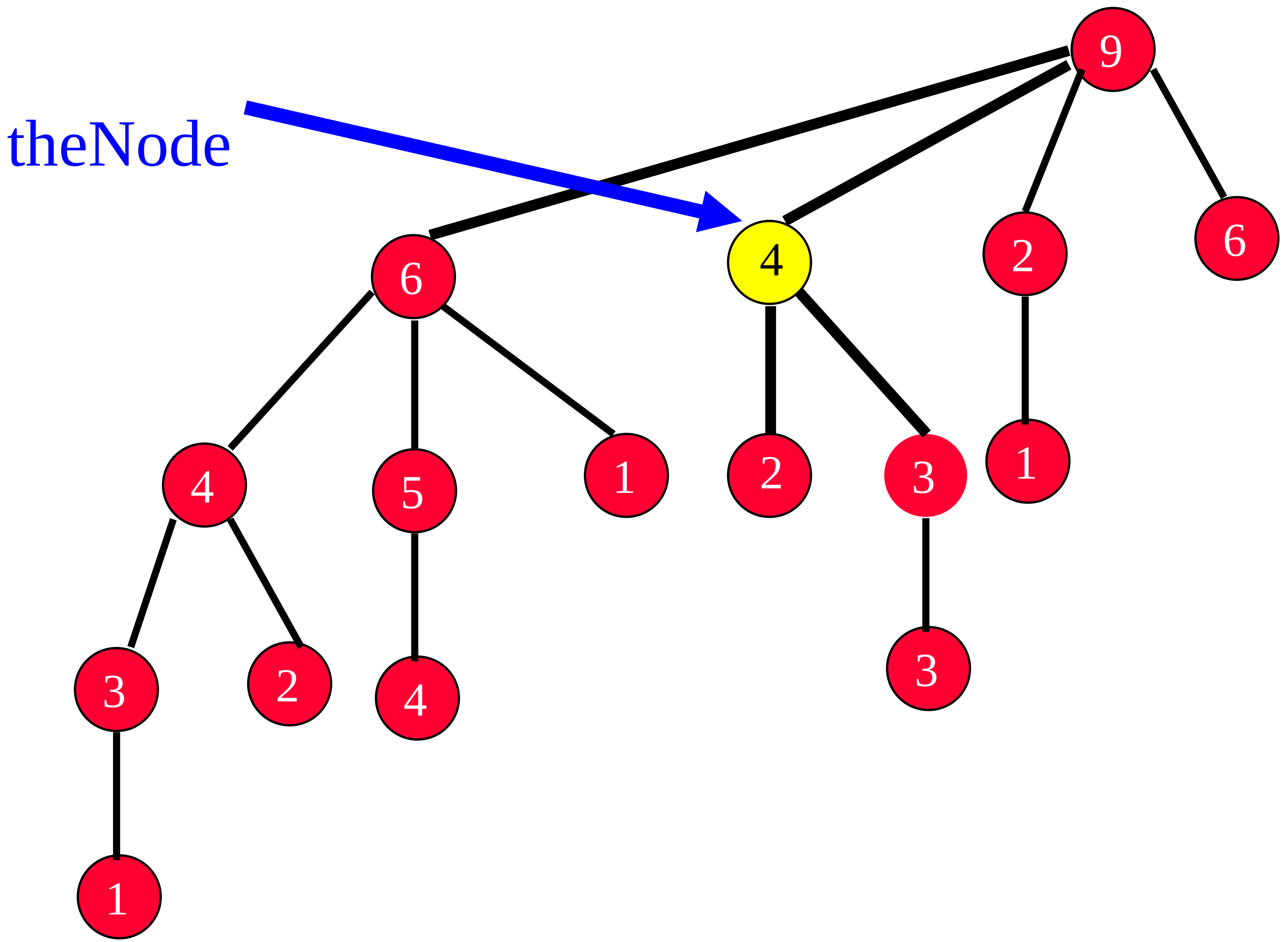


- Actual cost = O(n).

Delete Nonroot Element

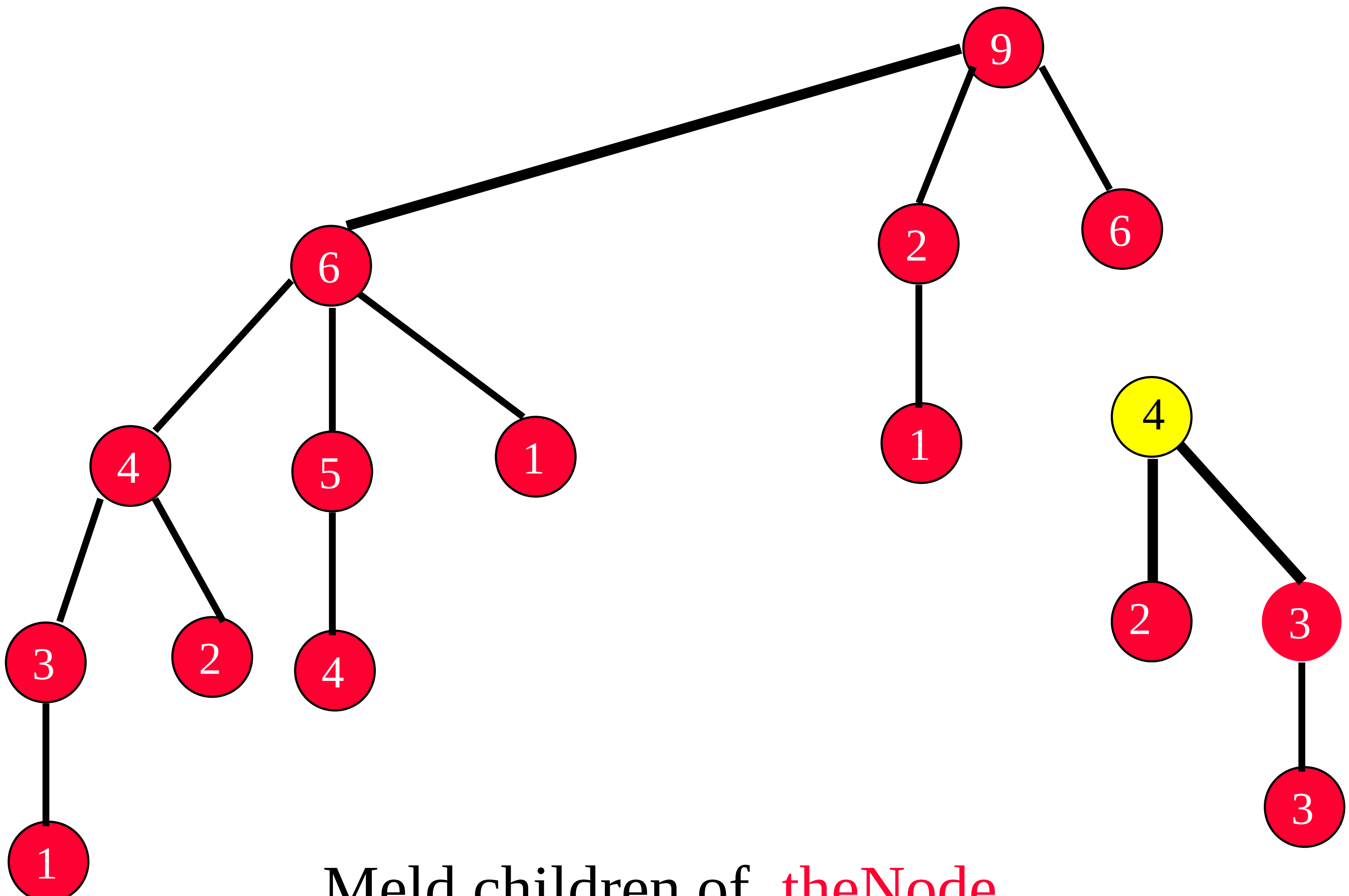
- Remove theNode from its sibling list.
- Meld children of theNode using either 2-pass or multipass scheme.
- Meld resulting tree with what's left of original tree.

Delete(theNode)



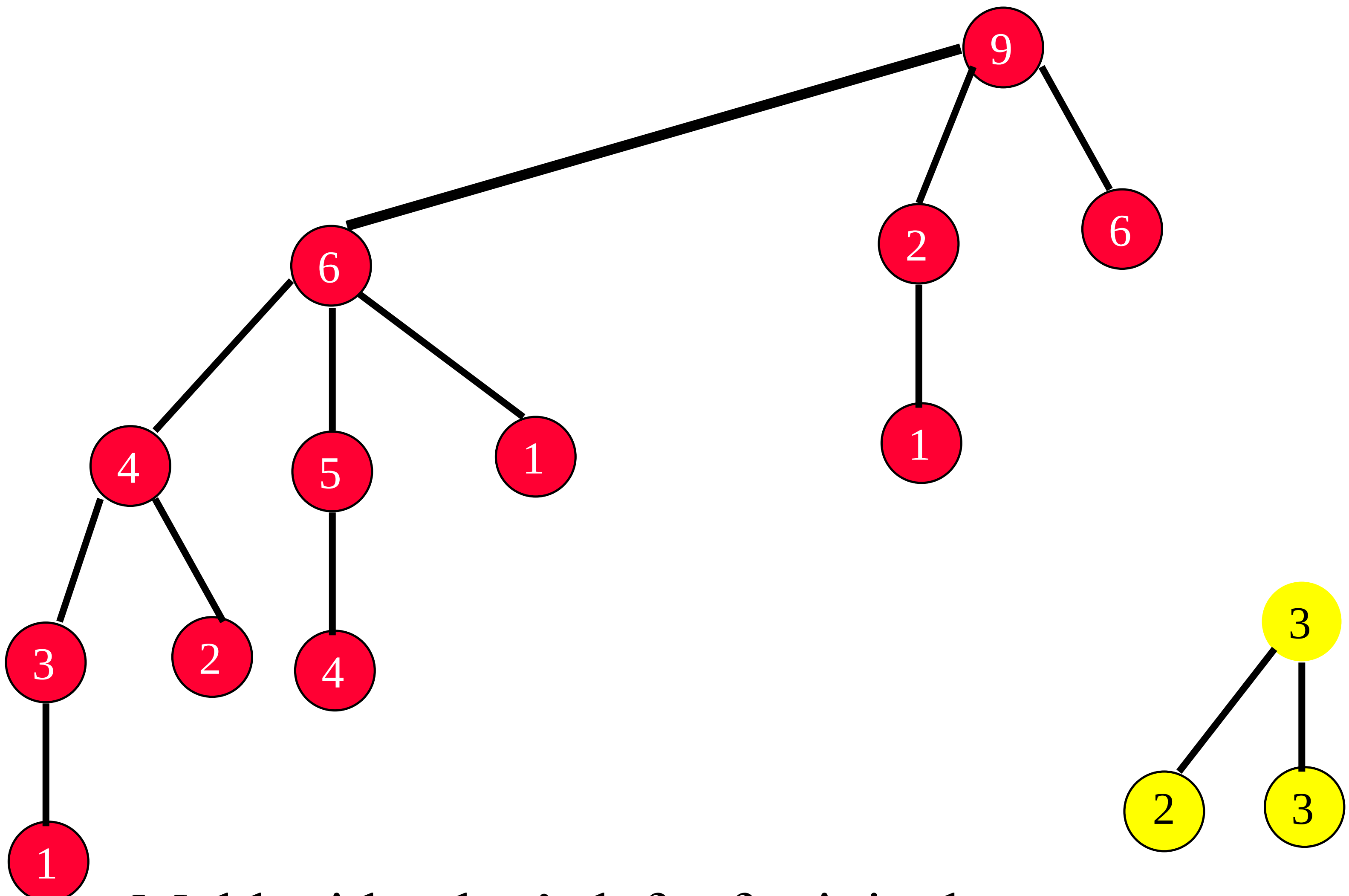
Remove theNode from its doubly-linked sibling list.

Delete(theNode)



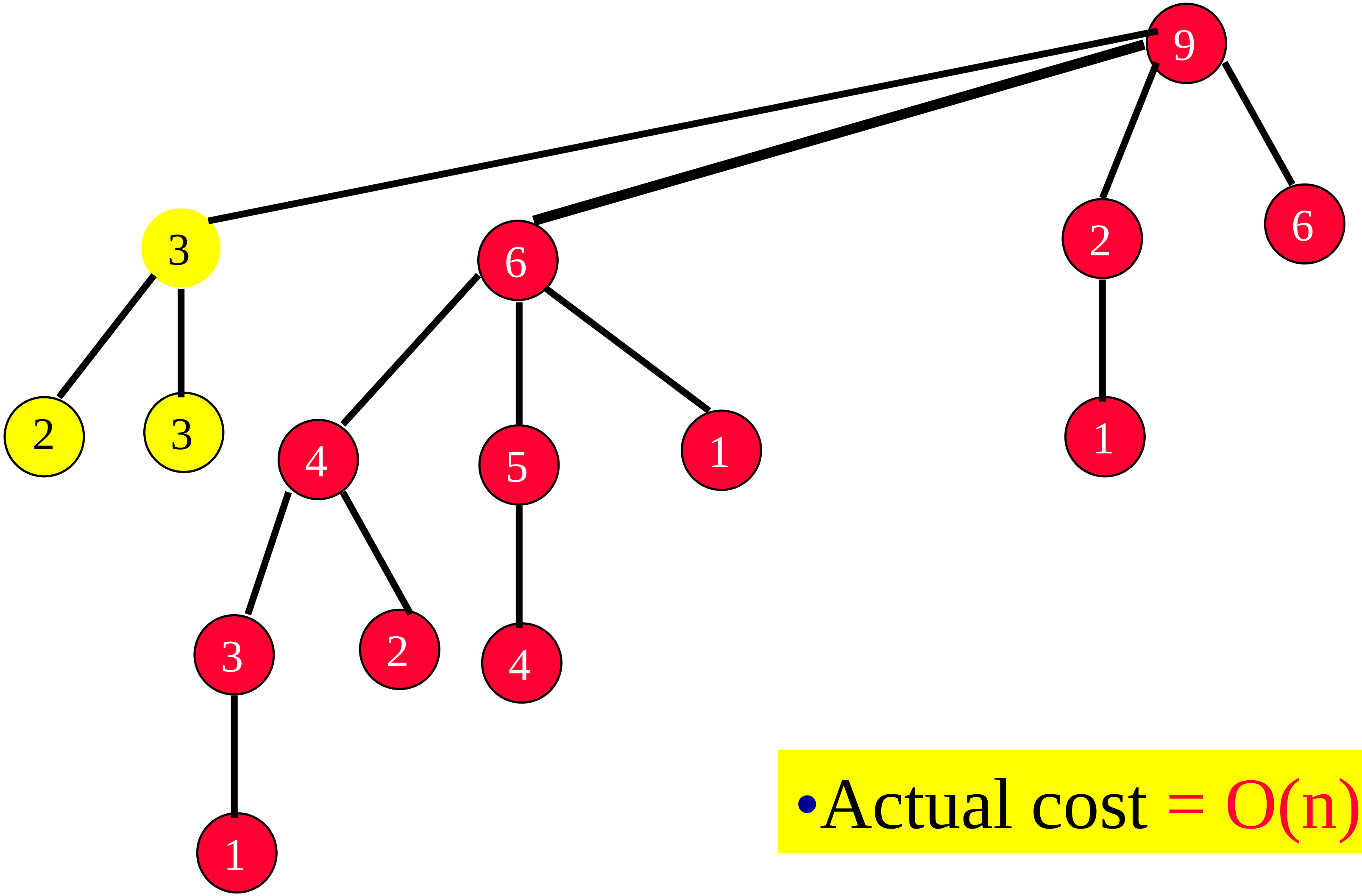
Meld children of theNode.

Delete(theNode)



Meld with what's left of original tree.

Delete(theNode)



- Actual cost = O(n).