Welcome To . . .

Advanced Data Structures

Sartaj Sahni

Clip Art Sources

- www.barrysclipart.com
- www.livinggraphics.com
- www.rad.kumc.edu
- www.livinggraphics.com

What The Course Is About

- Study data structures for:
  - External sorting
  - Single and double ended priority queues
  - Dictionaries
  - Multidimensional search
  - Computational geometry
  - Image processing
  - Packet routing and classification
  - …

What The Course Is About

- Concerned with:
  - Worst-case complexity
  - Average complexity
  - Amortized complexity

Prerequisites

- ✓ C, C++, Java, or Python
- ✓ Asymptotic Complexity
  - Big Oh, Theta, and Omega notations
- ✓ Undergraduate data structures
  - Stacks and Queues
  - Linked lists
  - Trees
  - Graphs

Web Site

- www.cise.ufl.edu/~sahni/cop5536
- http://elearning.ufl.edu
- Handouts, syllabus, readings, assignments, past exams, past exam solutions, TAs, Internet lectures, PowerPoint presentations, etc.

Assignments, Tests, & Grades

- 25% for assignments
  - There will be two assignments.
- 25% for each test
  - There will be three tests.

Grades (Rough Cutoffs)

- A >= 85%
- A- >= 81%
- B+ >= 77%
- B >= 72%
- B- >= 67%
- C+ >= 63%
- C >= 60%
- C- >= 55%

Kinds Of Complexity

- ✓ Worst-case complexity.
- ✓ Average complexity.
- Amortized complexity.

Data Structure Z

- Operations
  - Initialize
  - Insert
  - Delete
- Examples
  - Linear List
  - Stack
  - Queue
  - …

Data Structure Z

- Suppose that the worst-case complexity is
  - Initialize O(1)
  - Insert O(s)
  - Delete O(s)

  where s is the size of Z.
- How much time does it take to perform a sequence of 1 initialize followed by n inserts and deletes?
- $O(n^2)$

Data Structure Z

- Suppose further that the average complexity is
  - Initialize $O(1)$
  - Insert $O(\log s)$
  - Delete $O(\log s)$
- How much time does it take to perform a sequence of 1 initialize followed by n inserts and deletes?
- $O(n^2)$

An Application P

- Initialize Z
- Solve P by performing many inserts and deletes plus other tasks.
- Examples
  - Dijkstra's single-source shortest paths
  - Minimum cost spanning trees

An Application P

- Total time to solve P using Z is

time for inserts/deletes + time for other tasks

$= O(n^2)$ + time for other tasks

where n is the number of inserts/deletes

At times a better bound can be obtained using amortized complexity.

Amortized Complexity

- The amortized complexity of a task is the amount you charge the task.
- The conventional way to bound the cost of doing a task n times is to use one of the expressions
  - n*(worst-case cost of task)
  - Σ(worst-case cost of task i)
- The amortized complexity way to bound the cost of doing a task n times is to use one of the expressions
  - n*(amortized cost of task)
  - Σ(amortized cost of task i)

Amortized Complexity

- The amortized complexity of a task may bear no direct relationship to the actual complexity of the task. I.e., it may be <, =, or > actual task complexity.

Amortized Complexity

- In worst-case complexity analysis, each task is charged an amount that is >= its cost. So,

  Σ(actual cost of task i)

  <= Σ(worst-case cost of task i)

- In amortized analysis, some tasks may be charged an amount that is < their cost. The amount charged must ensure:

  Σ(actual cost of task i)

  <= Σ(amortized cost of task i)

Potential Function P()

- $P(i) = amortizedCost(i) - actualCost(i) + P(i - 1)$
- $\Sigma(P(i) - P(i-1)) =$

  $\Sigma(amortizedCost(i) - actualCost(i))$
- $P(n) - P(0) = \Sigma(amortizedCost(i) - actualCost(i))$
- $P(n) - P(0) >= 0$
- When $P(0) = 0$, $P(i)$ is the amount by which the first i tasks/operations have been over charged.

Arithmetic Statements

- Rewrite an arithmetic statement as a sequence of statements that do not use parentheses.
- a = x+((a+b)*c+d)+y;

  is equivalent to the sequence:

  z1 = a+b;

  z2 = z1*c+d;

  a = x+z2+y;

Arithmetic Statements

a = x+((a+b)*c+d)+y;

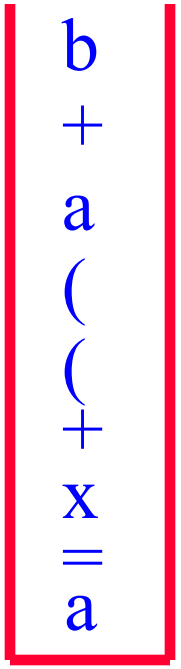
- The rewriting is done using a stack and a method processNextSymbol.
- create an empty stack;

```
for (int i = 1; i <= n; i++)
    // n is number of symbols in statement
    processNextSymbol();
```

Arithmetic Statements

a = x+((a+b)*c+d)+y;

- processNextSymbol extracts the next symbol from the input statement.
- Symbols other than) and ; are simply pushed on to the stack.

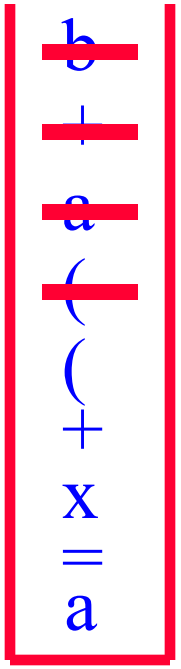


Arithmetic Statements

a = x+((a+b)*c+d)+y;

- If the next symbol is), symbols are popped from the stack up to and including the first (, an assignment statement is generated, and the left hand symbol is added to the stack.

z1 = a+b;

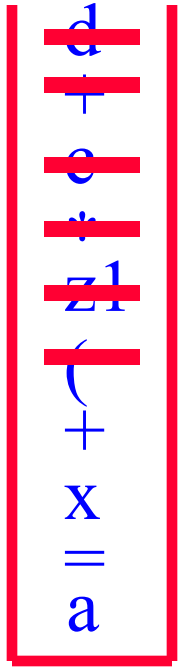


Arithmetic Statements

a = x+((a+b)*c+d)+y;

- If the next symbol is), symbols are popped from the stack up to and including the first (, an assignment statement is generated, and the left hand symbol is added to the stack.

z1 = a+b;
z2 = z1*c+d;

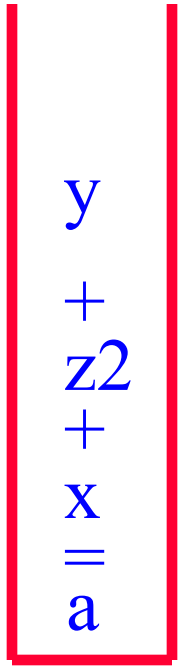


Arithmetic Statements

a = x+((a+b)*c+d)+y;

- If the next symbol is), symbols are popped from the stack up to and including the first (, an assignment statement is generated, and the left hand symbol is added to the stack.

z1 = a+b;
z2 = z1*c+d;

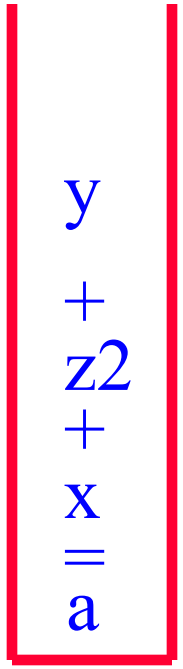


Arithmetic Statements

a = x+((a+b)*c+d)+y;

- If the next symbol is ;, symbols are popped from the stack until the stack becomes empty. The final assignment statement
  a = x+z2+y;
  is generated.

z1 = a+b;
z2 = z1*c+d;



Complexity Of processNextSymbol

a = x+((a+b)*c+d)+y;

- O(number of symbols that get popped from stack)
- O(i), where i is for loop index.

Overall Complexity (Conventional Analysis)

```
create an empty stack;
 for (int i = 1; i <= n; i++)
     // n is number of symbols in statement
     processNextSymbol();
```

- So, overall complexity is $O(\Sigma i) = O(n^2)$.
- Alternatively, $O(n*n) = O(n^2)$.
- Although correct, a more careful analysis permits us to conclude that the complexity is $O(n)$.

Ways To Determine Amortized Complexity

- Aggregate method.
- Accounting method.
- Potential function method.

Aggregate Method

- Somehow obtain a good upper bound on the actual cost of the n invocations of processNextSymbol()
- Divide this bound by n to get the amortized cost of one invocation of processNextSymbol()
- Easy to see that

  Σ(actual cost) <= Σ(amortized cost)

Aggregate Method

- The actual cost of the n invocations of processNextSymbol()

  equals number of stack pop and push operations.
- The n invocations cause at most n symbols to be pushed on to the stack.
- This count includes the symbols for new variables, because each new variable is the result of a) being processed. Note that no)s get pushed on to the stack.

Aggregate Method

- The actual cost of the n invocations of processNextSymbol() is at most 2n.
- So, using 2n/n = 2 as the amortized cost of processNextSymbol() is OK, because this cost results in Σ(actual cost) <= Σ(amortized cost)
- Since the amortized cost of processNextSymbol() is 2, the actual cost of all n invocations is at most 2n.

Aggregate Method

- The aggregate method isn't very useful, because to figure out the amortized cost we must first obtain a good bound on the aggregate cost of a sequence of invocations.
- Since our objective was to use amortized complexity to get a better bound on the cost of a sequence of invocations, if we can obtain this better bound through other techniques, we can omit dividing the bound by n to obtain the amortized cost.

Amortized Complexity

- ✓ Aggregate method.
- Accounting method.
- Potential function method.

Potential Function P()

- P(i) = amortizedCost(i) – actualCost(i) + P(i – 1)
- Σ(P(i) – P(i – 1)) =

  Σ(amortizedCost(i) –actualCost(i))
- P(n) – P(0) = Σ(amortizedCost(i) –actualCost(i))
- P(n) – P(0) >= 0
- When P(0) = 0, P(i) is the amount by which the first i operations have been over charged.

Potential Function Example

	a = x + ((a + b) * c + d) + y ;
actual cost	1 1 1 1 1 11 1 1 5 1 1 1 1 7 1 1 7
amortized cost	2 2 2 2 2 22 2 2 2 2 2 2 2 2 2 2 2
potential	1 2 3 4 5 67 8 9 6 7 8 9 10 5 6 7 2

Potential = stack size except at end.

Accounting Method

- Guess the amortized cost.
- Show that $P(n) - P(0) >= 0$.

Accounting Method Example

```
create an empty stack;
for (int i = 1; i <= n; i++)
    // n is number of symbols in statement
    processNextSymbol();
```

- Guess that amortized complexity of processNextSymbol is 2.
- Start with P(0) = 0.
- Can show that P(i) >= number of elements on stack after ith symbol is processed.

Accounting Method Example

	a = x + ((a + b) * c + d) + y ;
actual cost	1 1 1 1 1 1 1 1 5 1 1 1 1 7 1 1 7
amortized cost	2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
potential	1 2 3 4 5 6 7 8 9 6 7 8 9 10 5 6 7 2

- Potential >= number of symbols on stack.
- Therefore, P(i) >= 0 for all i.
- In particular, P(n) >= 0.

Potential Method

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

Potential Method Example

```
create an empty stack;
for (int i = 1; i <= n; i++)
    // n is number of symbols in statement
    processNextSymbol();
```

- Guess that the potential function is P(i) = number of elements on stack after i^{th} symbol is processed (exception is P(n) = 2).
- P(0) = 0 and P(i) – P(0) >= 0 for all i.

i^{th} Symbol Is Not) or ;

- Actual cost of processNextSymbol is 1.
- Number of elements on stack increases by 1.
- $\Delta P = P(i) - P(i-1) = 1$.
- amortized cost = actual cost + ΔP

  $= 1 + 1 = 2$

i^{th} Symbol Is)

- Actual cost of processNextSymbol is #pops + 1.
- Number of elements on stack decreases by #pops –1.
- $\Delta P = P(i) - P(i-1) = 1 - \#pops$.
- amortized cost = actual cost + ΔP

  $= \#pops + 1 + (1 - \#pops)$

  $= 2$

i^{th} Symbol Is ;

- Actual cost of processNextSymbol is

  #pops = P(n–1).
- Number of elements on stack decreases by P(n–1).
- $\Delta P = P(n) - P(n-1) = 2 - P(n-1)$.
- amortized cost = actual cost + ΔP

  $= P(n-1) + (2 - P(n-1))$

  $= 2$

Binary Counter

- n-bit counter
- Cost of incrementing counter is number of bits that change.
- Cost of 001011 => 001100 is 3.
- Counter starts at 0.
- What is the cost of incrementing the counter m times (m <= 2^n-1)?

Worst-Case Method

- Worst-case cost of an increment is n.
- Cost of 011111 => 100000 is 6.
- So, the cost of m increments is at most mn.

Aggregate Method

0 0 0 0 0

counter

- Each increment changes bit 0 (i.e., the right most bit).
- Exactly floor(m/2) increments change bit 1 (i.e., second bit from right).
- Exactly floor(m/4) increments change bit 2.

Aggregate Method

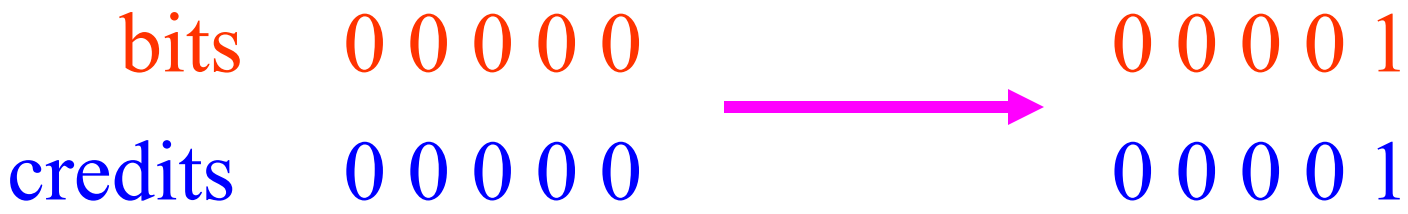
0 0 0 0 0

counter

- Exactly floor(m/8) increments change bit 3.
- So, the cost of m increments is
  m + floor(m/2) + floor(m/4) + ... < 2m
- Amortized cost of an increment is 2m/m = 2.

Accounting Method

- Guess that the amortized cost of an increment is 2.
- Now show that P(m) – P(0) >= 0 for all m.
- 1st increment:
  - one unit of amortized cost is used to pay for the change in bit 0 from 0 to 1.
  - the other unit remains as a credit on bit 0 and is used later to pay for the time when bit 0 changes from 1 to 0.



2nd Increment.

bits	0	0	0	0	1	→	0	0	0	1	0
credits	0	0	0	0	1		0	0	0	1	0

- one unit of amortized cost is used to pay for the change in bit 1 from 0 to 1
- the other unit remains as a credit on bit 1 and is used later to pay for the time when bit 1 changes from 1 to 0
- the change in bit 0 from 1 to 0 is paid for by the credit on bit 0

3rd Increment.

bits	0	0	0	1	0	→	0	0	0	1	1
credits	0	0	0	1	0		0	0	0	1	1

- one unit of amortized cost is used to pay for the change in bit 0 from 0 to 1
- the other unit remains as a credit on bit 0 and is used later to pay for the time when bit 1 changes from 1 to 0

4^{th} Increment.

bits	0	0	0	1	1	→	0	0	1	0	0
credits	0	0	0	1	1		0	0	1	0	0

- one unit of amortized cost is used to pay for the change in bit 2 from 0 to 1
- the other unit remains as a credit on bit 2 and is used later to pay for the time when bit 2 changes from 1 to 0
- the change in bits 0 and 1 from 1 to 0 is paid for by the credits on these bits

Accounting Method

- $P(m) - P(0) = \Sigma(\text{amortizedCost}(i) - \text{actualCost}(i))$
  $= $ amount by which the first m increments have been over charged
  $= $ number of credits
  $= $ number of 1s in binary rep. of m
  $>= 0$

Potential Method

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

Potential Method

- Guess P(i) = number of 1s in counter after ith increment.
- P(i) >= 0 and P(0) = 0.
- Let q = # of 1s at right end of counter just before ith increment (01001111 => q = 4).
- Actual cost of ith increment is 1+q.
- $\Delta P = P(i) - P(i-1) = 1 - q$ (0100111 => 0101000)
- amortized cost = actual cost + ΔP

$$= 1+q+(1-q) = 2$$

External Sorting

- Sort n records/elements that reside on a disk.
- Space needed by the n records is very large.
  - n is very large, and each record may be large or small.
  - n is small, but each record is very large.
- So, not feasible to input the n records, sort, and output in sorted order.

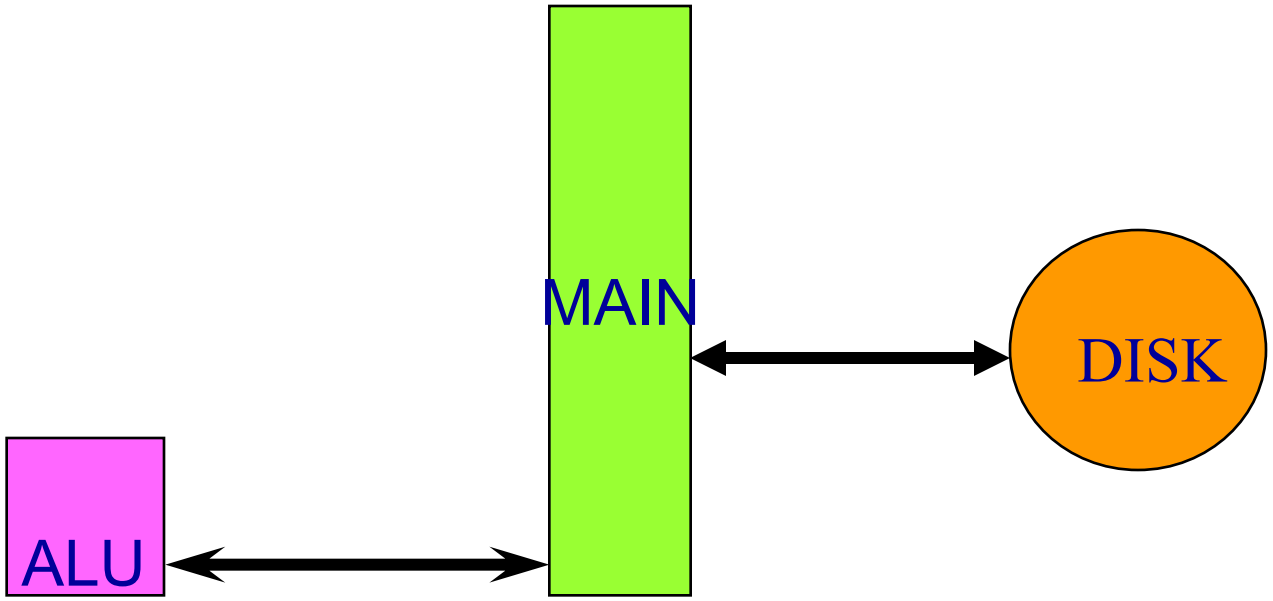
Small n But Large File

- Input the record keys.
- Sort the n keys to determine the sorted order for the n records.
- Permute the records into the desired order (possibly several fields at a time).
- We focus on the case: large n, large file.

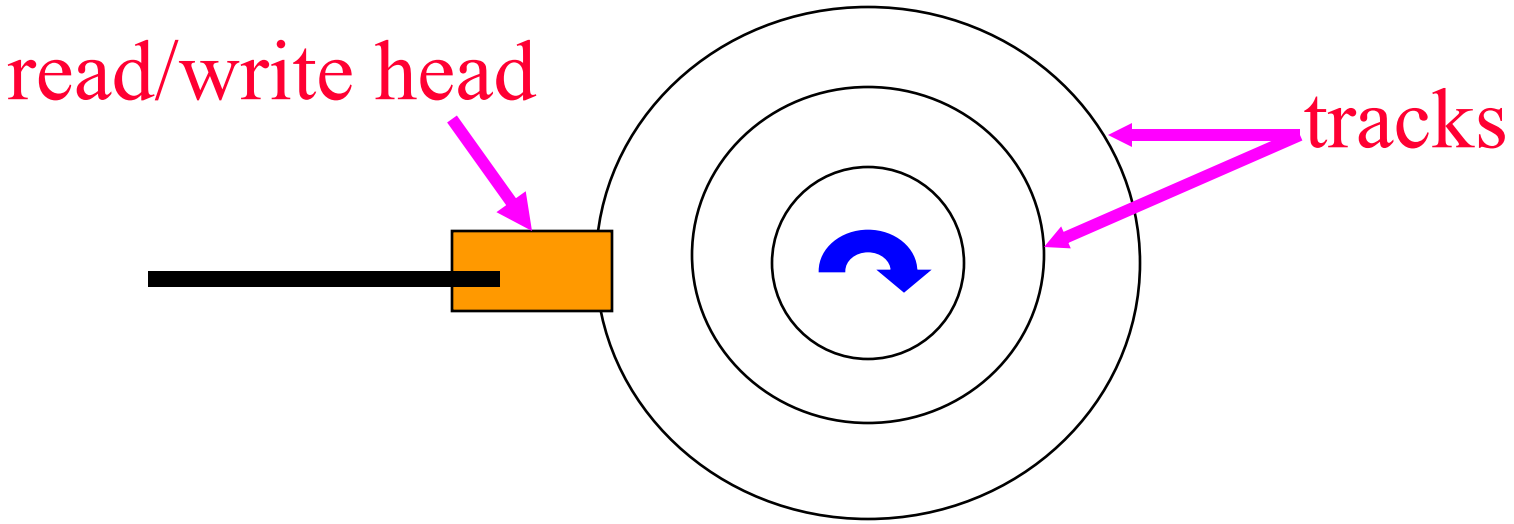
New Data Structures/Concepts

- Tournament trees.
- Huffman trees.
- Double-ended priority queues.
- Buffering.
- Ideas also may be used to speed algorithms for small instances by using cache more efficiently.

External Sort Computer Model

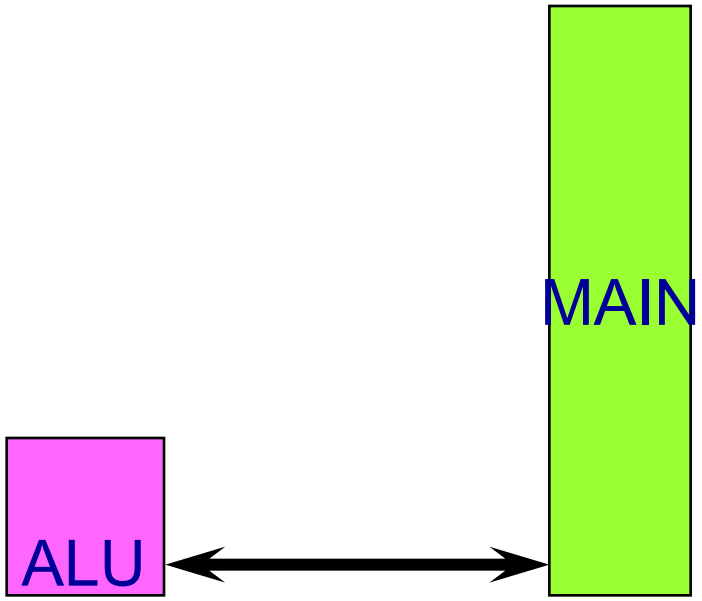


Disk Characteristics (HDD)



- Seek time
  - Approx. 100,000 arithmetics
- Latency time
  - Approx. 25,000 arithmetics
- Transfer time
- Data access by block

Traditional Internal Memory Model

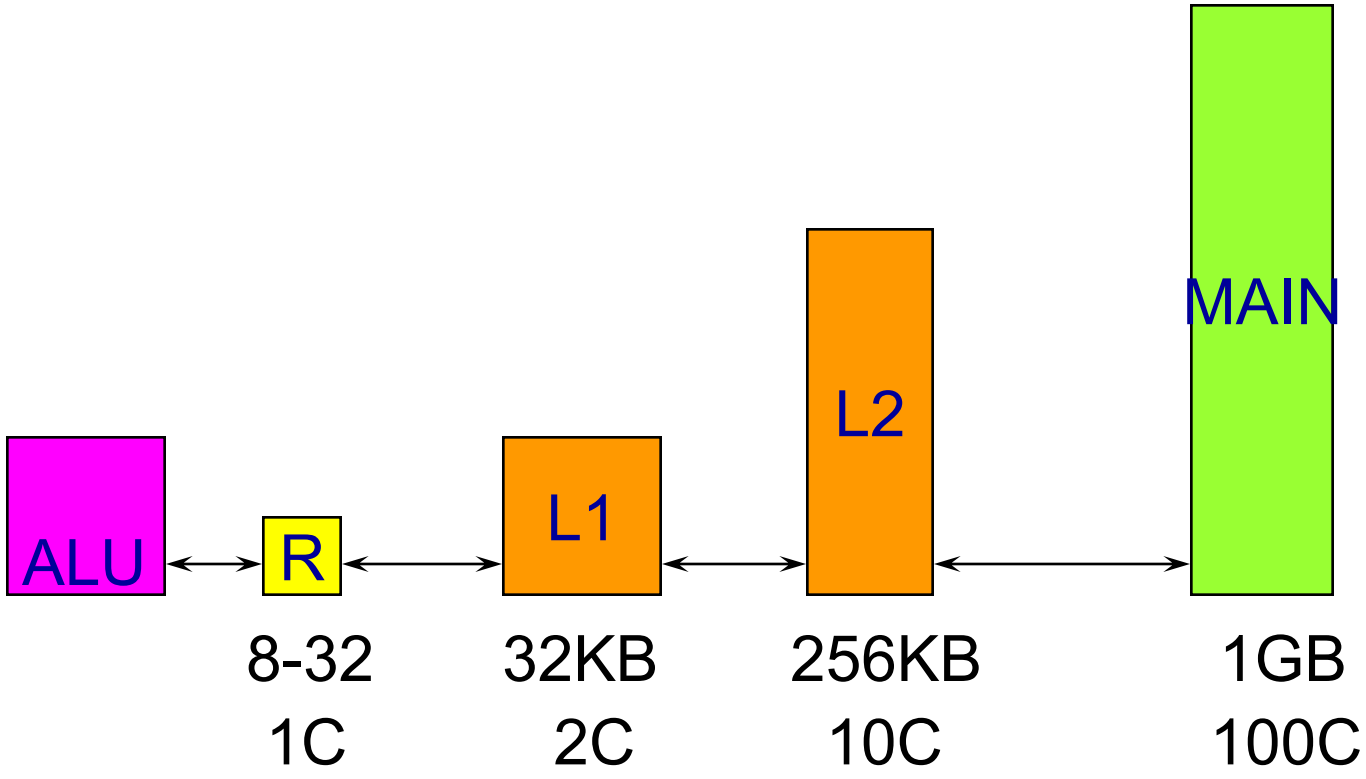


Matrix Multiplication

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
         c[i][j] += a[i][k] * b[k][j];
```

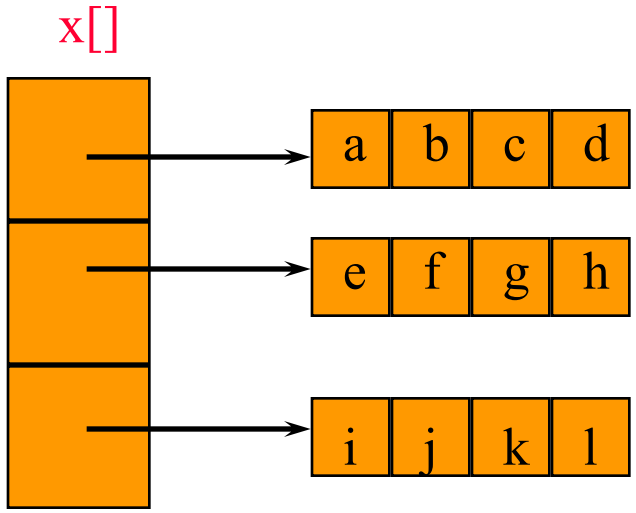
- ijk, ikj, jik, jki, kij, kji orders of loops yield same result.
- All perform same number of operations.
- But run time may differ significantly!

More Accurate Memory Model



2D Array Representation In Java, C, and C++

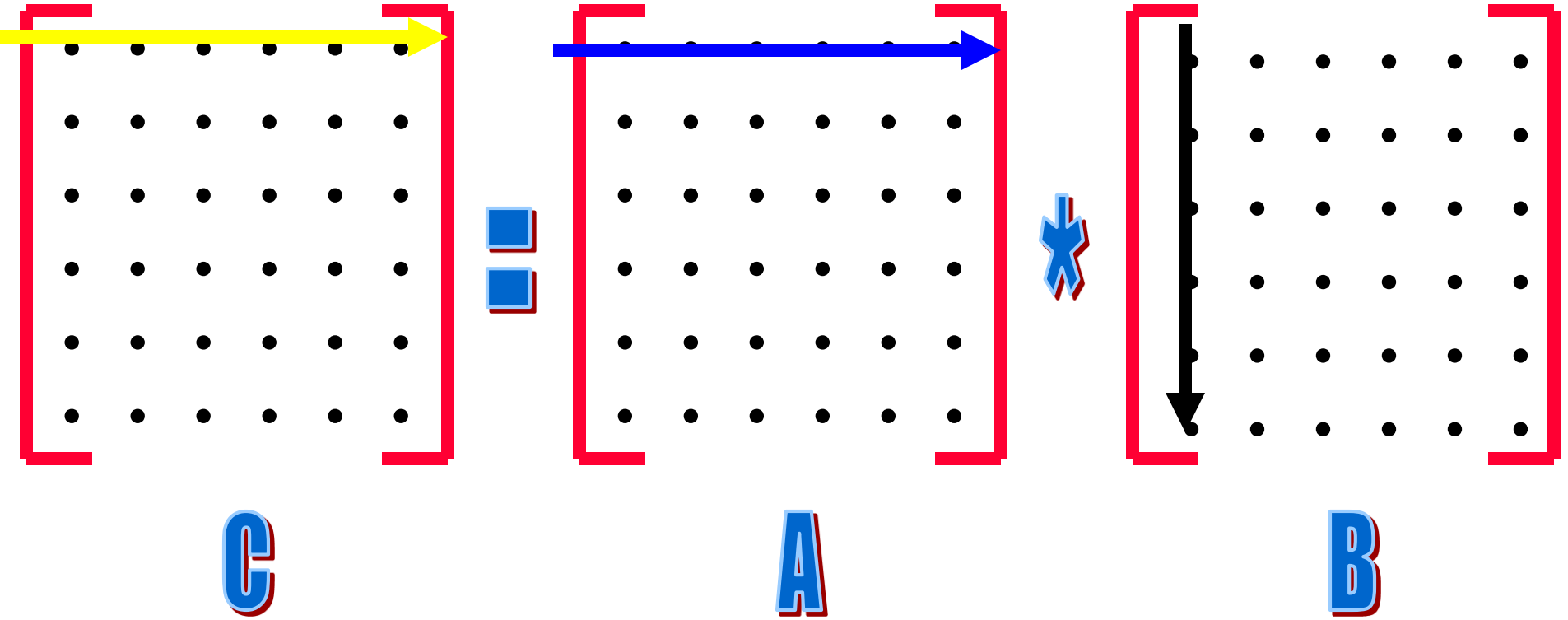
int x[3][4];



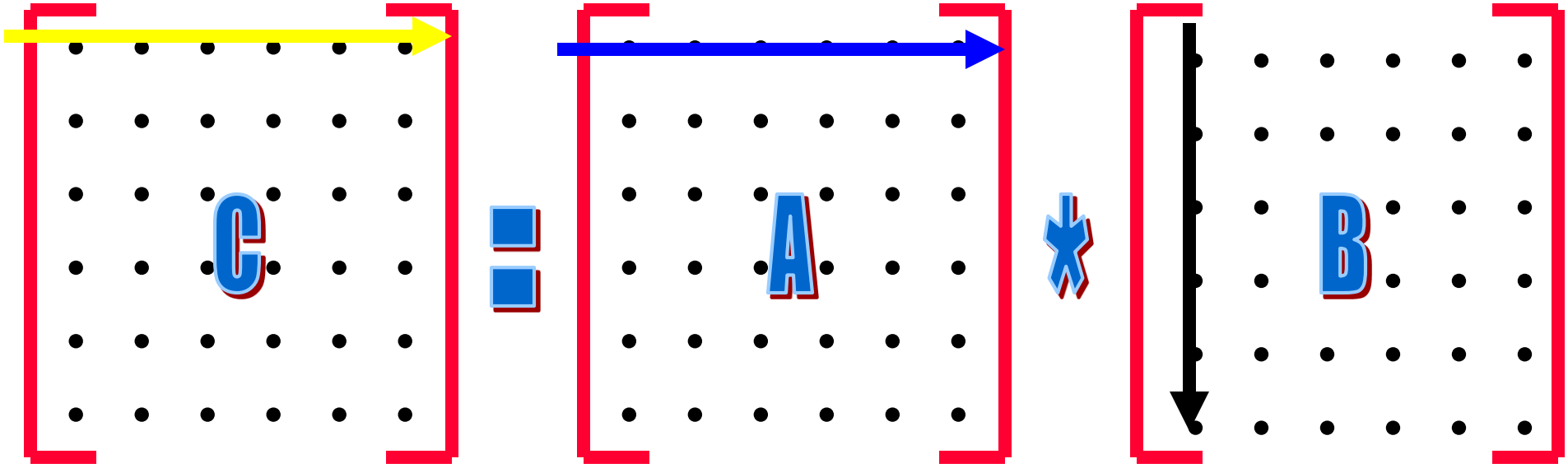
Array of Arrays Representation

ijk Order

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            c[i][j] += a[i][k] * b[k][j];
```



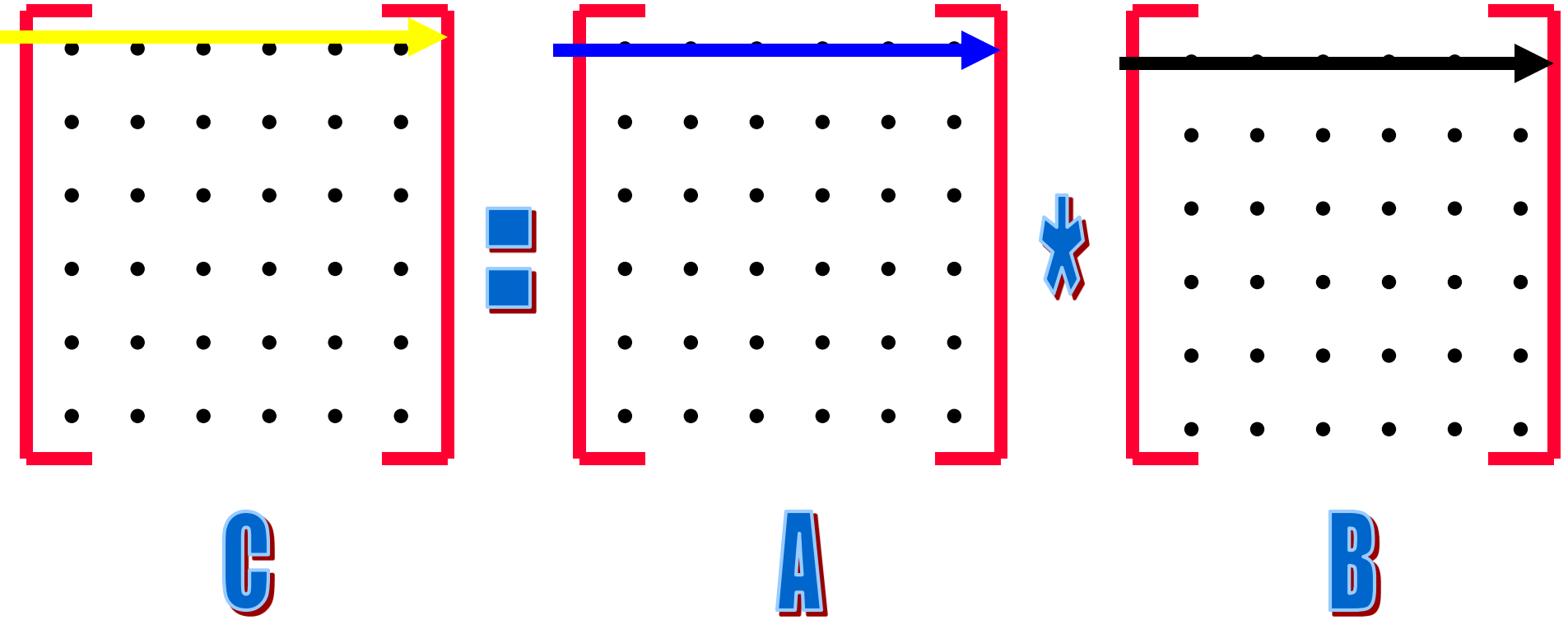
ijk Analysis



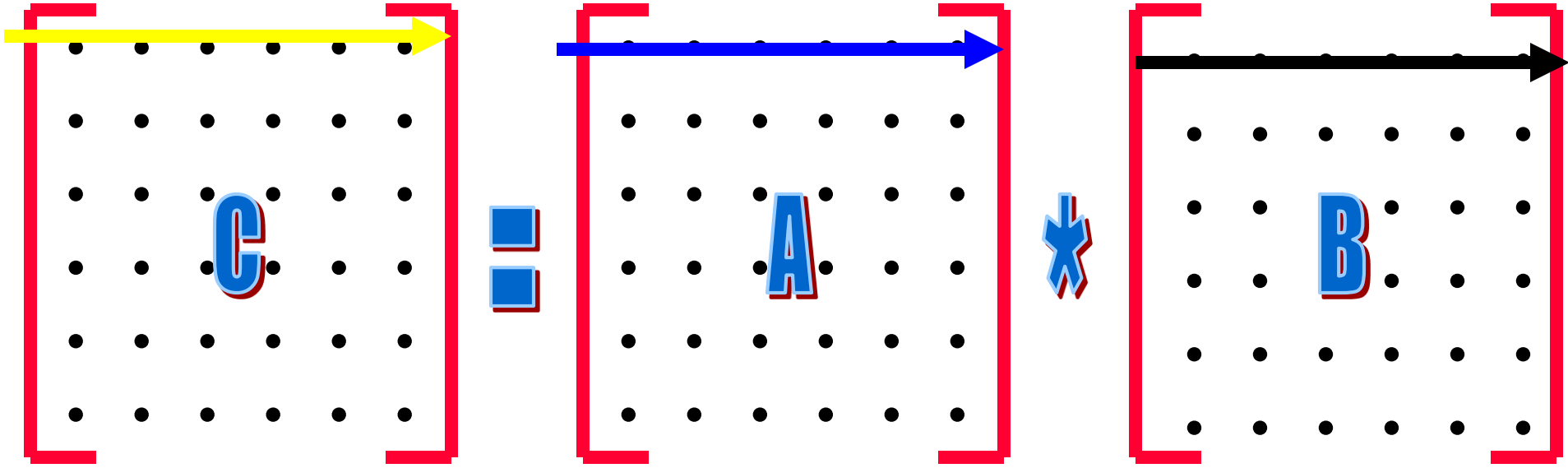
- Block size = width of cache line = w.
- Assume one-level cache.
- C => n^2/w cache misses.
- A => n^3/w cache misses, when n is large.
- B => n^3 cache misses, when n is large.
- Total cache misses = $n^3/w(1/n + 1 + w)$.

ikj Order

```
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
        for (int j = 0; j < n; j++)
            c[i][j] += a[i][k] * b[k][j];
```



ikj Analysis



- C => n^3/w cache misses, when n is large.
- A => n^2/w cache misses.
- B => n^3/w cache misses, when n is large.
- Total cache misses = $n^3/w(2 + 1/n)$.

ijk Vs. ikj Comparison

- ijk cache misses = $n^3/w(1/n + 1 + w)$.
- ikj cache misses = $n^3/w(2 + 1/n)$.
- ijk/ikj ~ $(1 + w)/2$, when n is large.
- w = 4 (32-byte cache line, double precision data)
  - ratio ~ 2.5.
- w = 8 (64-byte cache line, double precision data)
  - ratio ~ 4.5.
- w = 16 (64-byte cache line, integer data)
  - ratio ~ 8.5.

Prefetch

- Prefetch can hide memory latency
- Successful prefetch requires ability to predict a memory access much in advance
- Prefetch cannot reduce energy as prefetch does not reduce number of memory accesses

Faster Internal Sorting

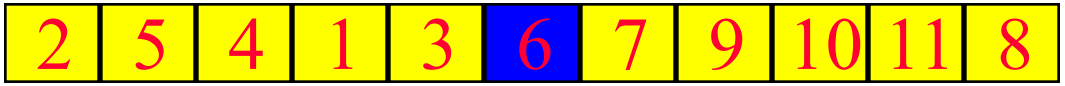
- May apply external sorting ideas to internal sorting.
- Internal tiled merge sort gives 2x (or more) speedup over traditional merge sort.

External Sort Methods

- Base the external sort method on a fast internal sort method.
- Average run time
  - Quick sort
- Worst-case run time
  - Merge sort

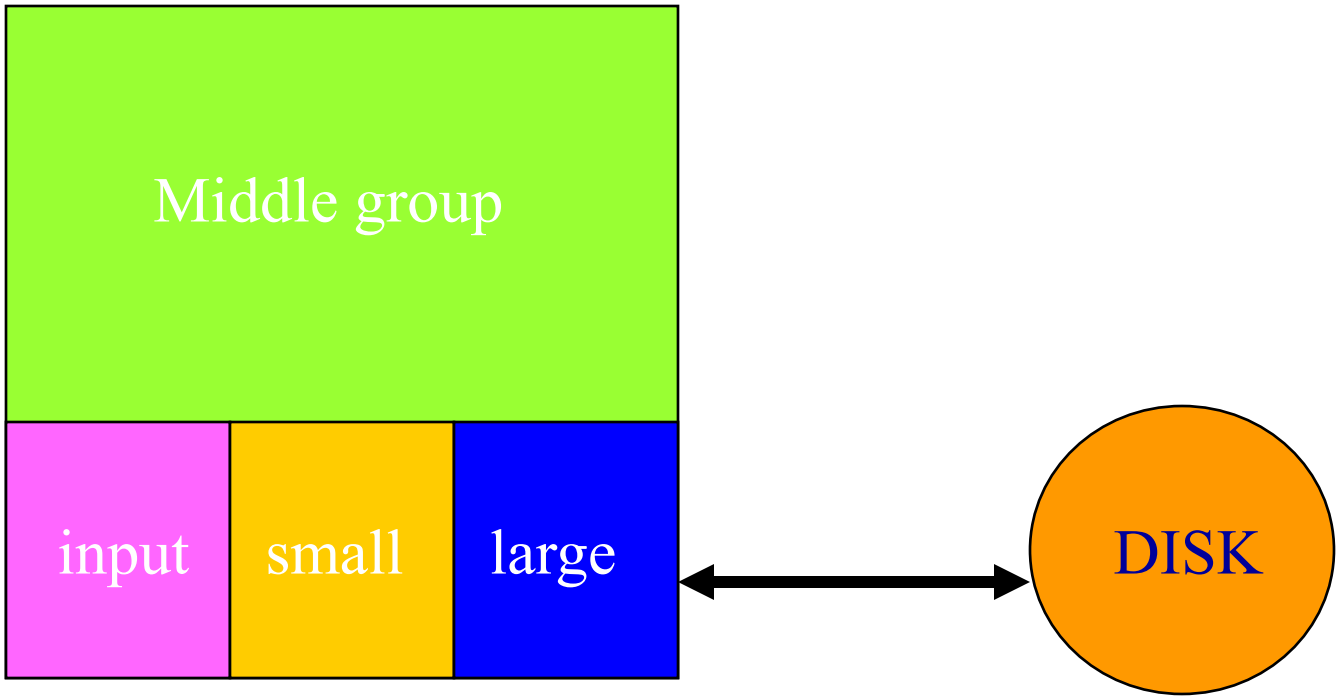
Internal Quick Sort

Use 6 as the pivot.



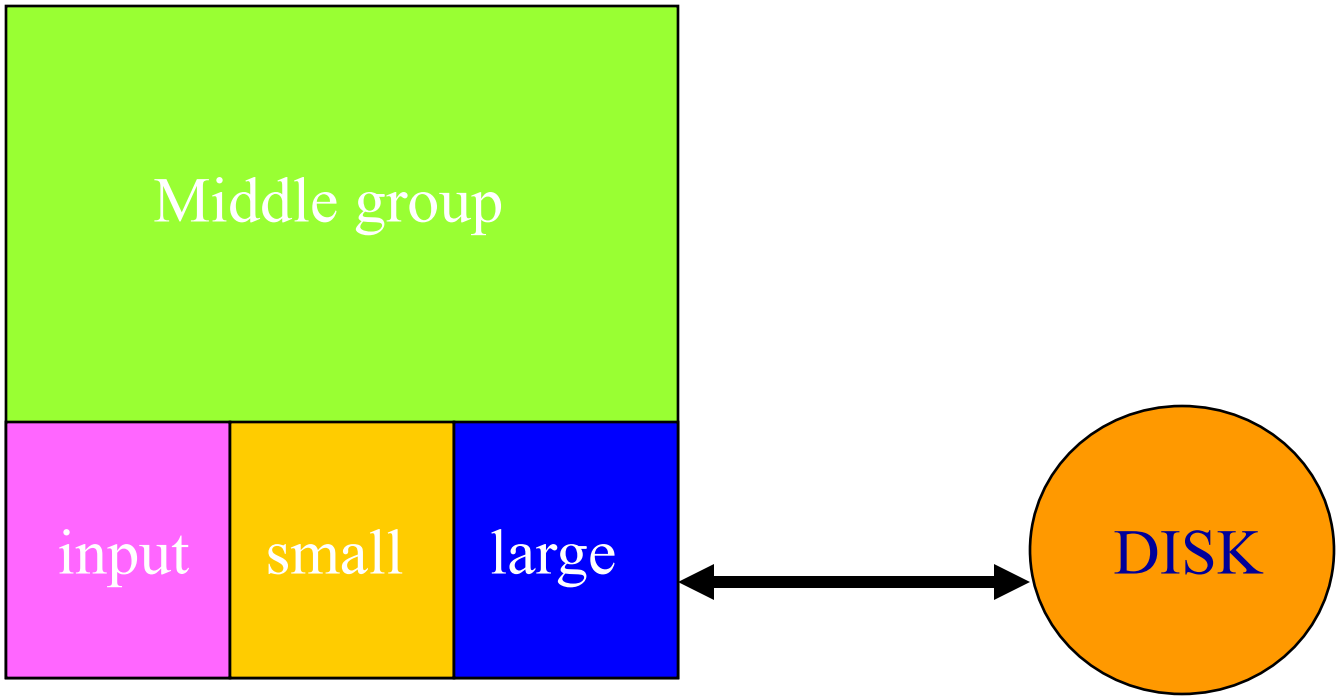
Sort left and right groups recursively.

Quick Sort – External Adaptation



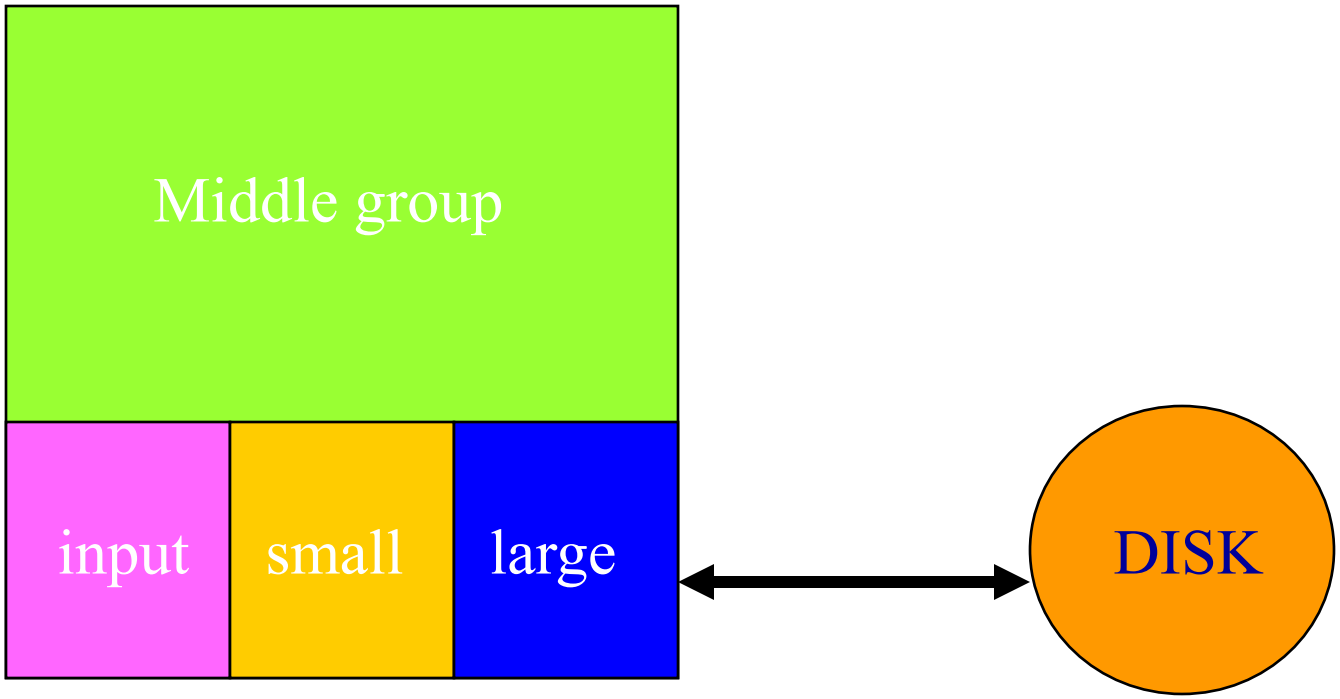
- 3 input/output buffers
  - input, small, large
- rest is used for middle group

Quick Sort – External Adaptation



- fill middle group and input buffer from disk
- if next record <= $middle_{min}$ send to small
- else if next record >= $middle_{max}$ send to large
- else remove $middle_{min}$ or $middle_{max}$ from middle and add new record to middle group

Quick Sort – External Adaptation



- Fill input buffer when it gets empty.
- Write small/large buffer when full.
- Write middle group in sorted order when done.
- Double-ended priority queue.
- Use additional buffers to reduce I/O wait time.

External Sorting

- Adapt fastest internal-sort methods.

✓ Quick sort …best average run time.

- Merge sort … best worst-case run time.

Internal Merge Sort Review

- Phase 1
  - Create initial sorted segments
    - Natural segments
    - Insertion sort
- Phase 2
  - Merge pairs of sorted segments, in merge passes, until only 1 segment remains.

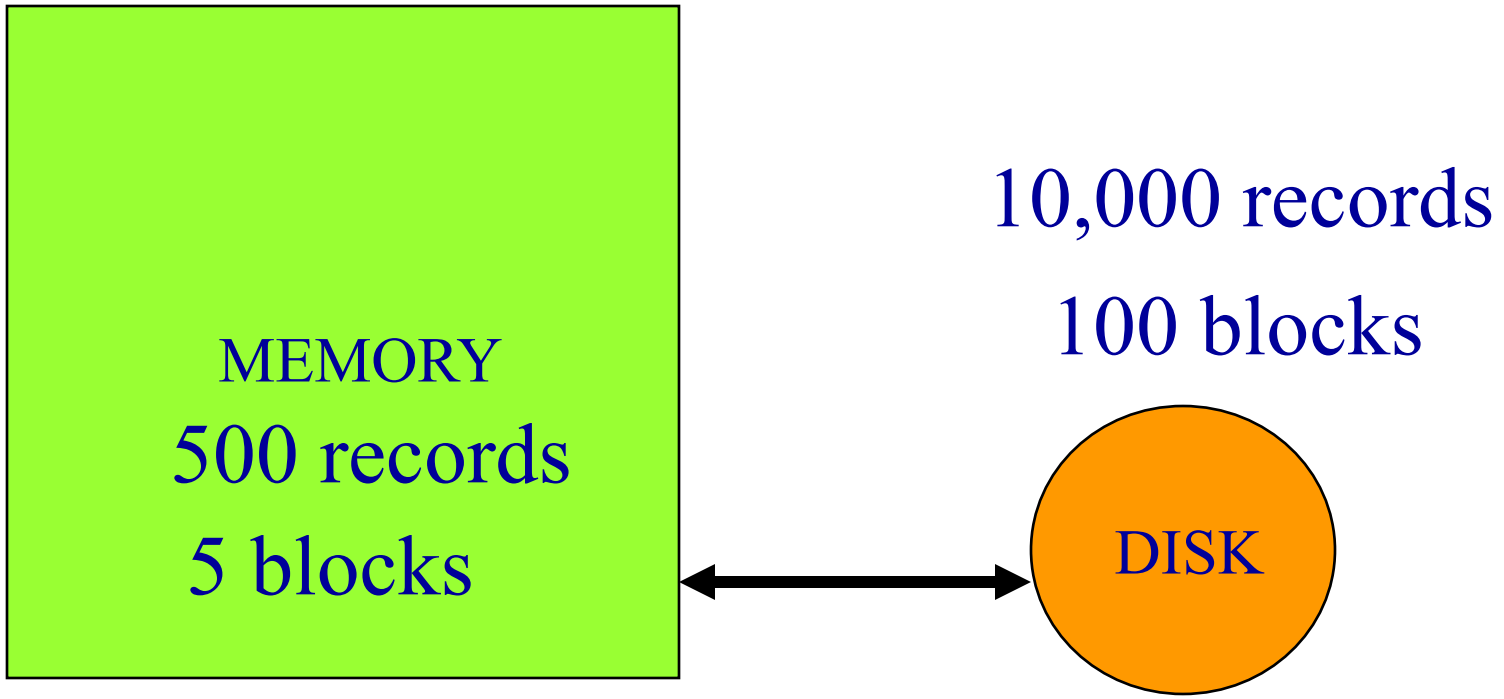
External Merge Sort

- Sort 10,000 records.
- Enough memory for 500 records.
- Block size is 100 records.
- t_{IO} = time to input/output 1 block
  (includes seek, latency, and transmission times)
- t_{IS} = time to internally sort 1 memory load
- t_{IM} = time to internally merge 1 block load

External Merge Sort

- Two phases.
  - Run generation.
    - A run is a sorted sequence of records.
  - Run merging.

Run Generation



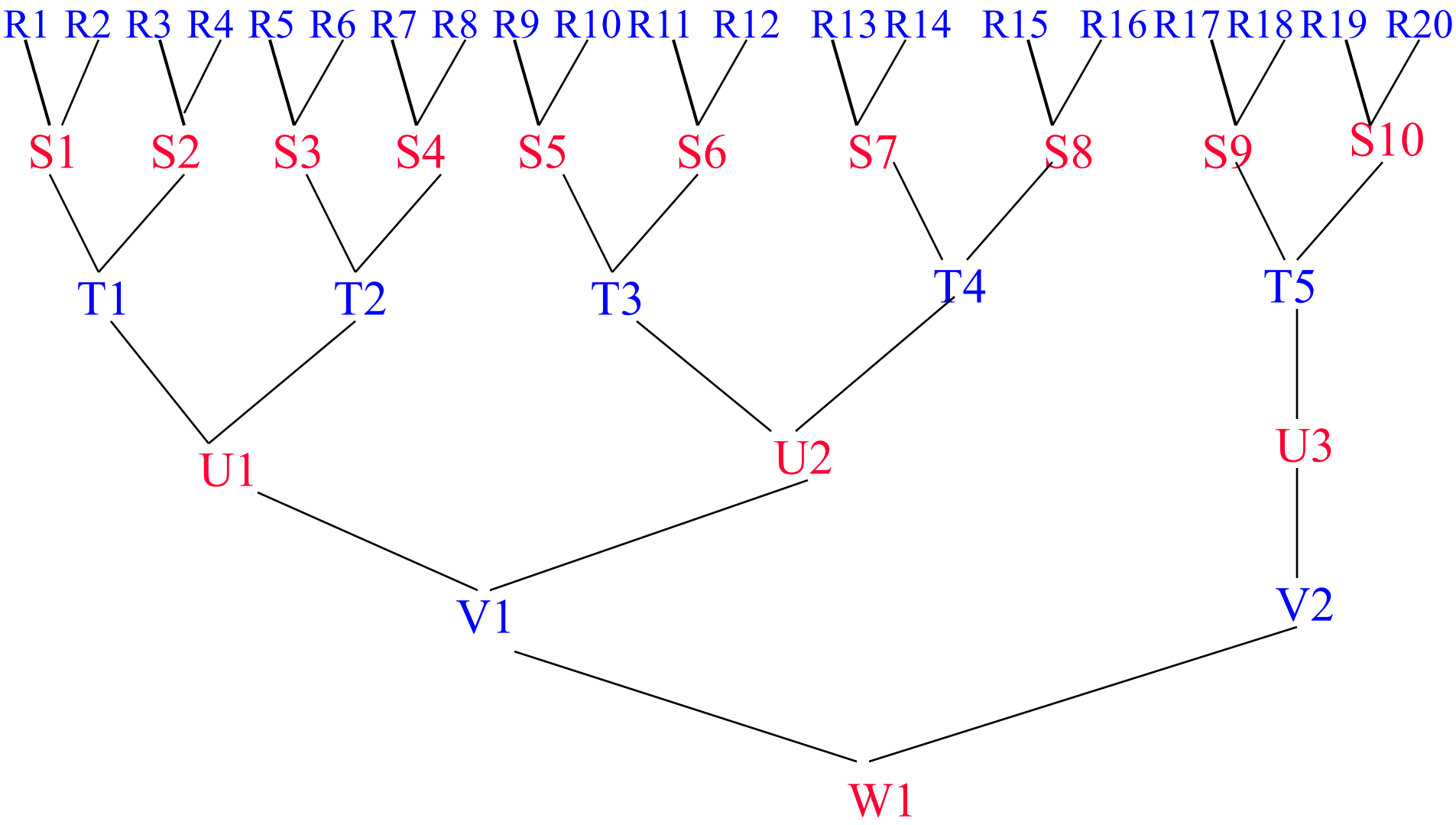
- Input 5 blocks.
- Sort.
- Output as a run.
- Do 20 times.

- $5t_{IO}$
- t_{IS}
- $5t_{IO}$
- $200t_{IO} + 20t_{IS}$

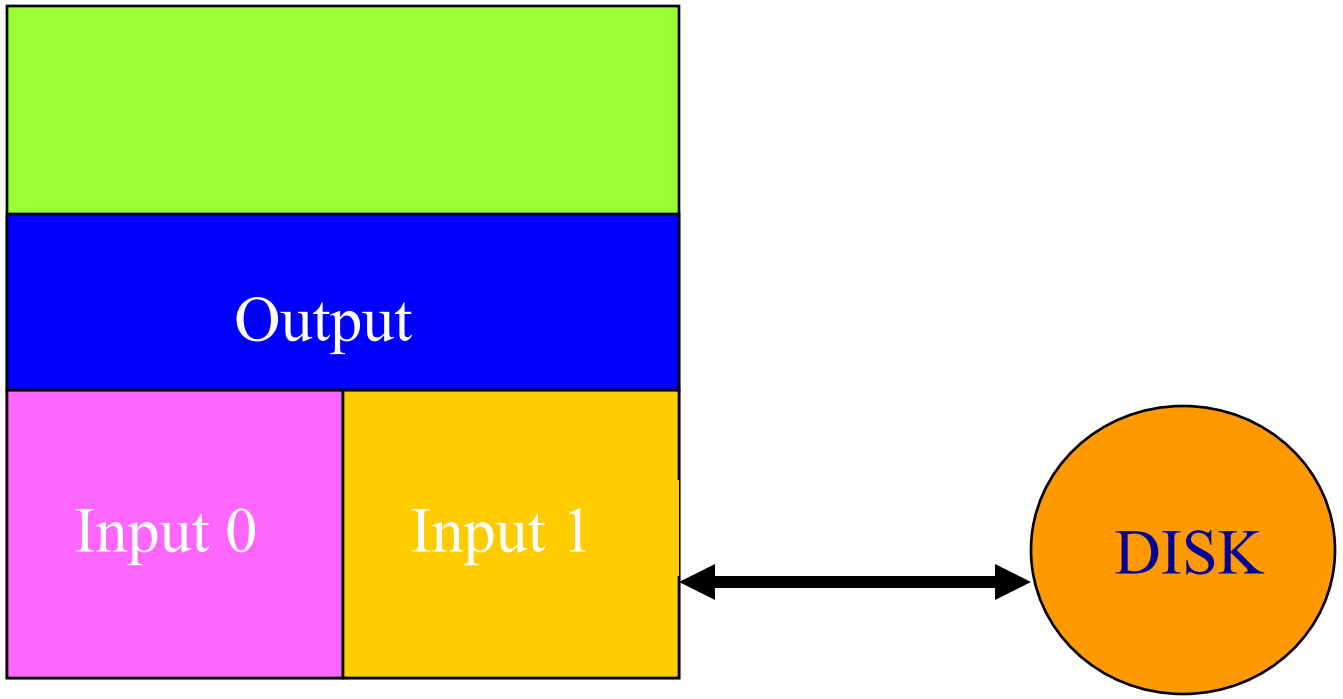
Run Merging

- Merge Pass.
  - Pairwise merge the 20 runs into 10.
  - In a merge pass all runs (except possibly one) are pairwise merged.
- Perform 4 more merge passes, reducing the number of runs to 1.

Merge 20 Runs



Merge R1 and R2



- Fill I0 (Input 0) from R1 and I1 from R2.
- Merge from I0 and I1 to output buffer.
- Write whenever output buffer full.
- Read whenever input buffer empty.

Time To Merge R1 and R2

- Each is 5 blocks long.
- Input time = $10t_{IO}$.
- Write/output time = $10t_{IO}$.
- Merge time = $10t_{IM}$.
- Total time = $20t_{IO} + 10t_{IM}$.

Time For Pass 1 (R → S)

- Time to merge one pair of runs
  $= 20t_{IO} + 10t_{IM}$.
- Time to merge all 10 pairs of runs
  $= 200t_{IO} + 100t_{IM}$.

Time To Merge S1 and S2

- Each is 10 blocks long.
- Input time = $20t_{IO}$.
- Write/output time = $20t_{IO}$.
- Merge time = $20t_{IM}$.
- Total time = $40t_{IO} + 20t_{IM}$.

Time For Pass 2 (S → T)

- Time to merge one pair of runs
  $= 40t_{IO} + 20t_{IM}$.
- Time to merge all 5 pairs of runs
  $= 200t_{IO} + 100t_{IM}$.

Time For One Merge Pass

- Time to input all blocks = $100t_{IO}$.
- Time to output all blocks = $100t_{IO}$.
- Time to merge all blocks = $100t_{IM}$.
- Total time for a merge pass = $200t_{IO} + 100t_{IM}$.

Total Run-Merging Time

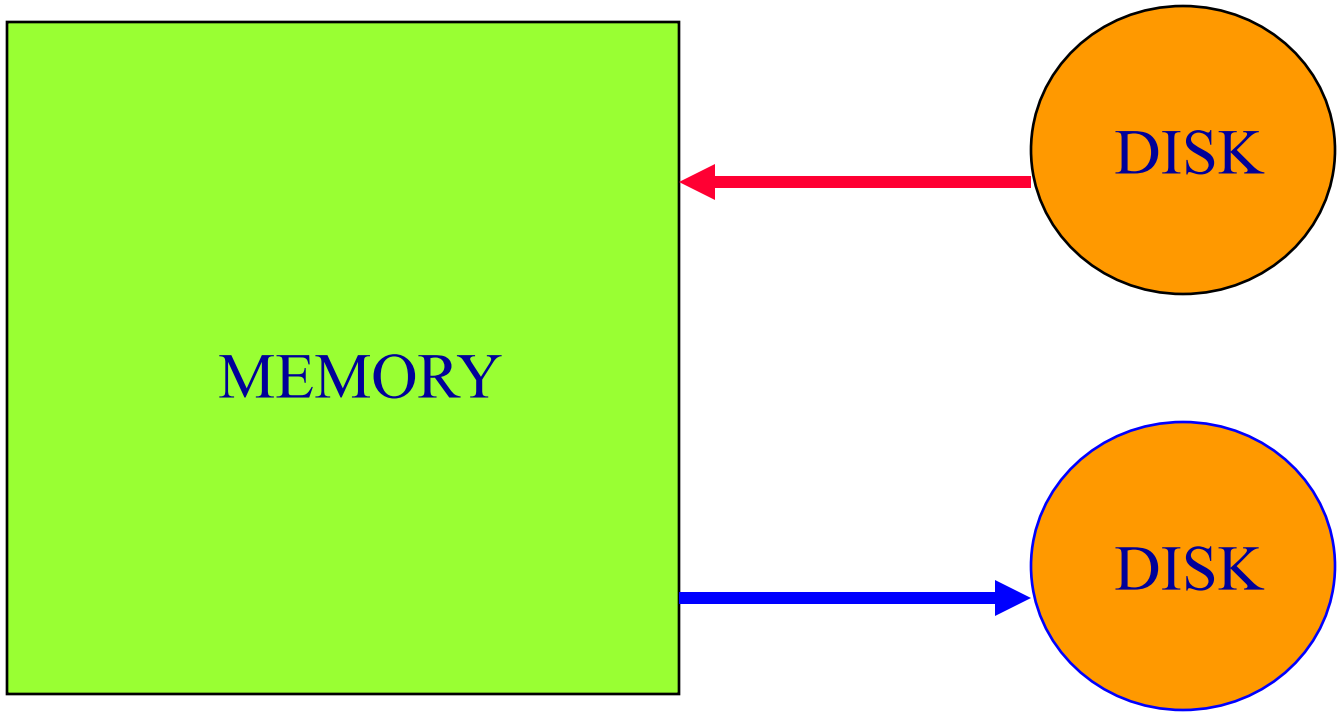
- (time for one merge pass) * (number of passes)
  = (time for one merge pass)
  * ceil($\log_2$(number of initial runs))
  = $(200t_{IO} + 100t_{IM})$ * ceil($\log_2(20)$)
  = $(200t_{IO} + 100t_{IM})$ * 5

Factors In Overall Run Time

- Run generation. $200t_{IO} + 20t_{IS}$
  - Internal sort time.
  - Input and output time.
- Run merging. $(200t_{IO} + 100t_{IM}) * \text{ceil}(\log_2(20))$
  - Internal merge time.
  - Input and output time.
  - Number of initial runs.
  - Merge order (number of merge passes is determined by number of runs and merge order)

Improve Run Generation

- Overlap input, output, and internal sorting.

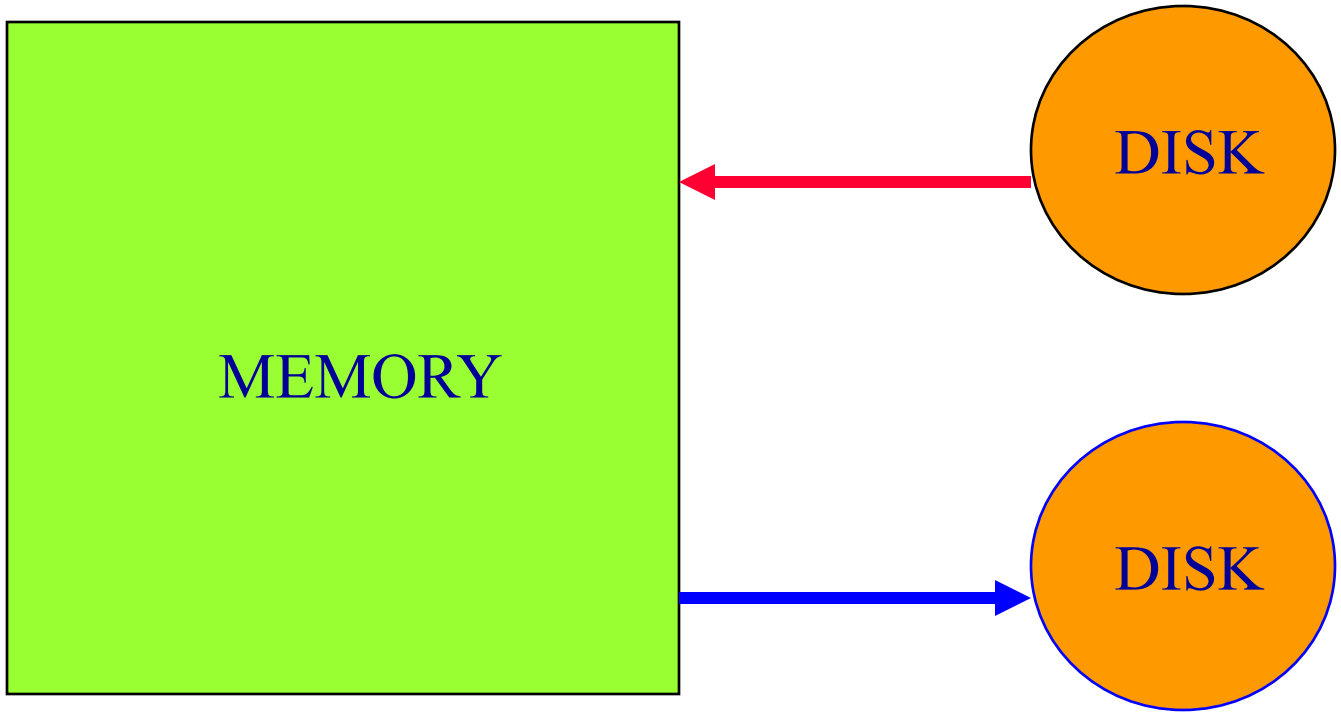


Improve Run Generation

- Generate runs whose length (on average) exceeds memory size.
- Equivalent to reducing number of runs generated.

Improve Run Merging

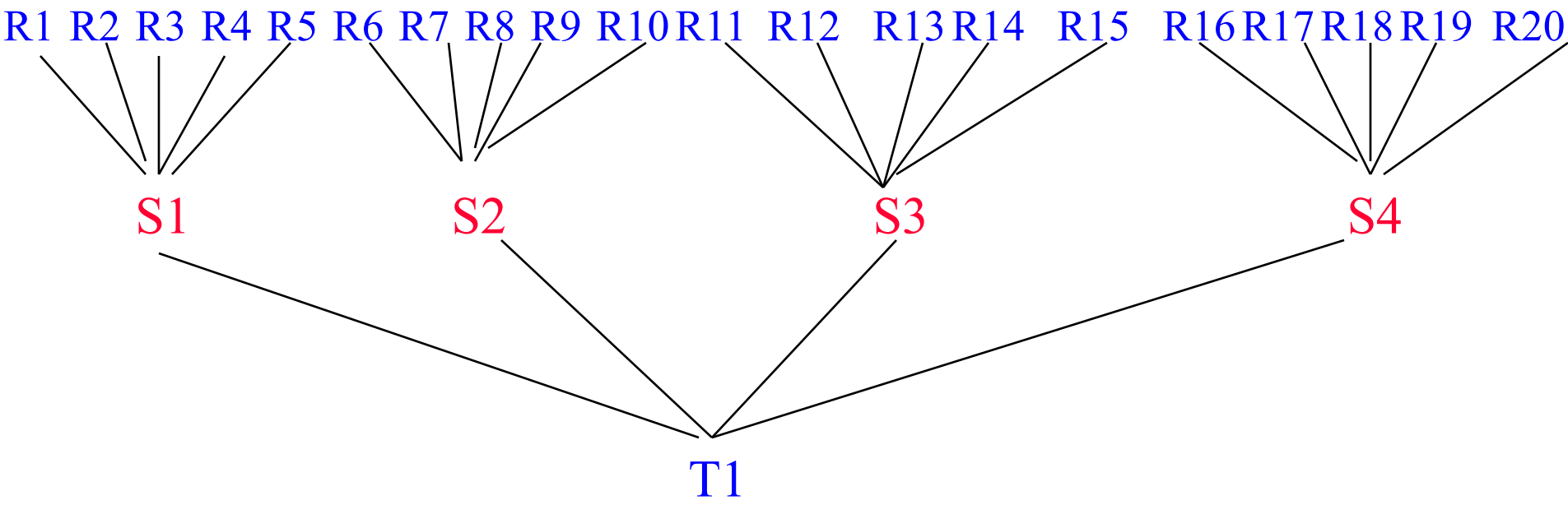
- Overlap input, output, and internal merging.



Improve Run Merging

- Reduce number of merge passes.
  - Use higher-order merge.
  - Number of passes
    = ceil($\log_k$(number of initial runs))
    where k is the merge order.

Merge 20 Runs Using 5-Way Merging

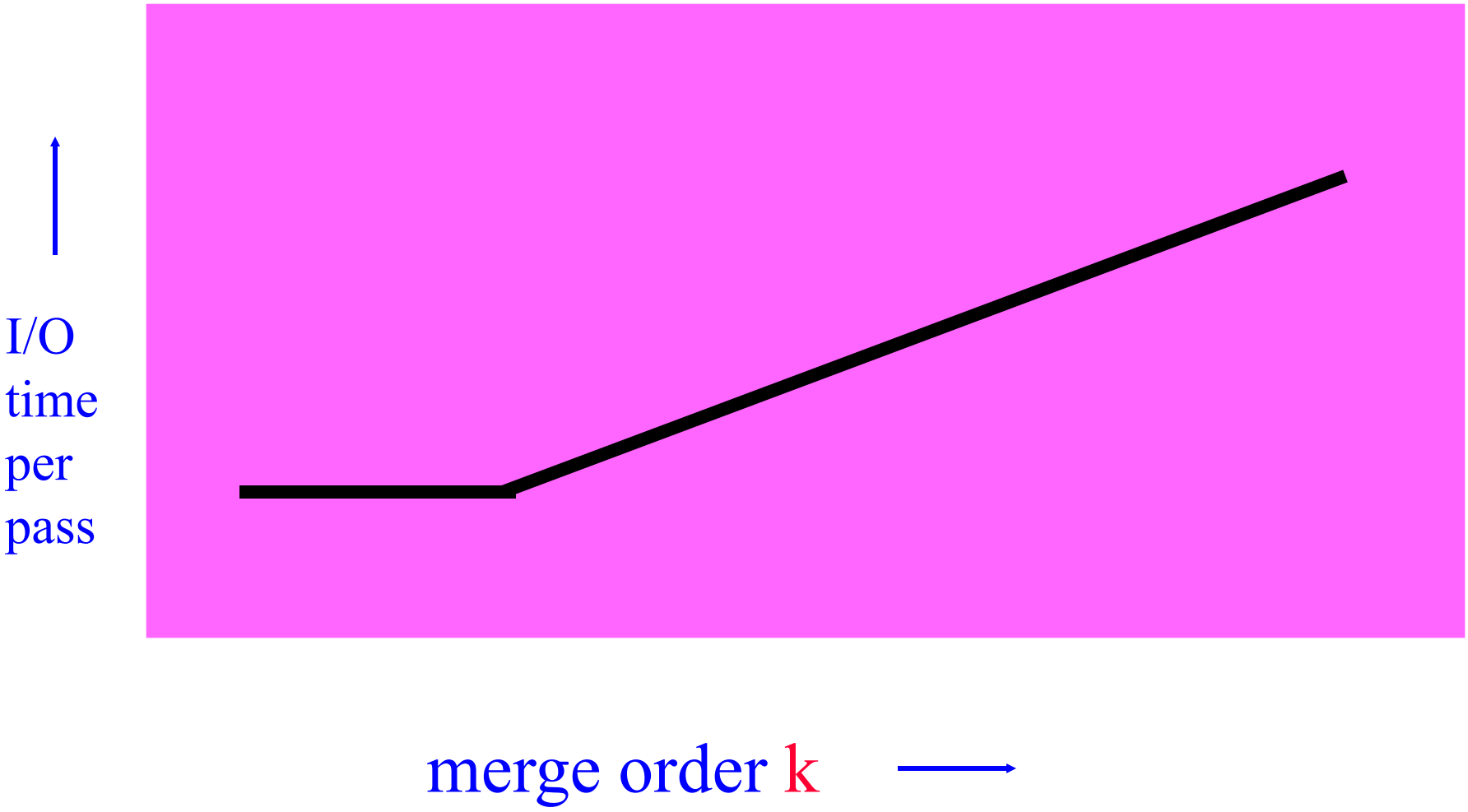


Number of passes = 2

I/O Time Per Merge Pass

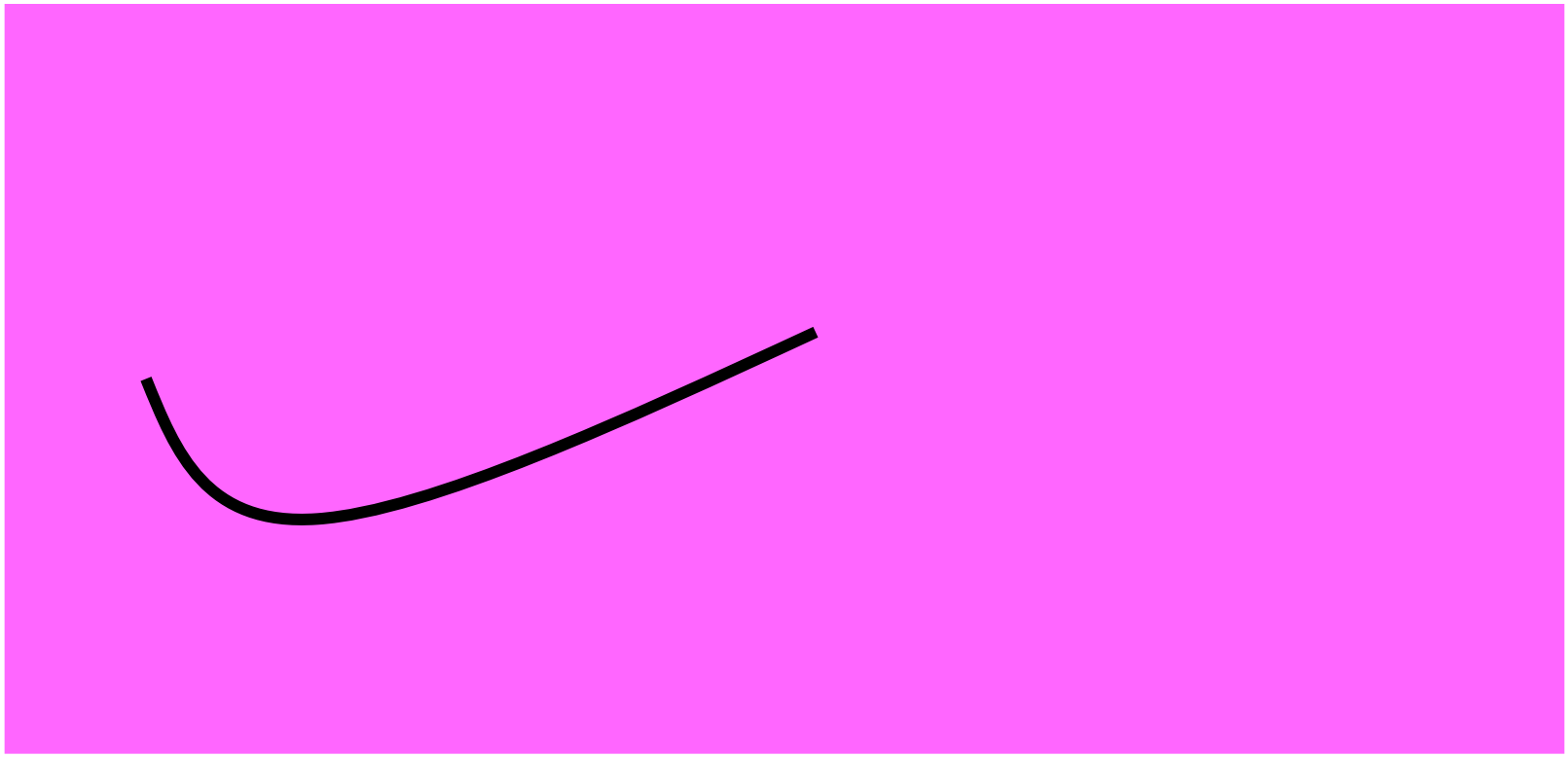
- Number of input buffers needed is linear in merge order k.
- Since memory size is fixed, block size decreases as k increases (after a certain k).
- So, number of blocks increases.
- So, number of seek and latency delays per pass increases.

I/O Time Per Merge Pass

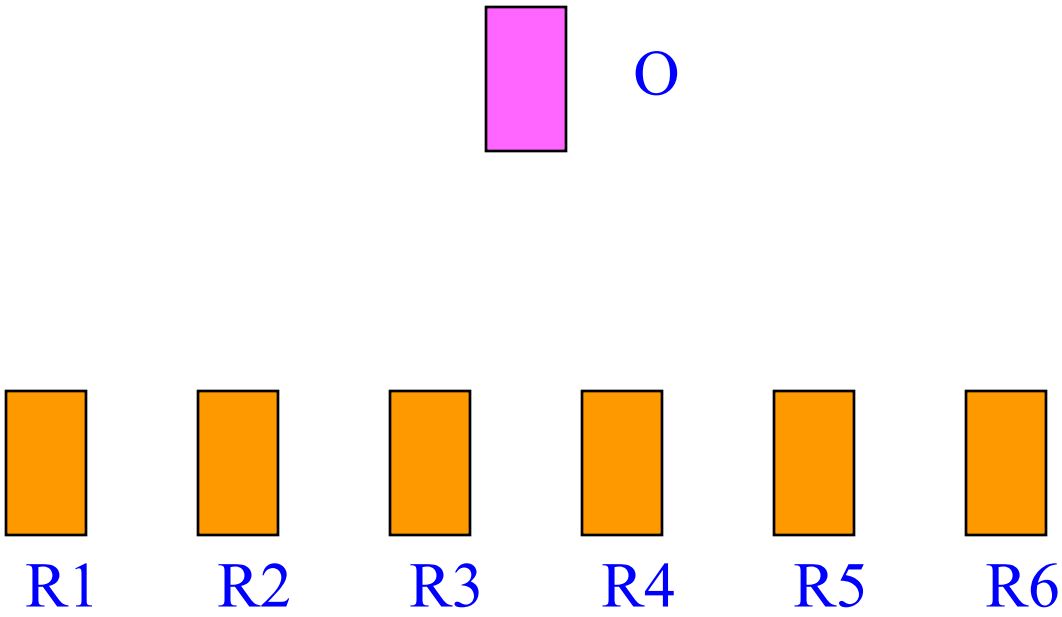


Total I/O Time To Merge Runs

- (I/O time for one merge pass)
  * ceil($\log_k$(number of initial runs))

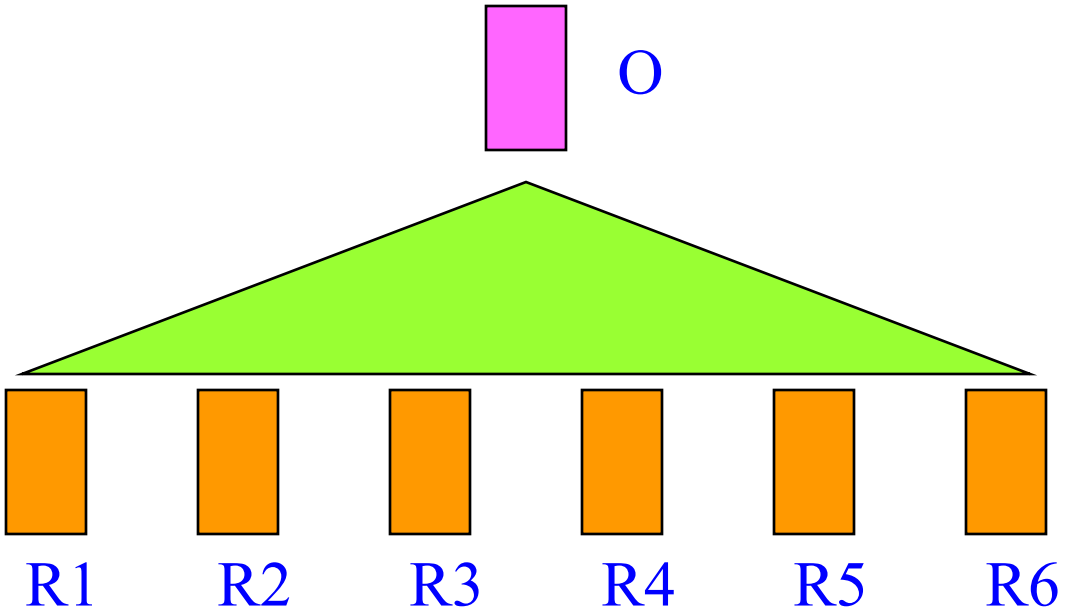


Internal Merge Time



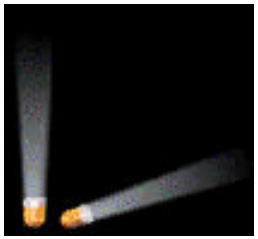
- Naïve way => $k - 1$ compares to determine next record to move to the output buffer.
- Time to merge n records is $c(k - 1)n$, where c is a constant.
- Merge time per pass is $c(k - 1)n$.
- Total merge time is $c(k - 1)n\log_k r \sim cn(k/\log_2 k)\log_2 r$.

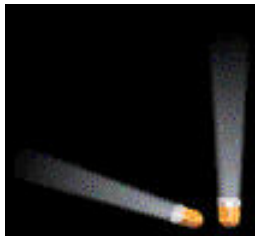
Merge Time Using A Tournament Tree



- Time to merge n records is $dn\log_2 k$, where d is a constant.
- Merge time per pass is $dn\log_2 k$.
- Total merge time is $(dn\log_2 k)\ \log_k r\ = dn\log_2 r$.

Tournament Trees





Winner trees.

Loser Trees.

Winner Tree – Definition

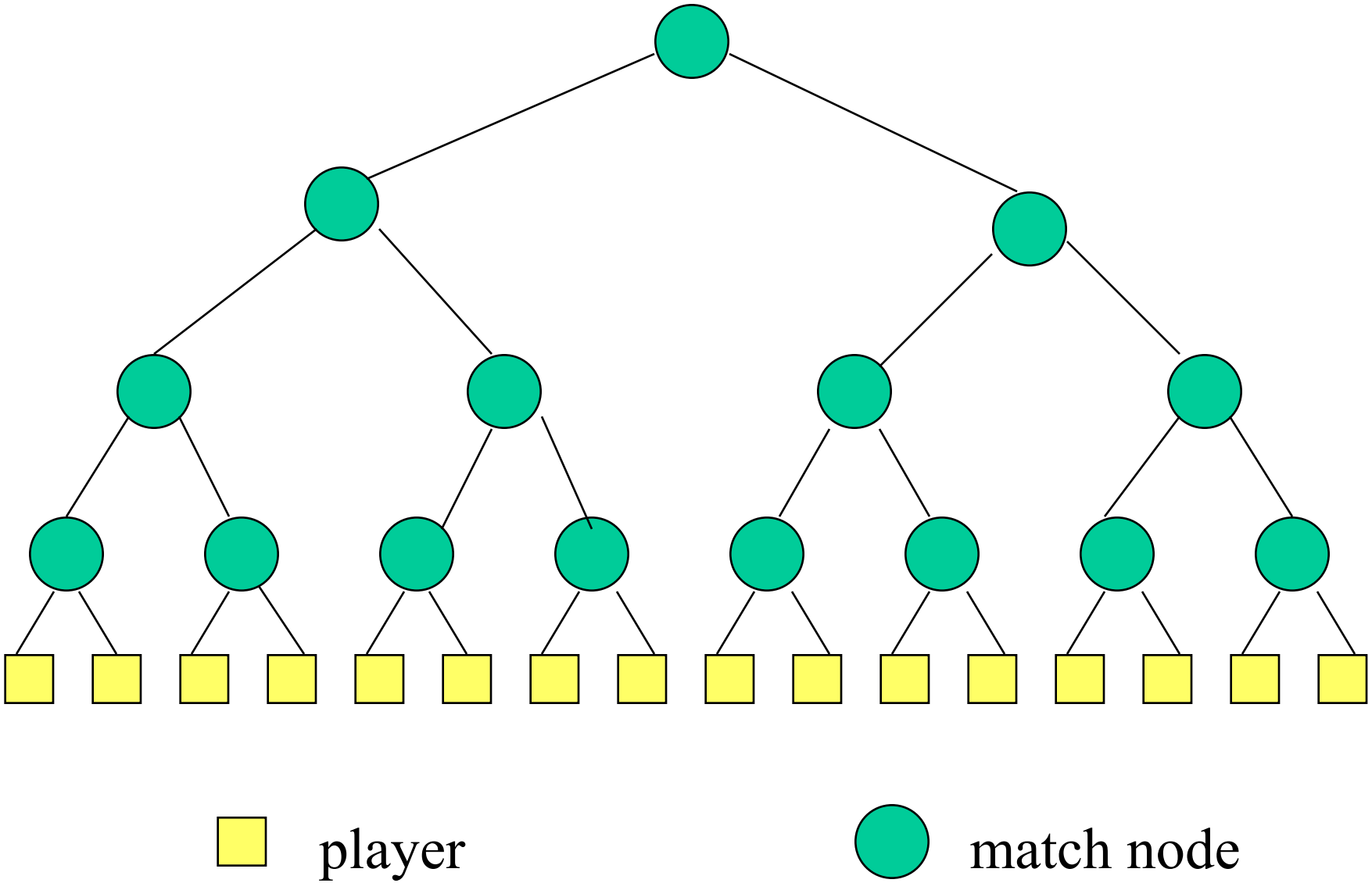
Complete binary tree with n-1 internal nodes and n external nodes.

External nodes represent tournament players.

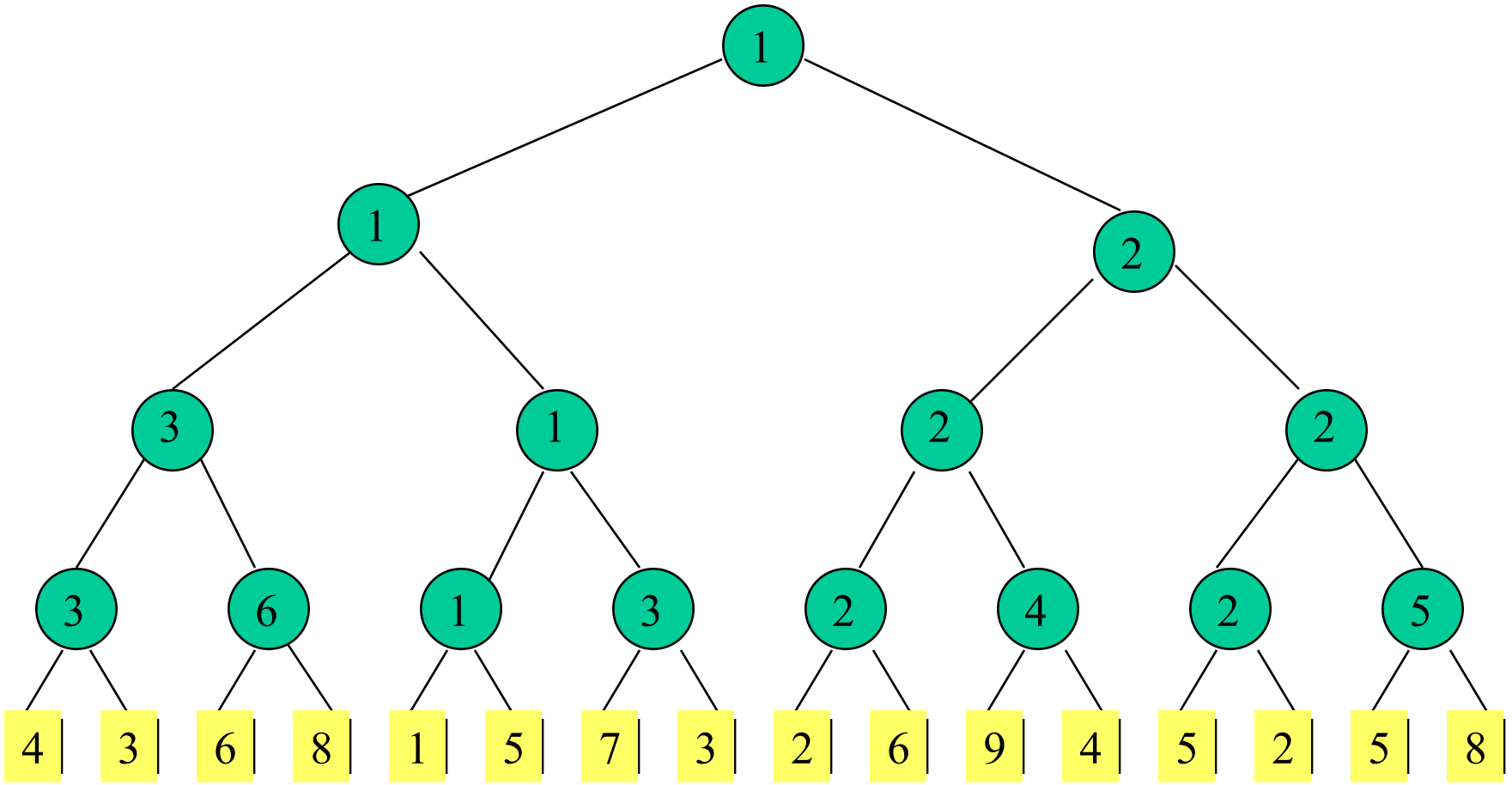
Each internal node represents a match played between its two children; the winner of the match is stored at the internal node.

Root has overall winner.

Winner Tree For 16 Players

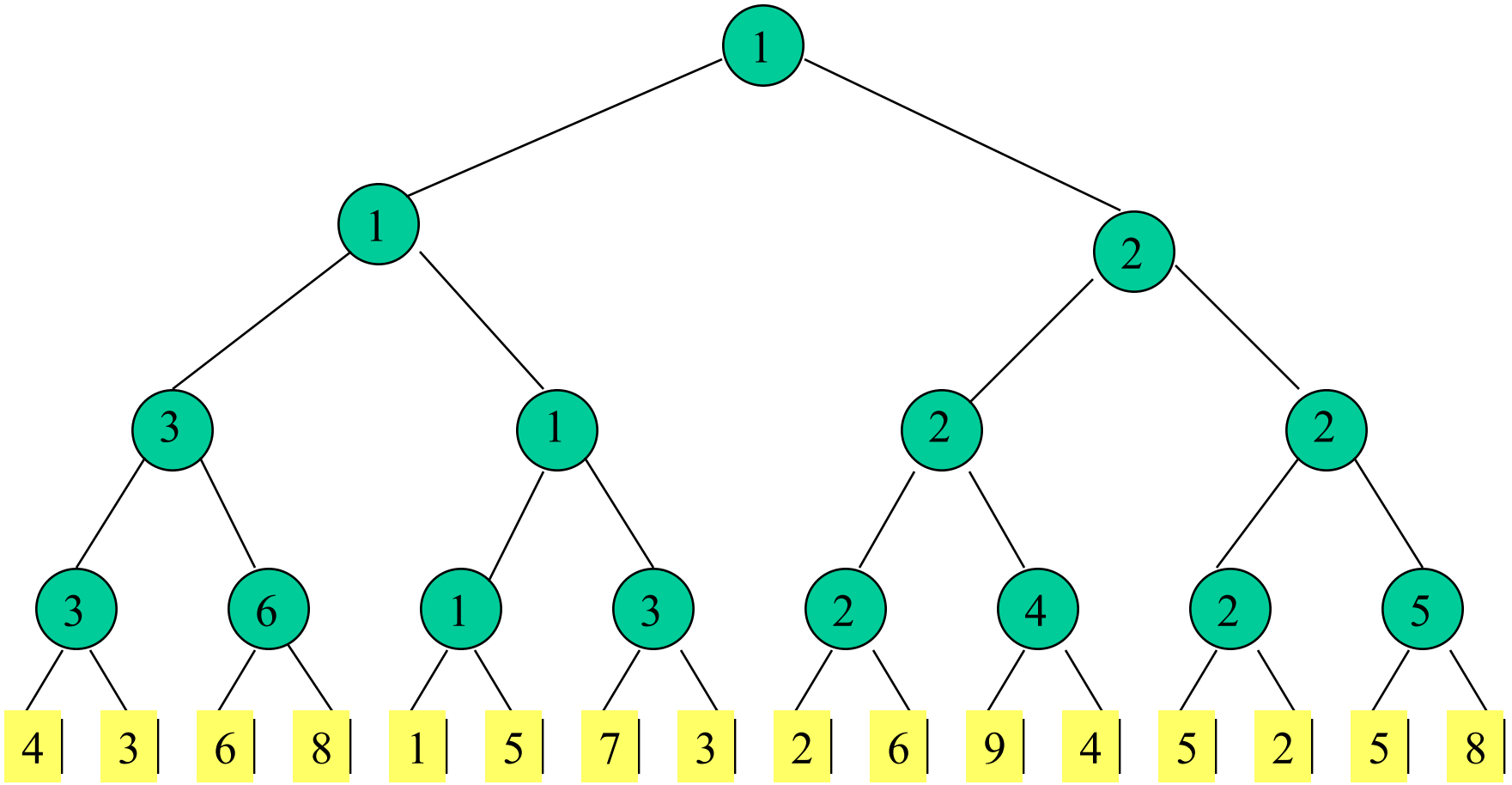


Winner Tree For 16 Players



Smaller element wins => min winner tree.

Winner Tree For 16 Players



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

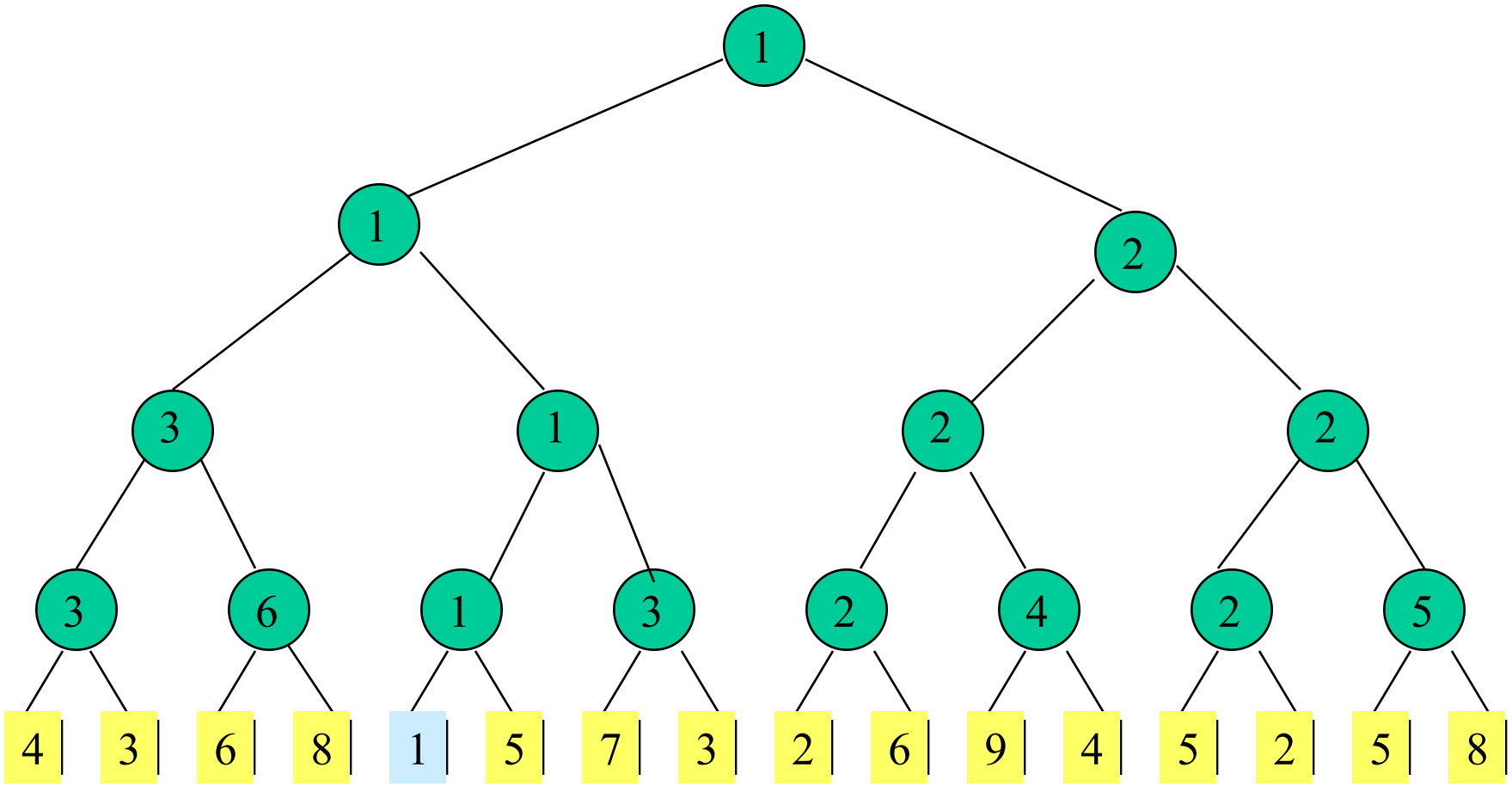
Complexity Of Initialize

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

Winner Tree Operations

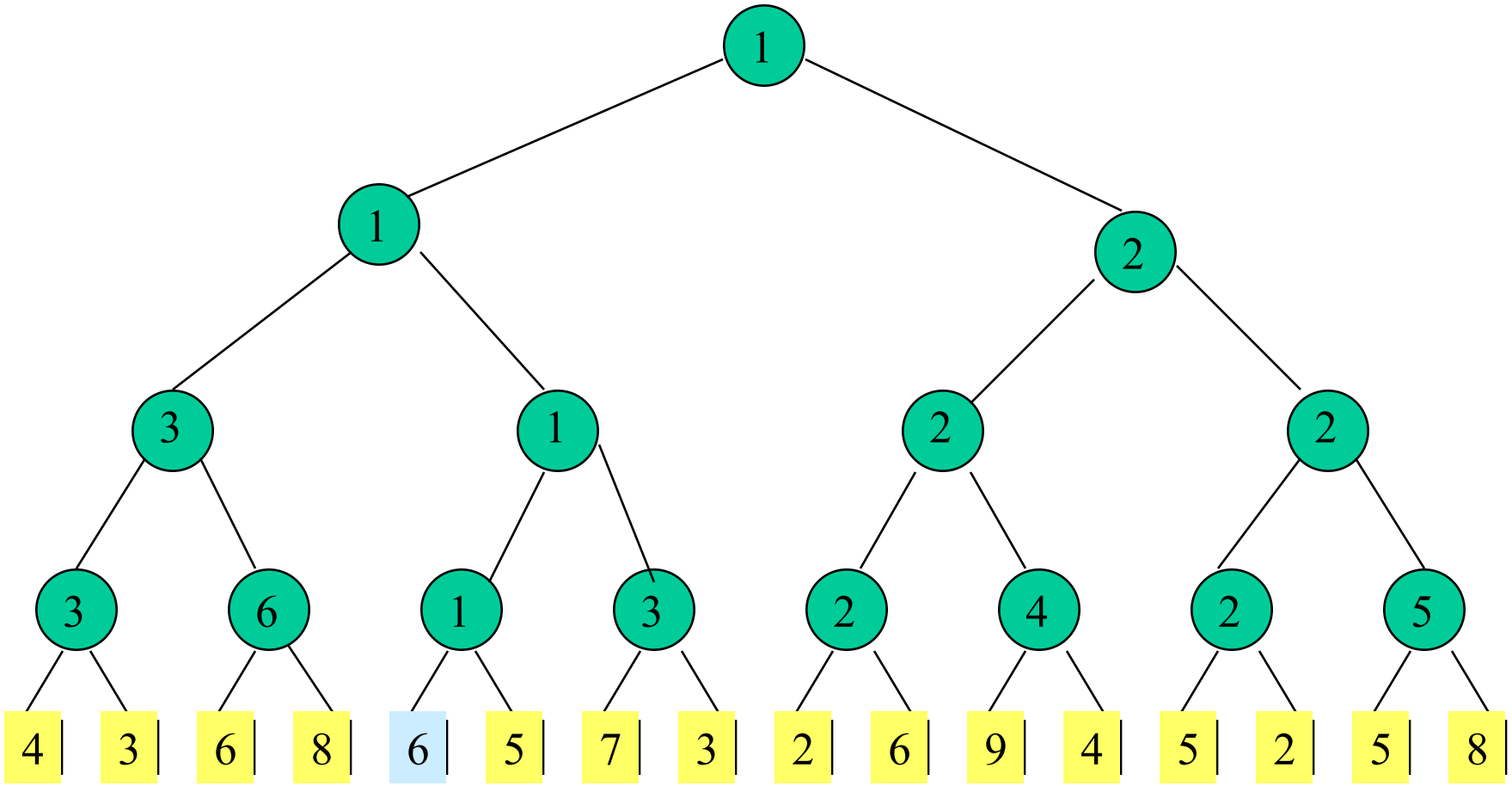
- Initialize
  - O(n) time
- Get winner
  - O(1) time
- Replace winner and replay
  - O(log n) time
  - More precisely Theta(log n)
- Tie breaker (player on left wins in case of a tie).

Replace Winner And Replay



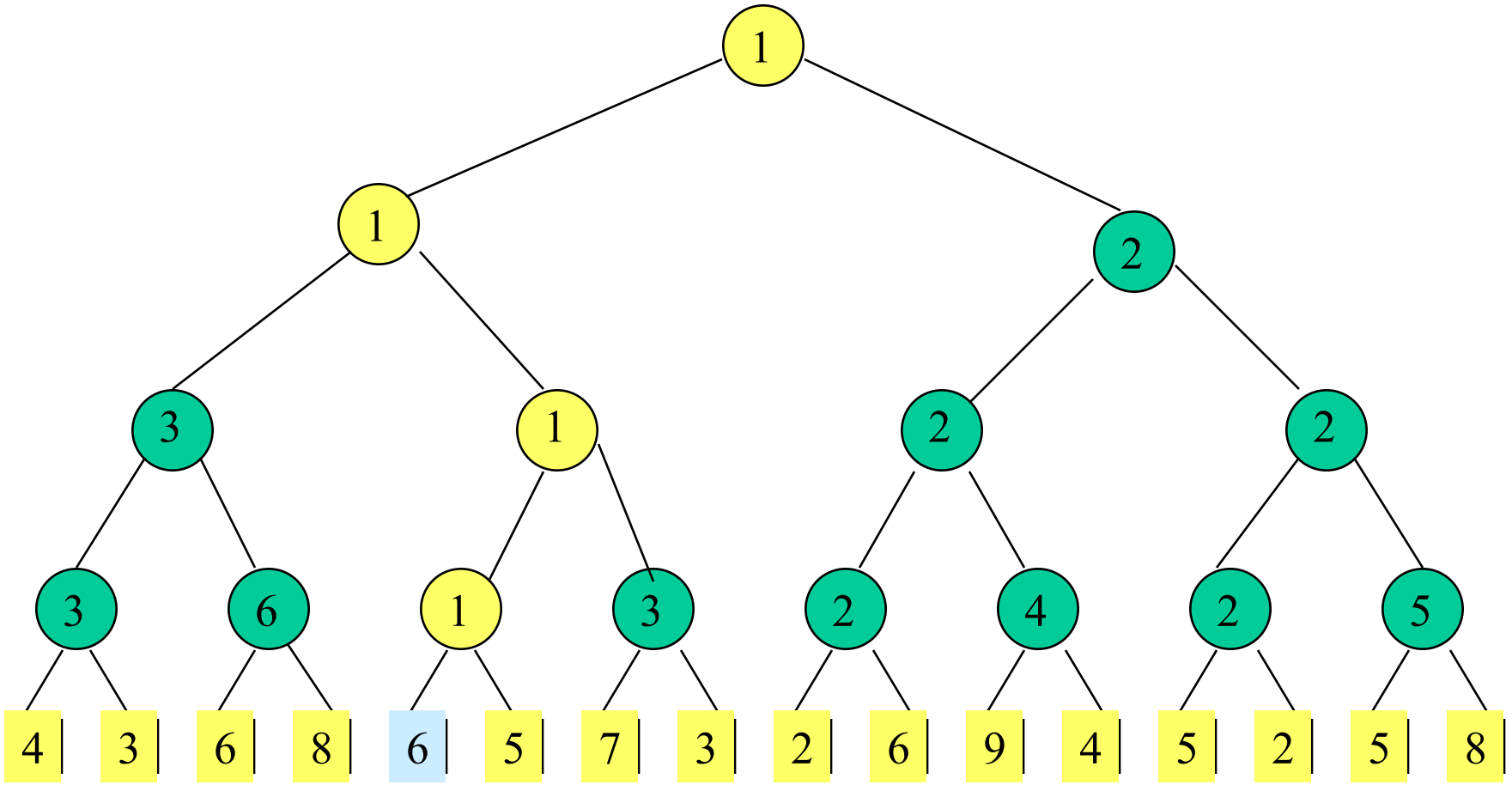
Replace winner with 6.

Replace Winner And Replay



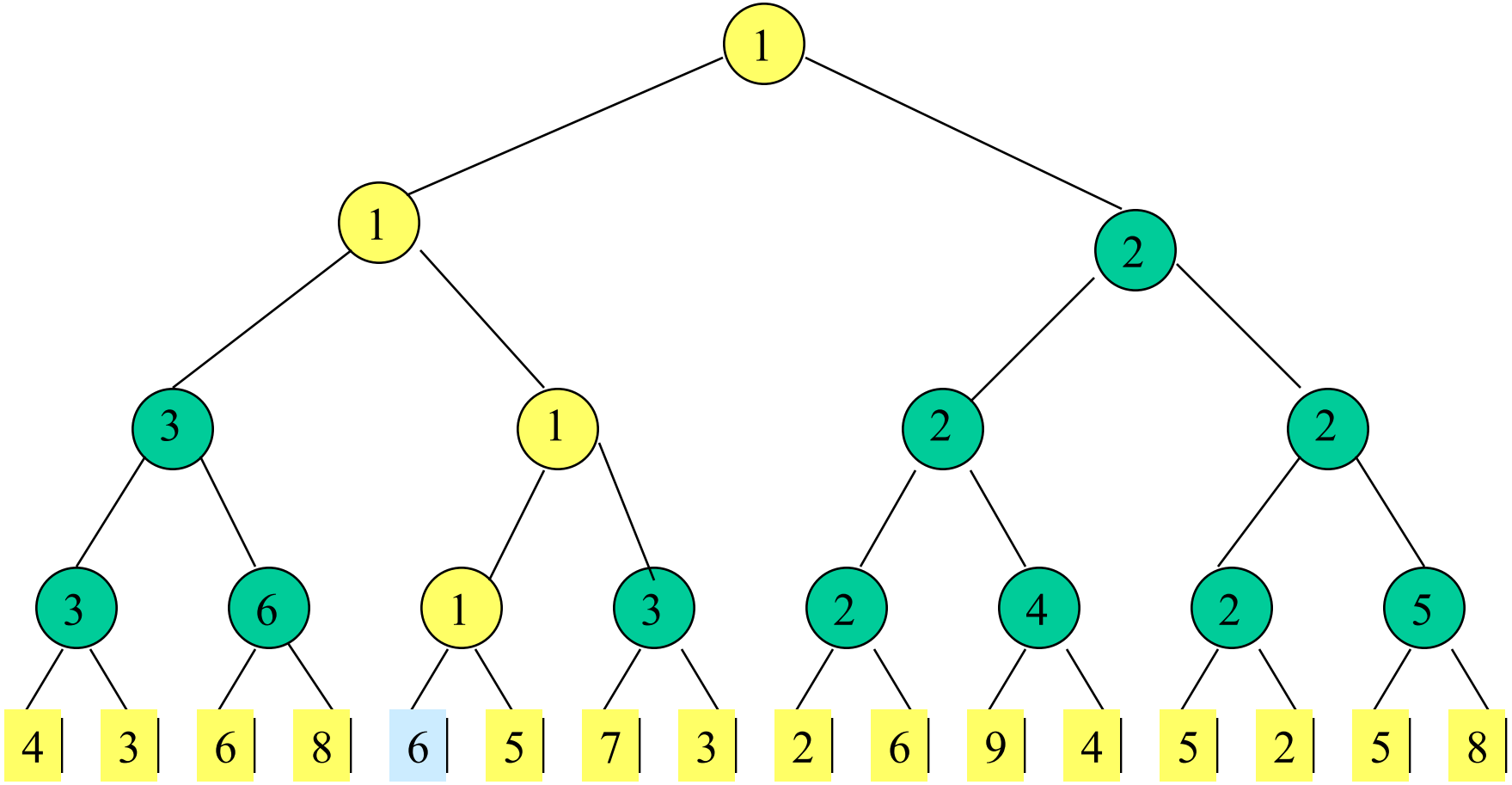
Replay matches on path to root.

Replace Winner And Replay



Replay matches on path to root.

Replace Winner And Replay

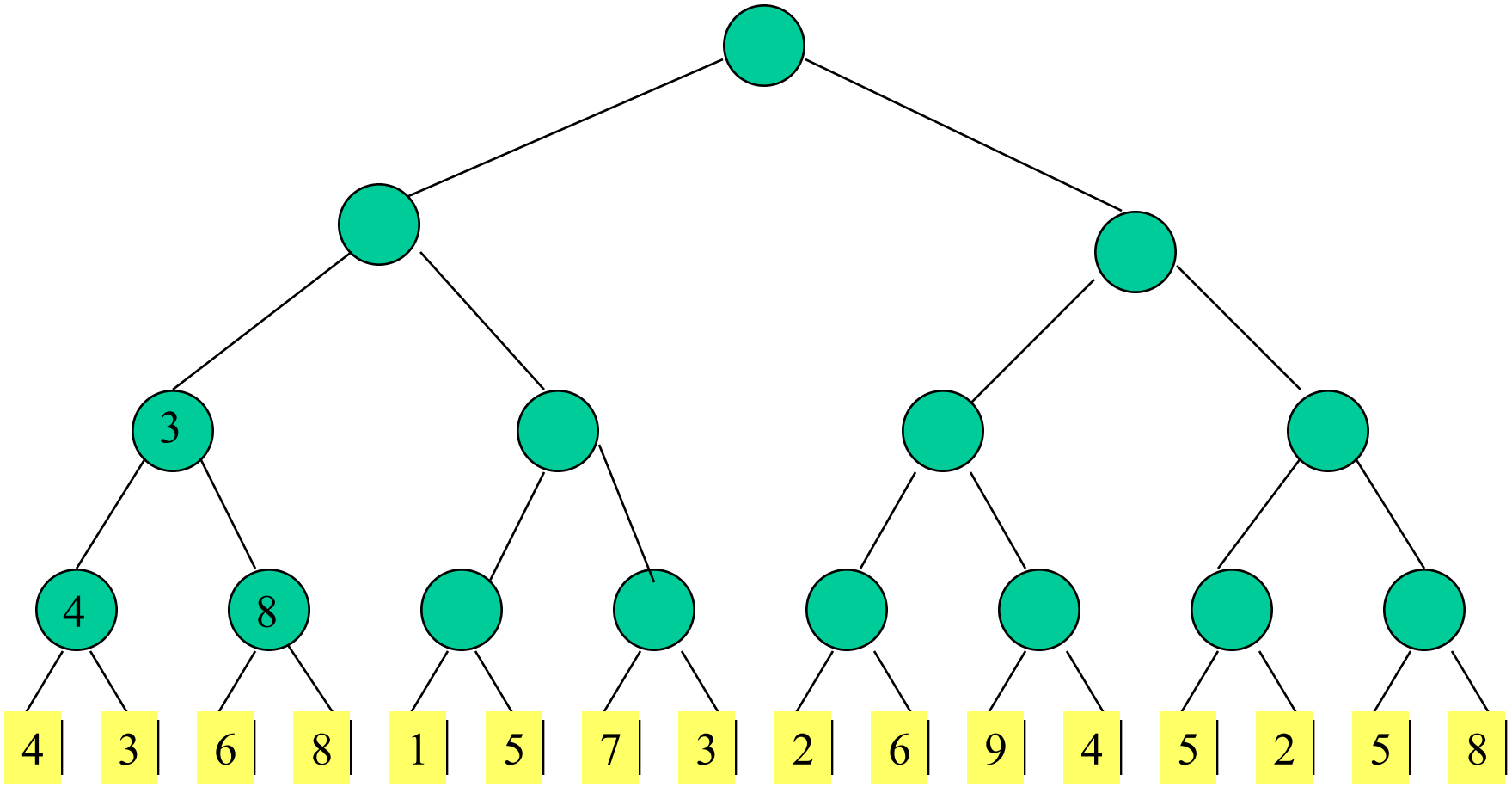


Opponent is player who lost last match played at this node.

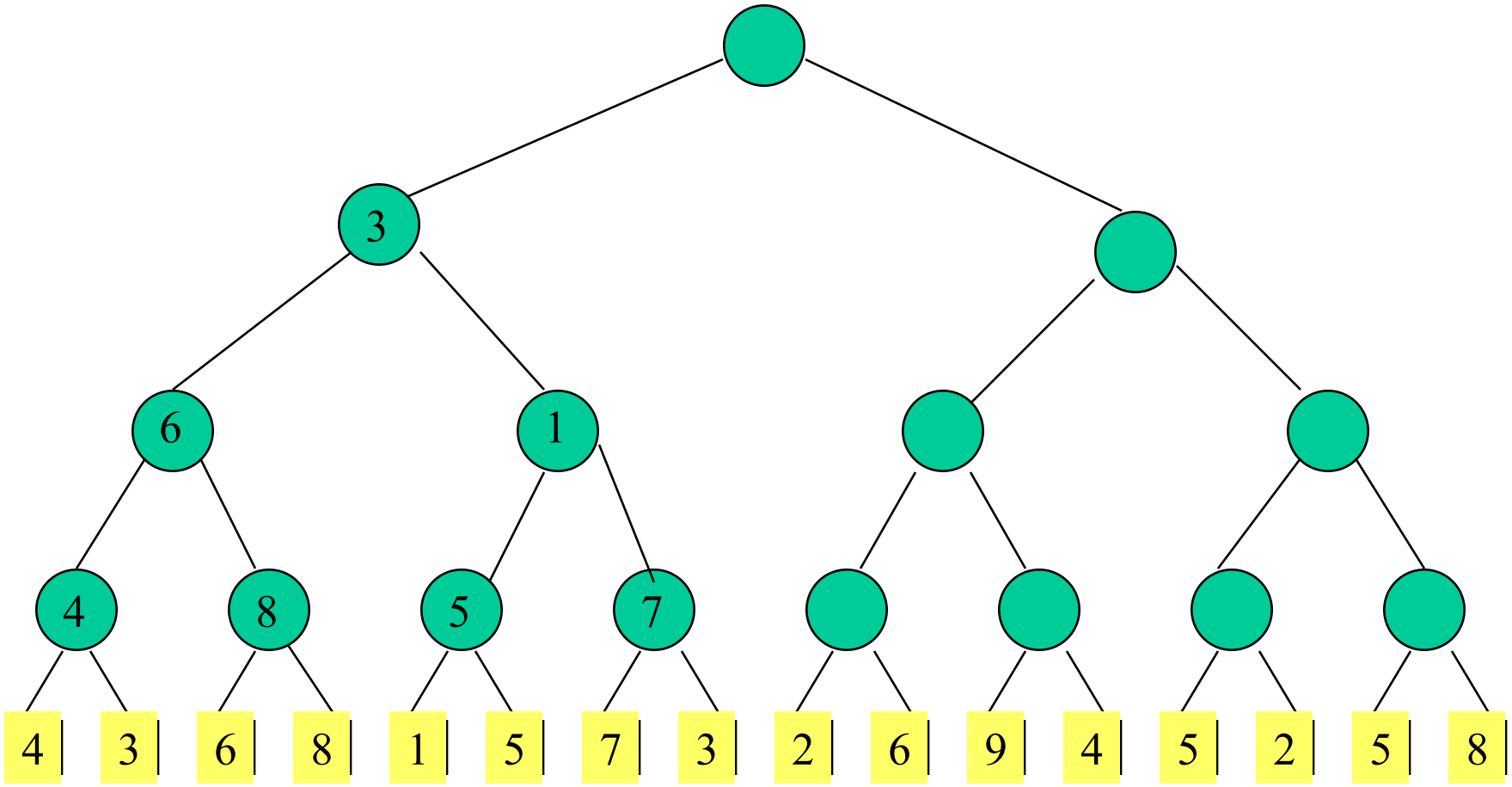
Loser Tree

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

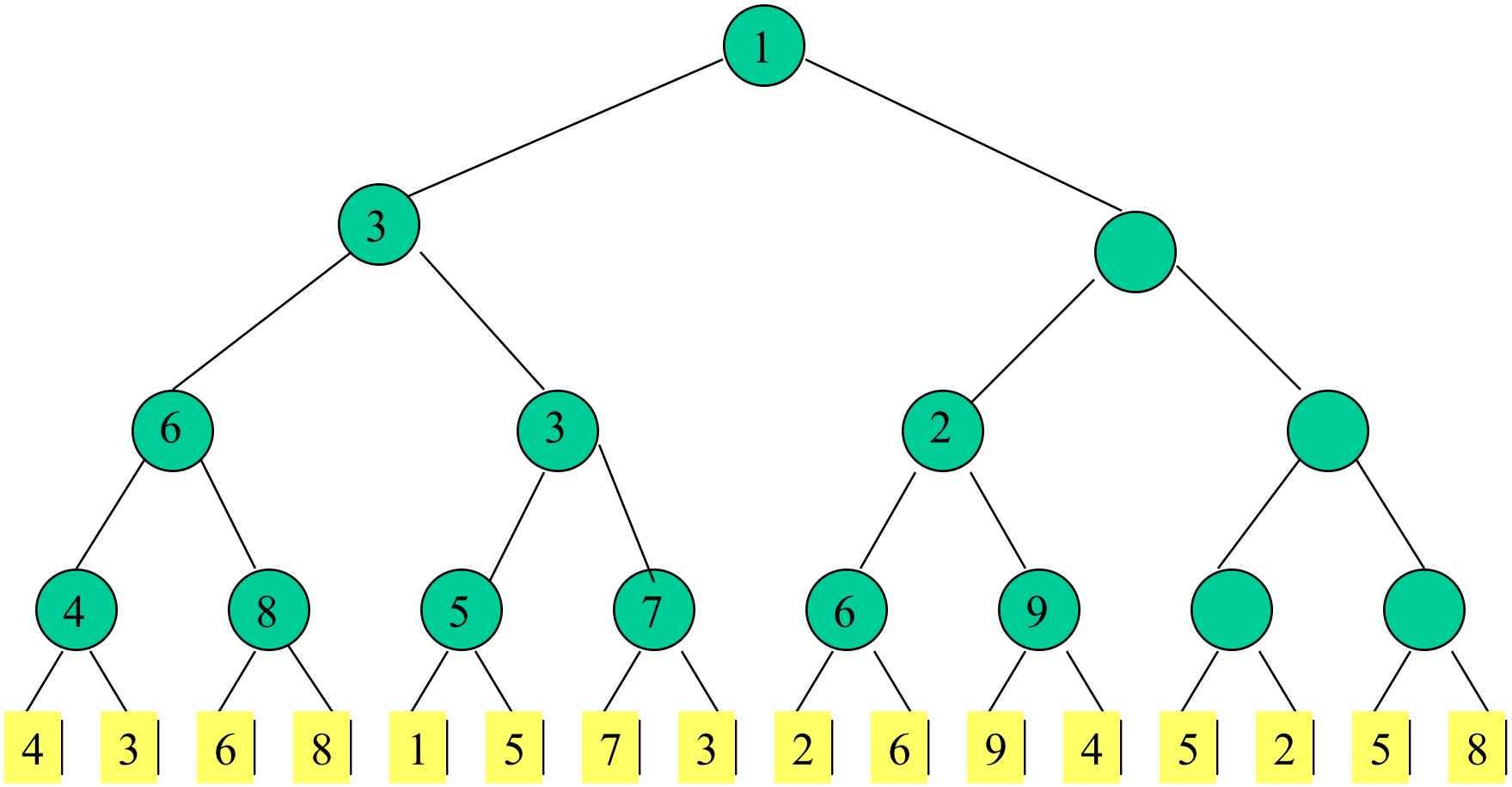
Min Loser Tree For 16 Players



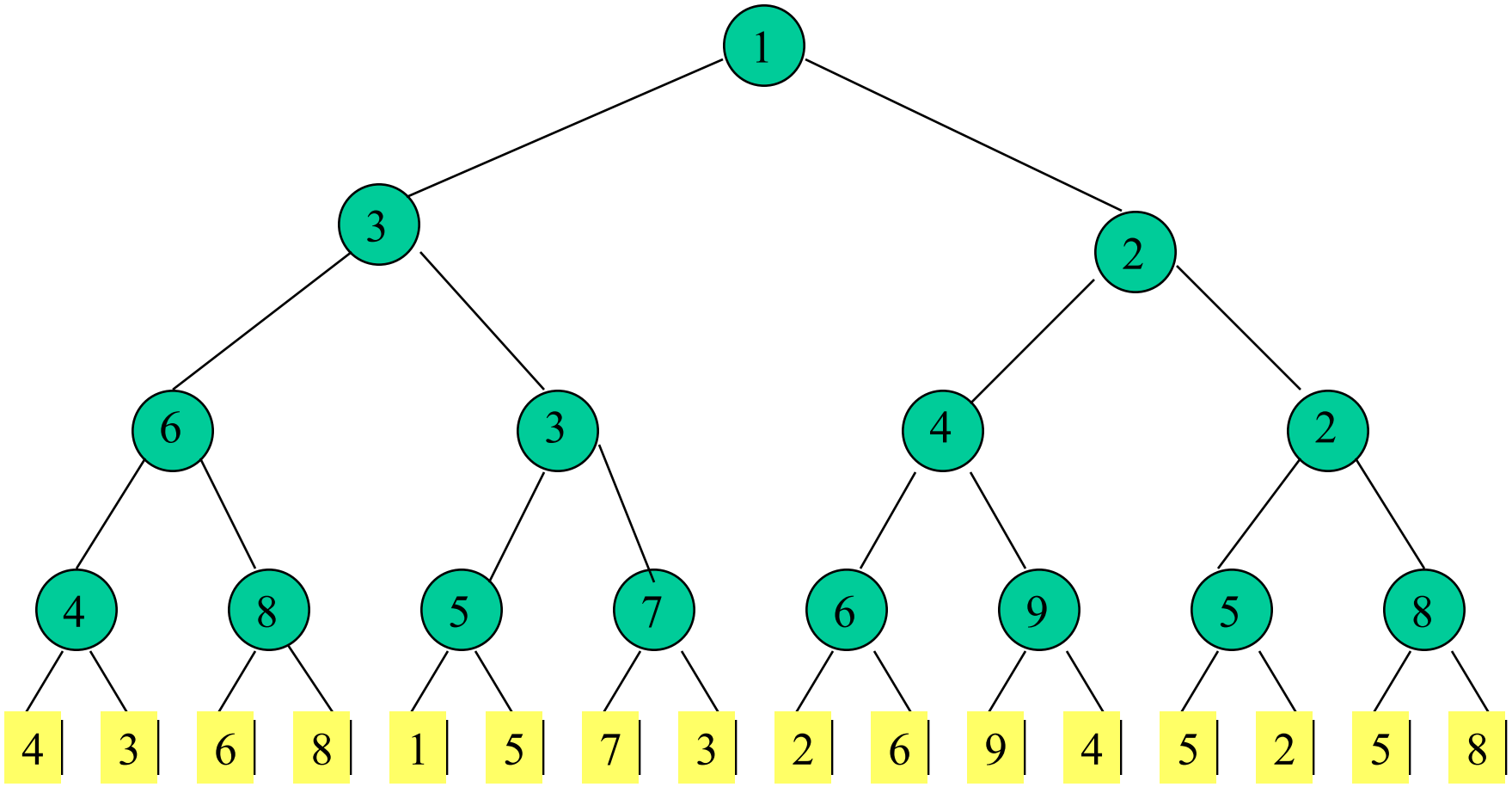
Min Loser Tree For 16 Players



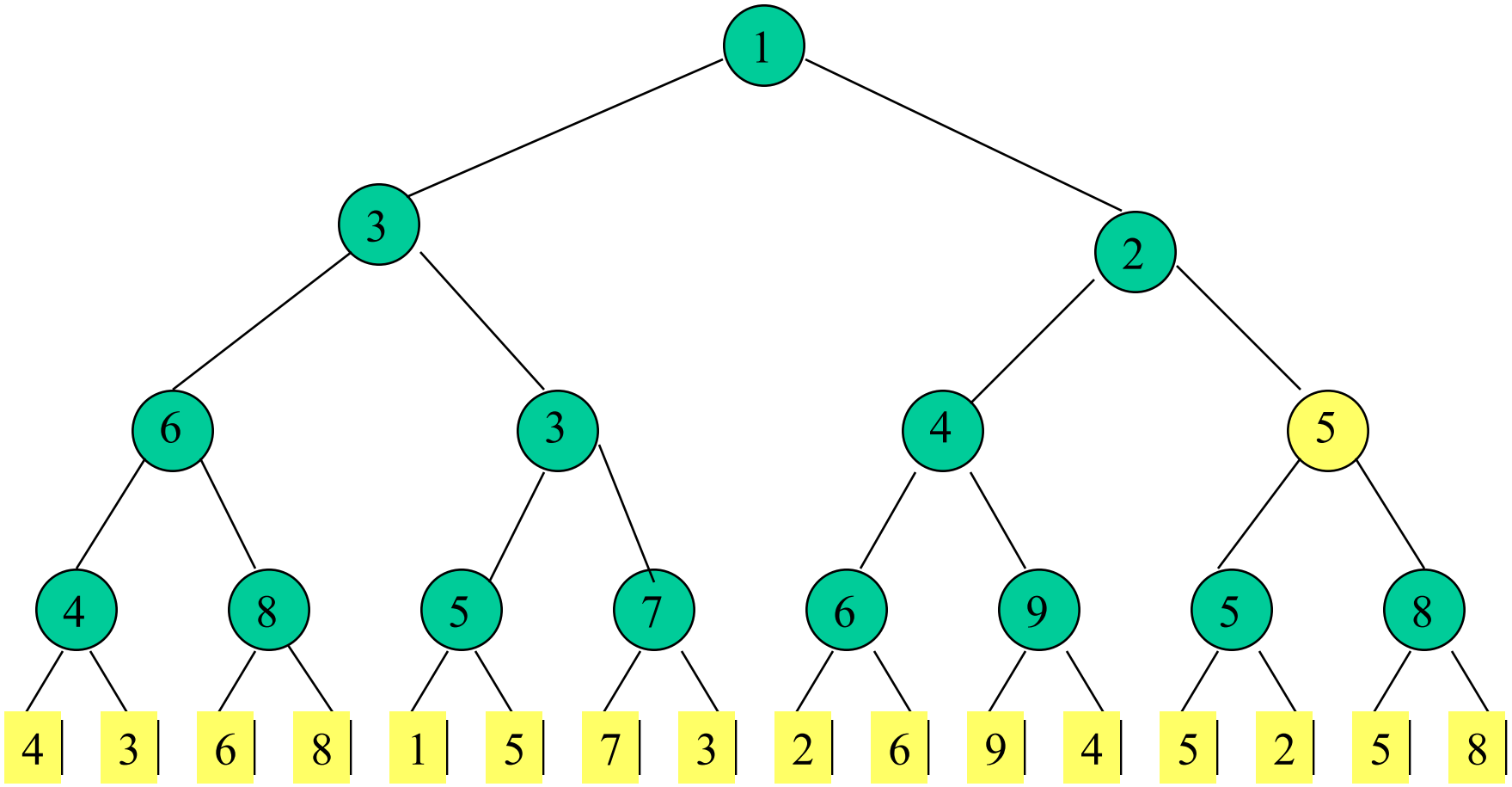
Min Loser Tree For 16 Players



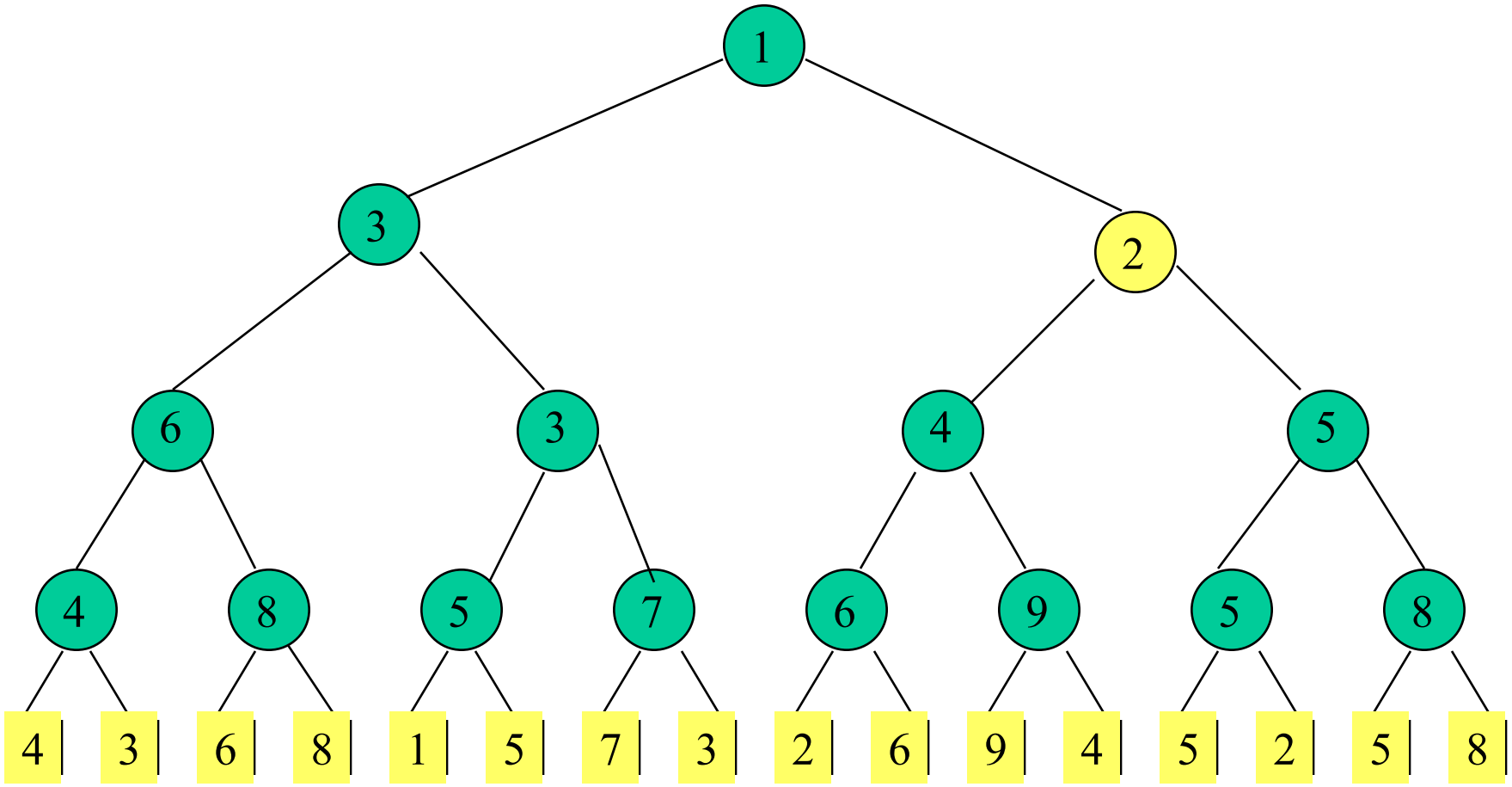
Min Loser Tree For 16 Players



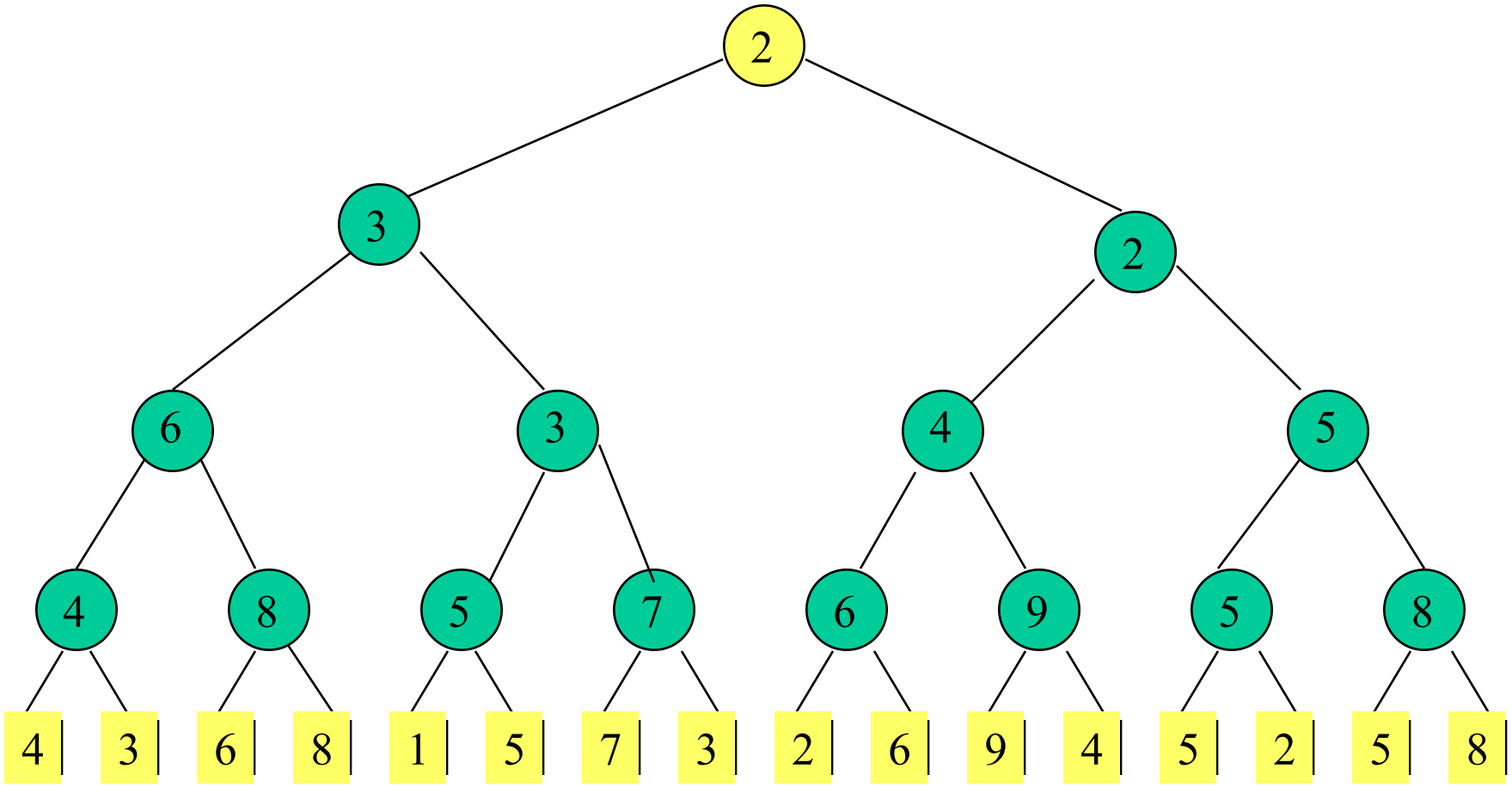
Min Loser Tree For 16 Players

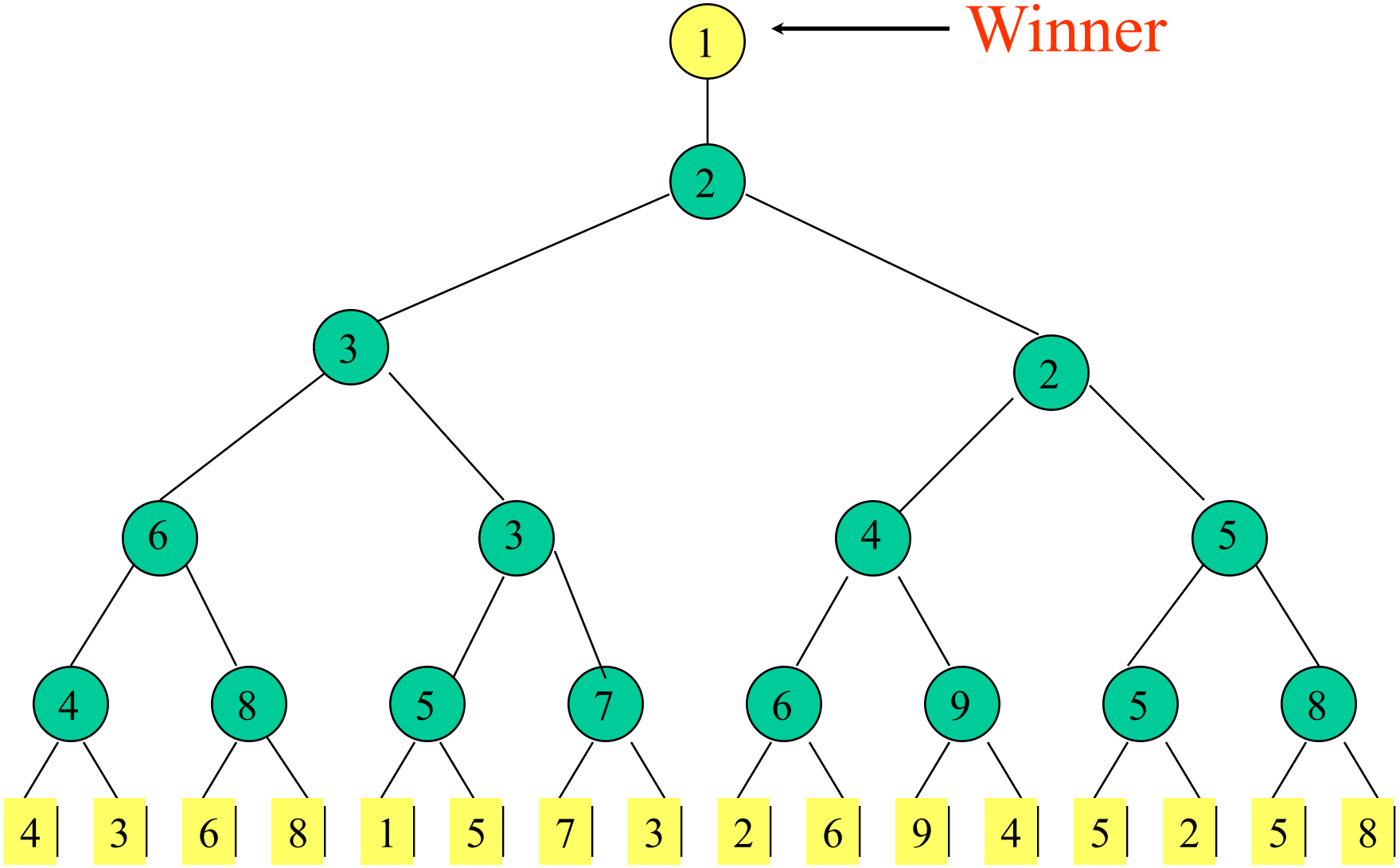


Min Loser Tree For 16 Players



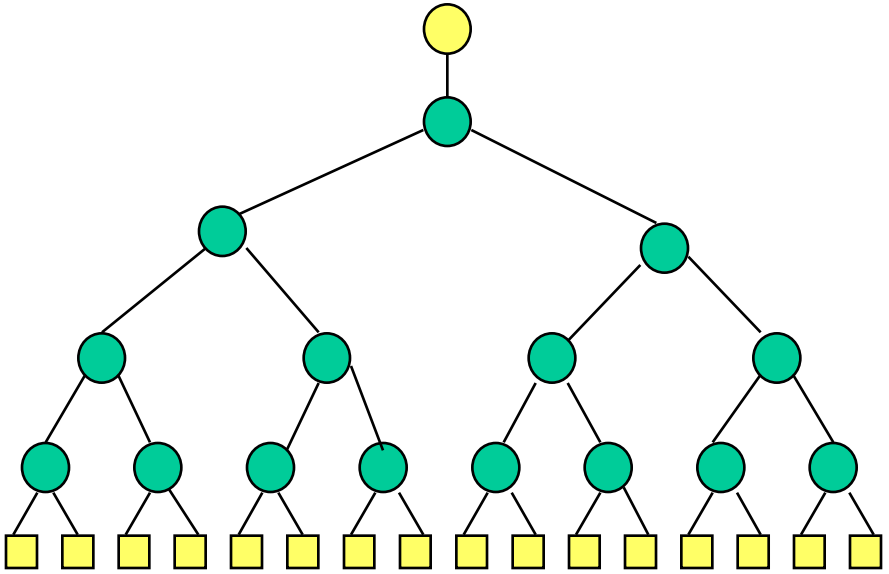
Min Loser Tree For 16 Players

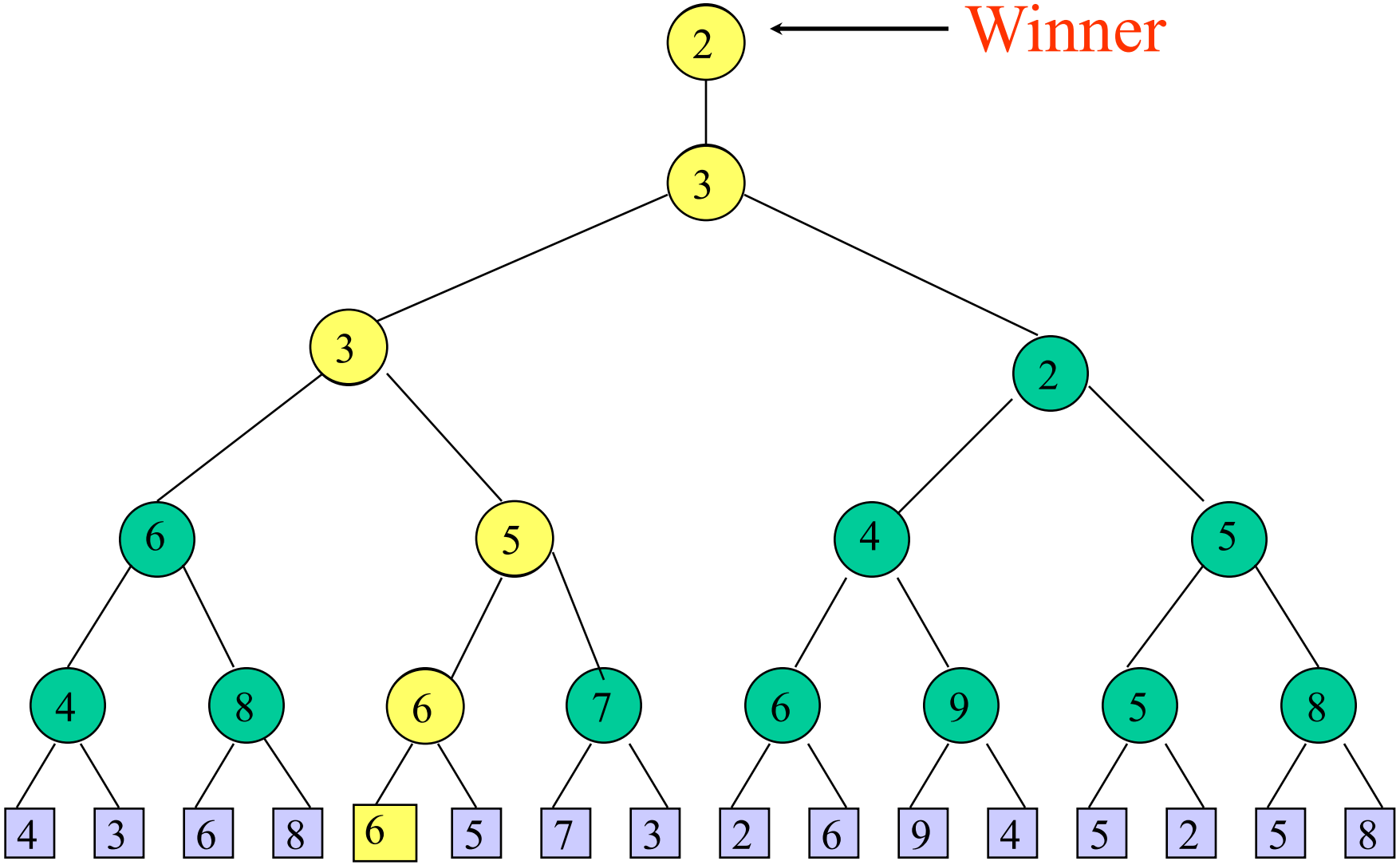



Winner
1
2
3
2
6
3
4
5
4
8
5
7
6
9
5
8
4
3
6
8
1
5
7
3
2
6
9
4
5
2
5
8

Complexity Of Loser Tree Initialize

- Start with 2 credits at each match node.
- Use one to pay for the match played at that node and the storing of the loser.
- Use the other to pay for the store of a left child winner.
- Total time is O(n).
- More precisely Theta(n).





Replace winner with 6 and replay matches.

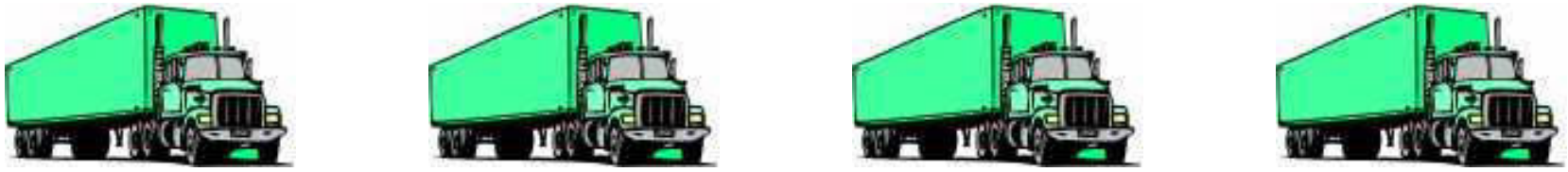
Complexity Of Replay

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

Tournament Tree Applications

- k-way merging of runs during an external merge sort.
- Truck loading.
- Run generation.
- …

Truck Loading



- n packages to be loaded into trucks
- each package has a weight
- each truck has a capacity of c tons
- minimize number of trucks

Bin Packing

- n items to be packed into bins
- each item has a size
- each bin has a capacity of c
- minimize number of bins

Bin Packing

Truck loading is same as bin packing.

- Truck is a bin that is to be packed (loaded).
- Package is an item/element.

Bin packing to minimize number of bins is NP-hard.

Several fast heuristics have been proposed.

Bin Packing Heuristics

- First Fit.
  - Bins are arranged in left to right order.
  - Items are packed one at a time in given order.
  - Current item is packed into leftmost bin into which it fits.
  - If there is no bin into which current item fits, start a new bin.

Bin Packing Heuristics

- First Fit Decreasing.
  - Items are sorted into decreasing order.
  - Then first fit is applied.

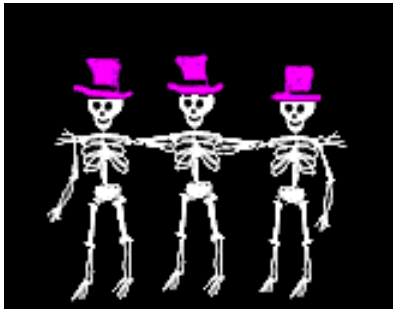
Bin Packing Heuristics

- Best Fit.
  - Items are packed one at a time in given order.
  - To determine the bin for an item, first determine set S of bins into which the item fits.
  - If S is empty, then start a new bin and put item into this new bin.
  - Otherwise, pack into bin of S that has least available capacity.

Bin Packing Heuristics

- Best Fit Decreasing.
  - Items are sorted into decreasing order.
  - Then best fit is applied.

Performance



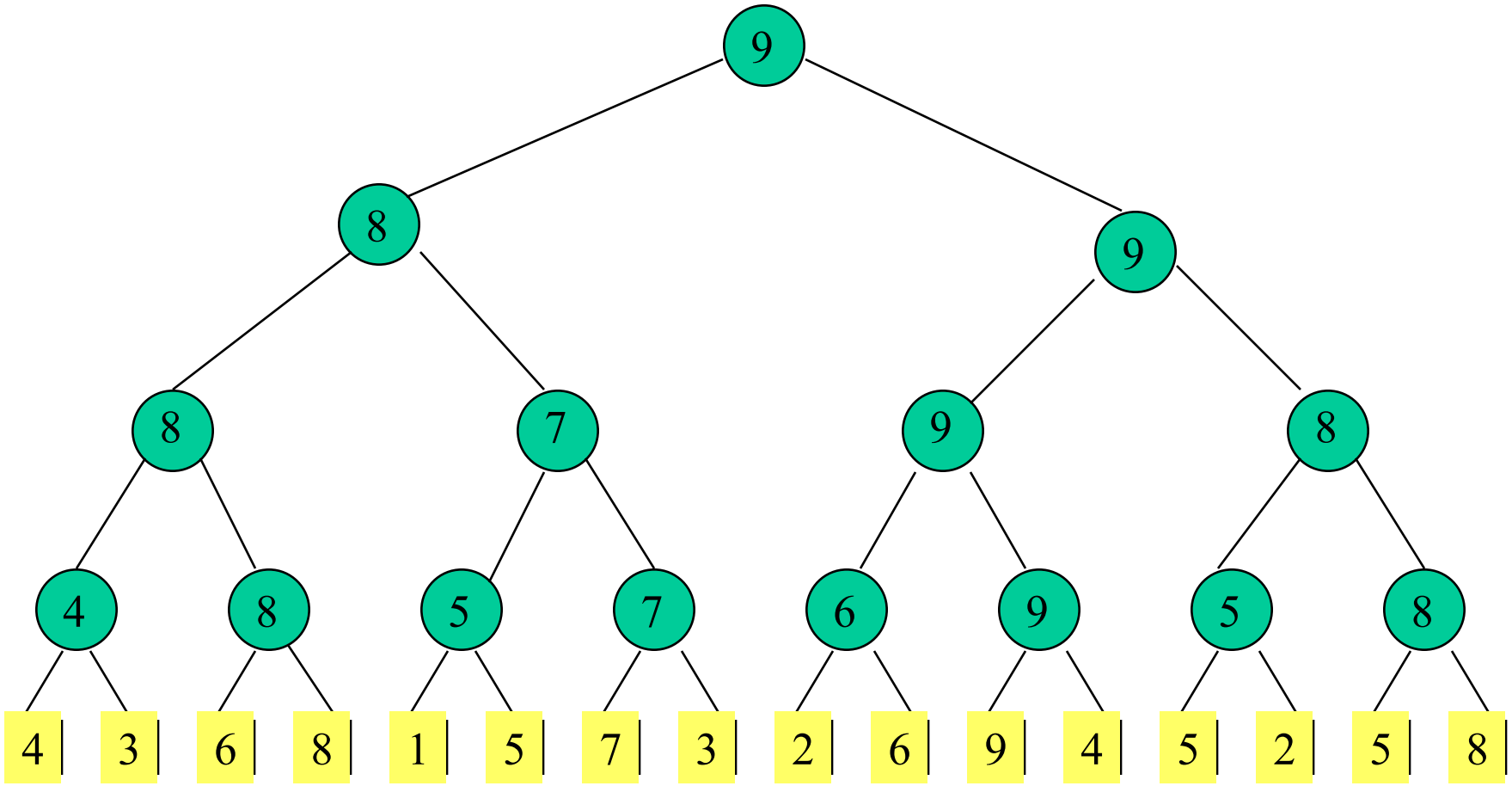
- For first fit and best fit:

  Heuristic Bins <= (17/10)(Minimum Bins) + 2

- For first fit decreasing and best fit decreasing:

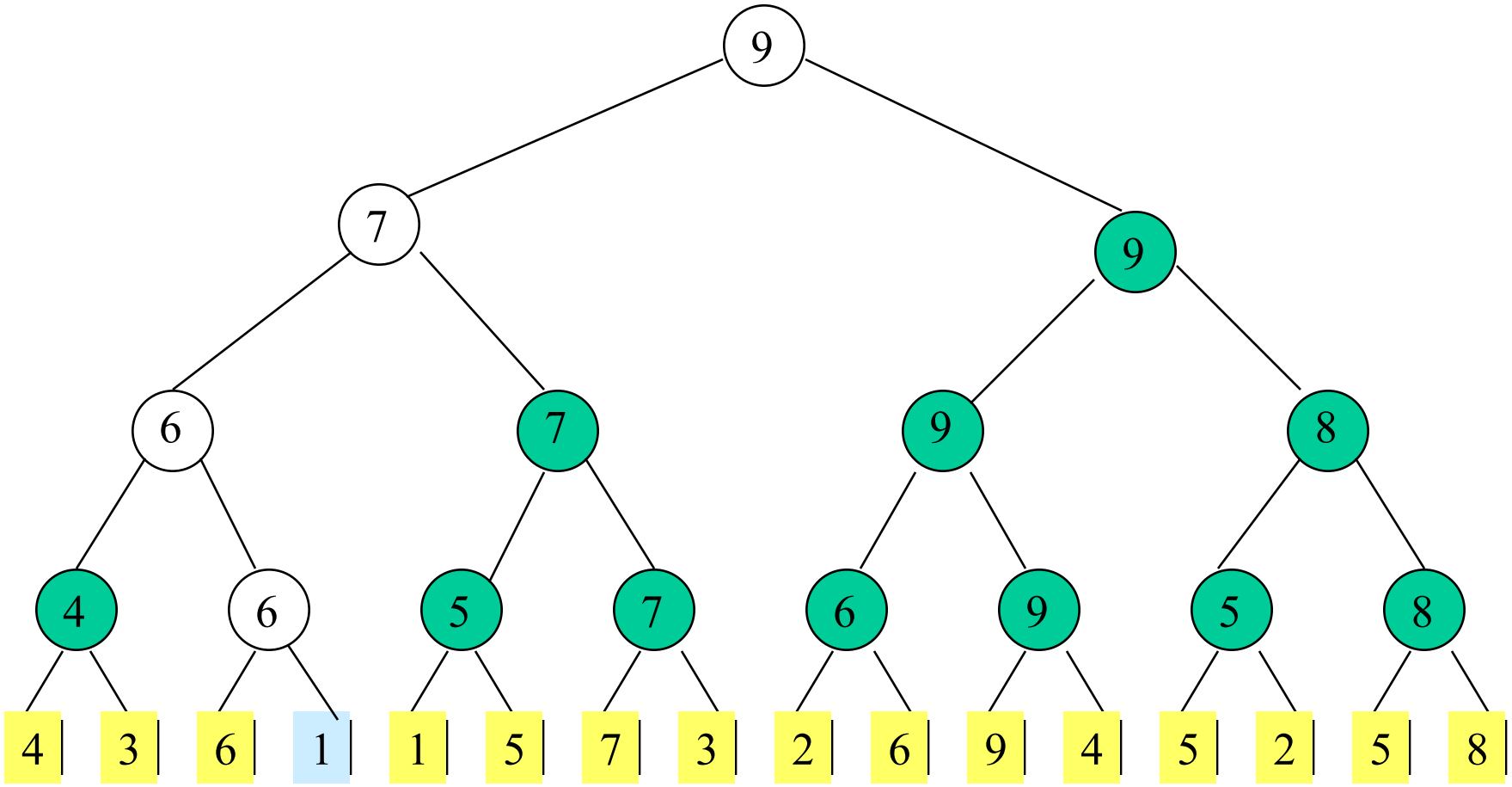
  Heuristic Bins <= (11/9)(Minimum Bins) + 4

Max Winner-Tree For 16 Bins



Item size = 7

Max Winner-Tree For 16 Bins

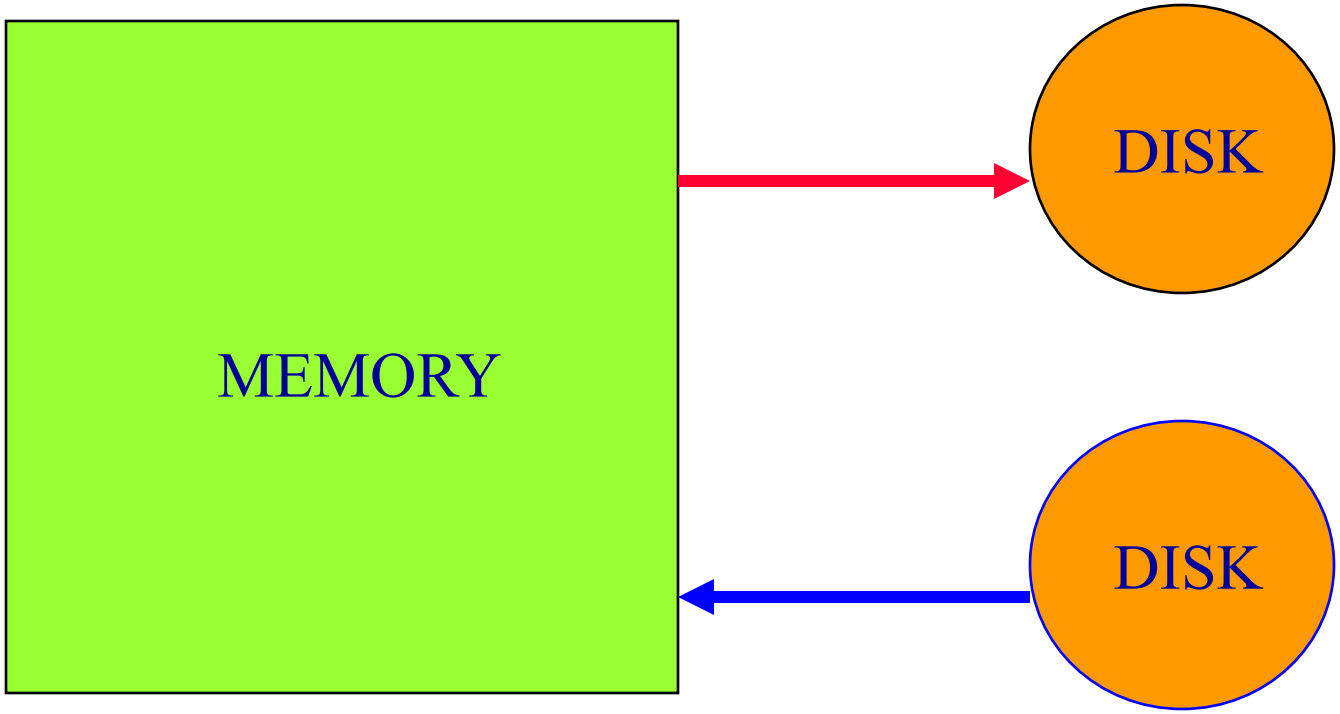


Complexity Of First Fit

$O(n \log n)$, where n is the number of items.

Improve Run Generation

- Overlap input,output, and internal CPU work.
- Reduce the number of runs (equivalently, increase average run length).



Internal Quick Sort

Use 6 as the pivot (median of 3).

Input first, middle, and last blocks first.

In-place partitioning.

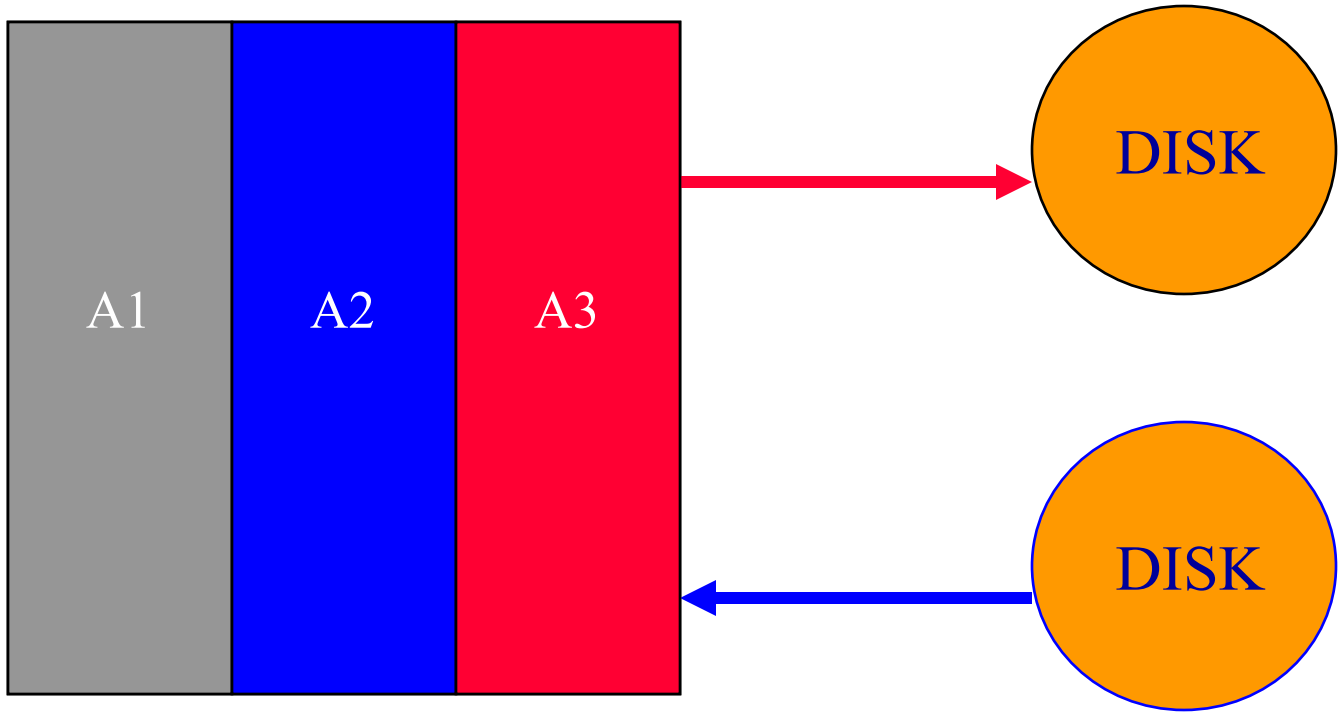
Input blocks from the ends toward the middle.

Sort left and right groups recursively.

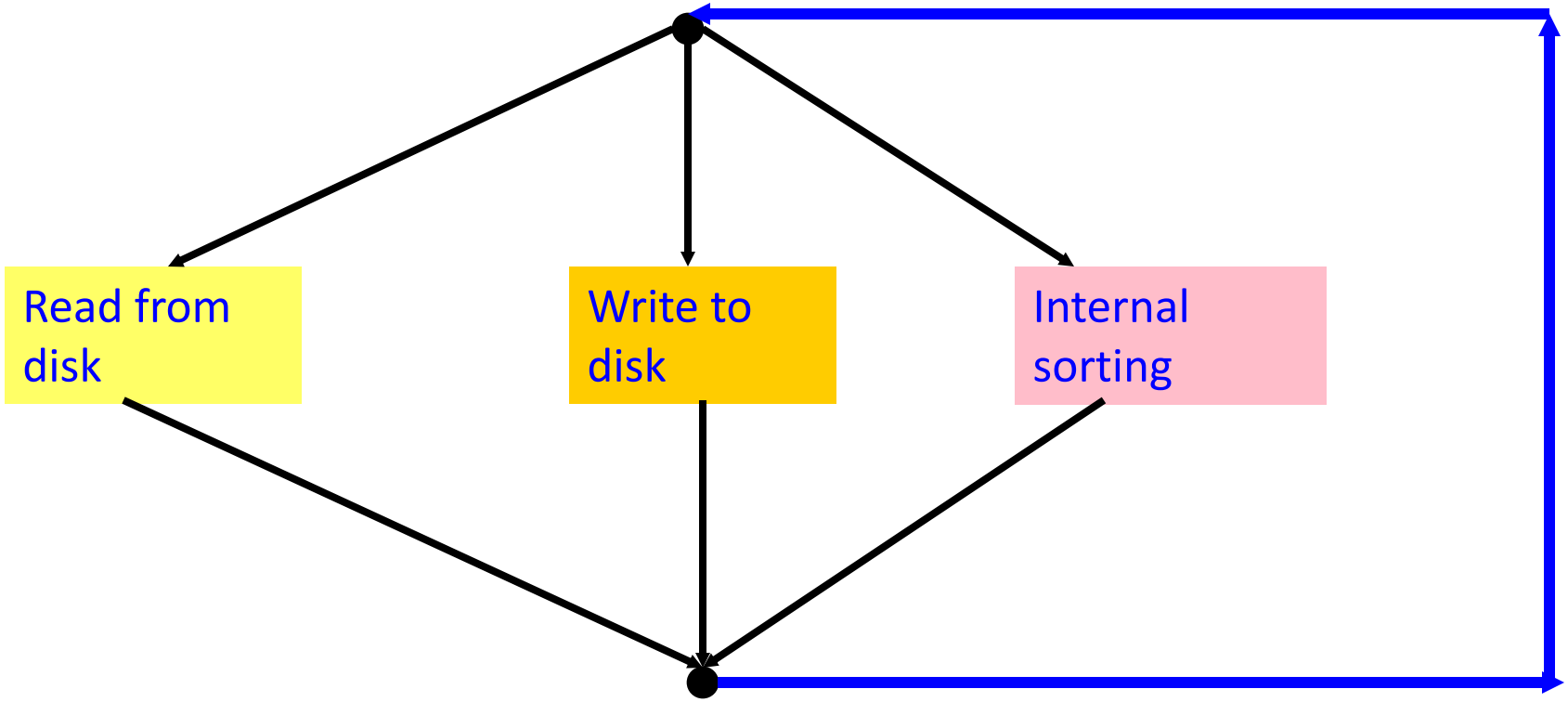
Can begin output as soon as leftmost block is ready.

Alternative Internal Sort Scheme

Partition into 3 areas, each may be more than 1 block in size.

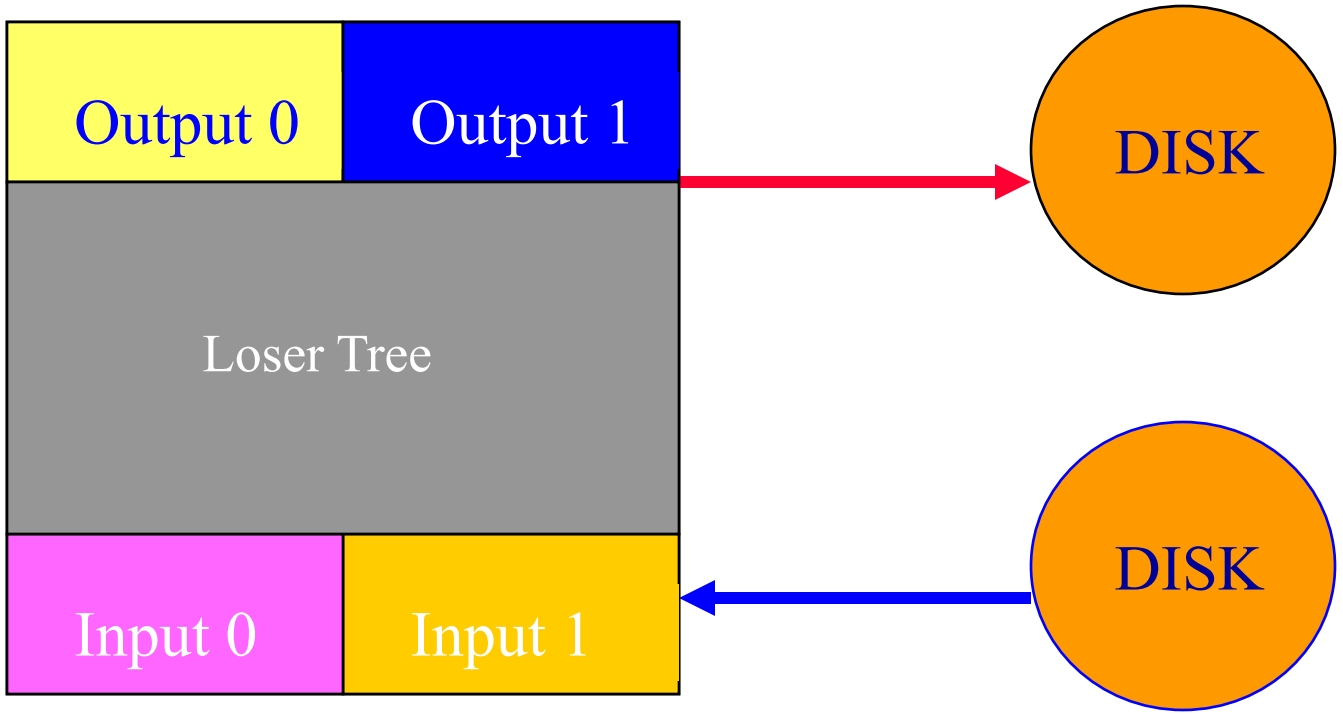


Steady State Operation



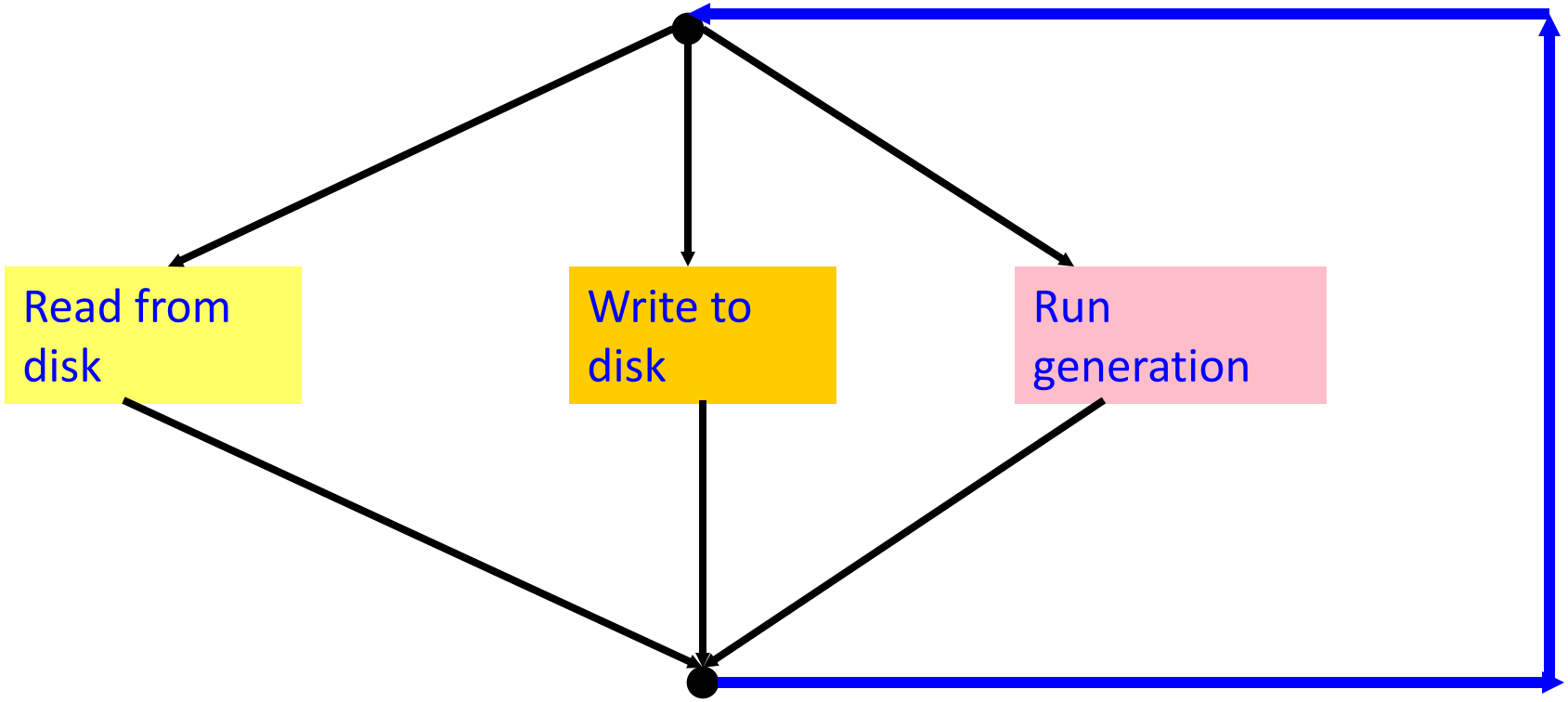
- Synchronization is done when the current internal sort terminates.

New Strategy



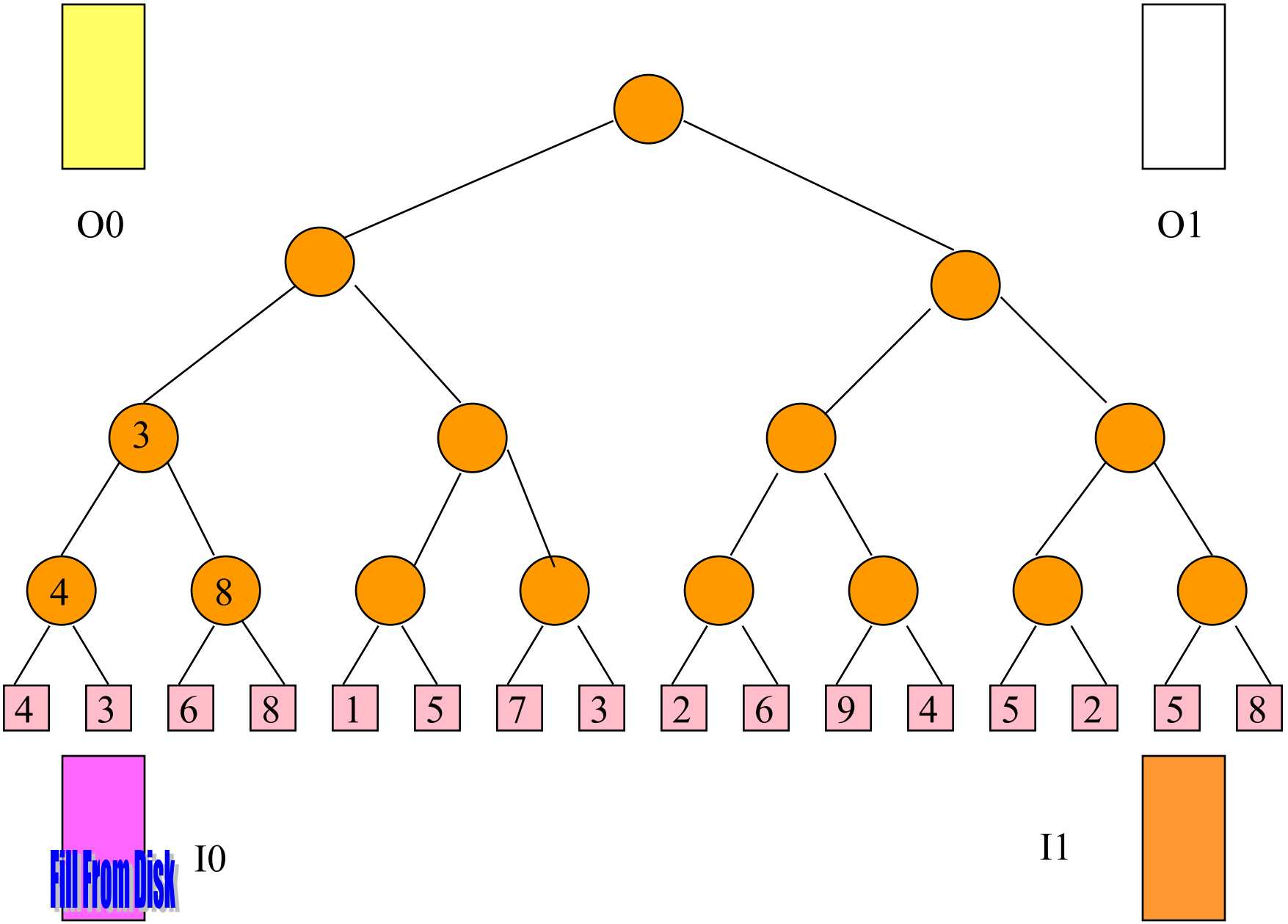
- Use 2 input and 2 output buffers.
- Rest of memory is used for a min loser tree.
- Actually, 3 buffers adequate.

Steady State Operation

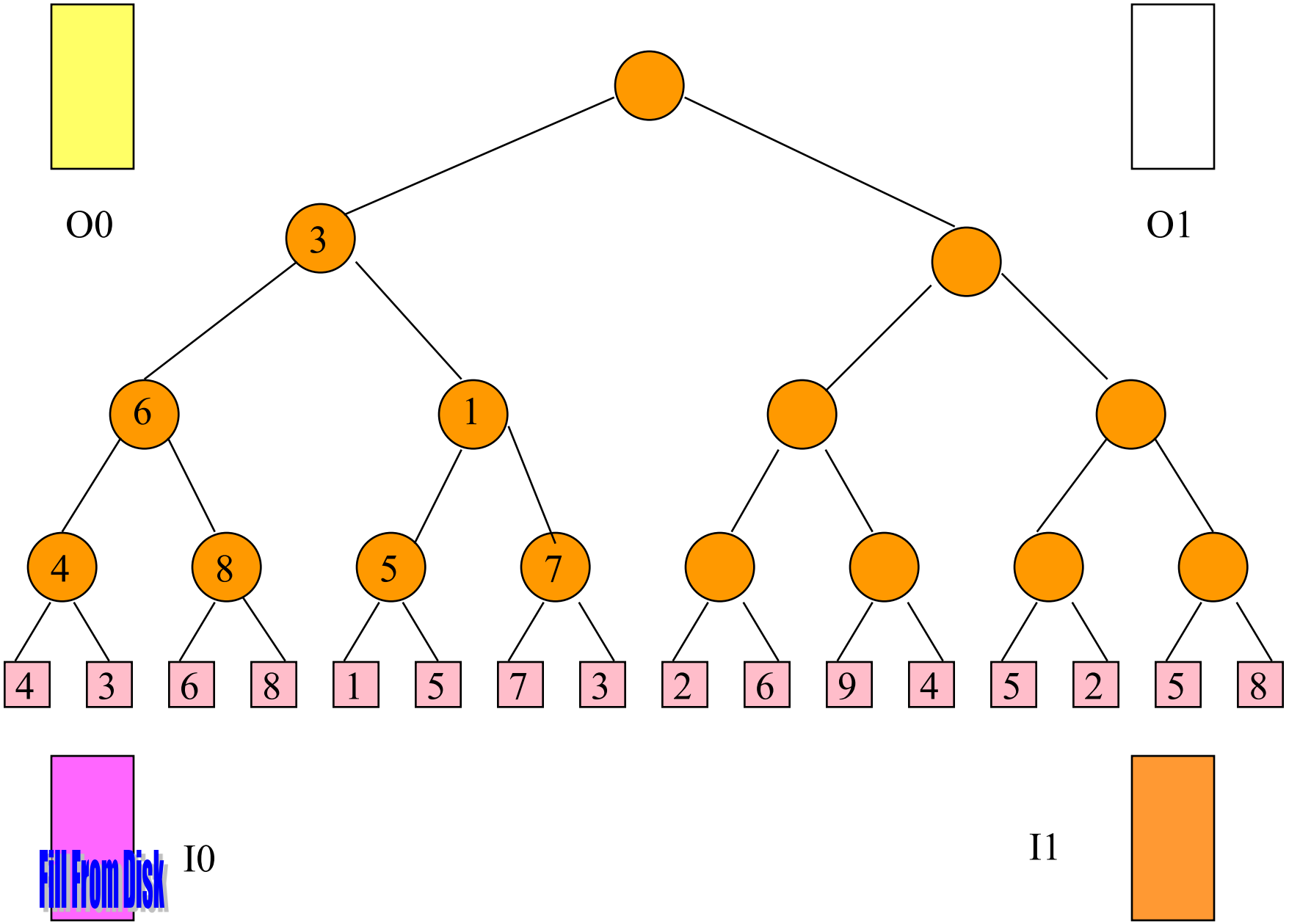


- Synchronization is done when the active input buffer gets empty (the active output buffer will be full at this time).

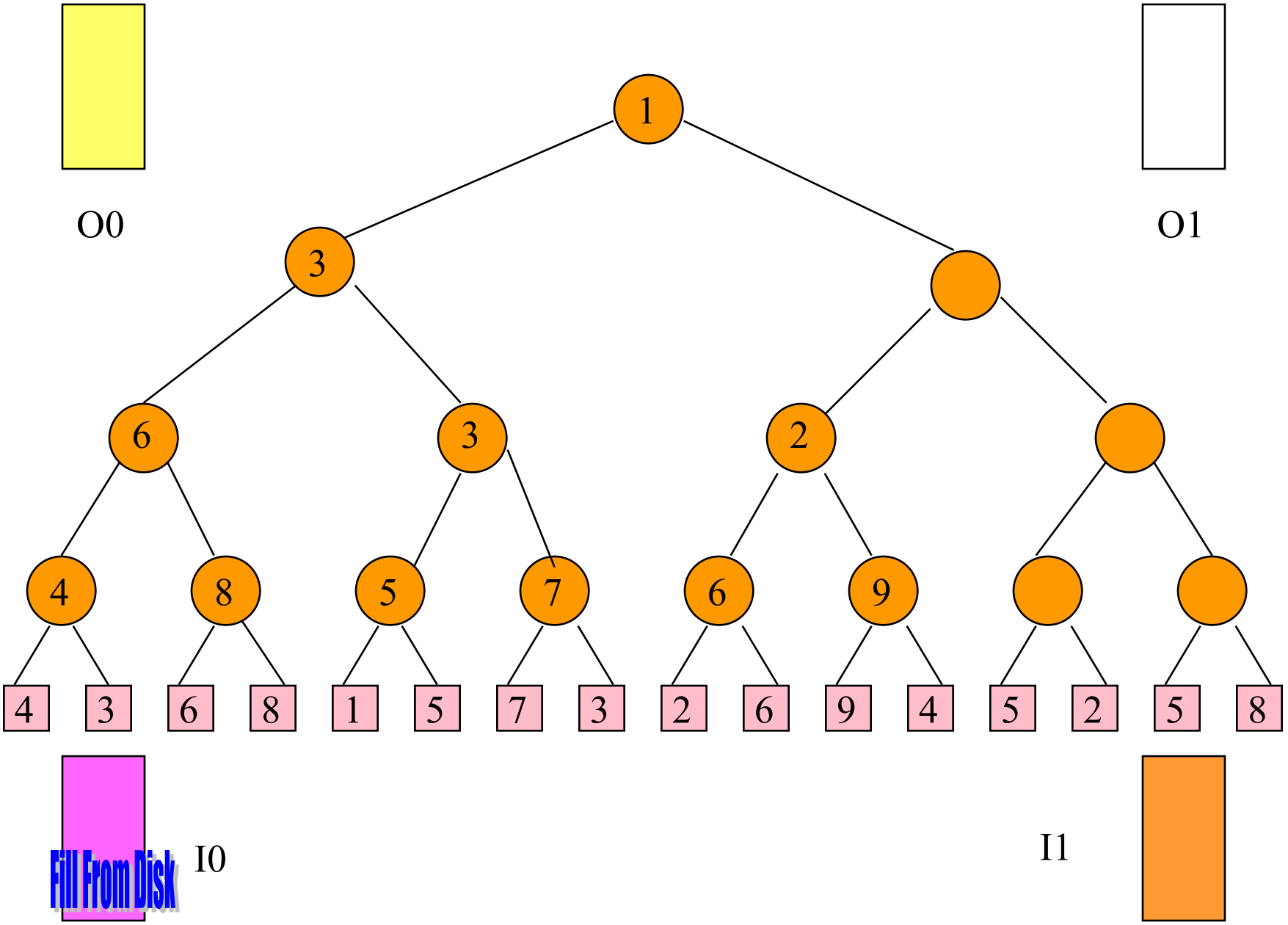
Initialize



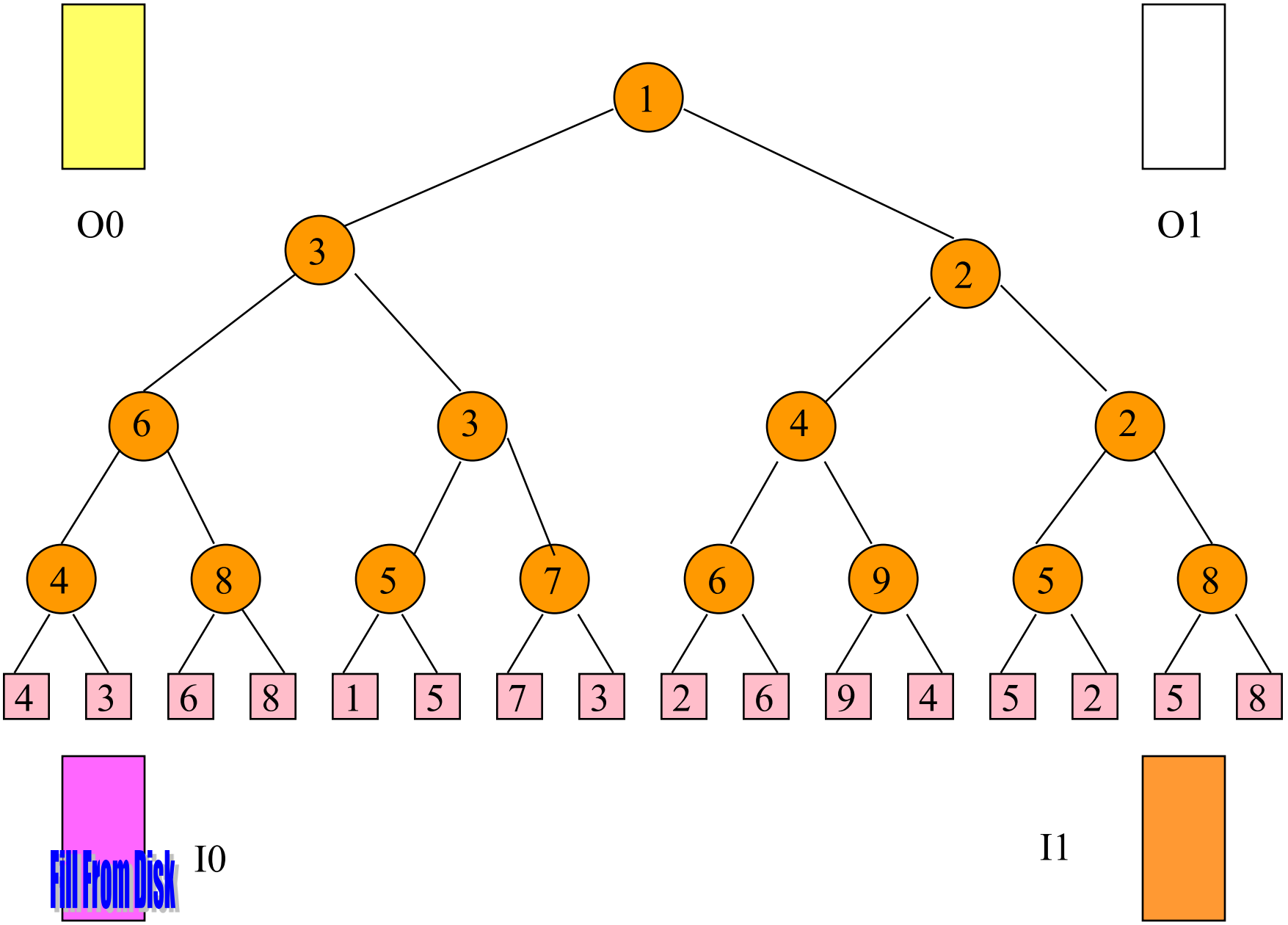
Initialize



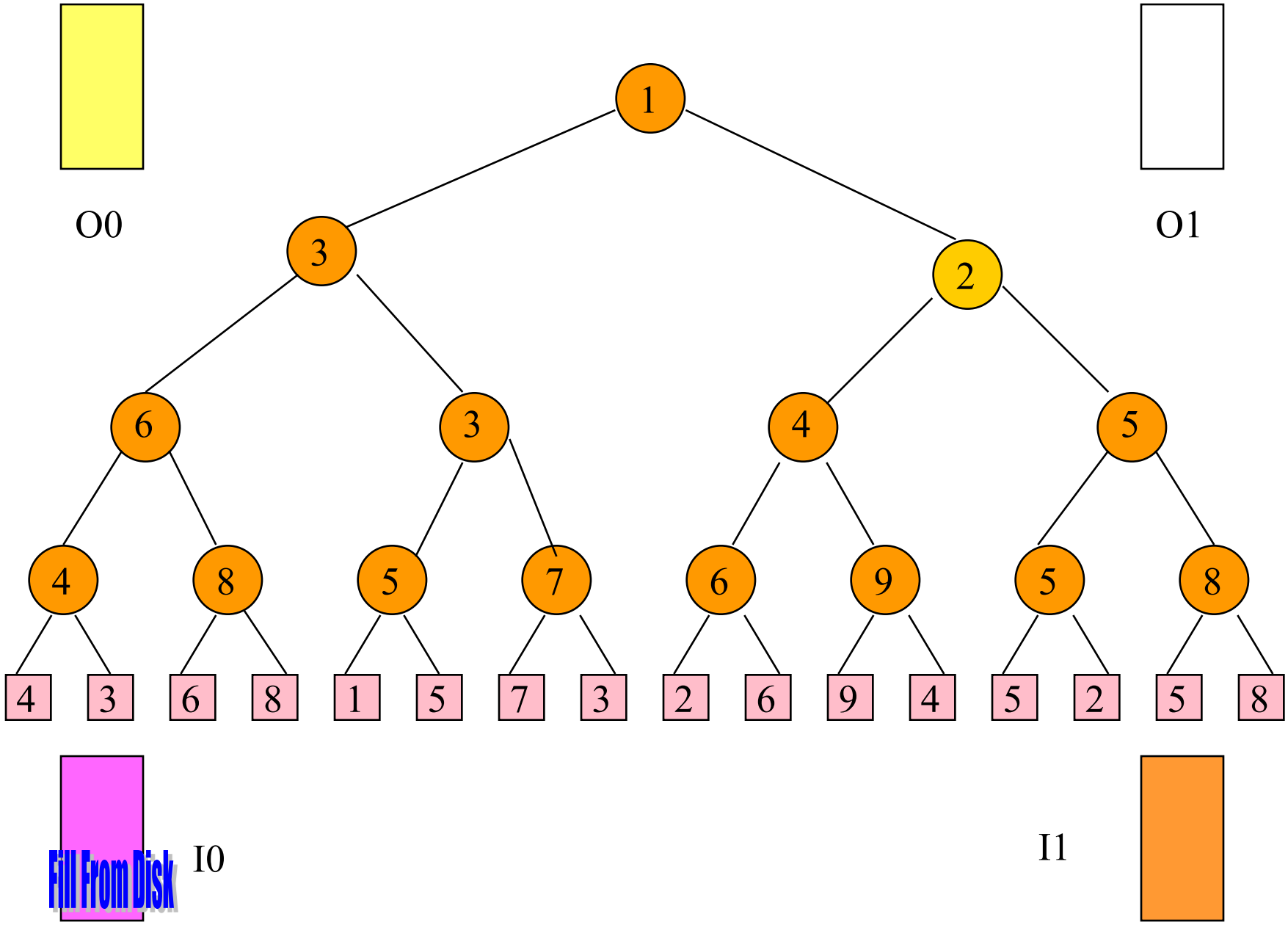
Initialize



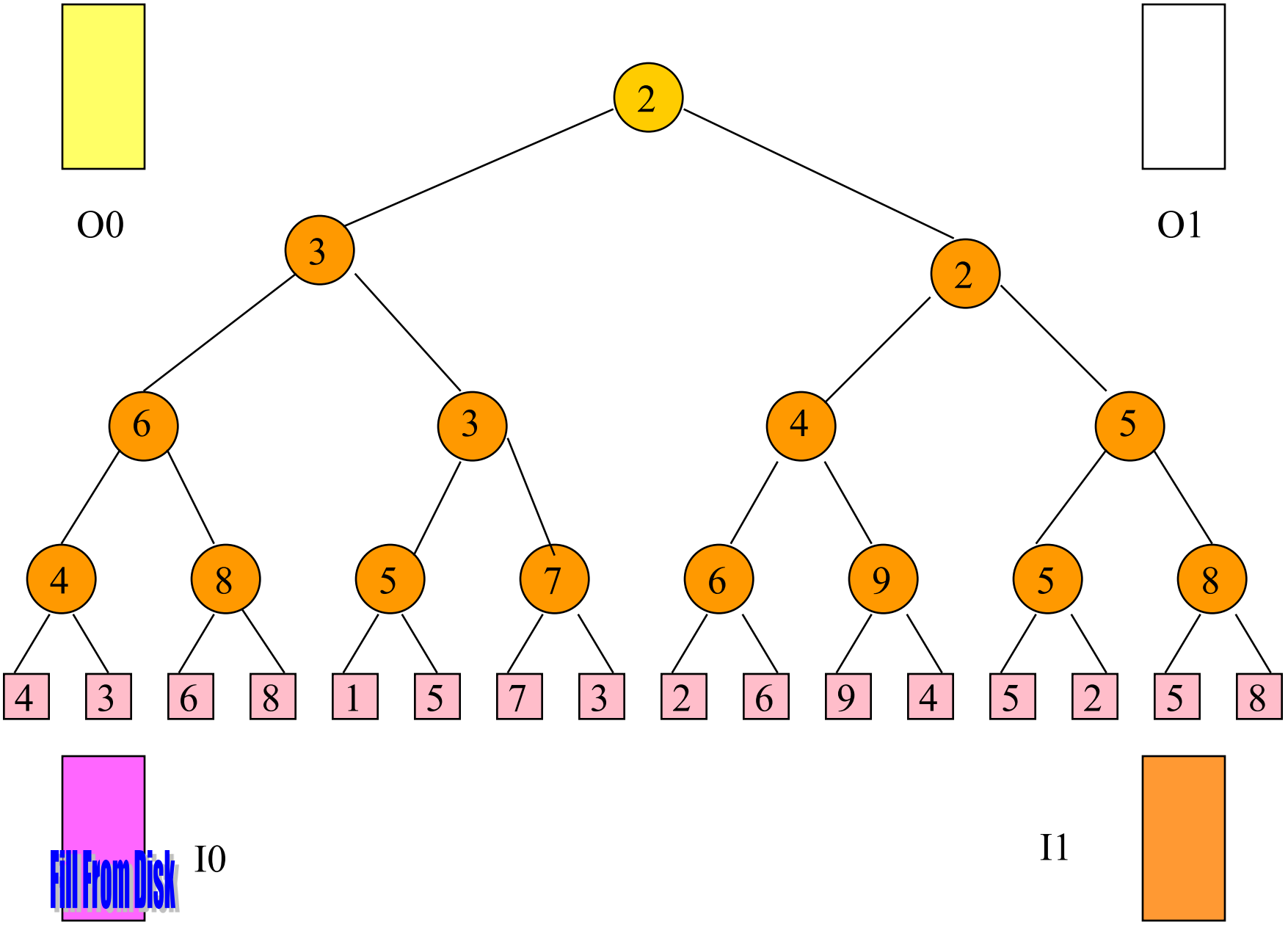
Initialize


O0
O1
1
3
2
6
3
4
2
4
8
5
7
6
9
5
8
4
3
6
8
1
5
7
3
2
6
9
4
5
2
5
8
Fill From Disk
I0
I1

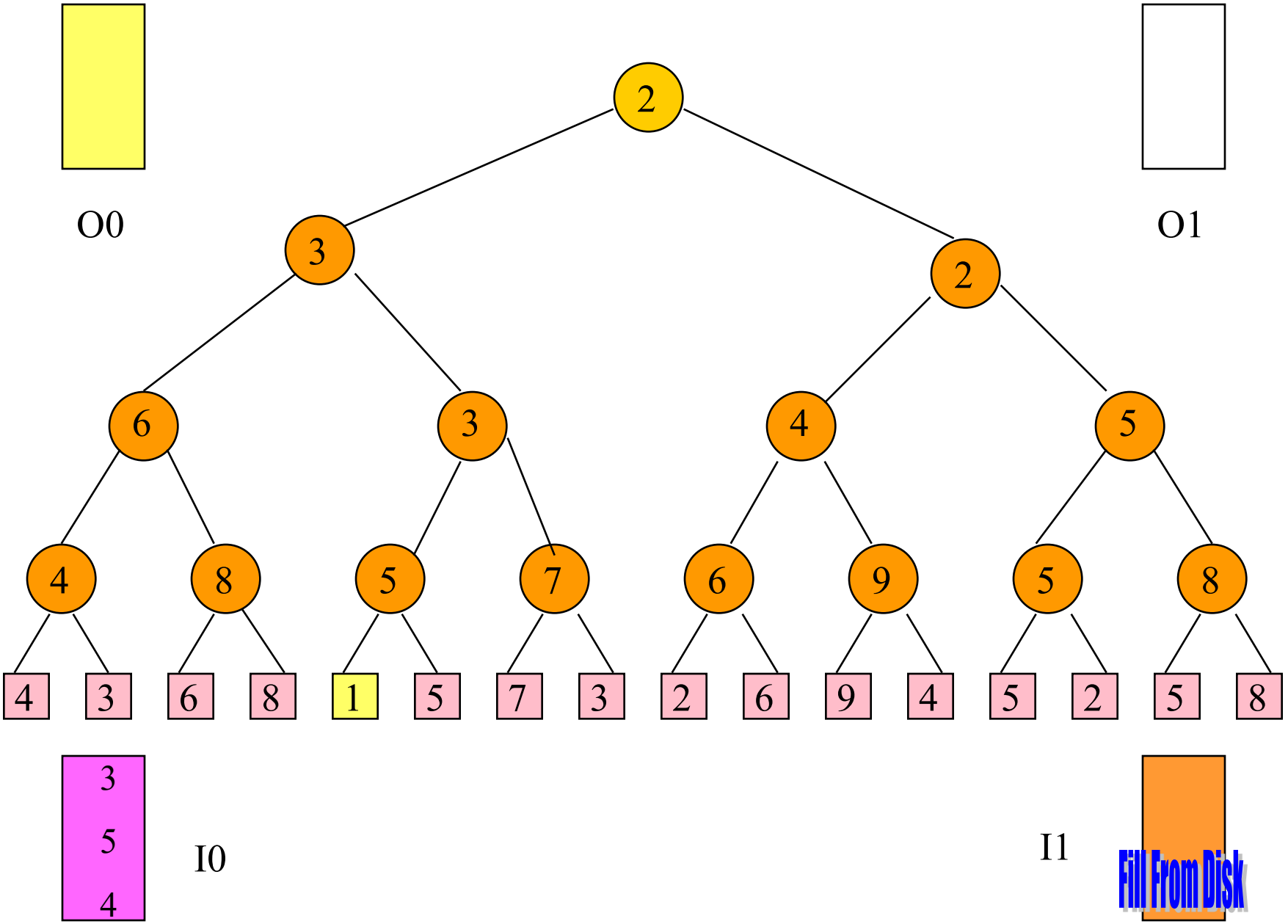
Initialize


O0
O1
1
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
Fill From Disk
I0
I1

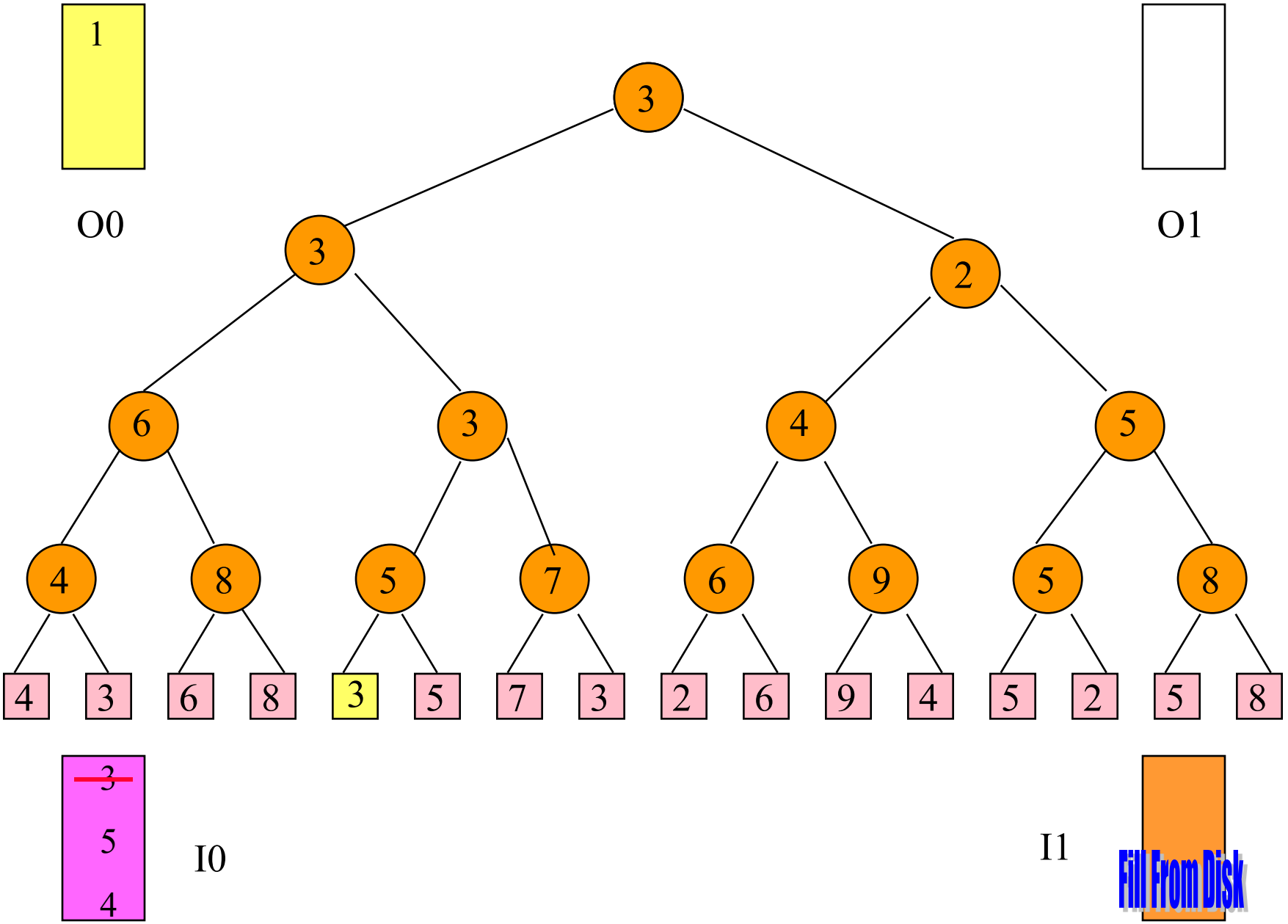
Initialize


O0
O1
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
Fill From Disk
I0
I1

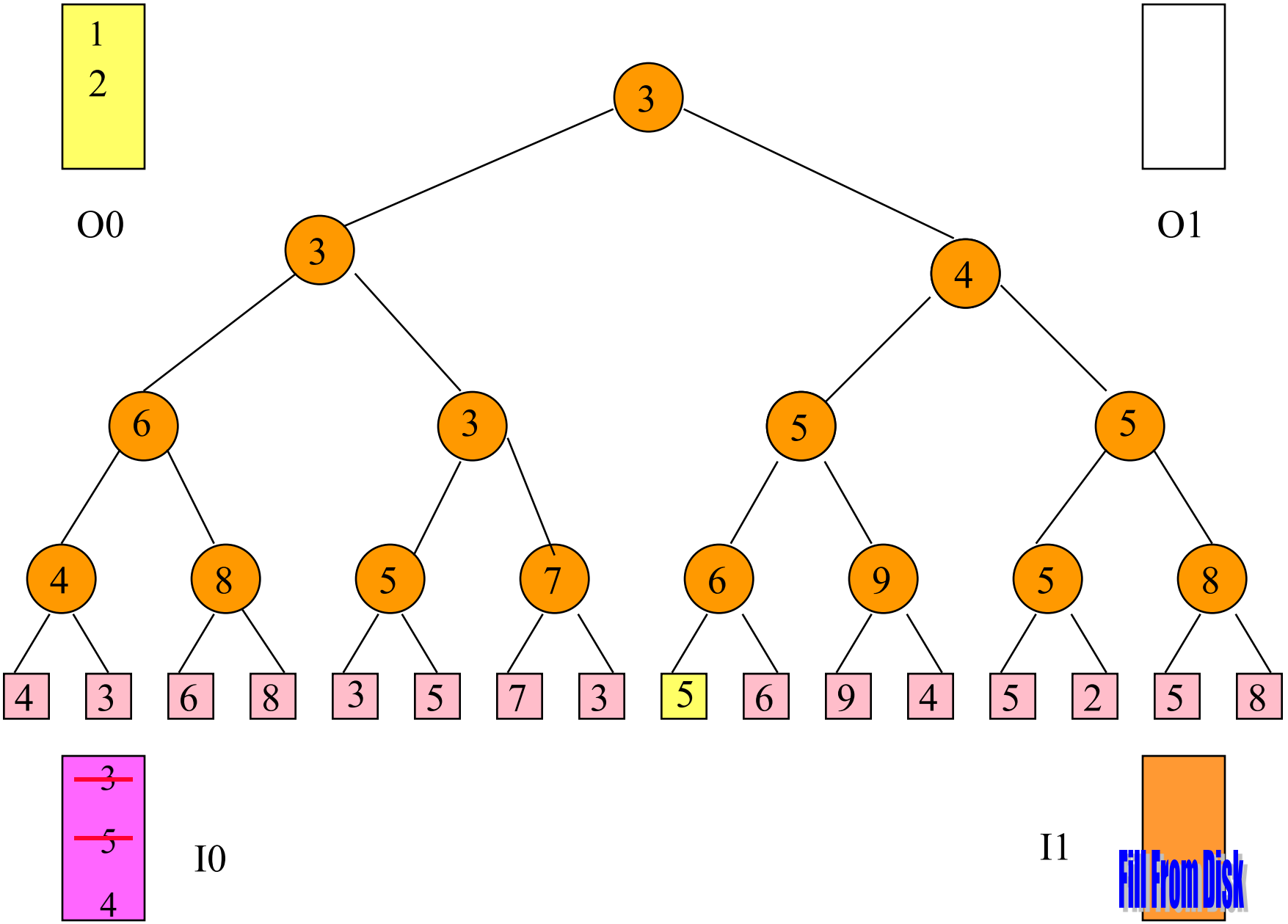
Generate Run 1



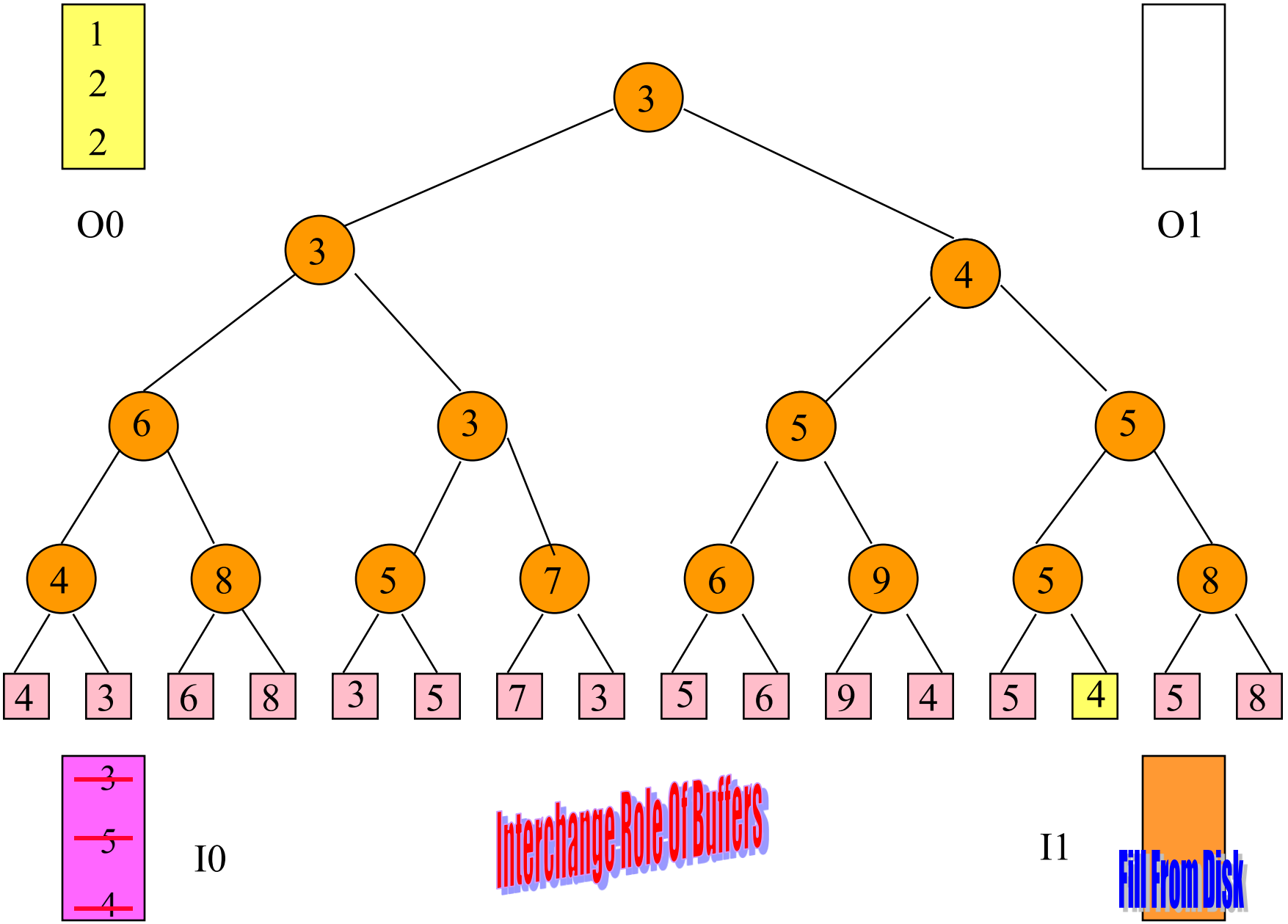
Generate Run 1

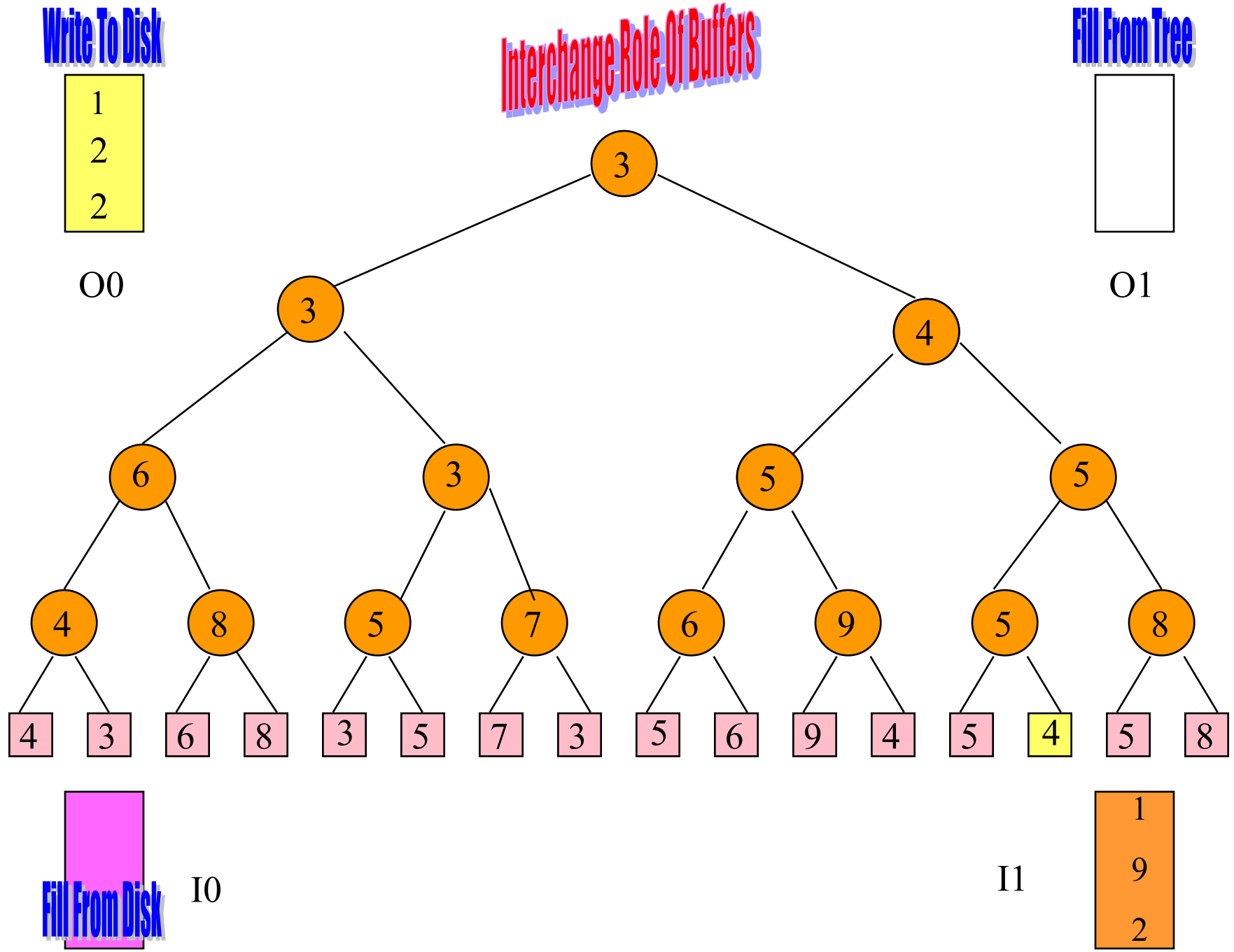


Generate Run 1

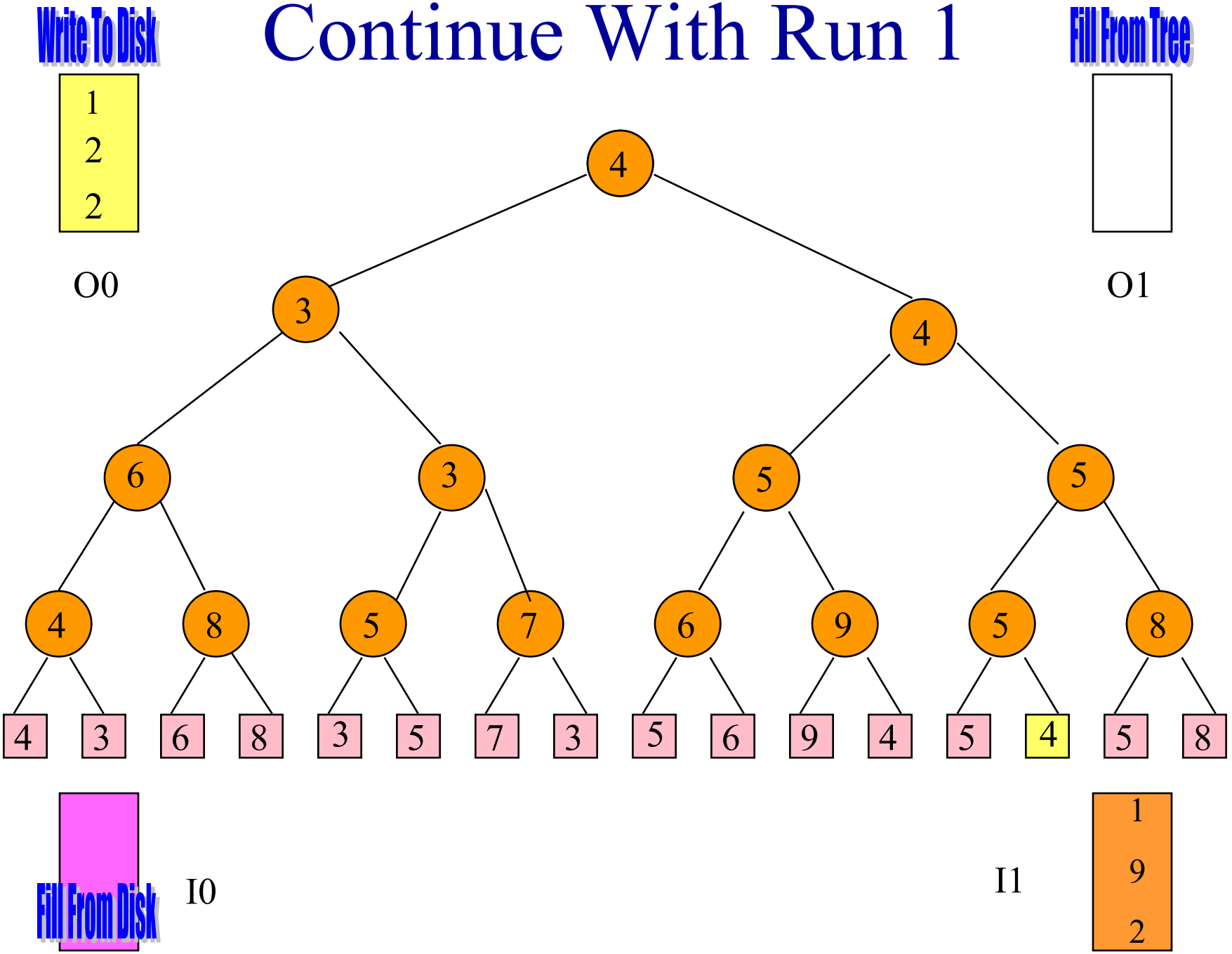


Generate Run 1




Write To Disk
Interchange Role Of Buffers
Fill From Tree
1
2
2
O0
O1
3
3
4
6
3
5
5
4
8
5
7
6
9
5
8
4
3
6
8
3
5
7
3
5
6
9
4
5
4
5
8
Fill From Disk
I0
I1
1
9
2

Continue With Run 1



Continue With Run 1

Write To Disk

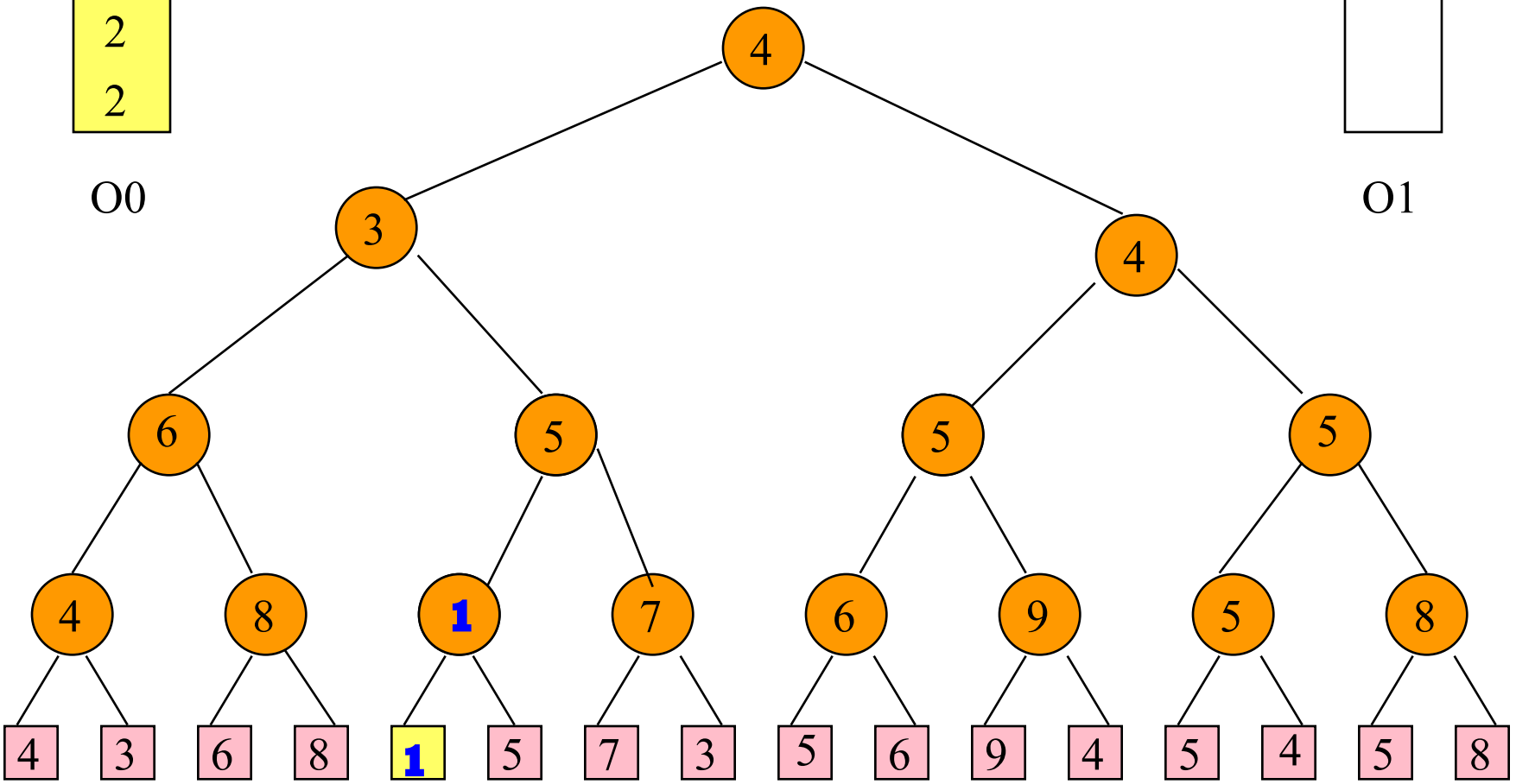
1
2
2

O0

Fill From Tree

3

O1



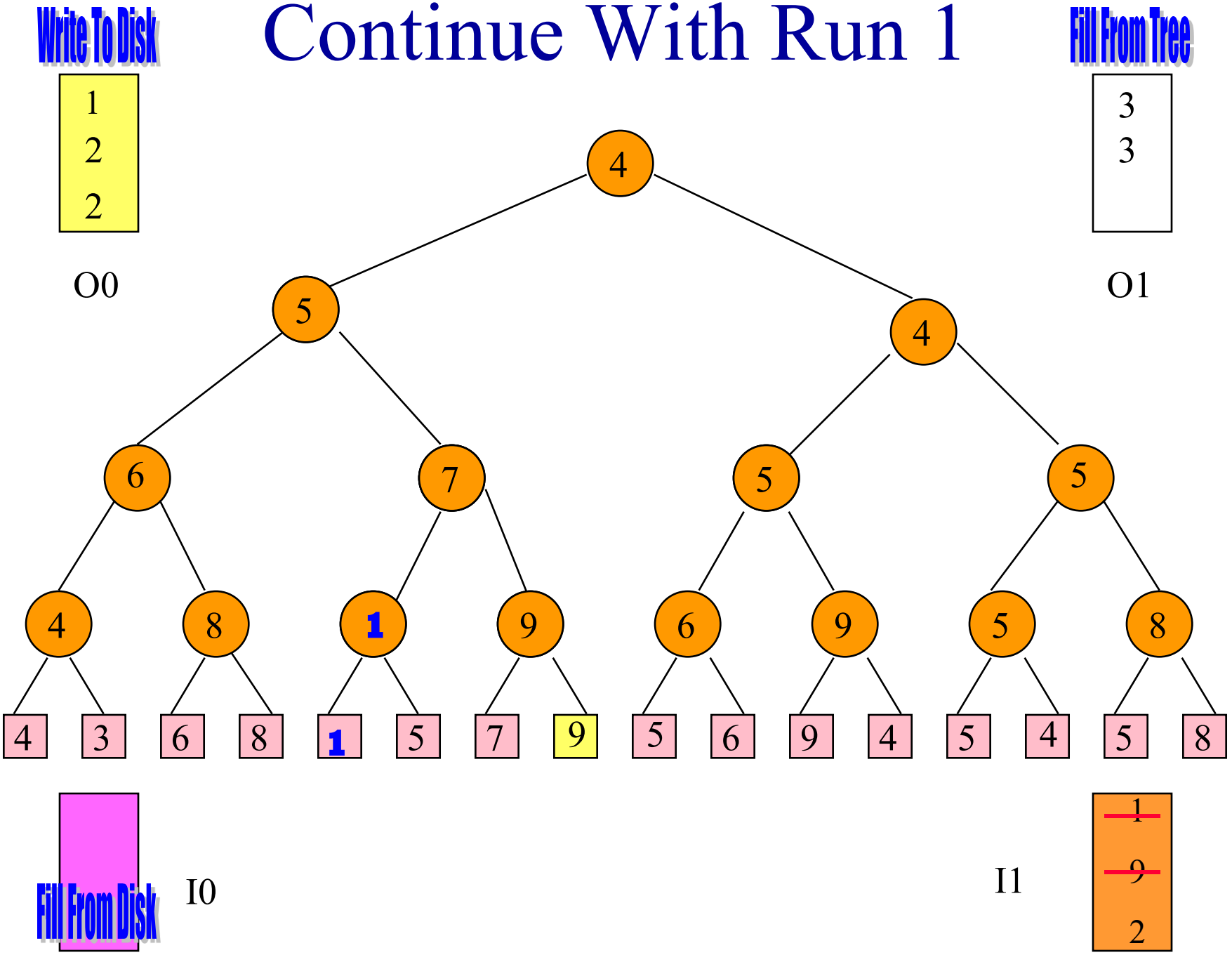
Fill From Disk

I0

I1

1
~~1~~
9
2

Continue With Run 1



Continue With Run 1

Write To Disk

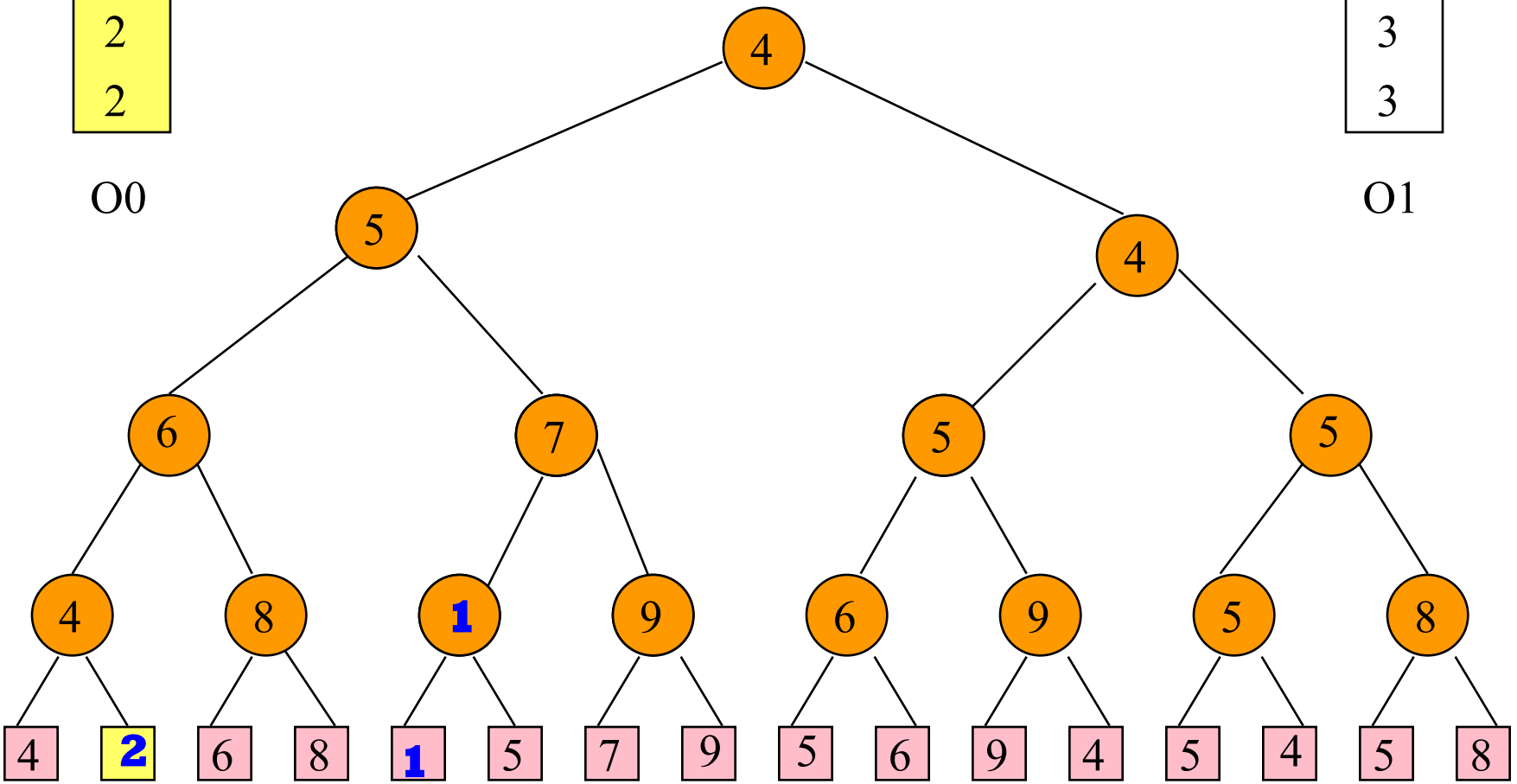
1
2
2

O0

Fill From Tree

3
3
3

O1



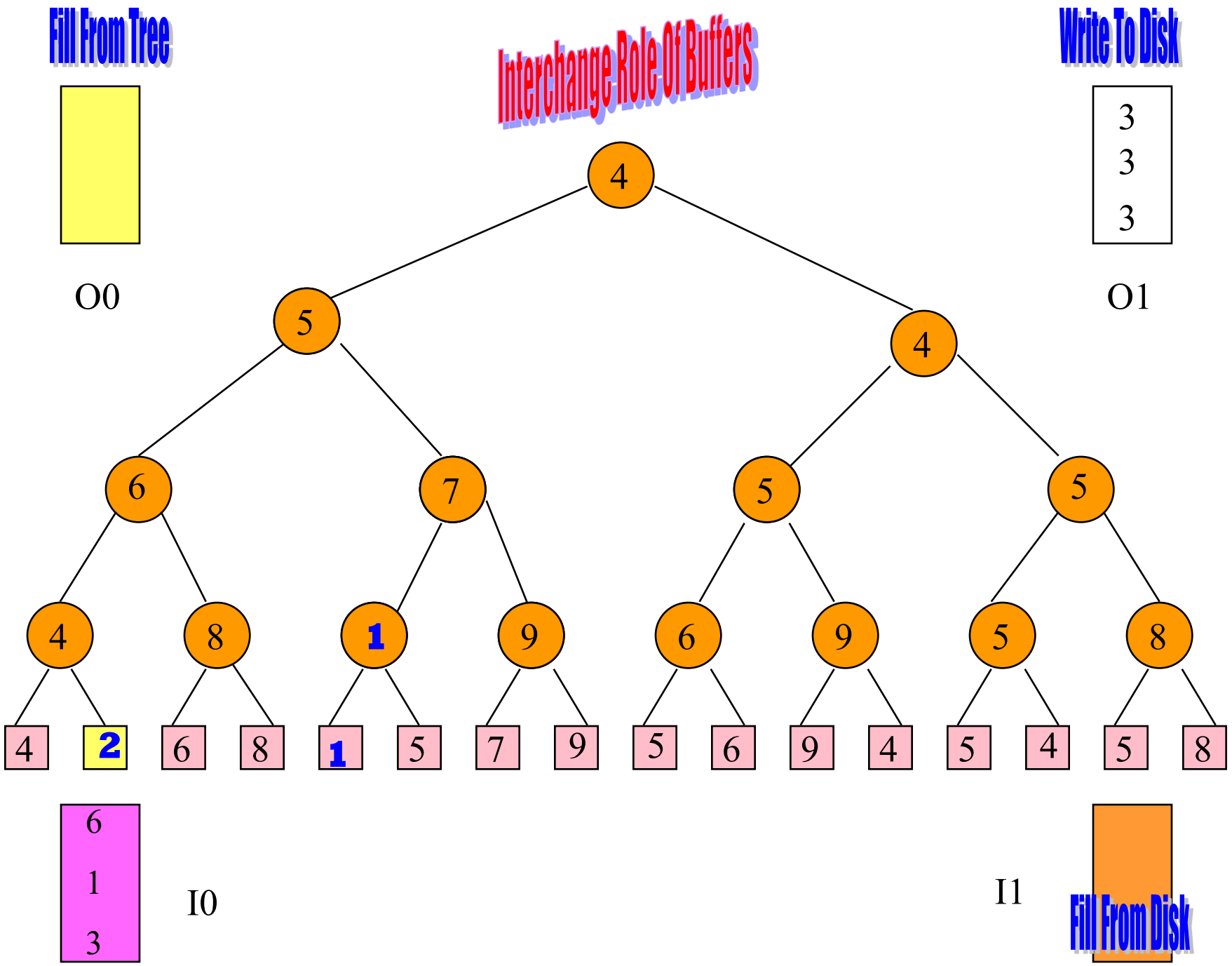
Fill From Disk

I0

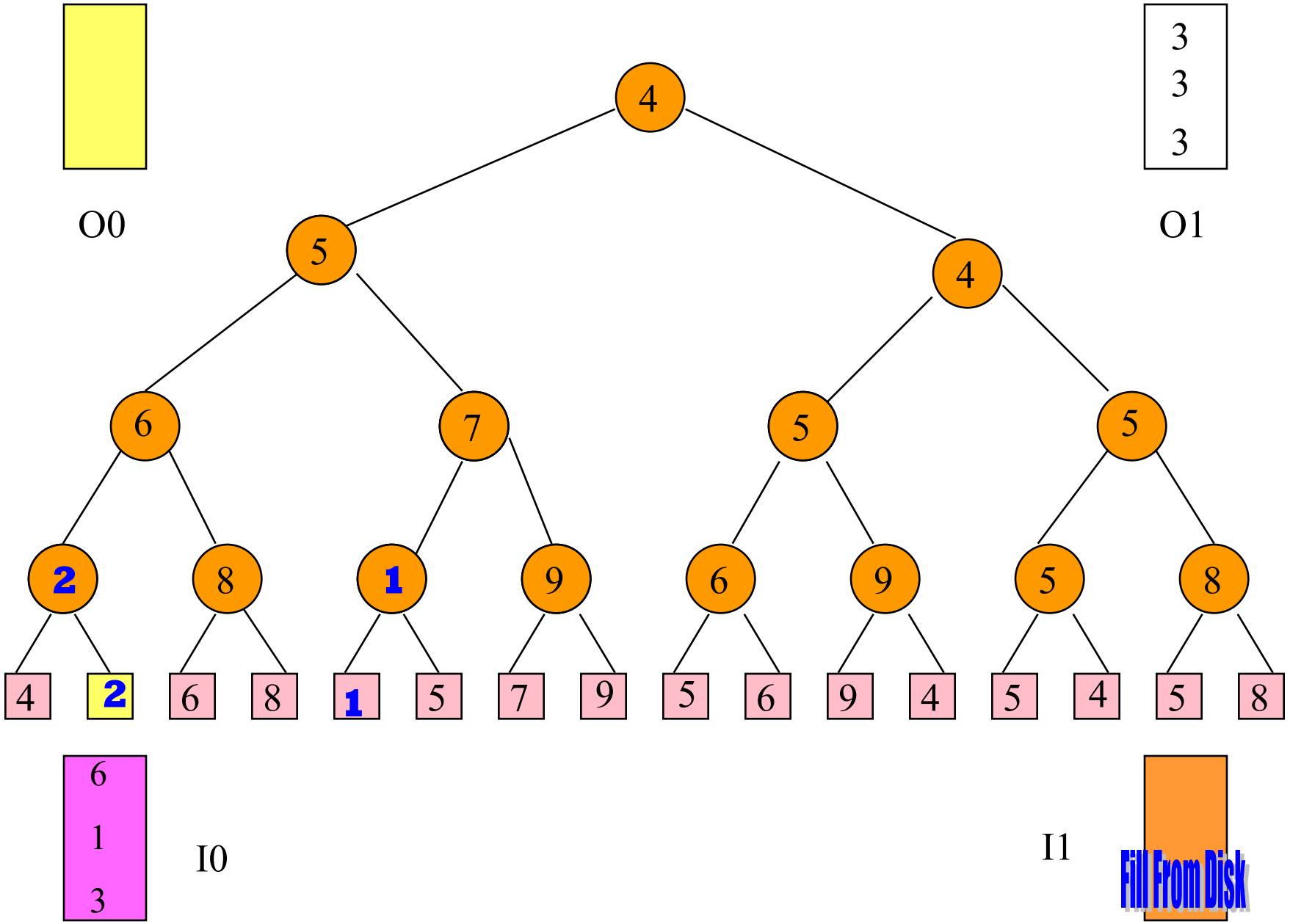
Interchange Role Of Buffers

I1

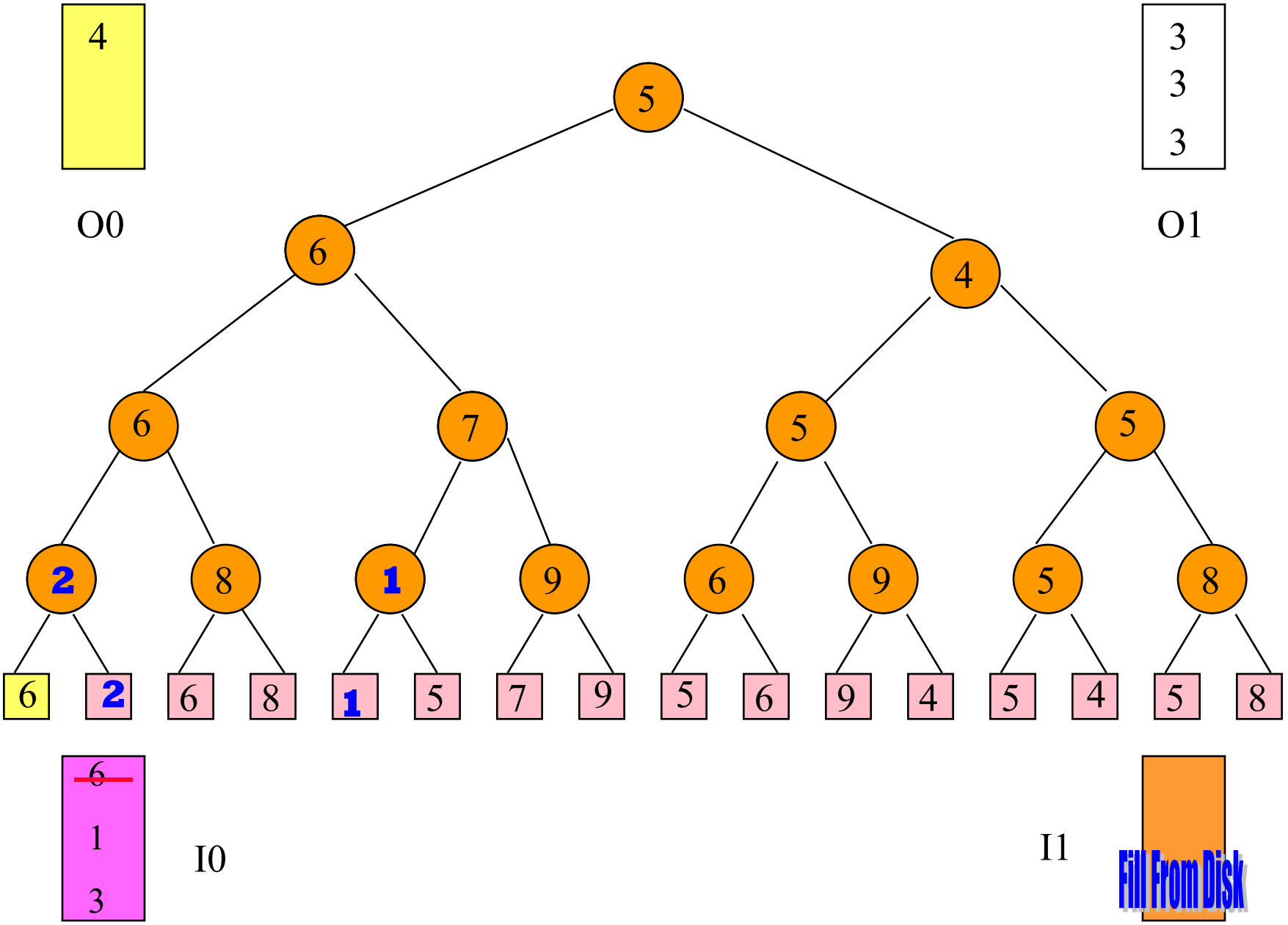
~~1~~
~~9~~
~~2~~


Fill From Tree
Interchange Role Of Buffers
Write To Disk
3
3
3
O0
O1
4
5
4
6
7
5
5
4
8
1
9
6
9
5
8
4
2
6
8
1
5
7
9
5
6
9
4
5
4
5
8
6
1
3
I0
I1
Fill From Disk

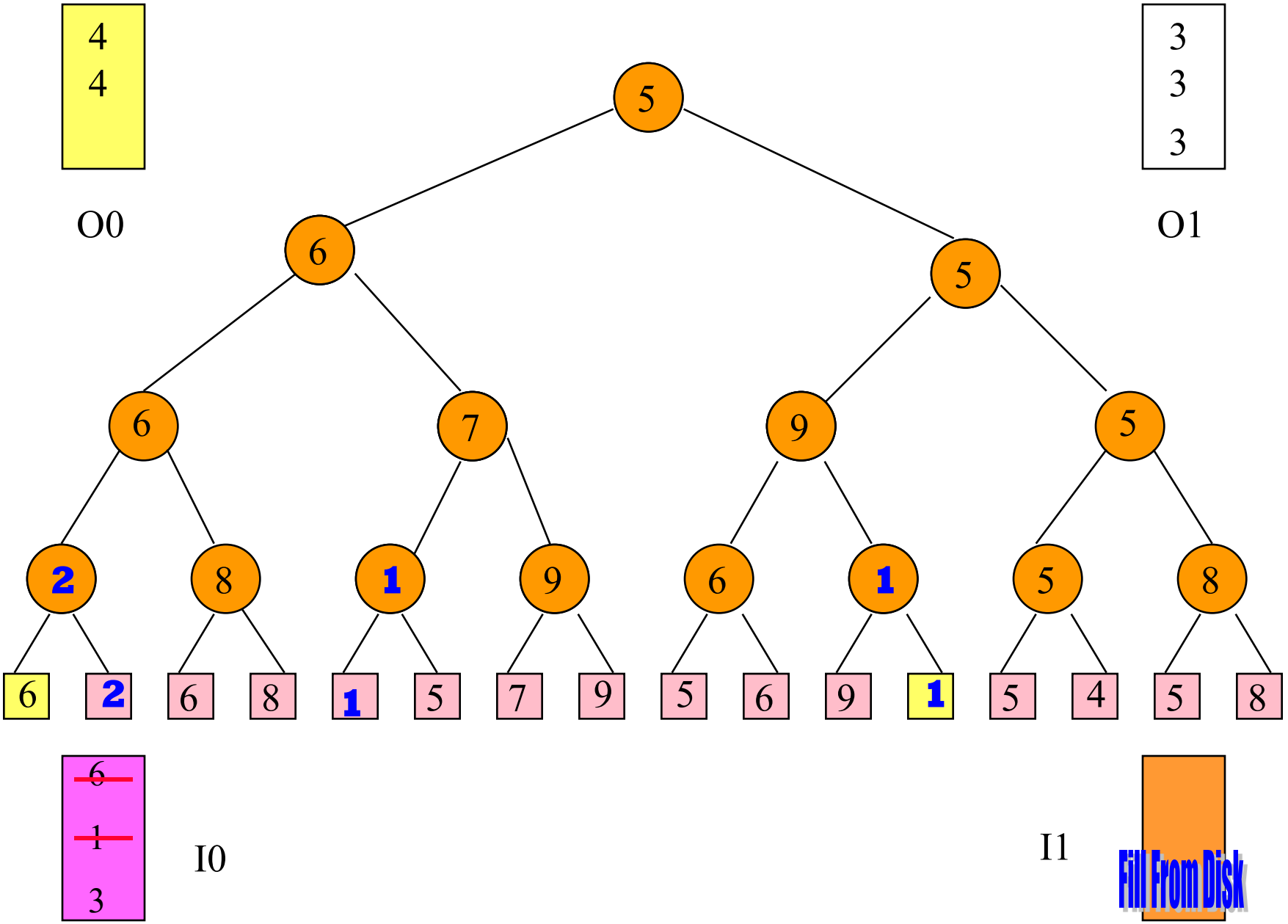
Continue With Run 1



Continue With Run 1



Continue With Run 1



RUN SIZE

- Let k be number of external nodes in the loser tree.
- Run size >= k.
- Sorted input => 1 run.
- Reverse of sorted input => n/k runs.
- Average run size is ~2k.

Comparison

- Memory capacity = m records.
- Run size using fill memory, sort, and output run scheme = m.
- Use loser tree scheme.
  - Assume block size is b records.
  - Need memory for 3 buffers (3b records).
  - Loser tree k = m – 3b.
  - Average run size = 2k = 2(m – 3b).
  - 2k >= m when m >= 6b.

Comparison

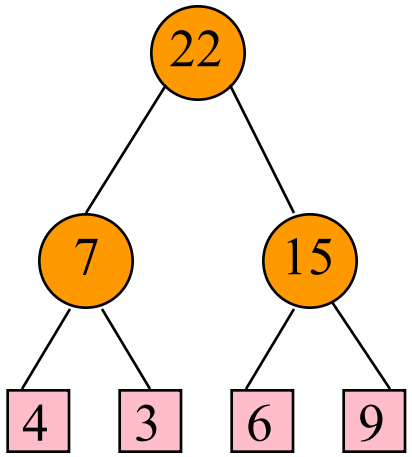
- Assume b = 100.

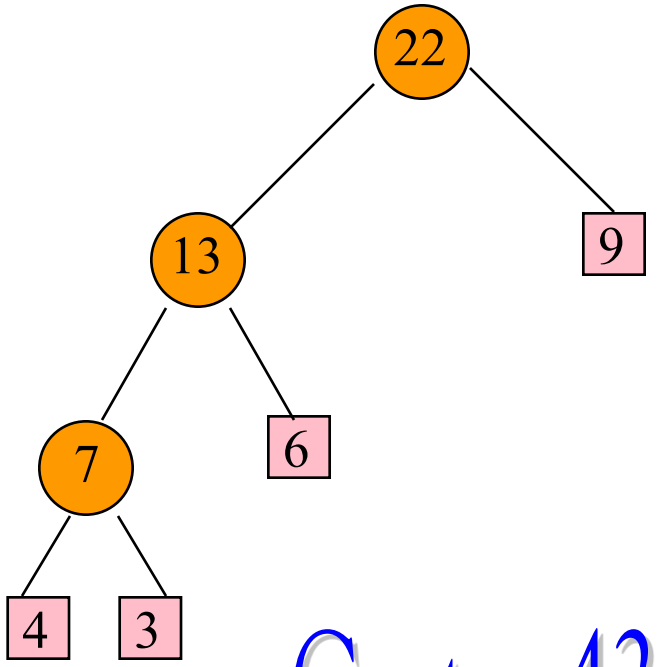
m	500	1000	5000	10000
k	200	700	4700	9700
2k	400	1400	9400	19400

Comparison

- Total internal processing time using fill memory, sort, and output run scheme = $O((n/m)\ m \log m) = O(n \log m)$.
- Total internal processing time using loser tree = $O(n \log k)$.
- Loser tree scheme generates runs that differ in their lengths.

Merging Runs Of Different Length



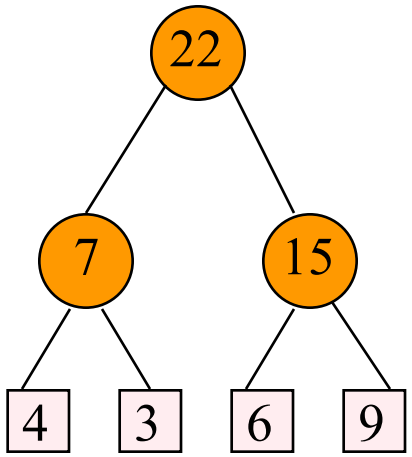


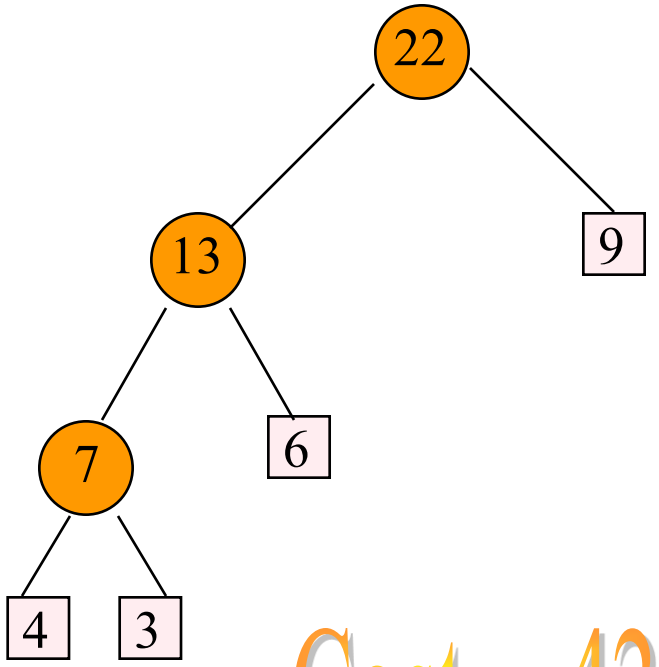
Cost = 44

Cost = 42

Best merge sequence?

Optimal Merging Of Runs





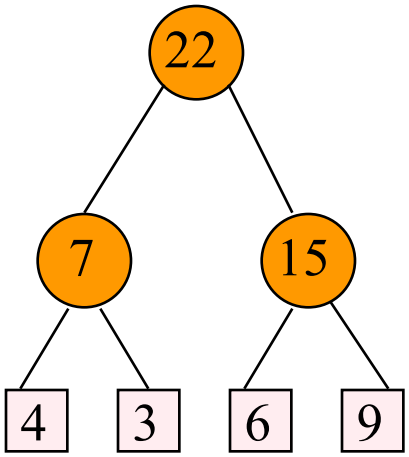
Cost = 44

Cost = 42

Best merge sequence?

Weighted External Path Length

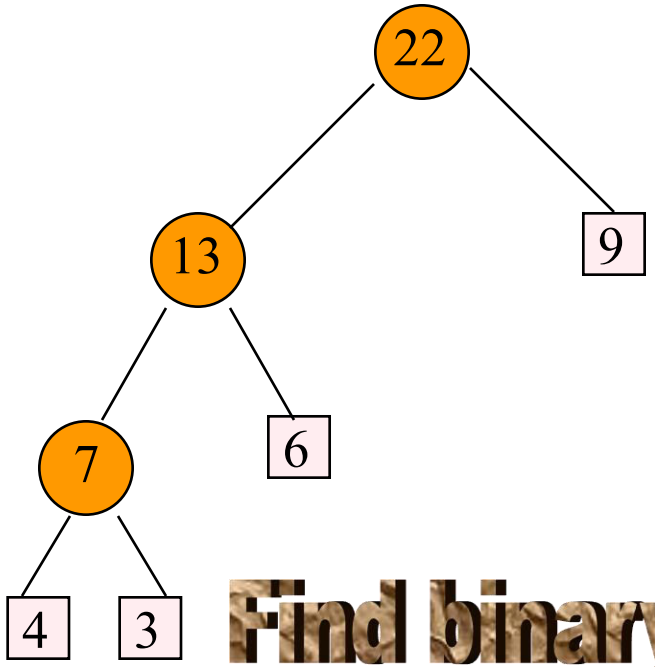
WEPL(T) = Σ(weight of external node i)
* (distance of node i from root of T)



WEPL(T) = 4 * 2 + 3*2 + 6*2 + 9*2
= 44
= Merge Cost

Weighted External Path Length

WEPL(T) = Σ(weight of external node i)
* (distance of node i from root of T)



WEPL(T) = 4 * 3 + 3*3 + 6*2 + 9*1
= 42
= Merge Cost

Find binary tree with minimum WEPL.

Other Applications

- Message coding and decoding.
- Lossless data compression.

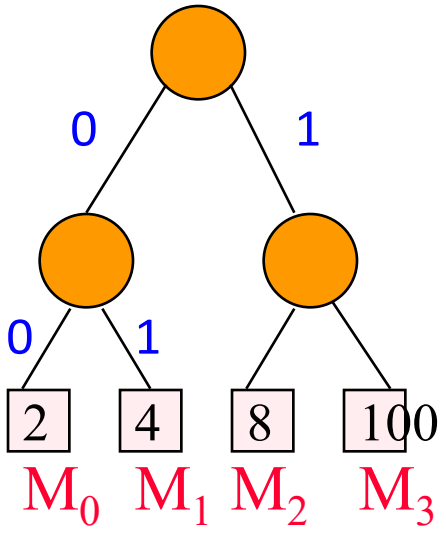
Message Coding & Decoding

- Messages M_0, M_1, M_2, …, M_{n-1} are to be transmitted.
- The messages do not change.
- Both sender and receiver know the messages.
- So, it is adequate to transmit a code that identifies the message (e.g., message index).
- M_i is sent with frequency f_i.
- Select message codes so as to minimize transmission and decoding times.

Example

- $n = 4$ messages.
- The frequencies are [2, 4, 8, 100].
- Use 2-bit codes [00, 01, 10, 11].
- Transmission cost $= 2*2 + 4*2 + 8*2 + 100*2$
  $= 228$.
- Decoding is done using a binary tree.

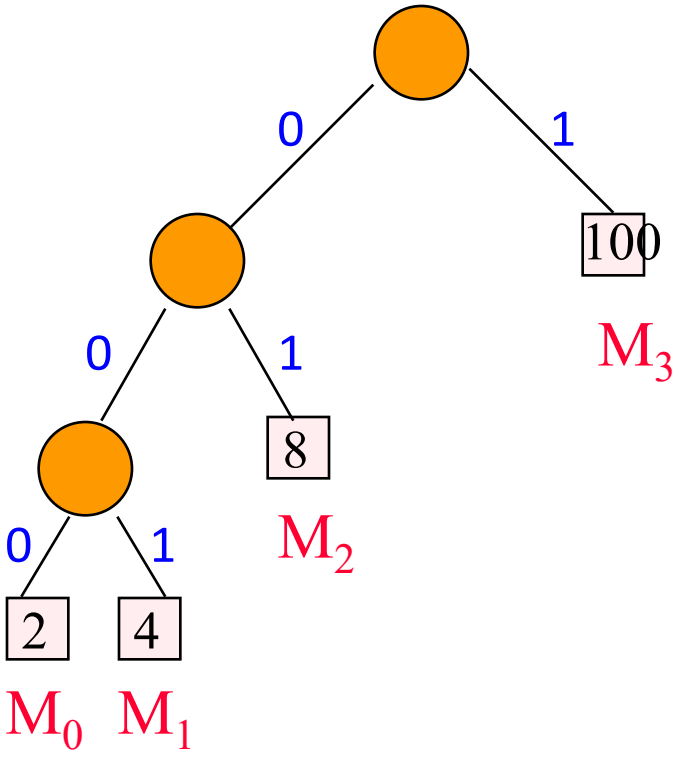
Example



- Decoding cost = 2*2 + 4*2 + 8*2 + 100*2
  = 228
  = transmission cost
  = WEPL

Example

- Every binary tree with n external nodes defines a code set for n messages.



- Decoding cost

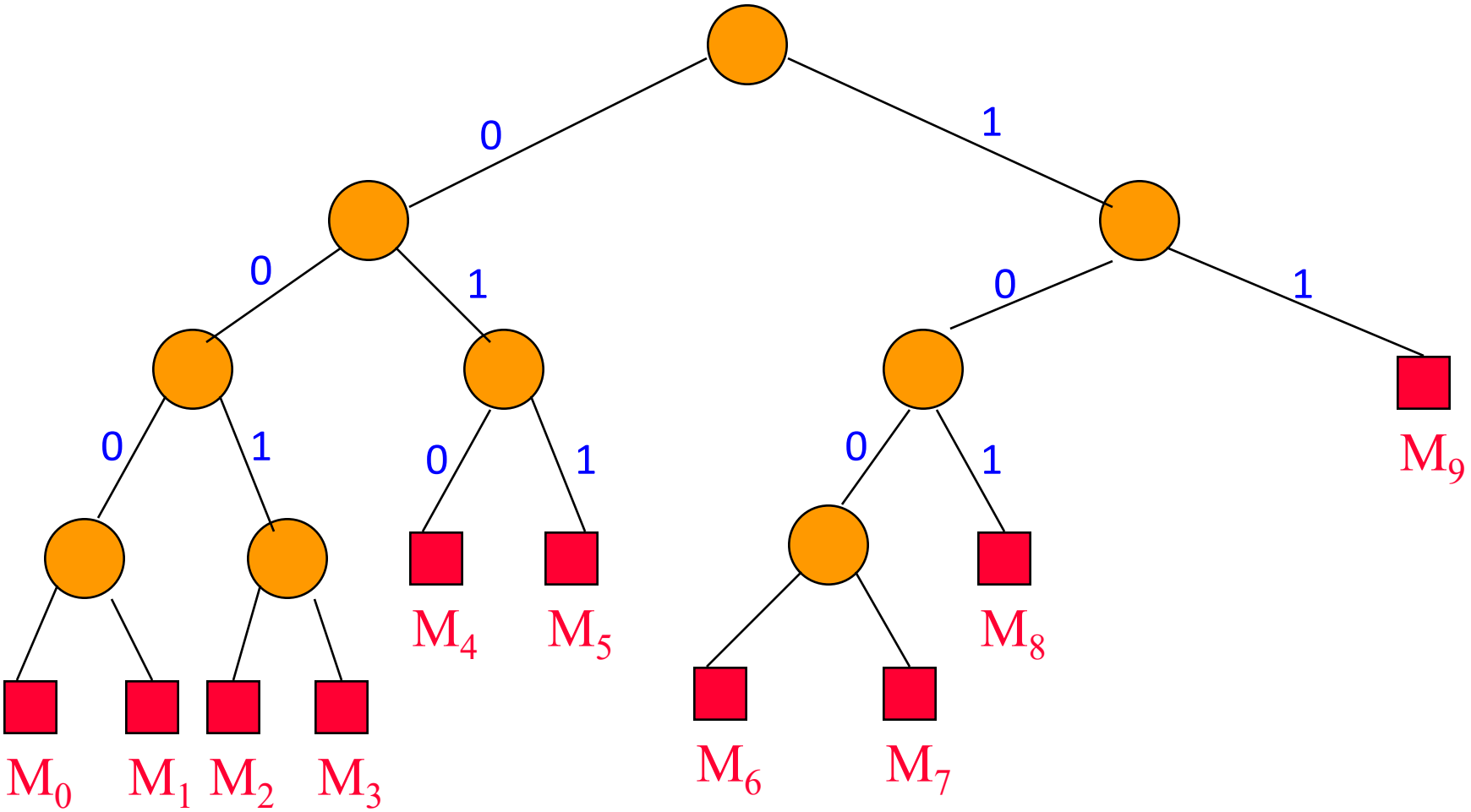
  $= 2*3 + 4*3 + 8*2 + 100*1$

  $= 134$

  = transmission cost

  = WEPL

Another Example



No code is a prefix of another!

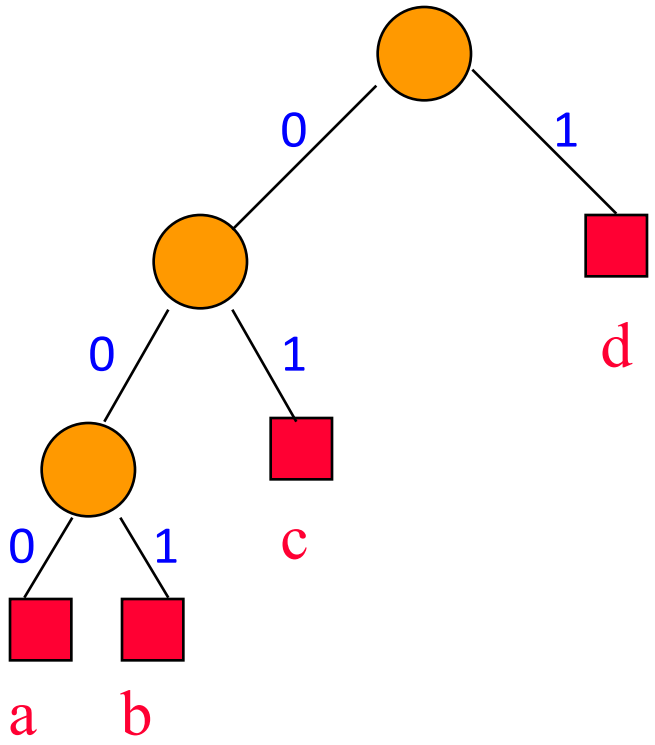
Lossless Data Compression

- Alphabet = {a, b, c, d}.
- String with 10 as, 5 bs, 100 cs, and 900 ds.
- Use a 2-bit code.
  - a = 00, b = 01, c = 10, d = 11.
  - Size of string = 10*2 + 5*2 + 100*2 + 900*2
    = 2030 bits.
  - Plus size of code table.

Lossless Data Compression

- Use a variable length code that satisfies prefix property (no code is a prefix of another).
  - a = 000, b = 001, c = 01, d = 1.
  - Size of string = 10*3 + 5*3 + 100*2 + 900*1
    = 1145 bits.
  - Plus size of code table.
  - Compression ratio is approx. 2030/1145 = 1.8.

Lossless Data Compression



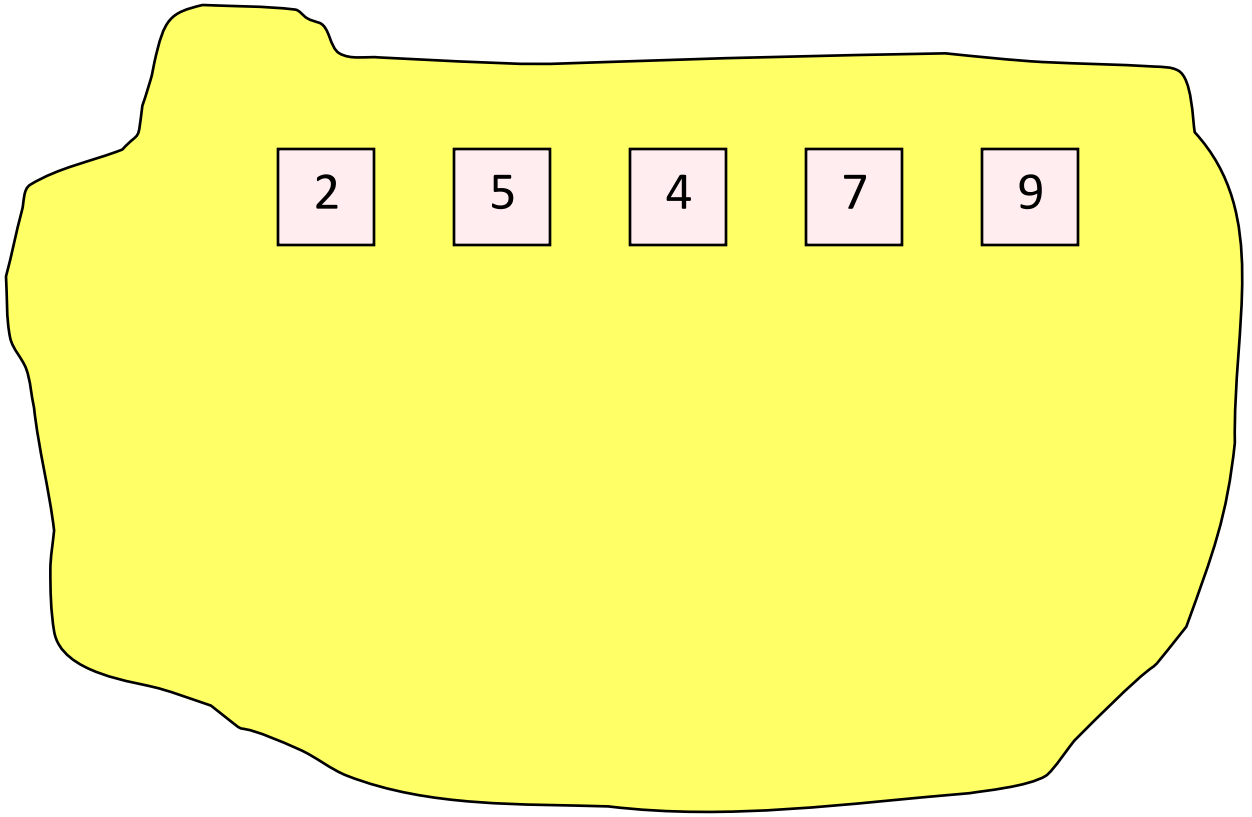
- Decode 0001100101…
- addbc…
- Compression ratio is maximized when the decode tree has minimum WEPL.

Huffman Trees

- Trees that have minimum WEPL.
- Binary trees with minimum WEPL may be constructed using a greedy algorithm.
- For higher order trees with minimum WEPL, a preprocessing step followed by the greedy algorithm may be used.
- Huffman codes: codes defined by minimum WEPL trees.

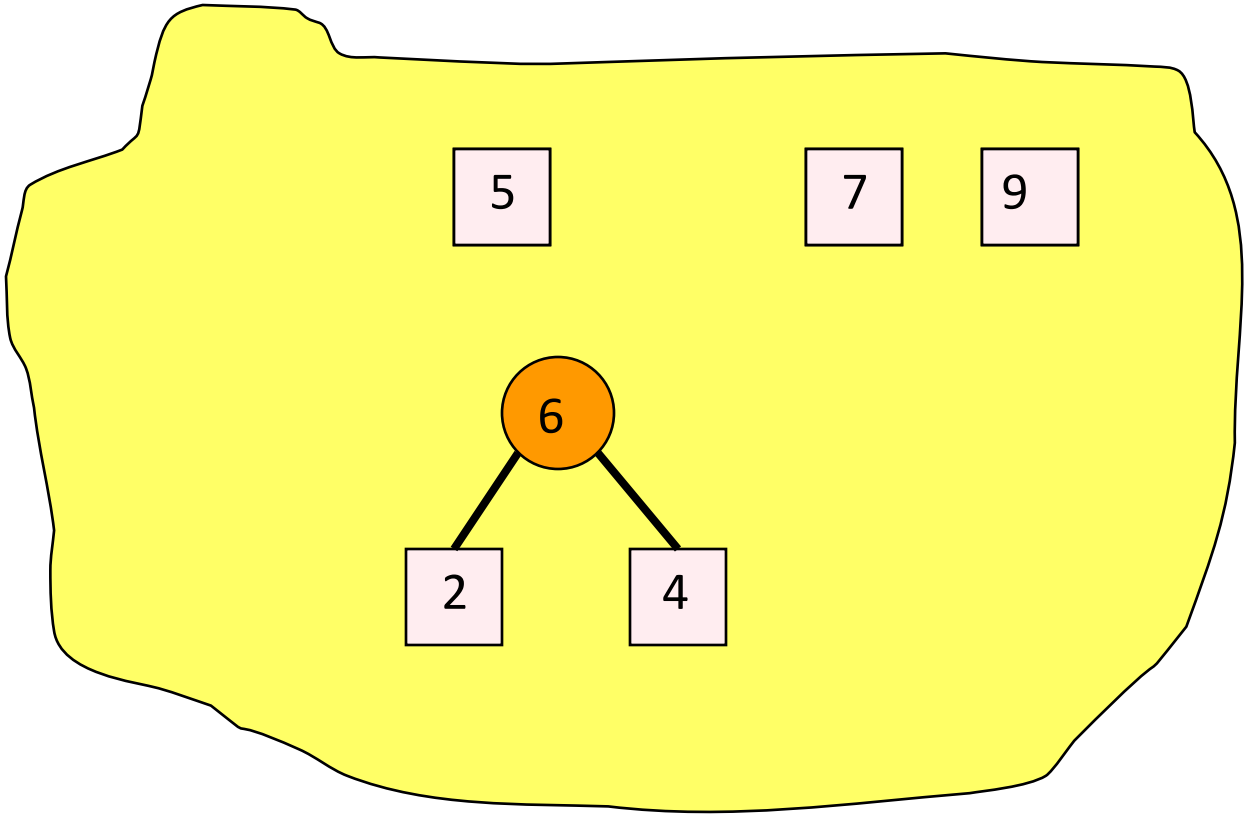
Example—Binary Trees

- n = 5, w[0:4] = [2, 5, 4, 7, 9].



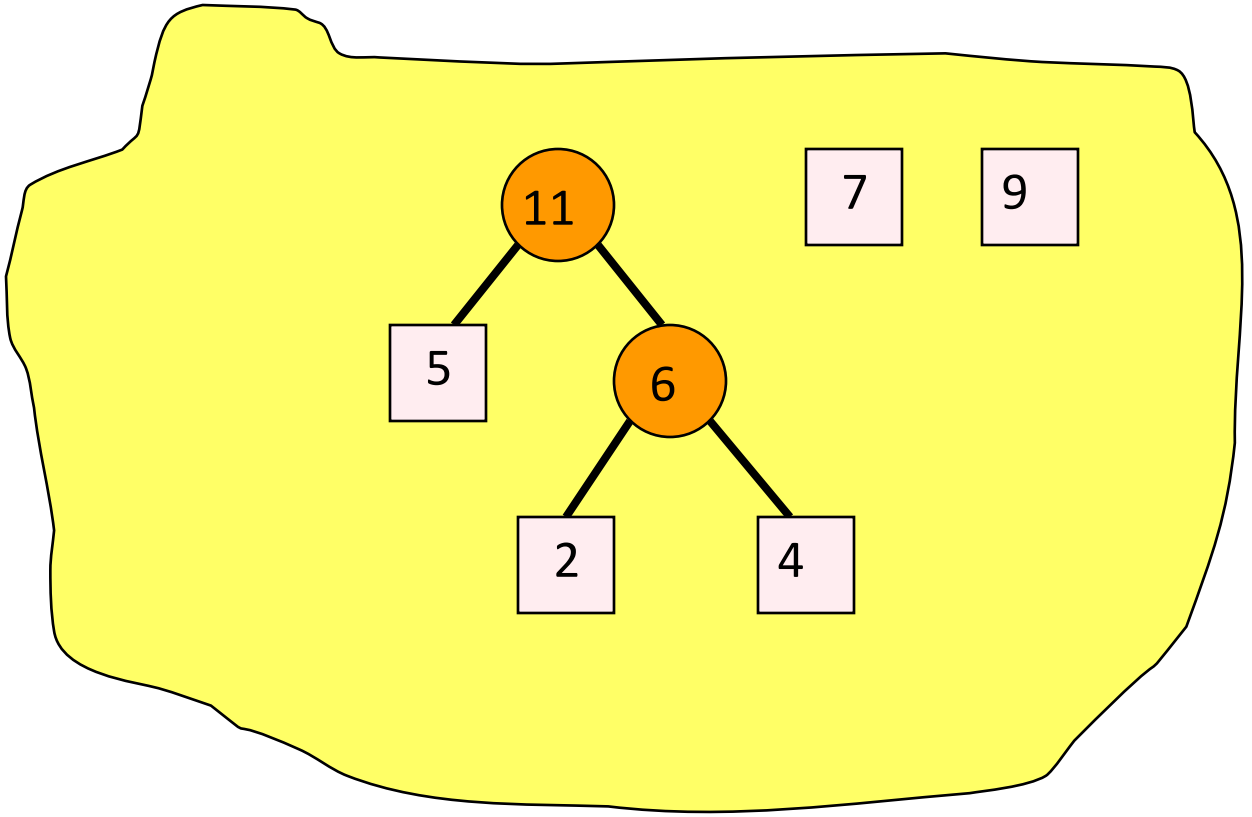
Example—Binary Trees

- n = 5, w[0:4] = [2, 5, 4, 7, 9].



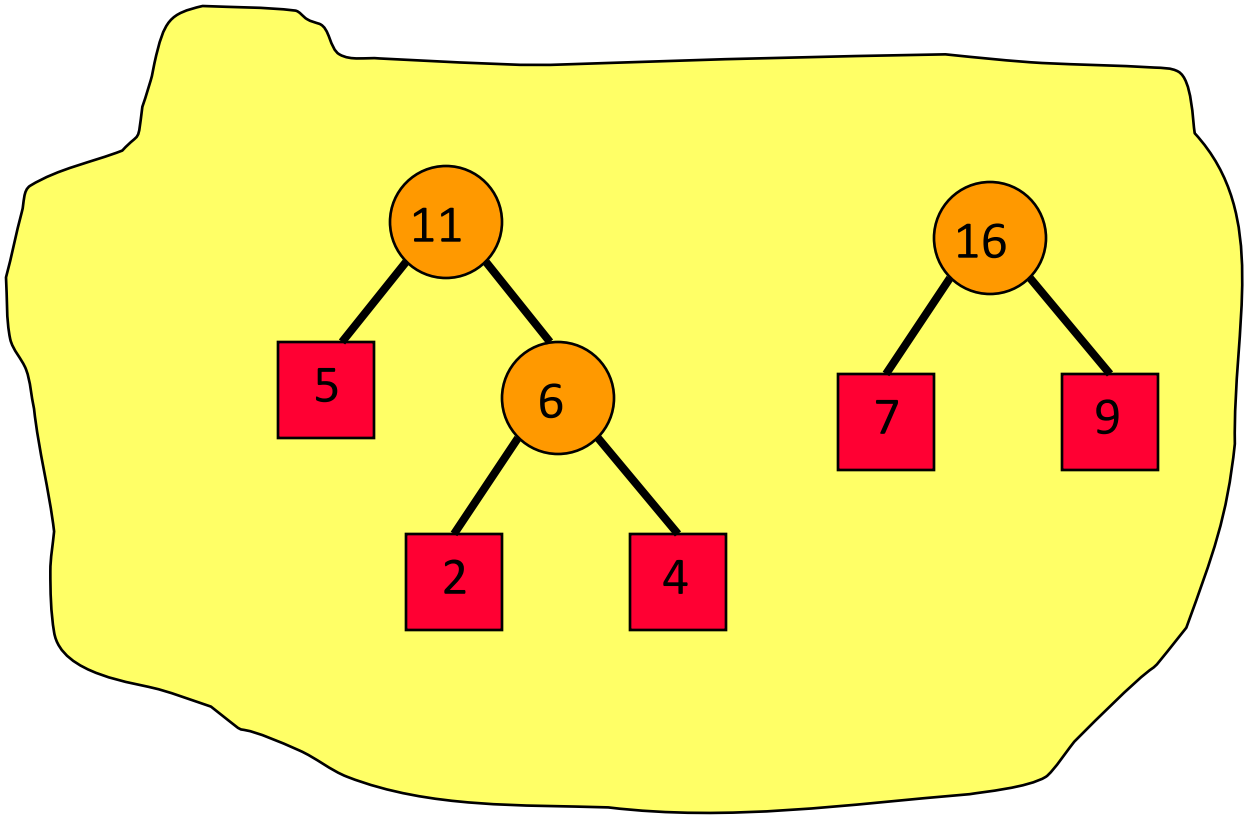
Example—Binary Trees

- n = 5, w[0:4] = [2, 5, 4, 7, 9].



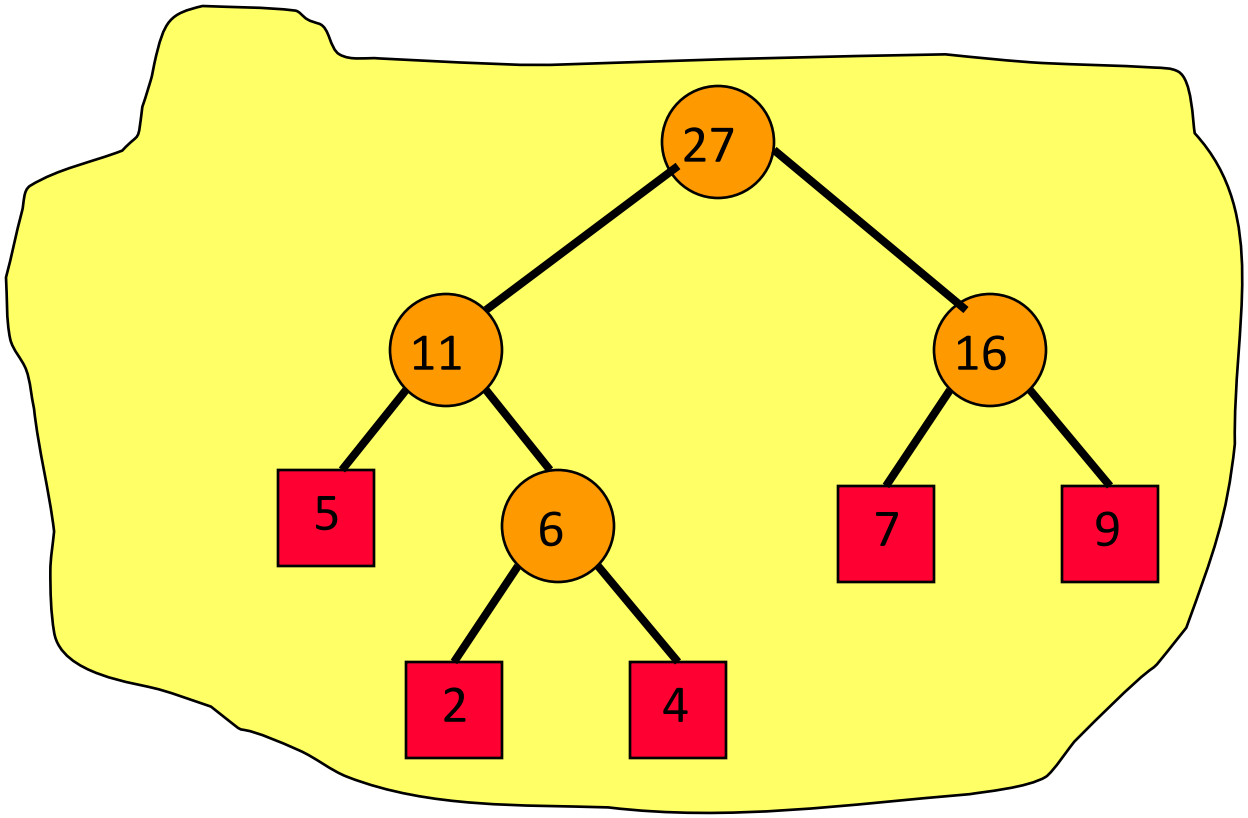
Example—Binary Trees

- n = 5, w[0:4] = [2, 5, 4, 7, 9].



Example—Binary Trees

- n = 5, w[0:4] = [2, 5, 4, 7, 9].



Greedy Algorithm For Binary Trees

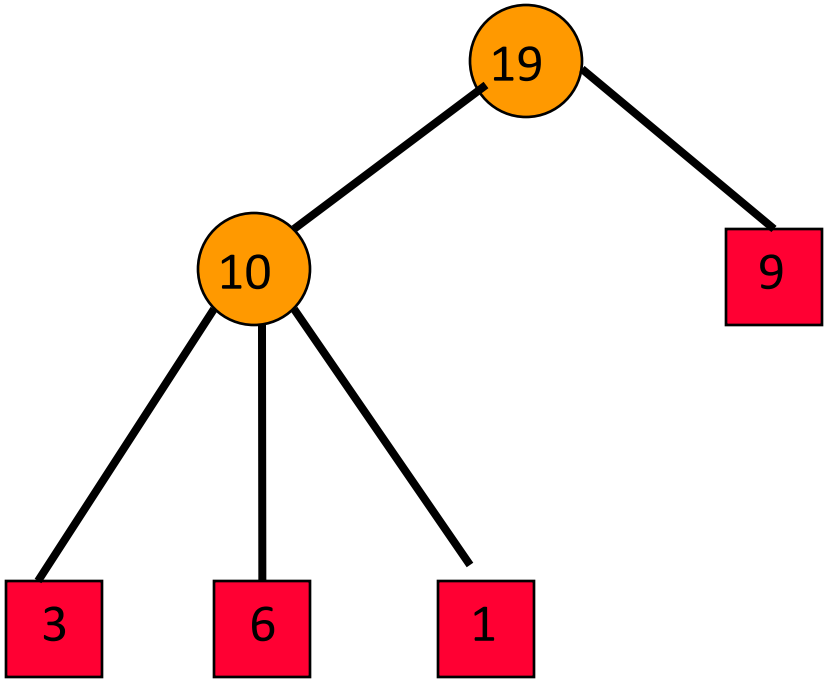
- Start with a collection of external nodes, each with one of the given weights. Each external node defines a different tree.
- Reduce number of trees by 1.
  - Remove 2 trees with minimum weight from the collection.
  - Combine them by making them children of a new root node.
  - The weight of the new tree is the sum of the weights of the individual trees.
  - Add new tree to tree collection.
- Repeat reduce step until only 1 tree remains.

Data Structure For Tree Collection

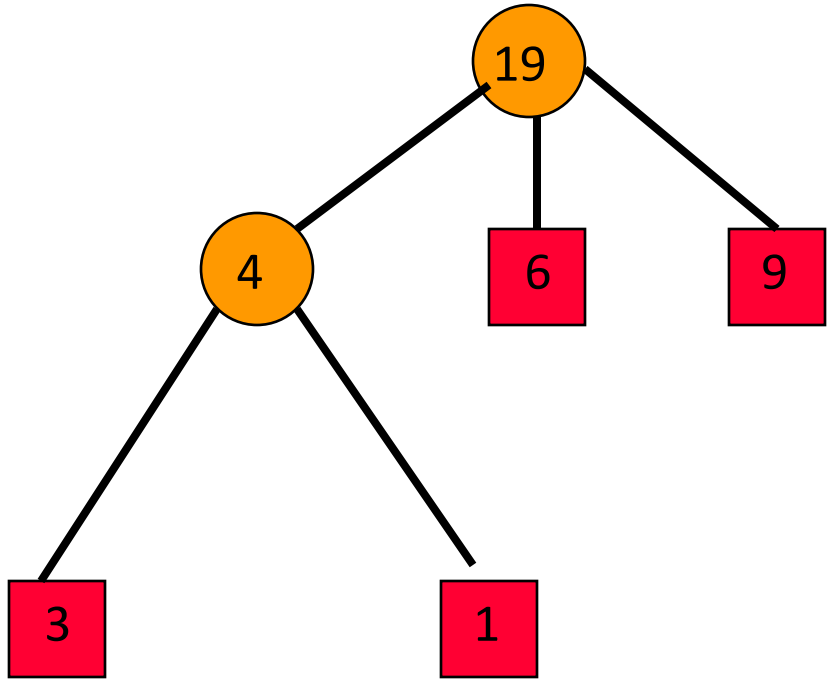
- Operations are:
  - Initialize with n trees.
  - Remove 2 trees with least weight.
  - Insert new tree.
- Use a min heap.
- Initialize … O(n).
- 2(n – 1) remove min operations … O(n log n).
- n – 1 insert operations … O(n log n).
- Total time is O(n log n).
- Or, (n – 1) remove mins and (n – 1) change mins.

Higher Order Trees

- Greedy scheme doesn't work!
- 3-way tree with weights [3, 6, 1, 9].

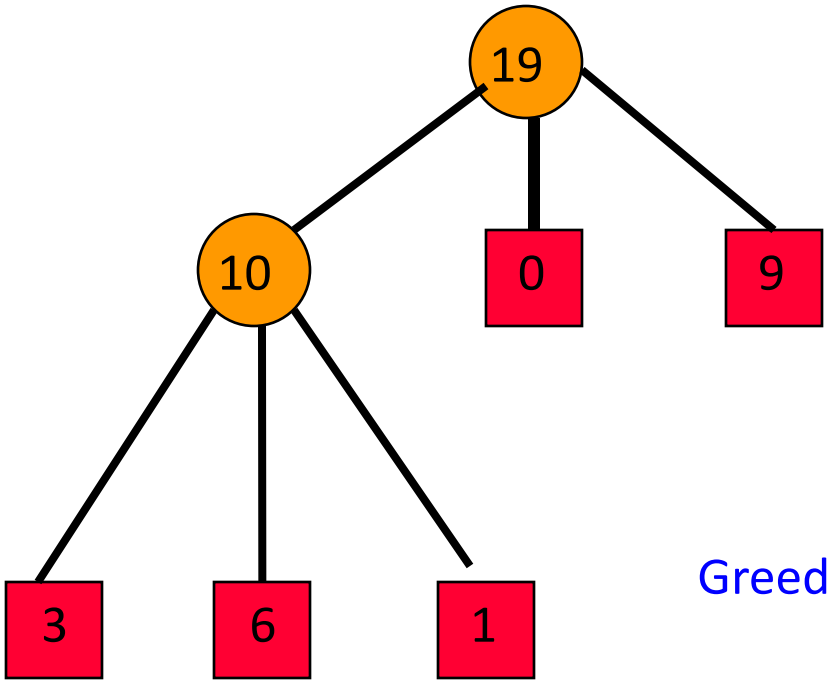


Greedy Tree Cost = 29



Optimal Tree Cost = 23

Cause Of Failure



Greedy Tree Cost = 29

- One node is not a 3-way node.
- A 2-way node is like a 3-way node, one of whose children has a weight of 0.
- Must start with enough runs/weights of length 0 so that all nodes are 3-way nodes.

How Many Length 0 Runs To Add?

- k-way tree, $k > 1$.
- Initial number of runs is r.
- Add least $q >= 0$ runs of length 0.
- Each k-way merge reduces the number of runs by $k - 1$.
- Number of runs after s k-way merges is

  $r + q - s(k - 1)$
- For some positive integer s, the number of remaining runs must become 1.

How Many Length 0 Runs To Add?

- So, we want

  $r + q - s(k-1) = 1$

  for some positive integer s.
- So, $r + q - 1 = s(k - 1)$.
- Or, $(r + q - 1) \bmod (k - 1) = 0$.
- Or, $r + q - 1$ is divisible by $k - 1$.
  - This implies that $q < k - 1$.
- $(r - 1) \bmod (k - 1) = 0 \Rightarrow q = 0$.
- $(r - 1) \bmod (k - 1) \mathrel{!}= 0 \Rightarrow$

  $q = k - 1 - (r - 1) \bmod (k - 1)$.
- Or, $q = (1 - r) \bmod (k - 1)$.

Examples

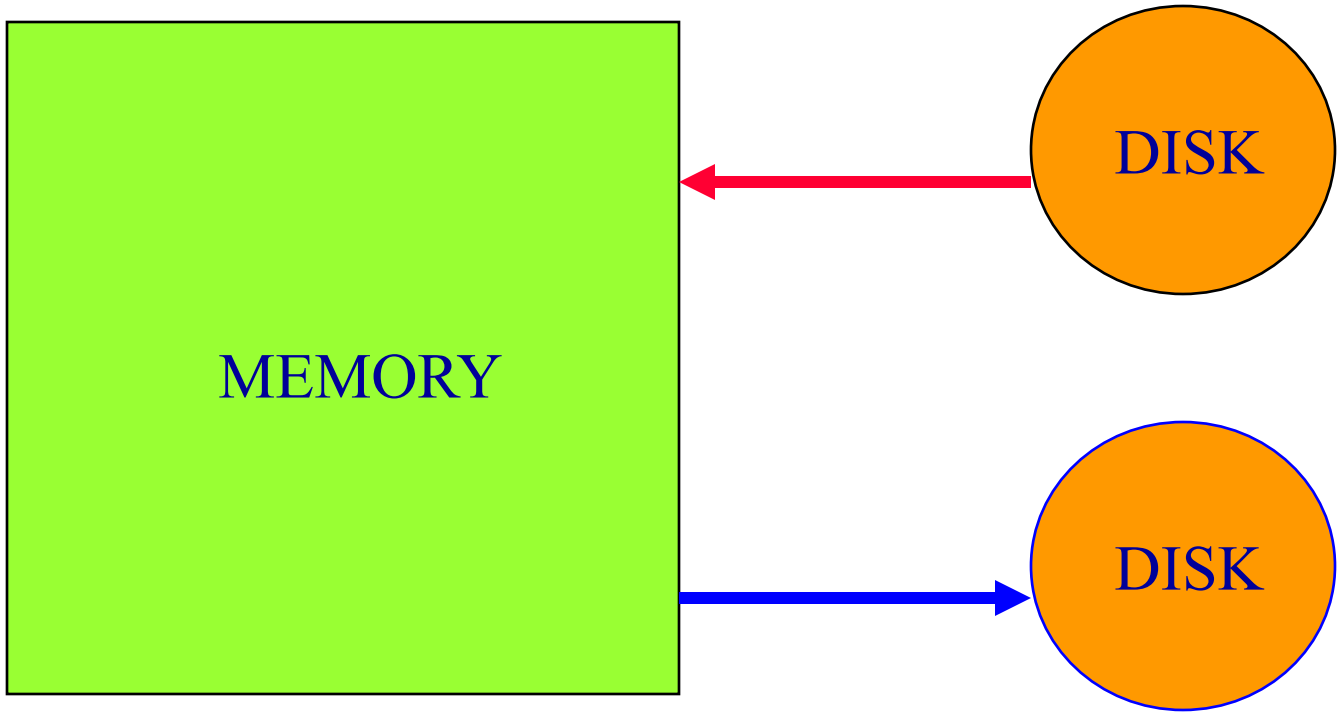
- $k = 2$.
  - $q = (1 - r) \bmod (k - 1) = (1 - r) \bmod 1 = 0$.
  - So, no runs of length 0 are to be added.
- $k = 4$, $r = 6$.
  - $q = (1 - r) \bmod (k - 1) = (1 - 6) \bmod 3$
    $= (-5) \bmod 3$
    $= (6 - 5) \bmod 3$
    $= 1$.
  - So, must start with 7 runs, and then apply greedy method.

Improve Run Merging

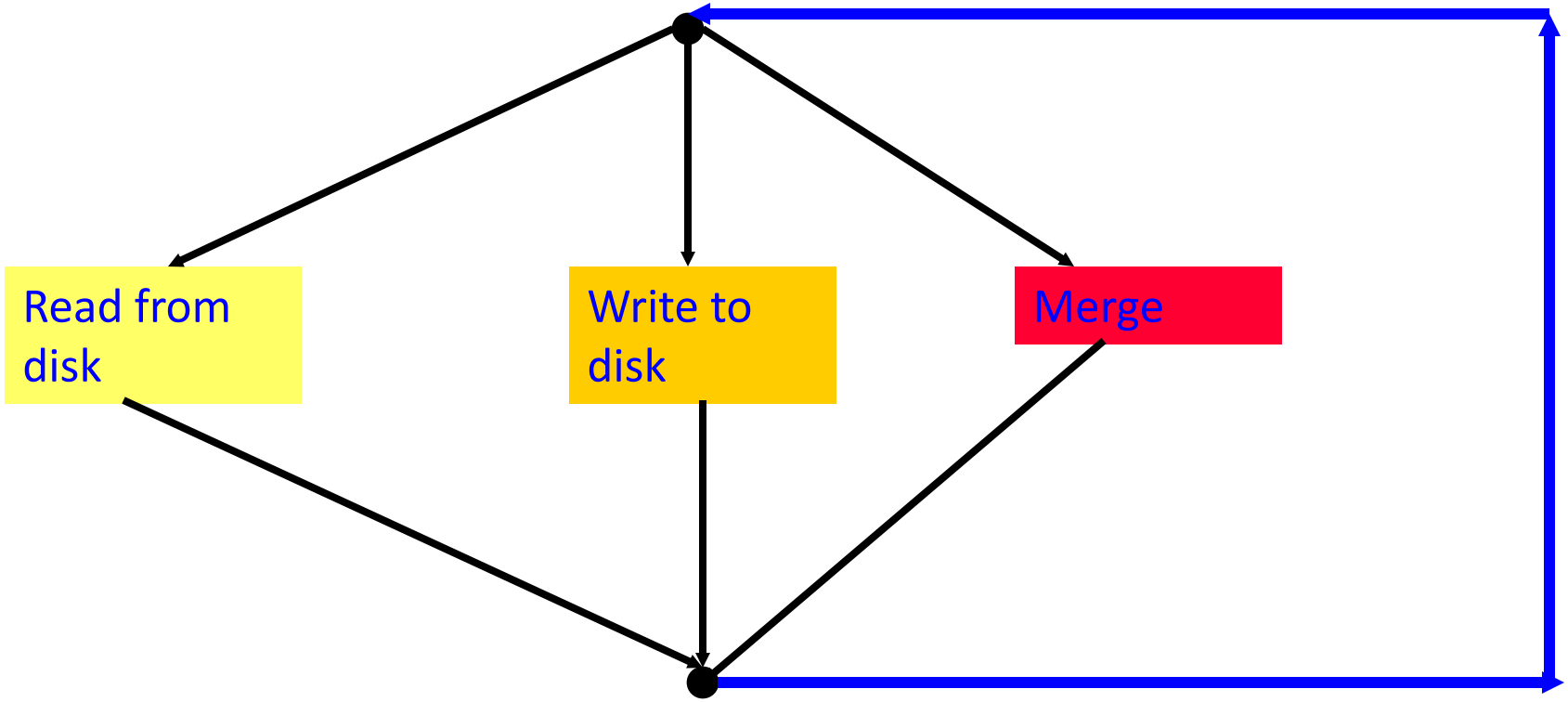
- Reduce number of merge passes.
  - Use higher order merge.
  - Number of passes
    = ceil($\log_k$(number of initial runs))
    where k is the merge order.
- More generally, a higher-order merge reduces the cost of the optimal merge tree.

Improve Run Merging

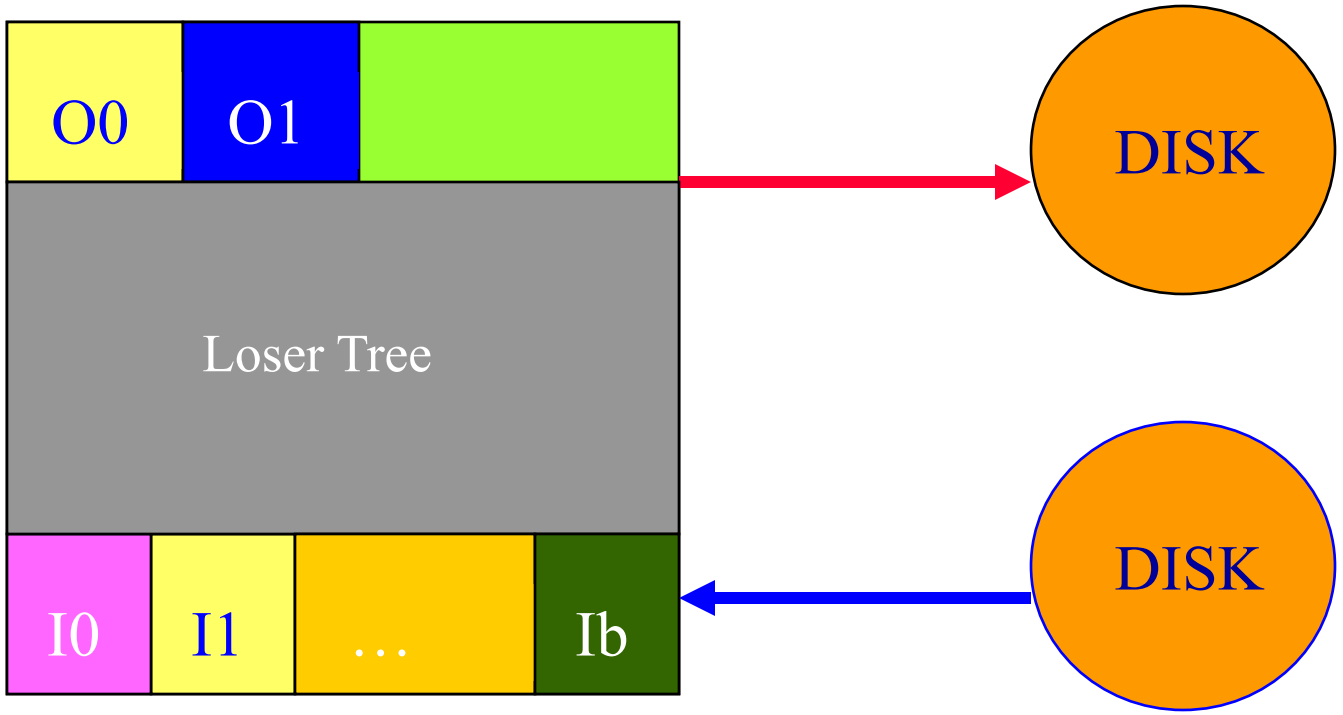
- Overlap input, output, and internal merging.



Steady State Operation



Partitioning Of Memory



- Need exactly 2 output buffers.
- Need at least k+1 (k is merge order) input buffers.
- 2k input buffers suffice.

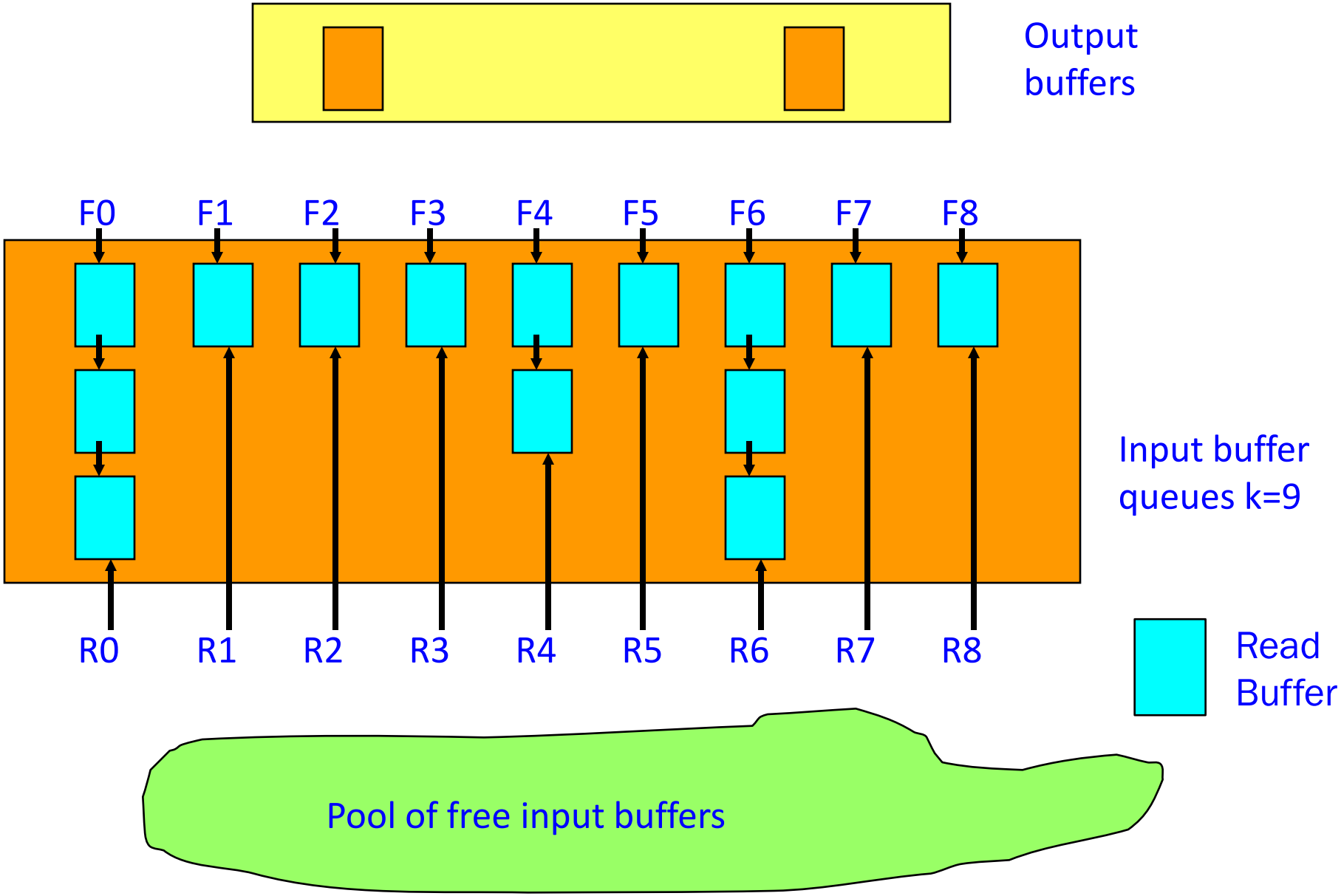
Number Of Input Buffers

- When 2 input buffers are dedicated to each of the k runs being merged, 2k buffers are not enough!
- Input buffers must be allocated to runs on an as needed basis.

Buffer Allocation

- When ready to read a buffer load, determine which run will exhaust first.
  - Examine key of the last record read from each of the k runs.
  - Run with smallest last key read will exhaust first.
  - Use an enforceable tie breaker.
- Next buffer load of input is to come from run that will exhaust first, allocate an input buffer to this run.

Buffer Layout



Initialize To Merge k Runs

- Initialize k queues of input buffers, 1 queue per run, 1 buffer per run.
- Input one buffer load from each of the k runs.
- Put k – 1 unused input buffers into pool of free buffers.
- Set activeOutputBuffer = 0.
- Initiate input of next buffer load from first run to exhaust. Use remaining unused input buffer for this input.

The Method kWayMerge

- k-way merge from input queues to the active output buffer.
- Merge stops when either the output buffer gets full or when an end-of-run key is merged into the output buffer.
- If merge hasn't stopped and an input buffer gets empty, advance to next buffer in queue and free empty buffer.

Merge k Runs

```
repeat
    kWayMerge;
    wait for input/output to complete;
    add new input buffer (if any) to queue for its run;
    determine run that will exhaust first;
    if (there is more input from this run)
        initiate read of next block for this run;
    initiate write of active output buffer;
    activeOutputBuffer = 1 – activeOutputBuffer;
until end-of-run key merged;
```

What Can Go Wrong?

- k-way merge from input queues to the active output buffer.
- Merge stops when either the output buffer gets full or when an end-of-run key is merged into the output buffer.
- If merge hasn't stopped and an input buffer gets empty, advance to next buffer in queue and free empty buffer. There may be no next buffer in the queue.

What Can Go Wrong?


```
repeat
    kWayMerge;
    wait for input/output to complete;
    add new input buffer (if any) to queue for its run;
    determine run that will exhaust first;
    if (there is more input from this run)
        initiate read of next block for this run;
    initiate write of active output buffer;
    activeOutputBuffer = 1 – activeOutputBuffer;
until end of run key merged;
```

There may be no free input buffer to read into.

kWayMerge

- If merge hasn't stopped and an input buffer gets empty, advance to next buffer in queue and free empty buffer. There may be no next buffer in the queue.

- If this type of failure were to happen, using two different and valid analyses, we will end up with inconsistent counts of the amount of data available to kWayMerge.
- Data available to kWayMerge is data in
  - Input buffer queues.
  - Active output buffer.
  - Excludes data in buffer being read or written.

No Next Buffer In Queue

```
repeat
   kWayMerge;
    wait for input/output to complete;
    add new input buffer (if any) to queue for its run;
    determine run that will exhaust first;
    if (there is more input from this run)
        initiate read of next block for this run;
    initiate write of active output buffer;
    activeOutputBuffer = 1 – activeOutputBuffer;
until end-of-run key merged;
```

- Exactly k buffer loads available to kWayMerge.

kWayMerge

- If merge hasn't stopped and an input buffer gets empty, advance to next buffer in queue and free empty buffer. There may be no next buffer in the queue.

- Alternative analysis of data available to kWayMerge at time of failure.
  - < 1 buffer load in active output buffer
  - <= k – 1 buffer loads in remaining k – 1 queues
  - Total data available to k-way merge is < k buffer loads.

Merge k Runs

initiate read of next block for this run;

There may be no free input buffer to read into.

- Suppose there is no free input buffer.
- One analysis will show there are exactly $k + 1$ buffer loads in memory (including newly read input buffer) at time of failure.
- Another analysis will show there are $> k + 1$ buffer loads in memory at time of failure.
- Note that at time of failure there is no buffer being read or written.

No Free Input Buffer

```
repeat
   kWayMerge;
    wait for input/output to complete;
    add new input buffer (if any) to queue for its run;
    determine run that will exhaust first;
    if (there is more input from this run)
        initiate read of next block for this run;
    initiate write of active output buffer;
    activeOutputBuffer = 1 – activeOutputBuffer;
until end-of-run key merged;
```

- Exactly k + 1 buffer loads in memory.

Merge k Runs

initiate read of next block for this run;

There may be no free input buffer to read into.

- Alternative analysis of data in memory.
  - 1 buffer load in the active output buffer.
  - 1 input queue may have an empty first buffer.
  - Remaining $k - 1$ input queues have a nonempty first buffer.
  - Remaining k input buffers must be in queues and full.
  - Since $k > 1$, total data in memory is $> k + 1$ buffer loads.

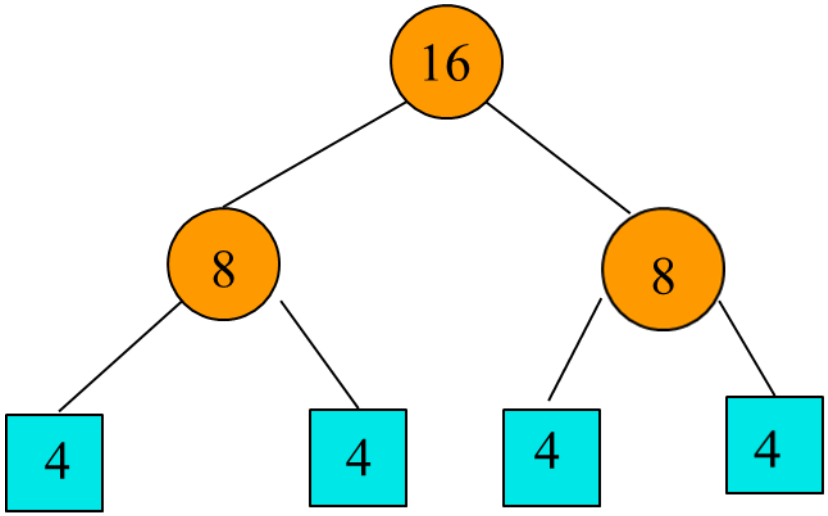
Minimize Wait Time For I/O To Complete

Time to fill an output buffer

~ time to read a buffer load

~ time to write a buffer load

Initializing For Next k-way Merge



Initializing For Next k-way Merge

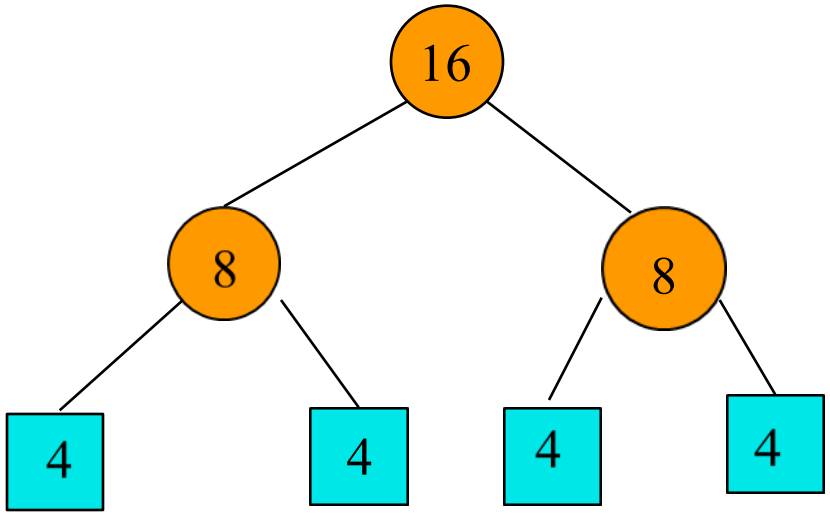
Change

```
if (there is more input from this run)
        initiate read of next block for this run;
```

to

```
if (there is more input from this run)
        initiate read of next block for this run;
else
        initiate read of a block for the next k-way merge if
        on current read disk;
```

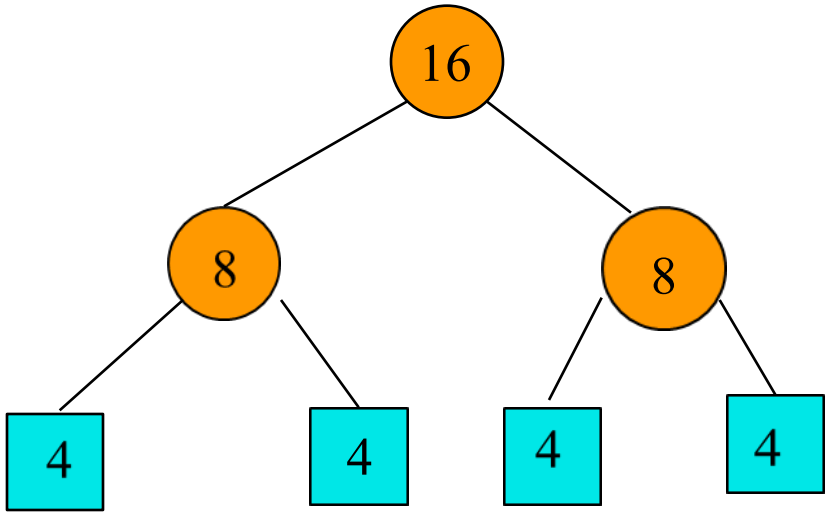
Phase 2 (Run Merging) Time



Using 1 output and k input buffers (k is merge order) and1 disk

$64t_{IO} + 32t_{IM}$

Phase 2 (Run Merging) Time



Using 2 output and 2k input buffers (k is merge order) and 2 disks

$\sim 38 t_{IO}$

External Merge Sort Steps

- Run Generation
  - Memory load scheme
  - Loser tree
- Merge Sequence For Runs & Run Distribution
  - Huffman Trees
  - Runs for external nodes on even levels on one disk; others on second disk if you have 2 disks
- Run merging
  - Loser tree
  - Double buffering if you have two disks

Double-Ended Priority Queues

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

General Methods

- Dual min and max single-ended priority queues.
- Correspondence based min and max single-ended priority queues.

Specialized Structures

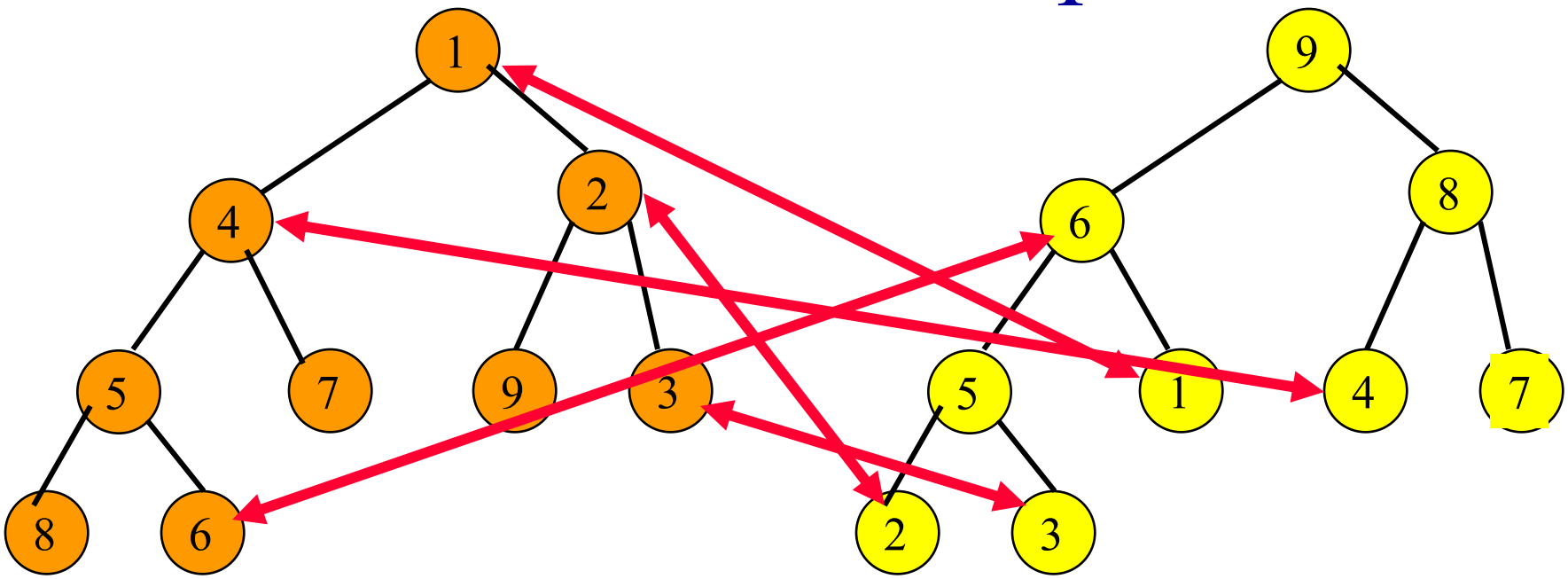
- Symmetric min-max heaps.
- Min-max heaps.
- Deaps.
- ✓Interval heaps.

…

Dual Single-Ended Priority Queues

- Use a min and a max single-ended priority queue.
- Single-ended priority queue also must support an arbitrary remove.
- Each element is in both single-ended priority queues.
- Each priority queue node has a pointer to the node in the other priority queue that has the same element.

9-Element Example



Min Heap

Max Heap

- Only 5 of 9 two-way pointers shown.
- Insert, remove min, remove max, initialize.
- Operation cost is more than doubled relative to heap.
- Space for 2n nodes.

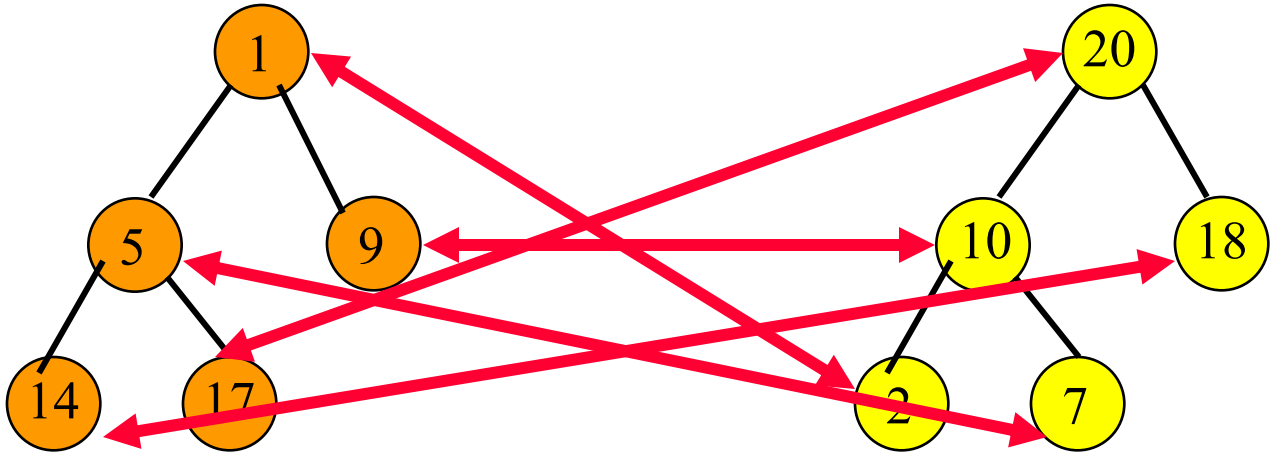
Correspondence Structures

- Use a min and a max single-ended priority queue.
- At most 1 element is in a buffer.
- Remaining elements are in the single-ended priority queues, which may be of different size.
- No element is in both the min and max single-ended priority queue.
- Establish a correspondence between the min and max single-ended priority queues.
  - Total correspondence.
  - Leaf correspondence.
- Single-ended priority queue also must support an arbitrary remove.

Total Correspondence

- The min- and max-priority queues are of the same size.
- Each element of the min priority queue is paired with a different and >= element in the max priority queue.

Total Correspondence Example

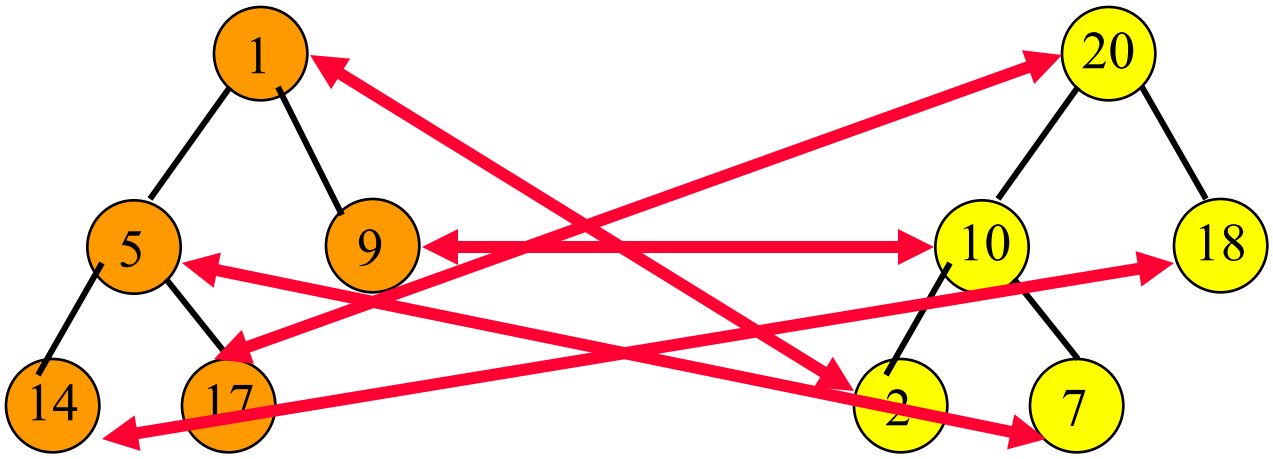


Min Heap

Max Heap

Buffer = 12

Insert



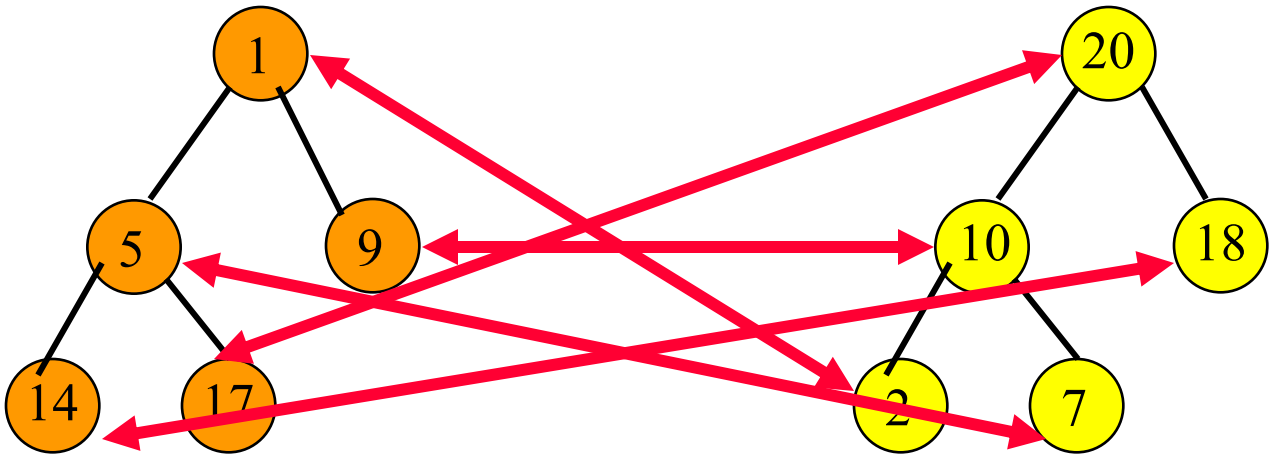
Min Heap

Max Heap

Buffer = 12

- Buffer empty => place in buffer.
- Else, insert smaller of new and buffer elements into min priority queue and larger into max priority queue; establish correspondence between the 2 elements.

Remove Min



Min Heap

Max Heap

Buffer = 12

- Buffer is min => empty buffer.
- Else, remove min from min PQ as well as corresponding element from max PQ; reinsert corresponding element.

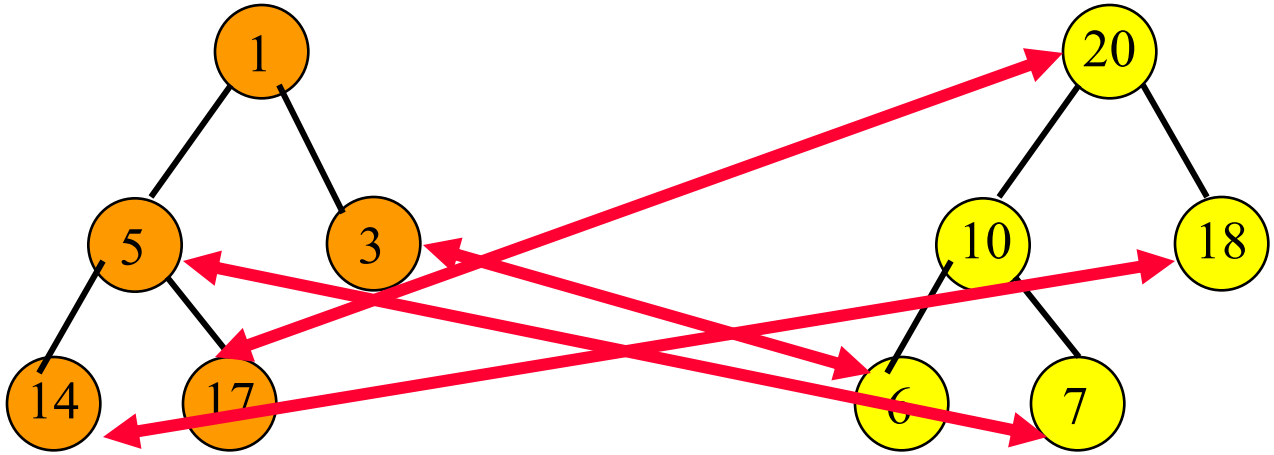
Leaf Correspondence

- Min- and max-priority queues may have different size.
- Each leaf element of the min priority queue is paired with a different and >= element in the max priority queue.
- Each leaf element of the max priority queue is paired with a different and <= element in the min priority queue.

Added Restrictions

- When an element is inserted into a single-ended PQ, only the newly inserted element can become a new leaf.
- When an element is deleted from a single-ended PQ, only the parent of the deleted element can become a new leaf.
- Min and max heaps do not satisfy these restrictions. So, leaf correspondence is harder to implement using min and max heaps.

Leaf Correspondence Example

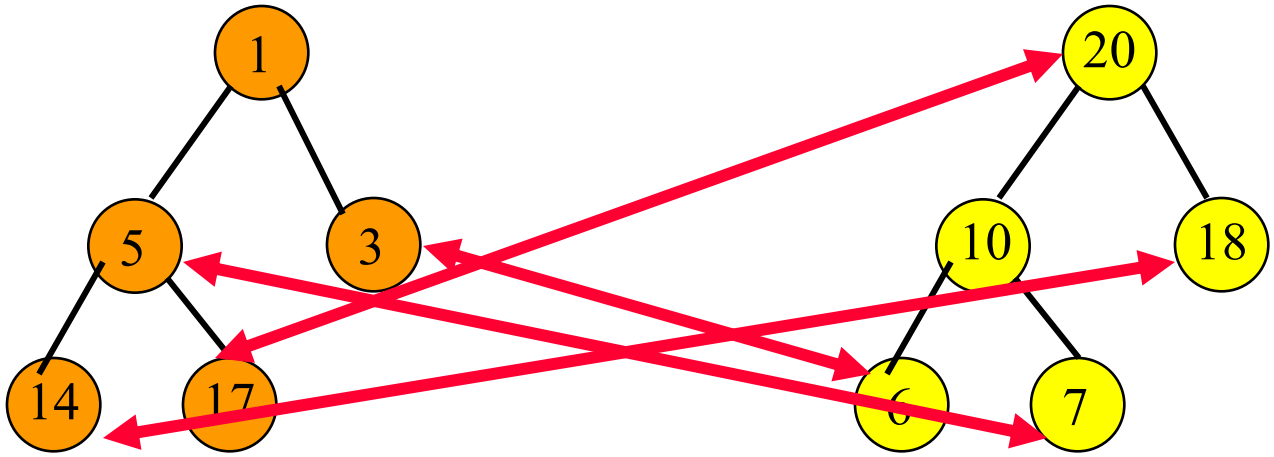


Min Heap

Max Heap

Buffer = 12

Insert



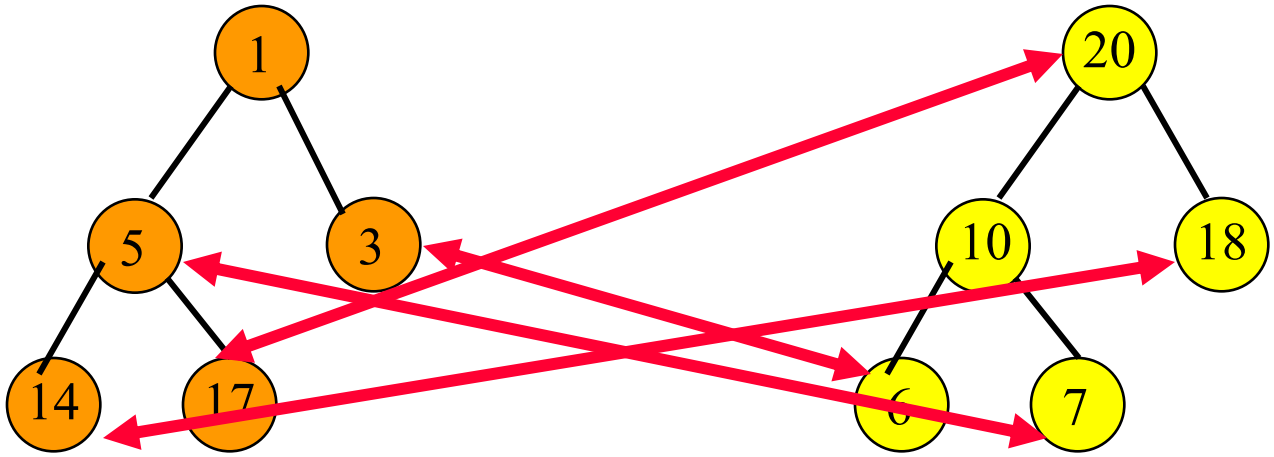
Min Heap

Max Heap

Buffer = 12

- Buffer empty => place in buffer.
- Else, insert smaller of new and buffer elements into min priority queue; insert larger into max priority queue only if smaller one is a leaf.

Insert



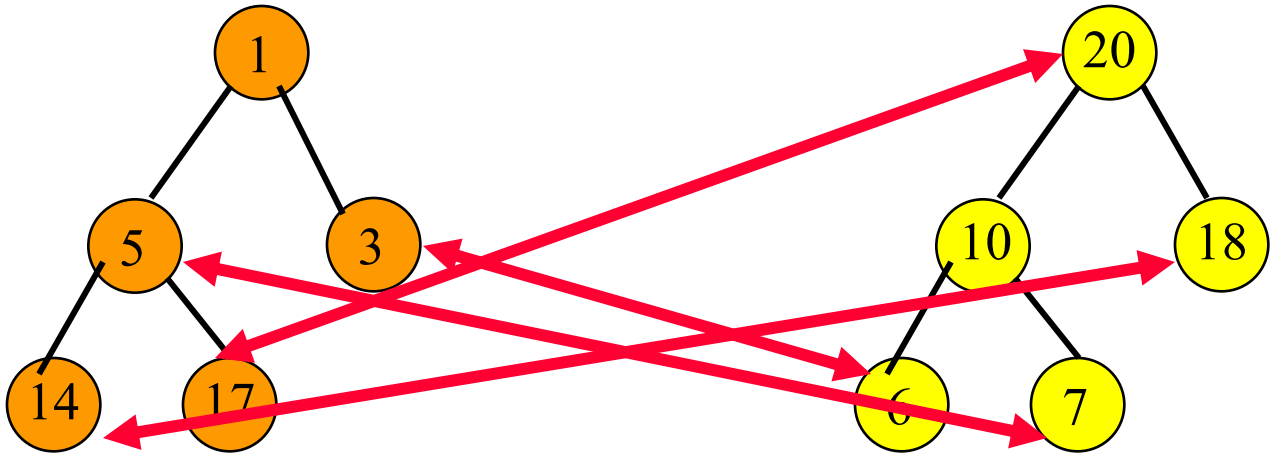
Min Heap

Max Heap

Buffer = 12

- Case when min and/or max heap originally have an even number of elements is more involved, because a nonleaf may become a leaf. See reference.

Remove Min



Min Heap

Max Heap

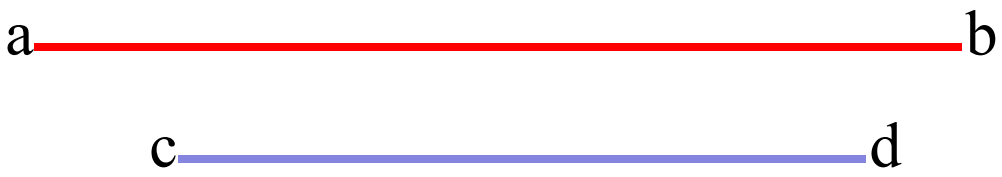
Buffer = 12

- Buffer is min => empty buffer.
- Else, remove min from min PQ as well as corresponding element (if any and a leaf) from max PQ; reinsert removed corresponding element (see reference for details).

Interval Heaps

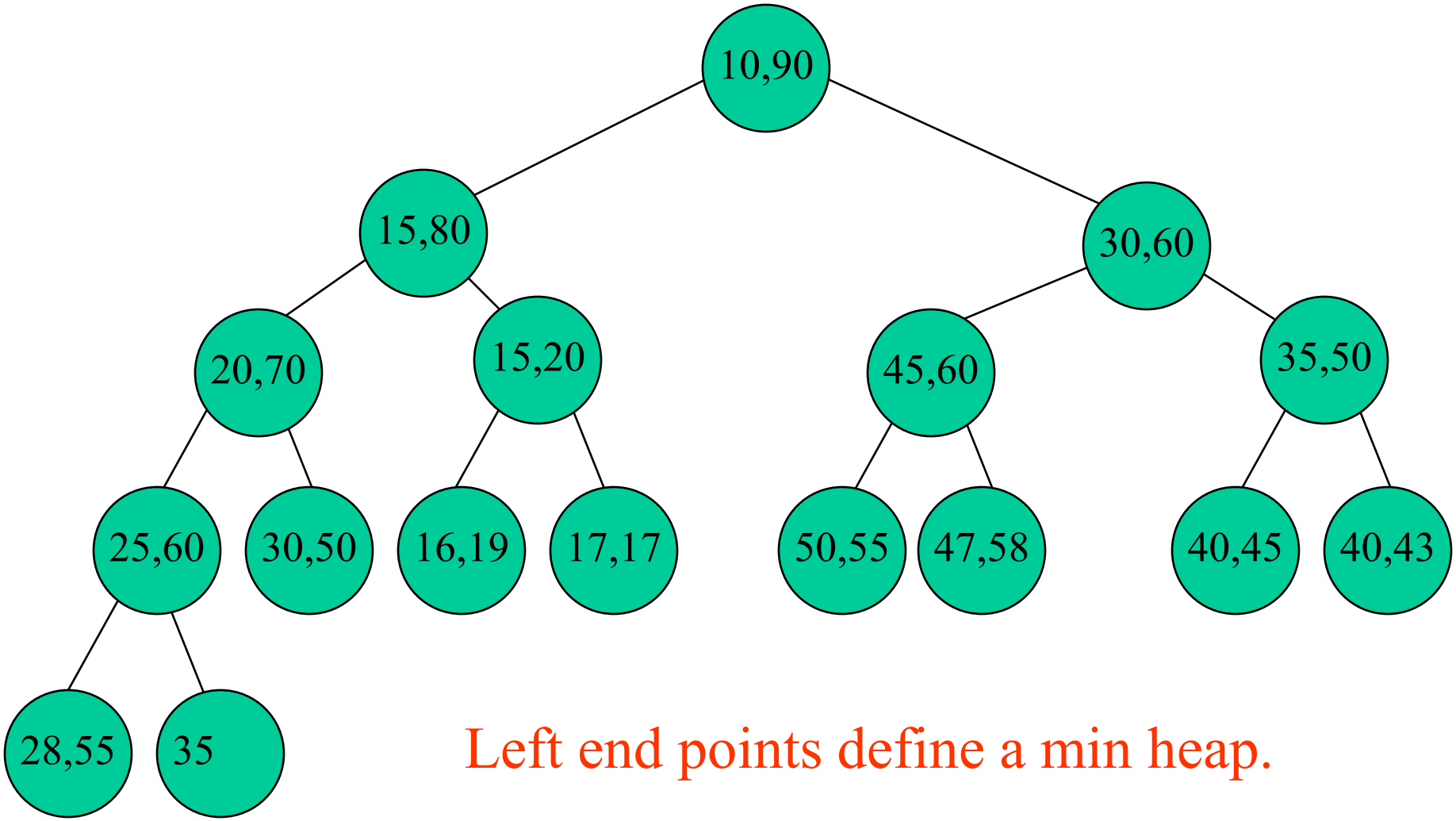
- Complete binary tree.
- Assume not empty.
- Each node (except possibly last one) has 2 elements.
- Last node has 1 or 2 elements.
- Let a and b be the elements in a node P, a <= b.
- [a, b] is the interval represented by P.
- The interval represented by a node that has just one element a is [a, a].
- 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.

Interval



- [c,d] is contained in [a,b]
- a <= c
- d <= b

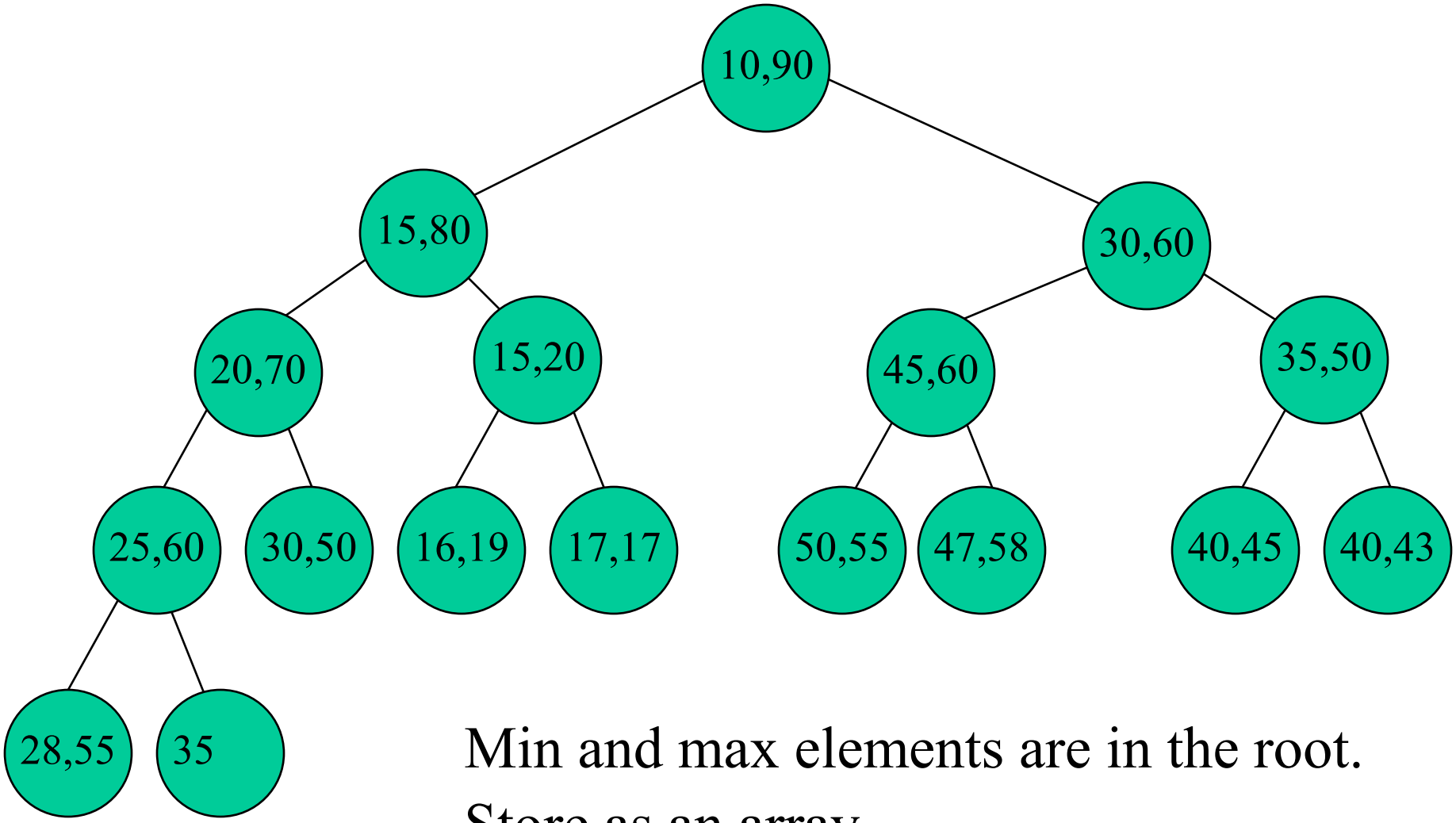
Example Interval Heap



Left end points define a min heap.

Right end points define a max heap.

Example Interval Heap

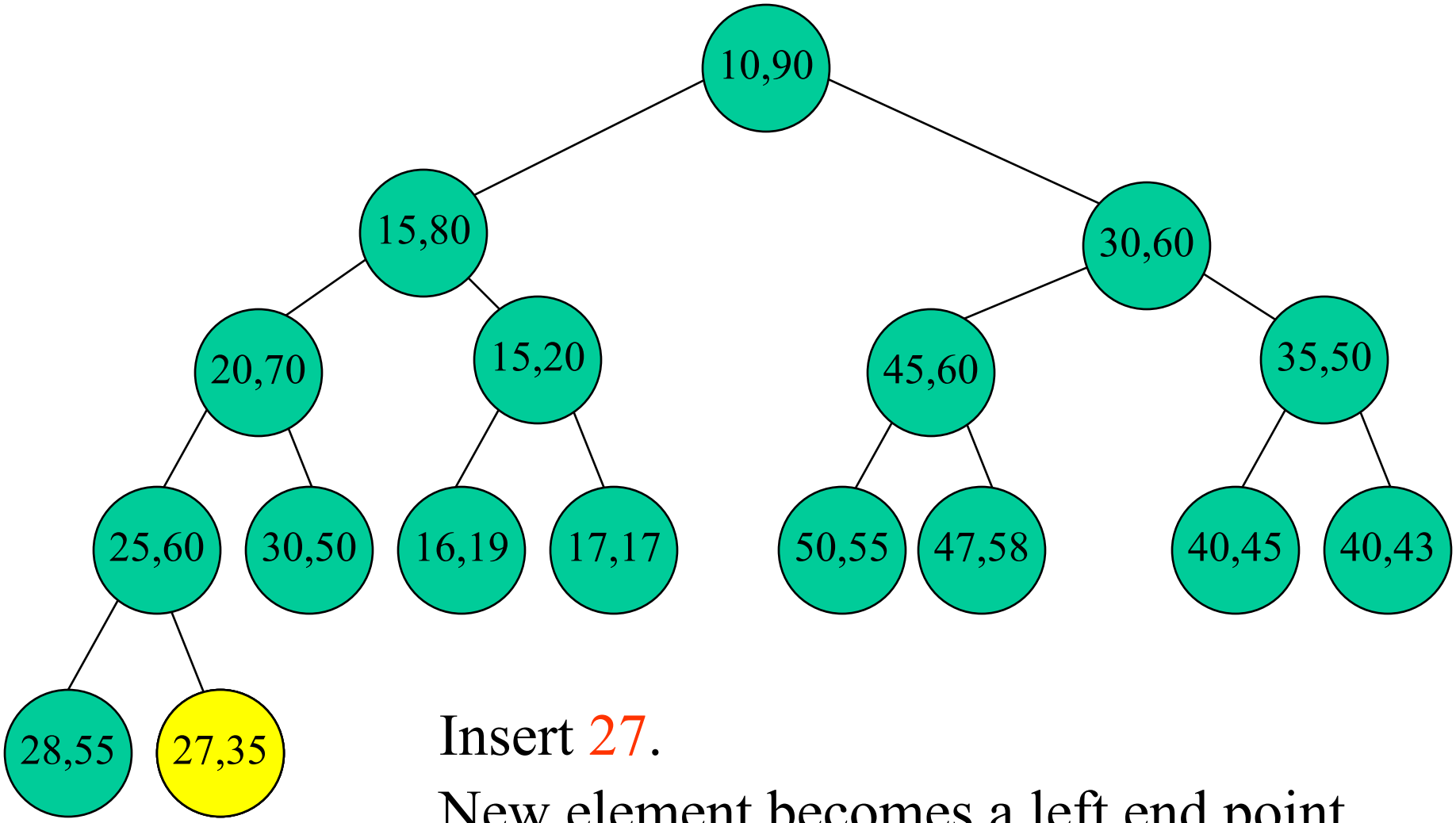


Min and max elements are in the root.

Store as an array.

Height is ~$\log_2 n$.

Insert An Element

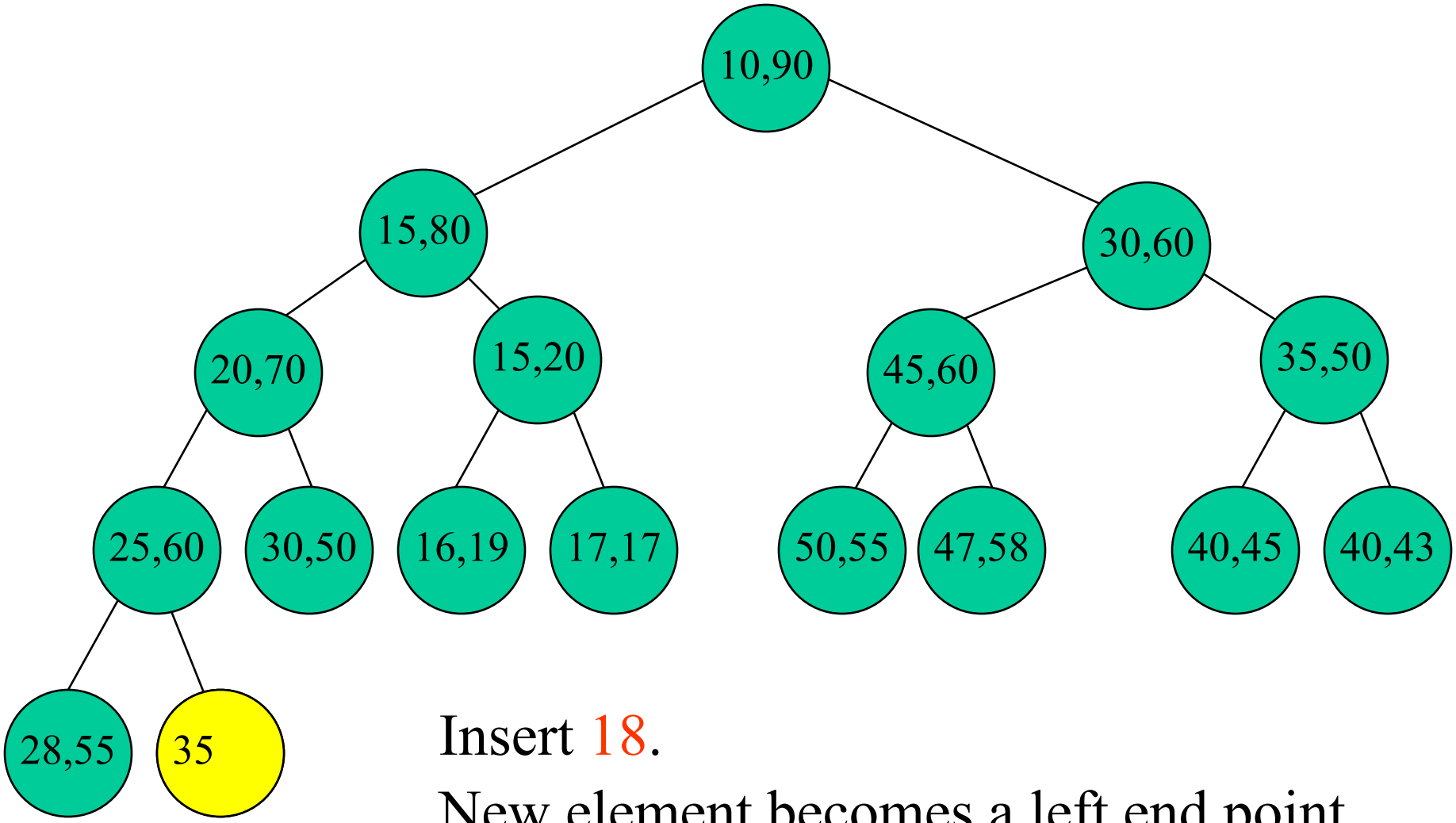


Insert 27.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

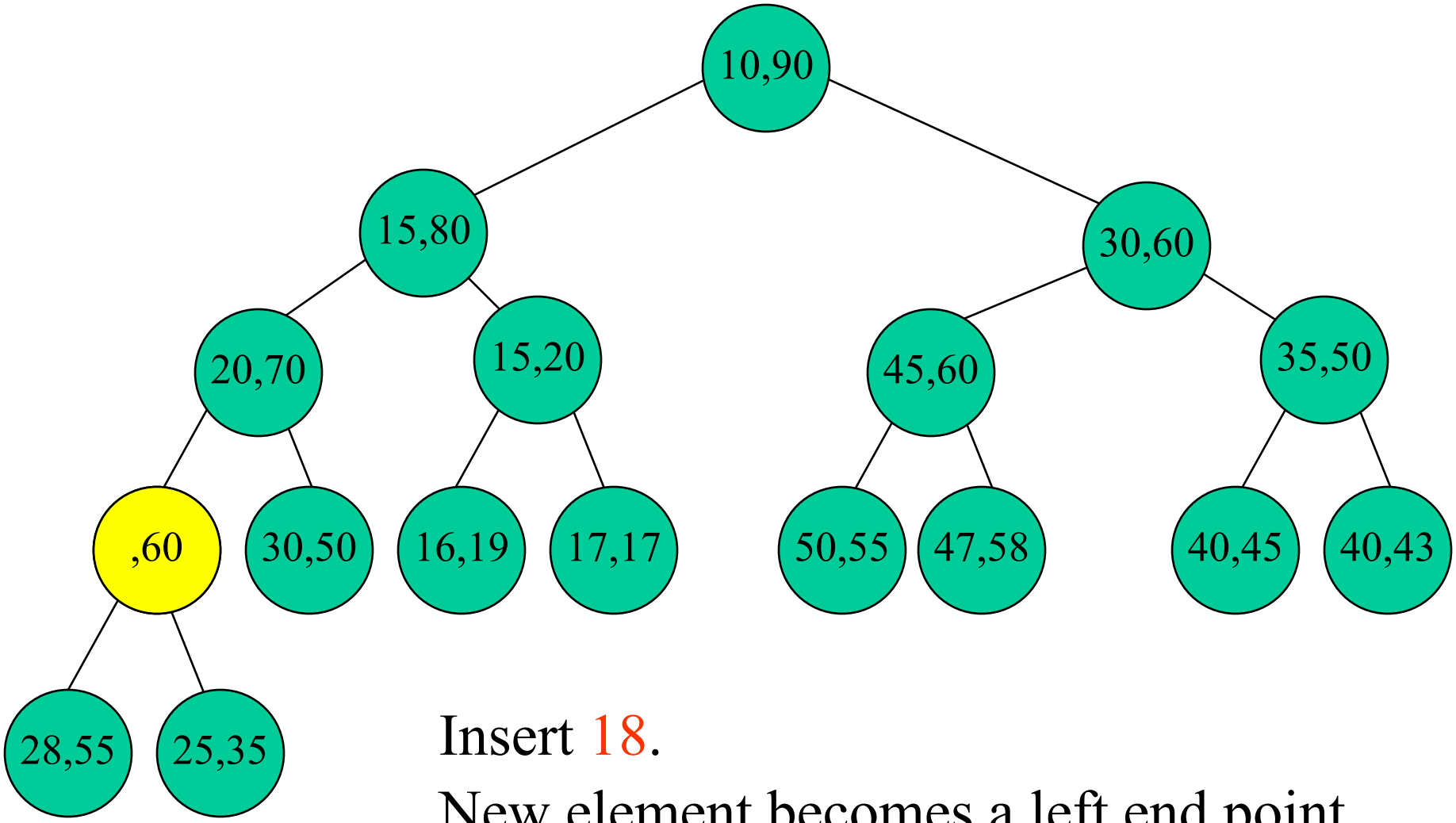


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

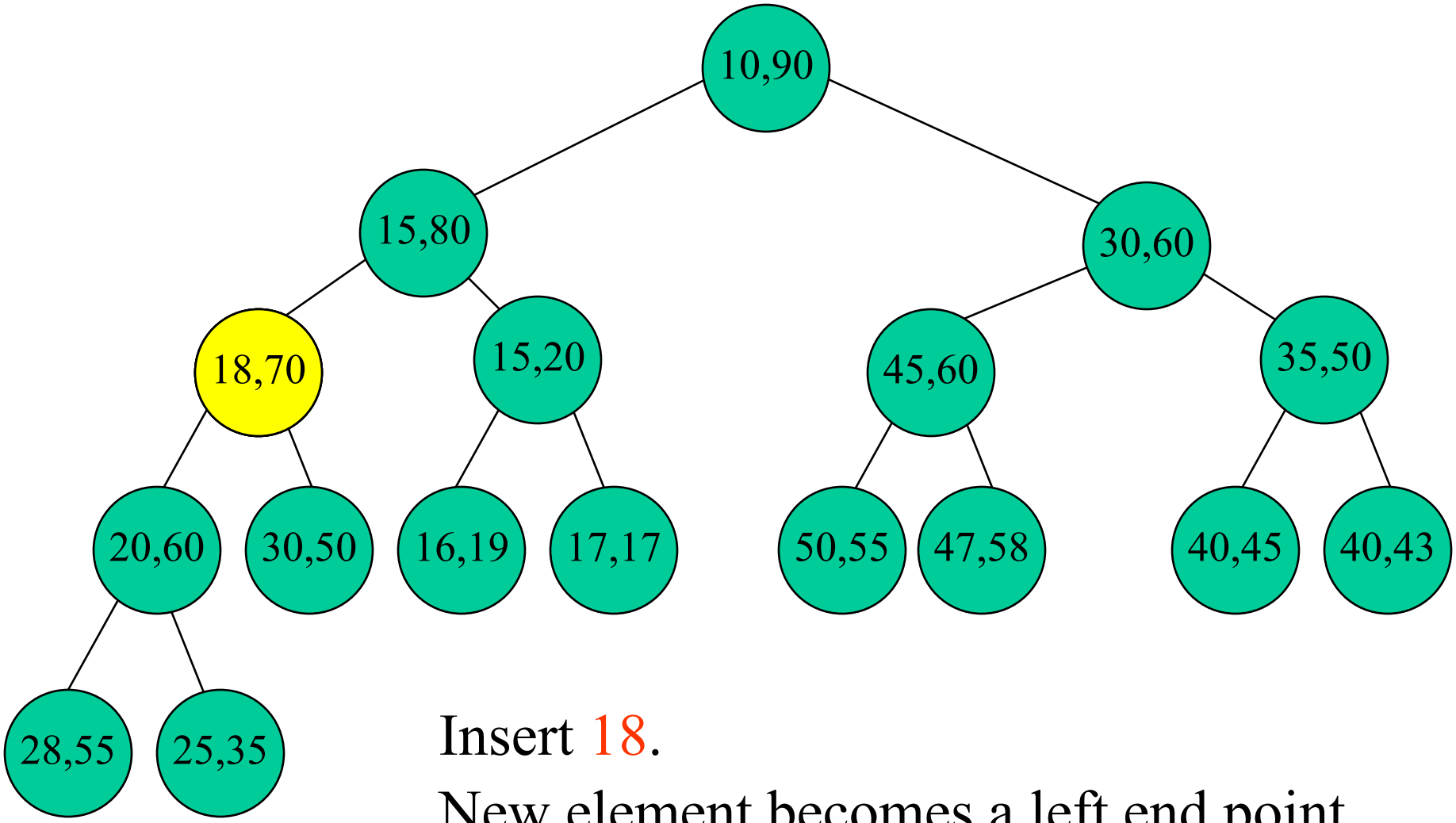


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

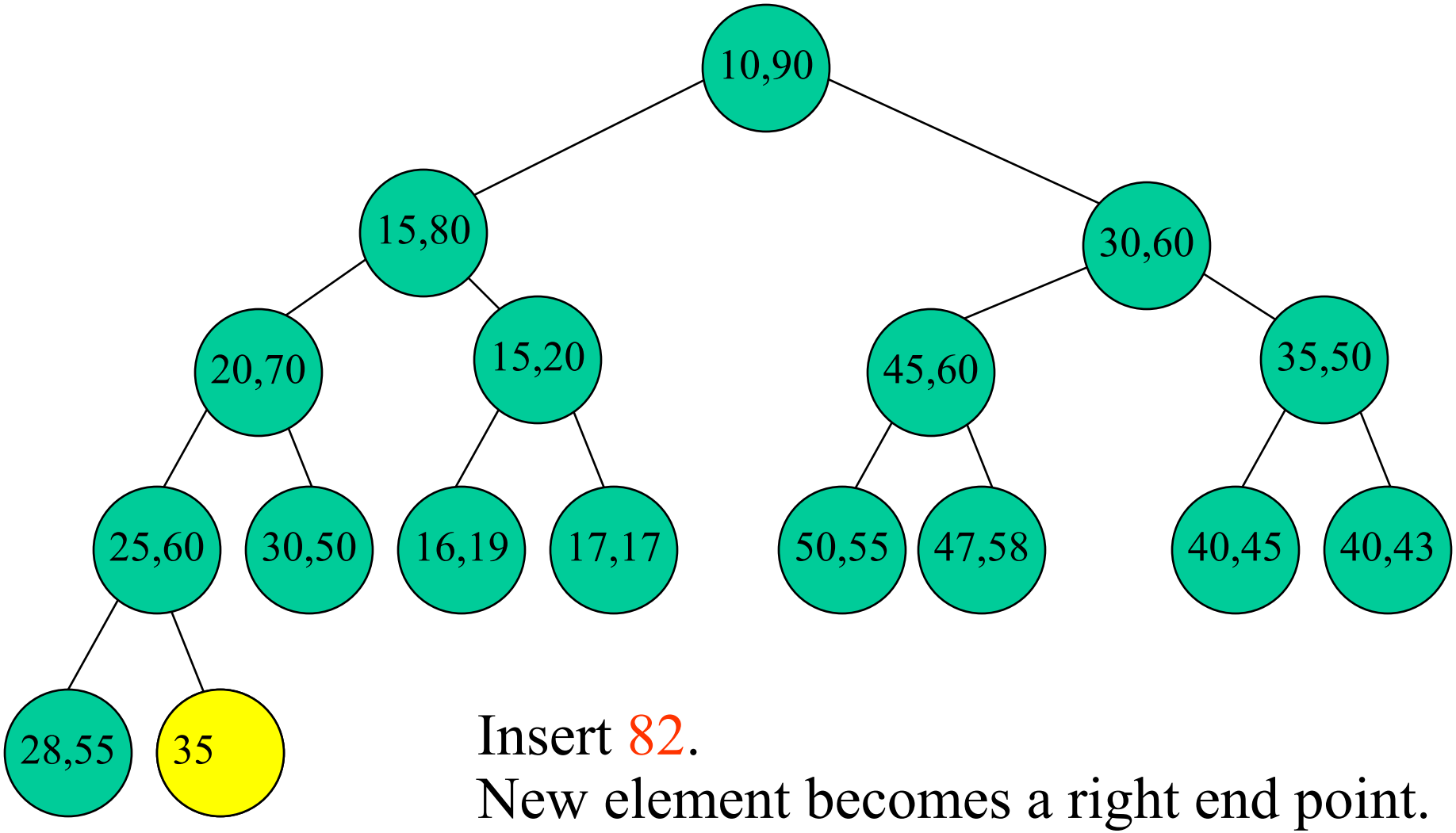


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

Yet Another Insert

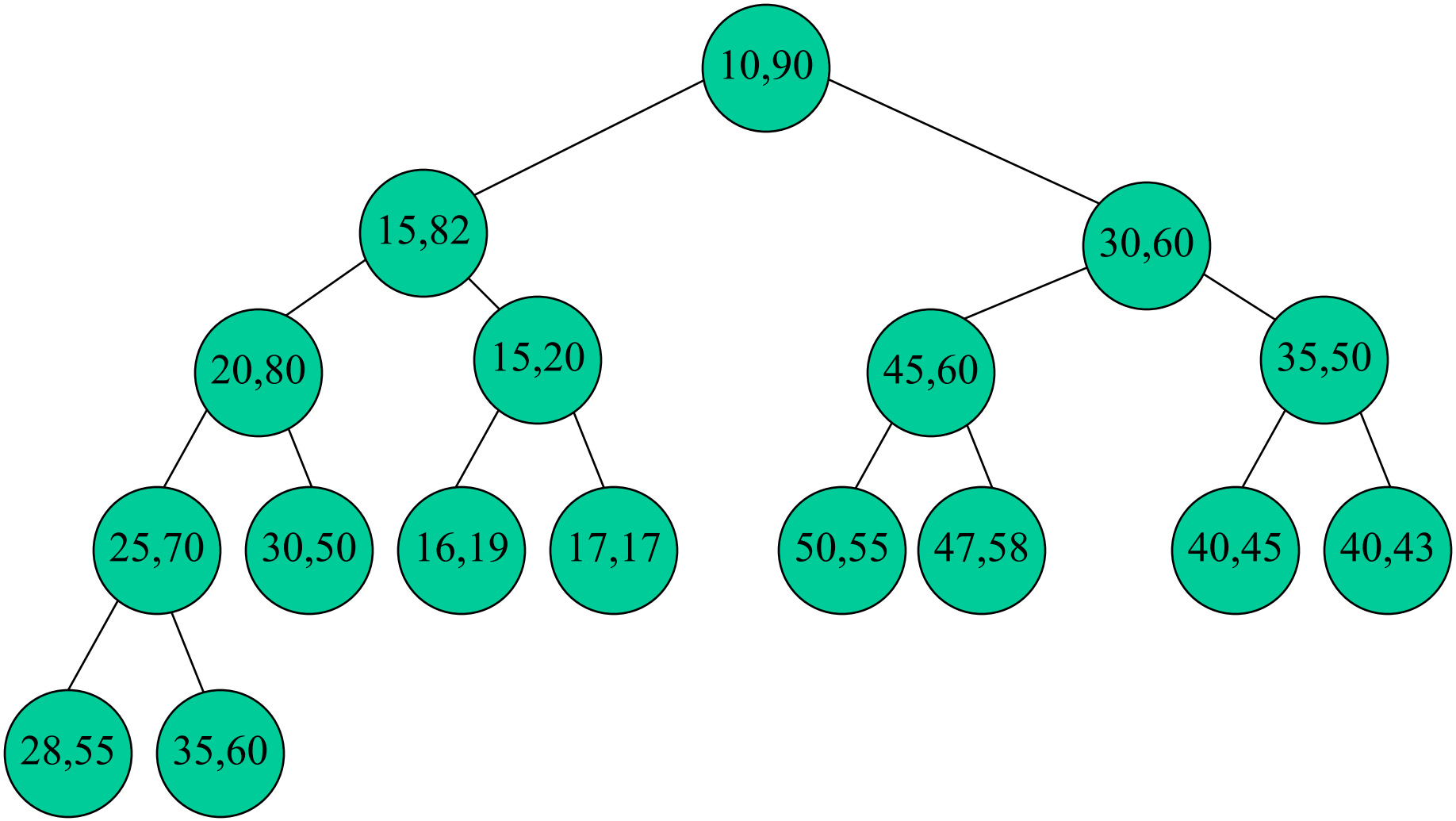


Insert 82.

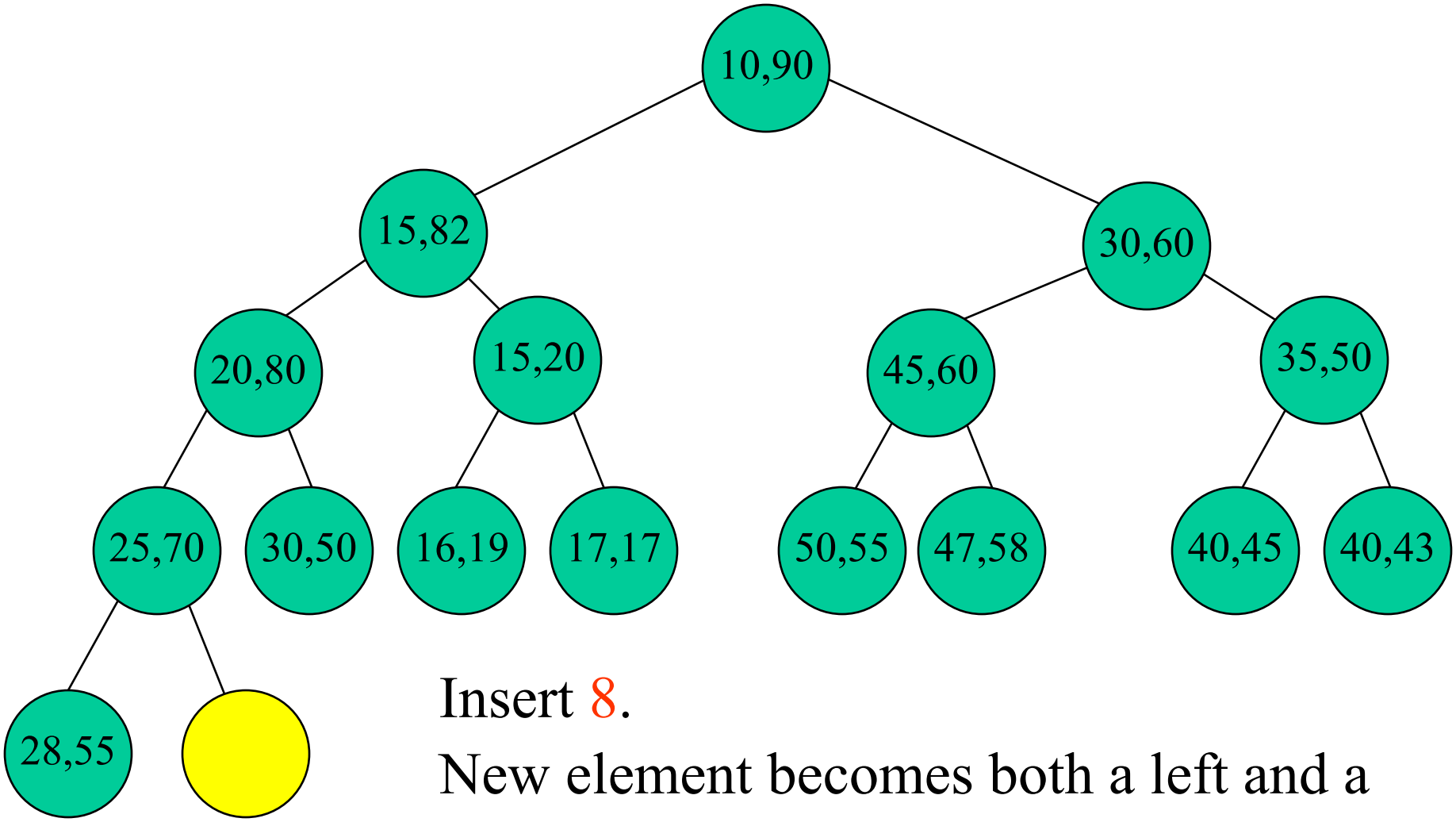
New element becomes a right end point.

Insert new element into max heap.

After 82 Inserted



One More Insert Example

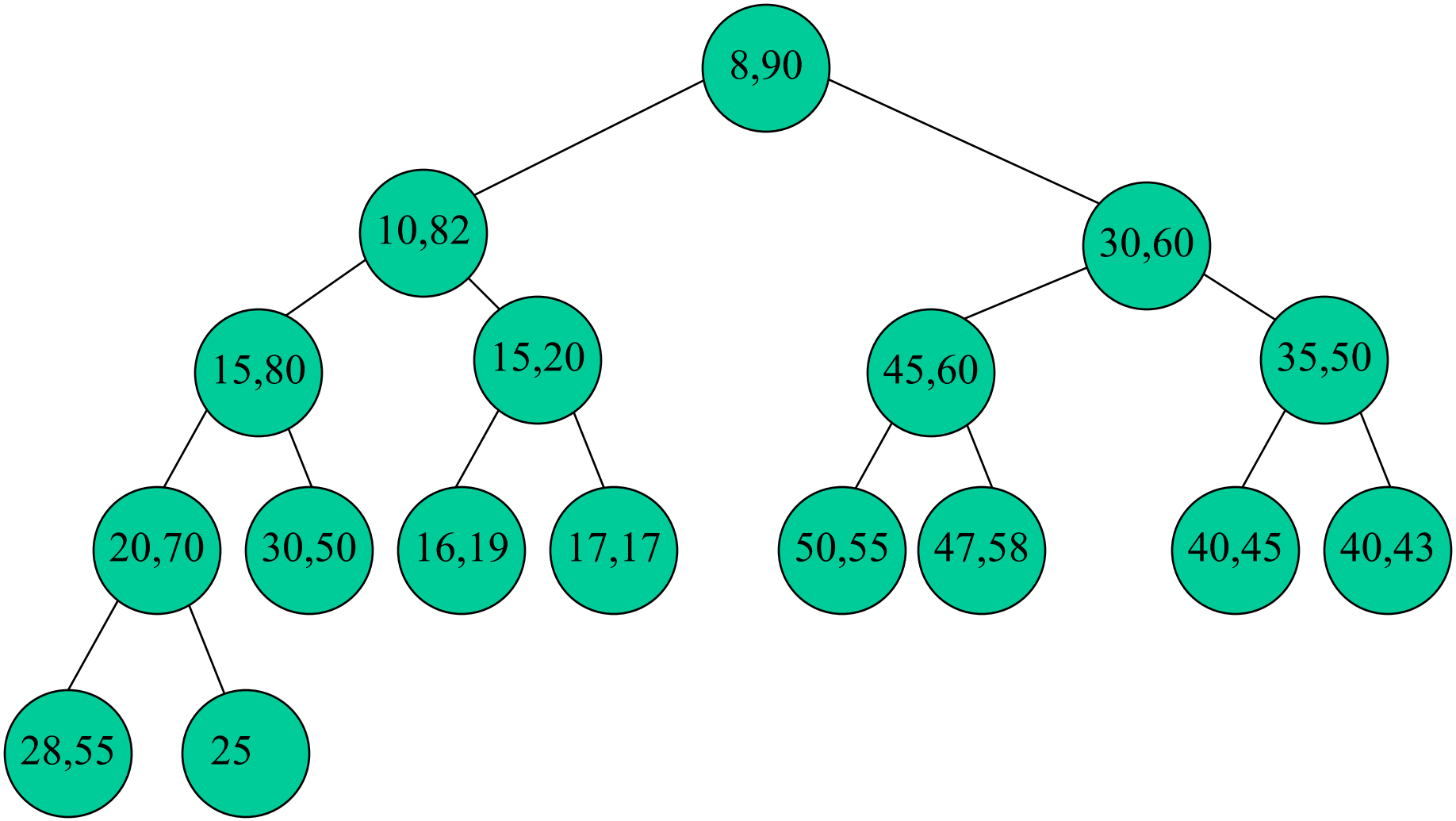


Insert 8.

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

Insert new element into min heap.

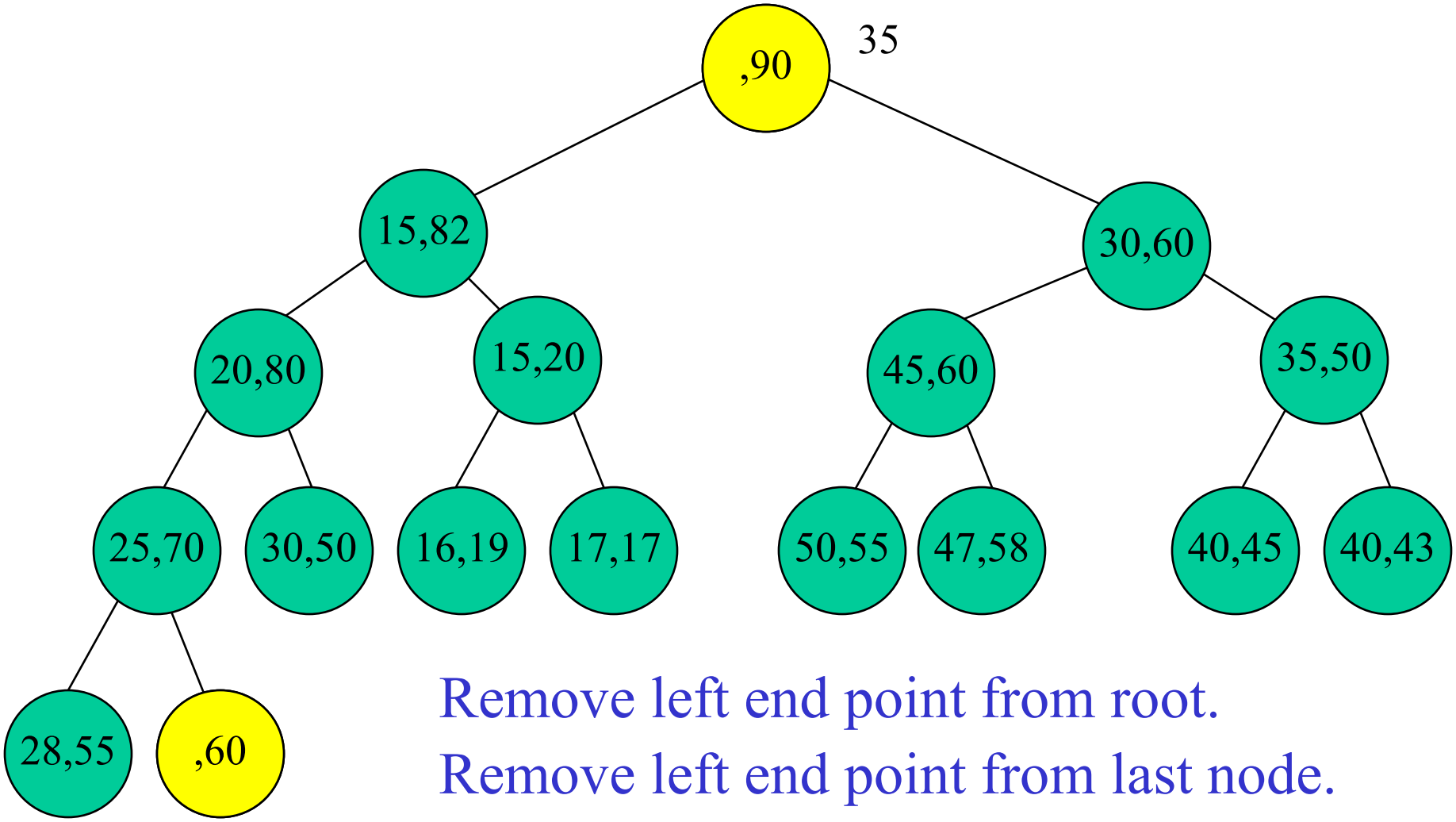
After 8 Is Inserted



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



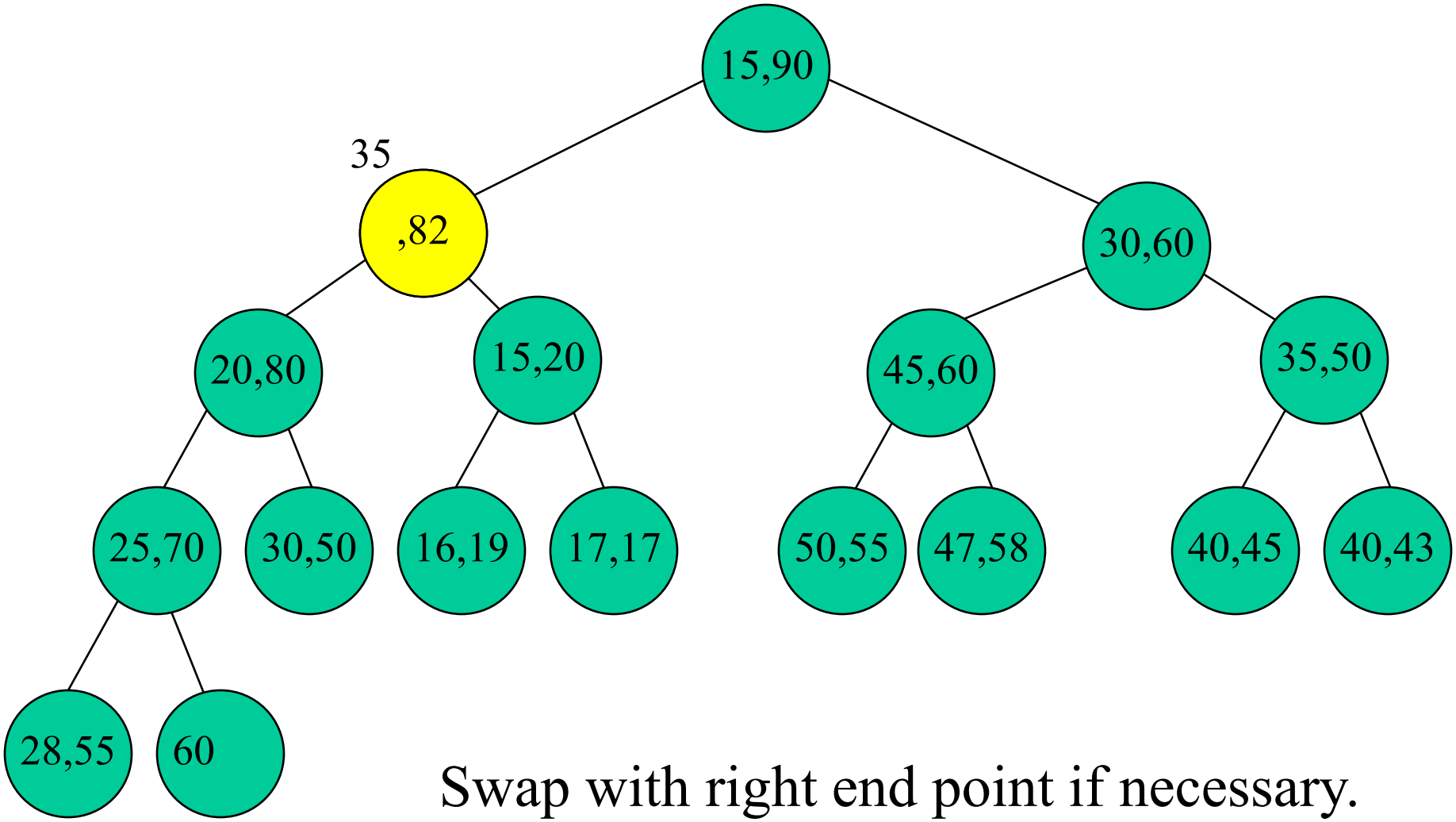
Remove left end point from root.

Remove left end point from last node.

Delete last node if now empty.

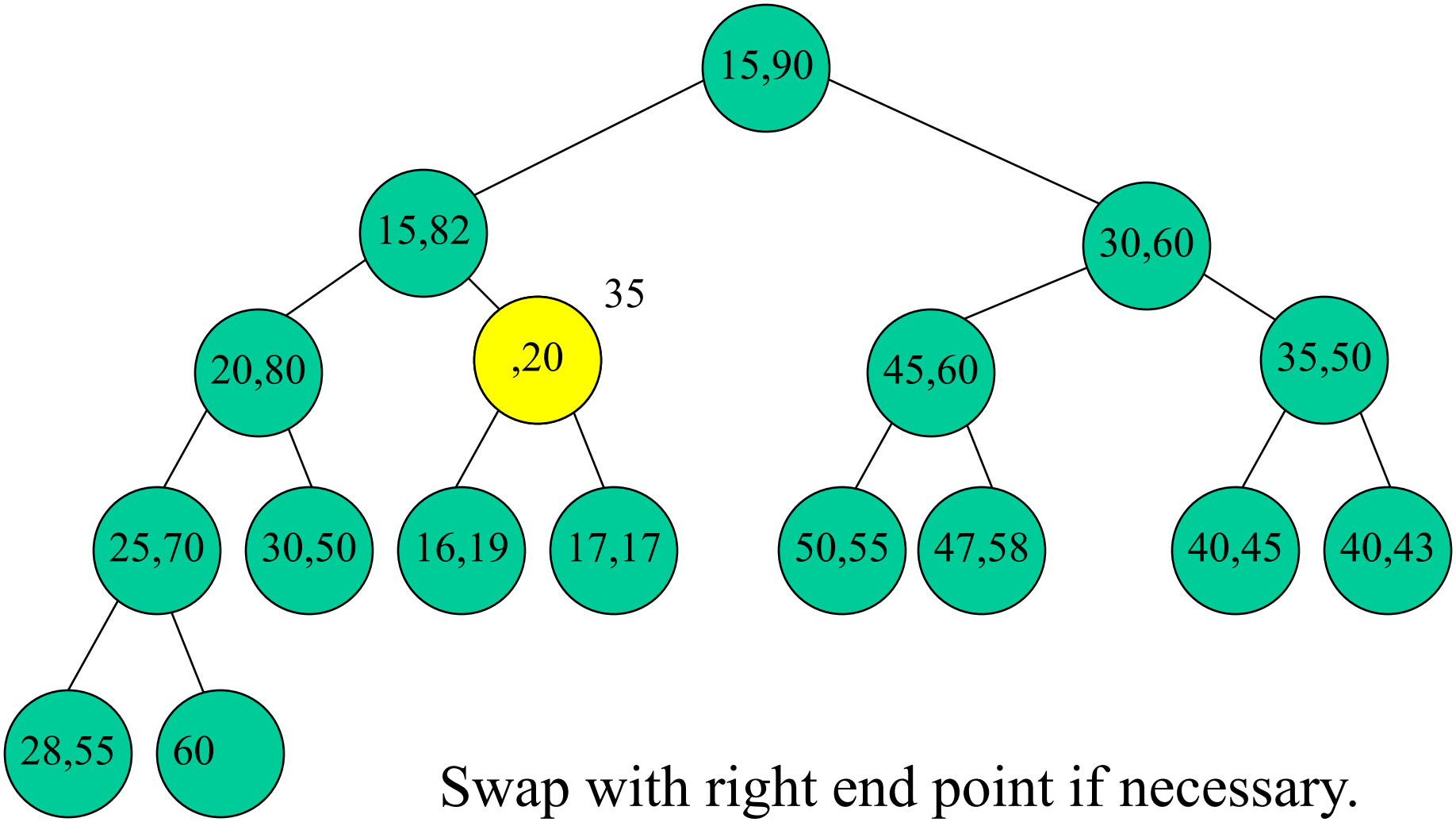
Reinsert into min heap, begin at root.

Remove Min Element Example



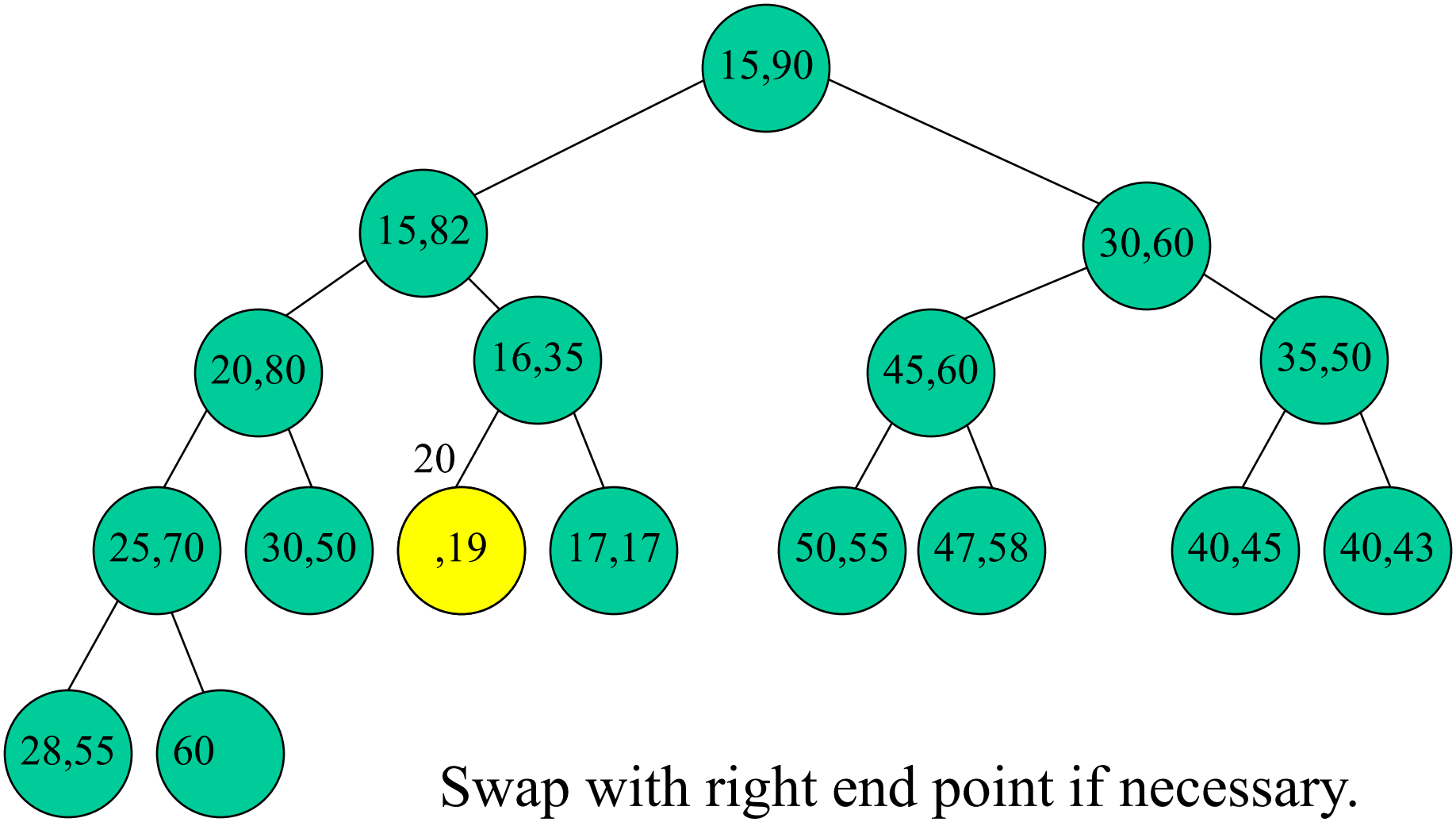
Swap with right end point if necessary.

Remove Min Element Example



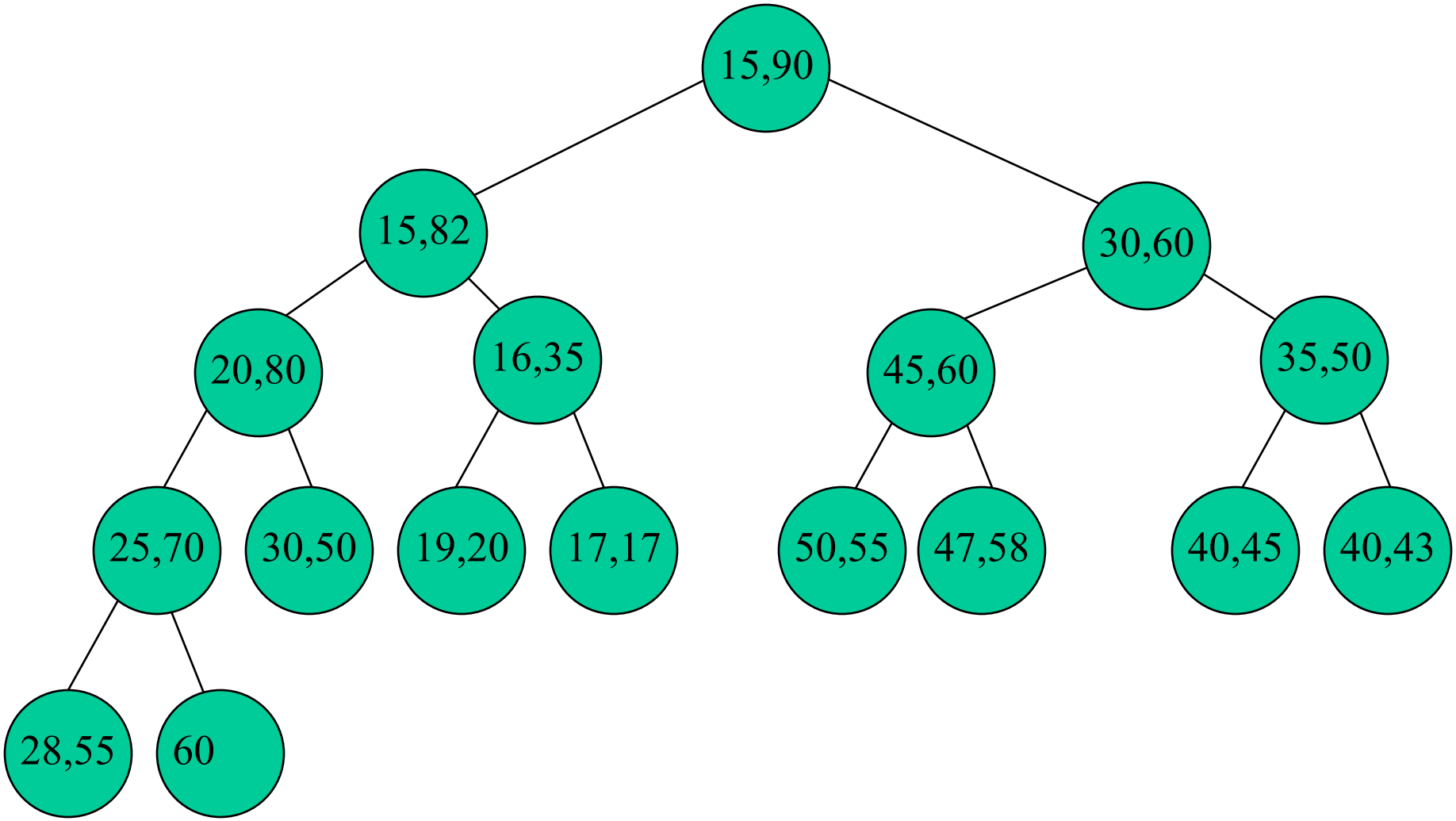
Swap with right end point if necessary.

Remove Min Element Example

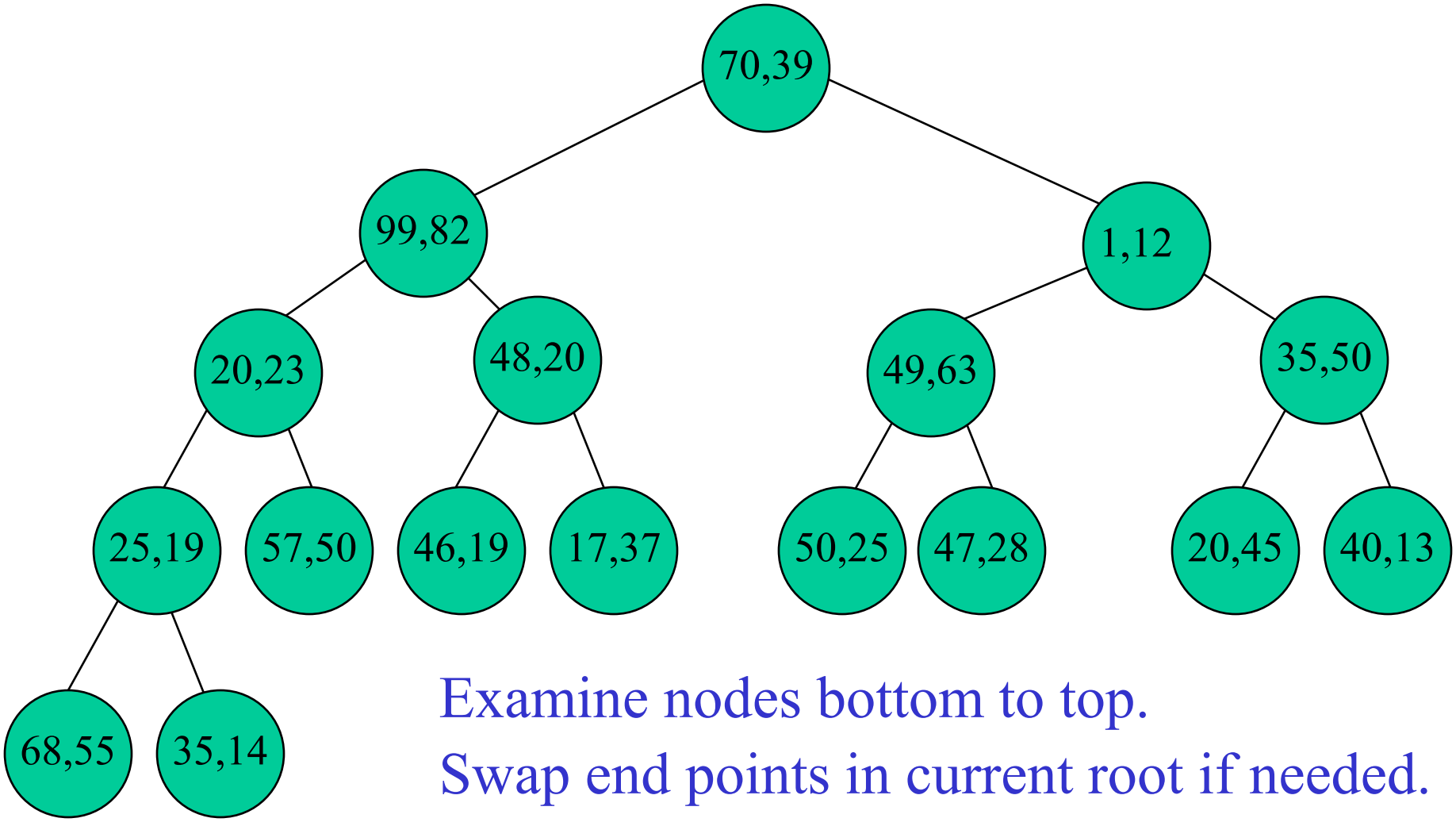


Swap with right end point if necessary.

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.

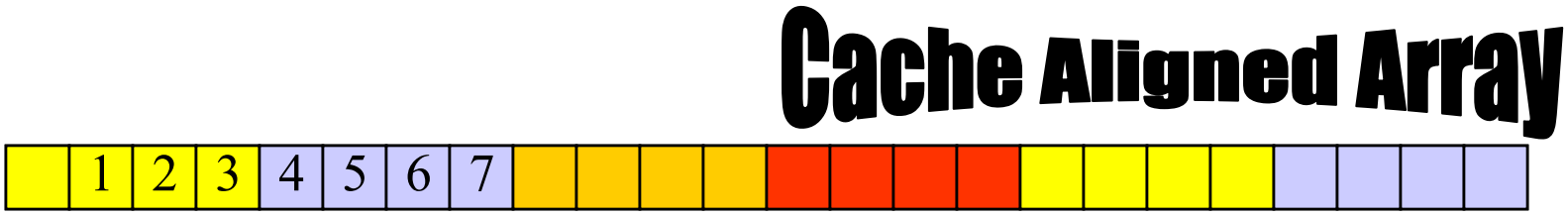
Reinsert right end point into max heap.

Cache Optimization

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

Cache Aligned Array

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



- A remove min (max) has ~h cache misses on average.
  - Root and its children are in the same cache line.
  - ~$\log_2 n$ 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.
- $\text{Parent}(i) = \text{ceil}((i - 1)/d)$.
- Children are $d*(i - 1) + 2, \ldots, \min\{d*i + 1, n\}$.
- Height is $\log_d n$.
- Height of 4-ary heap is half that of 2-ary heap.

d = 4, 4-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).

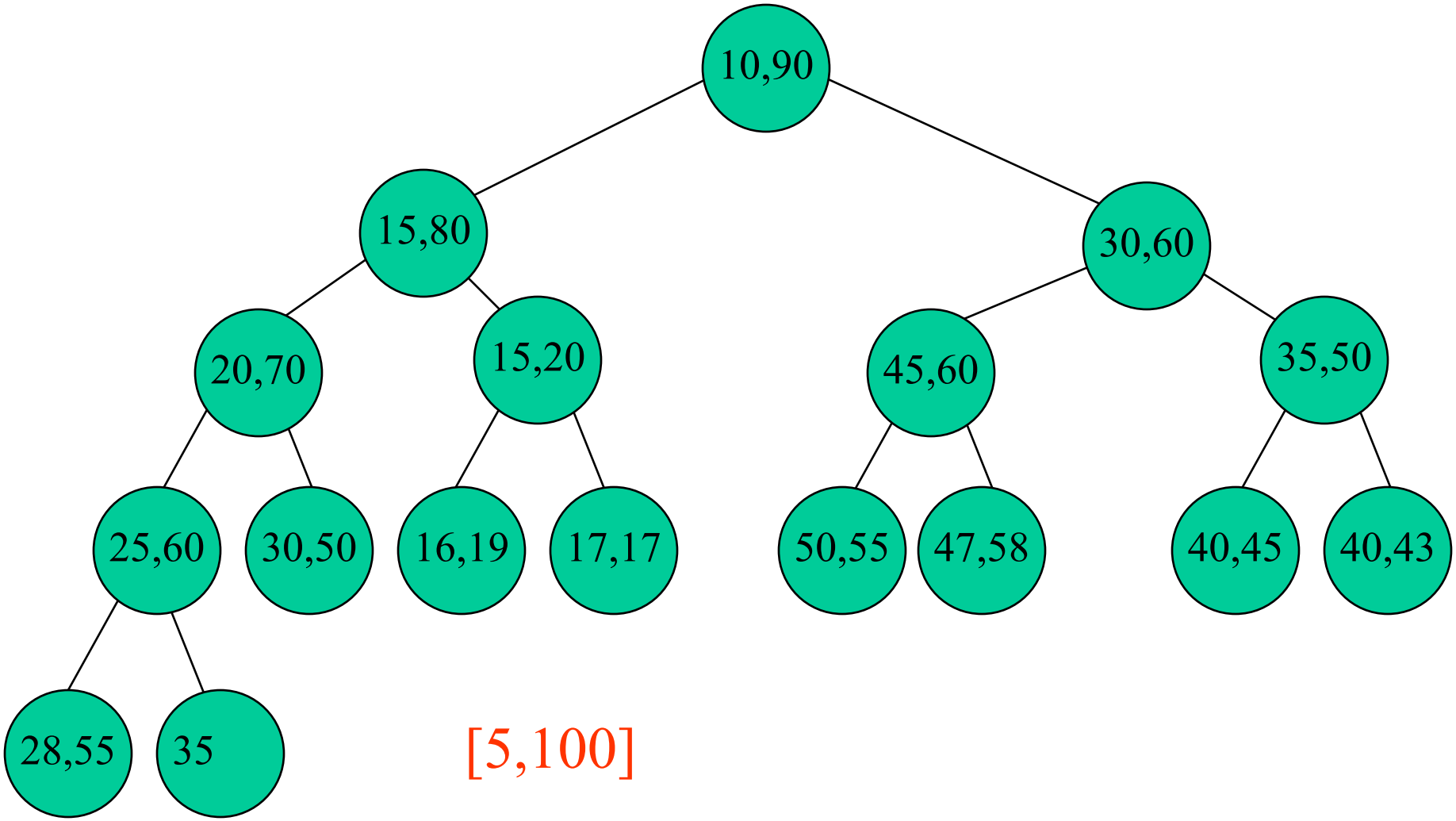
d-ary Heap Performance

- Speedup of about 1.5 to 1.8 when sorting a large number of 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.

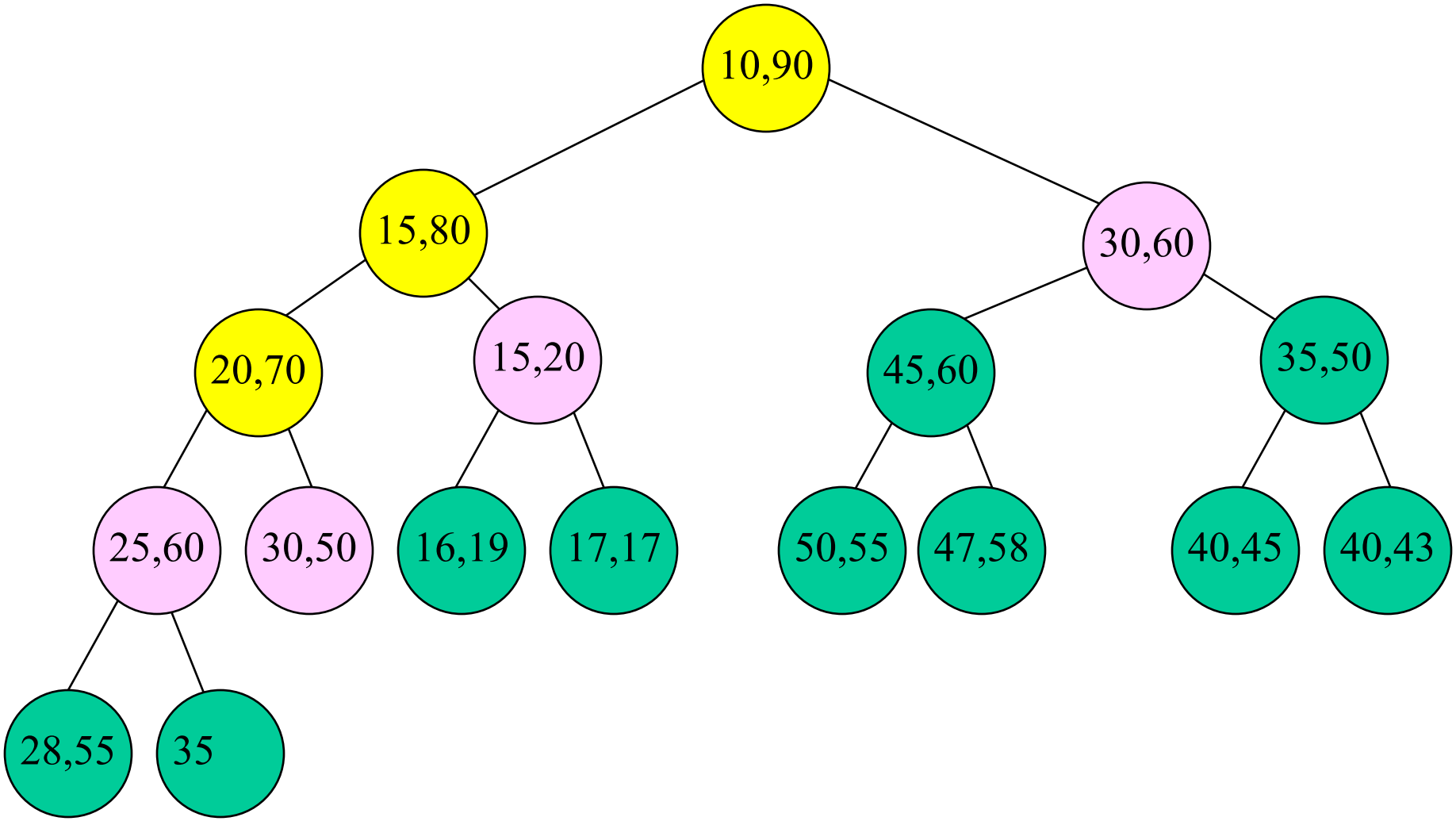
Example



[5,100]

[2,65]

Example


10,90
15,80
30,60
20,70
15,20
45,60
35,50
25,60
30,50
16,19
17,17
50,55
47,58
40,45
40,43
28,55
35

[2,65]

Leftist Trees

Linked binary tree.

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

- insert
- remove min (or max)
- arbitrary remove (need parent pointers)
- initialize

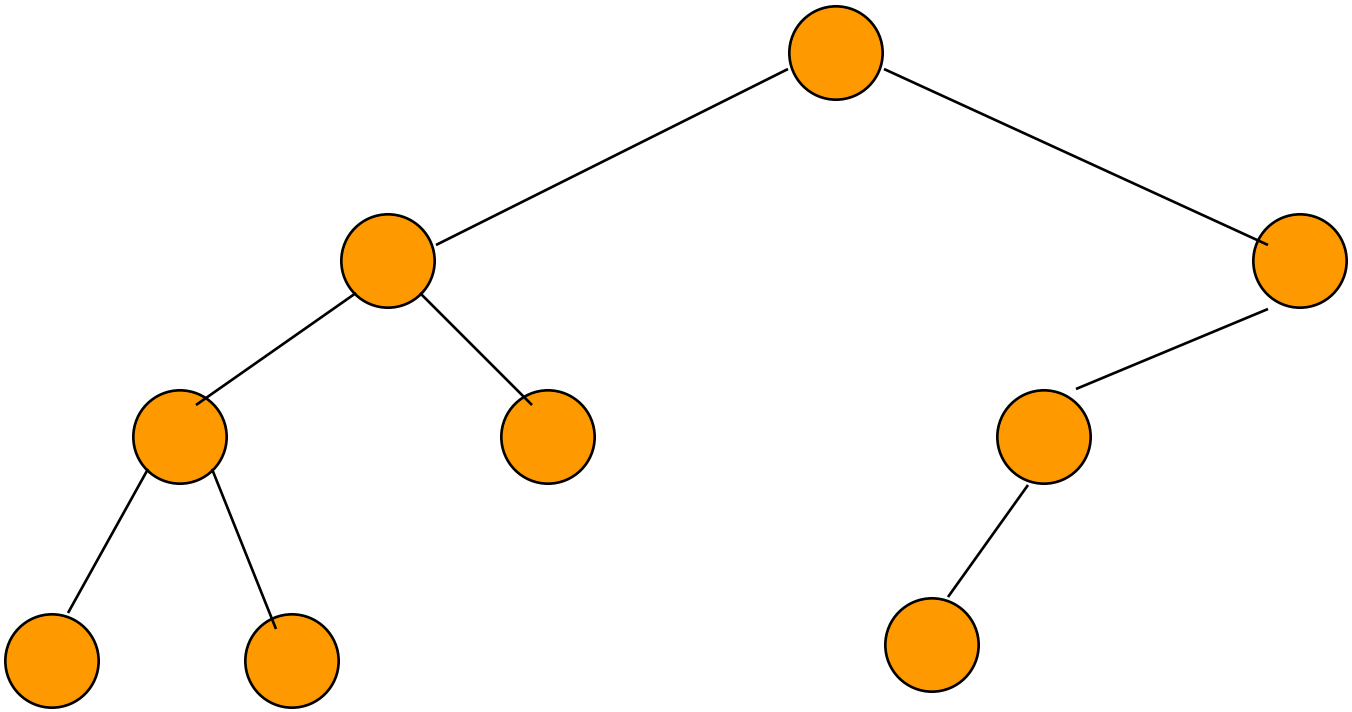
Can meld two leftist tree priority queues in O(log n) time.

Extended Binary Trees

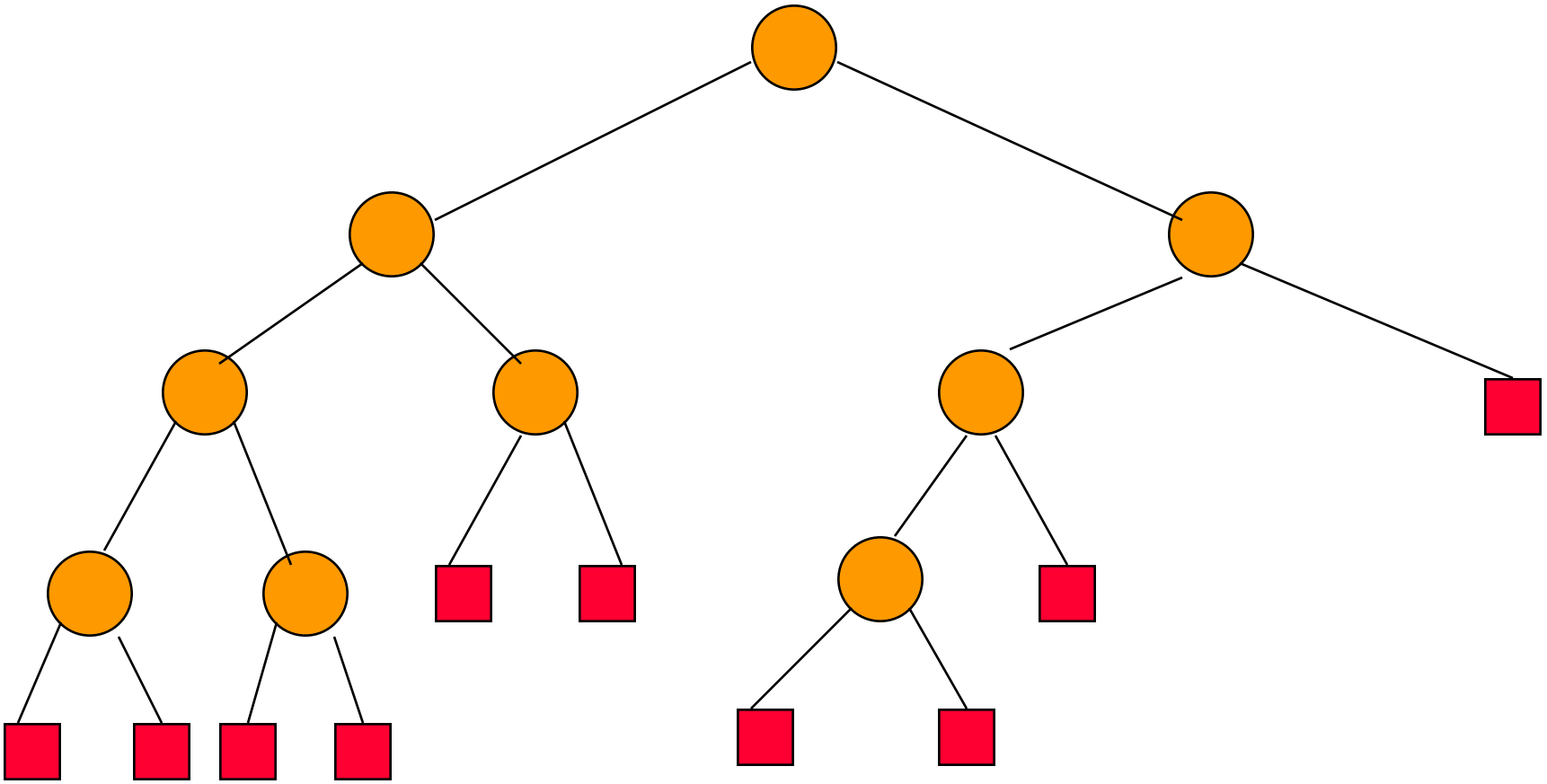
Start with any binary tree and add an external node wherever there is an empty subtree.

Result is an extended binary tree.

A Binary Tree



An Extended Binary Tree

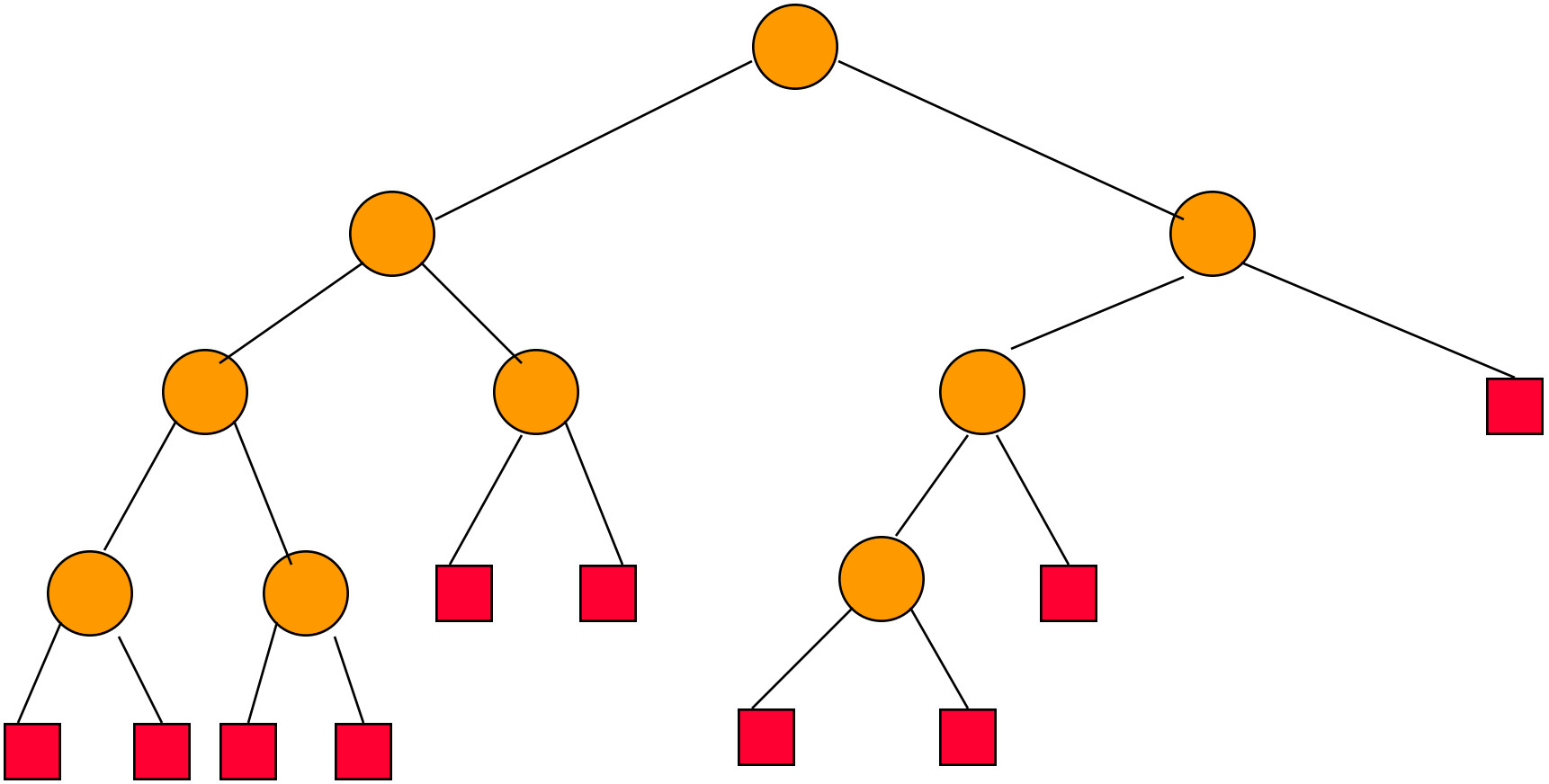


number of external nodes is n+1

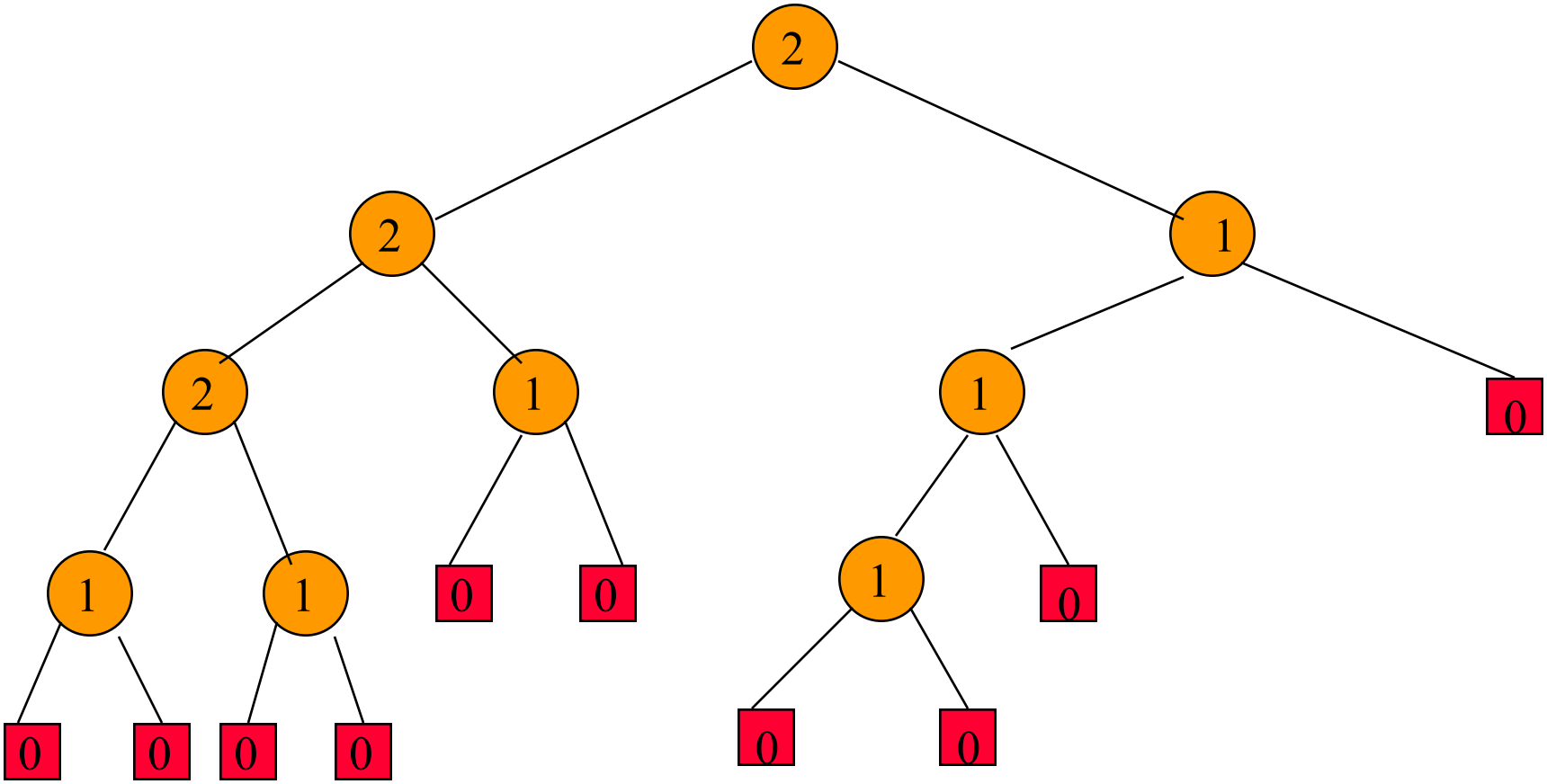
The Function s()

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,

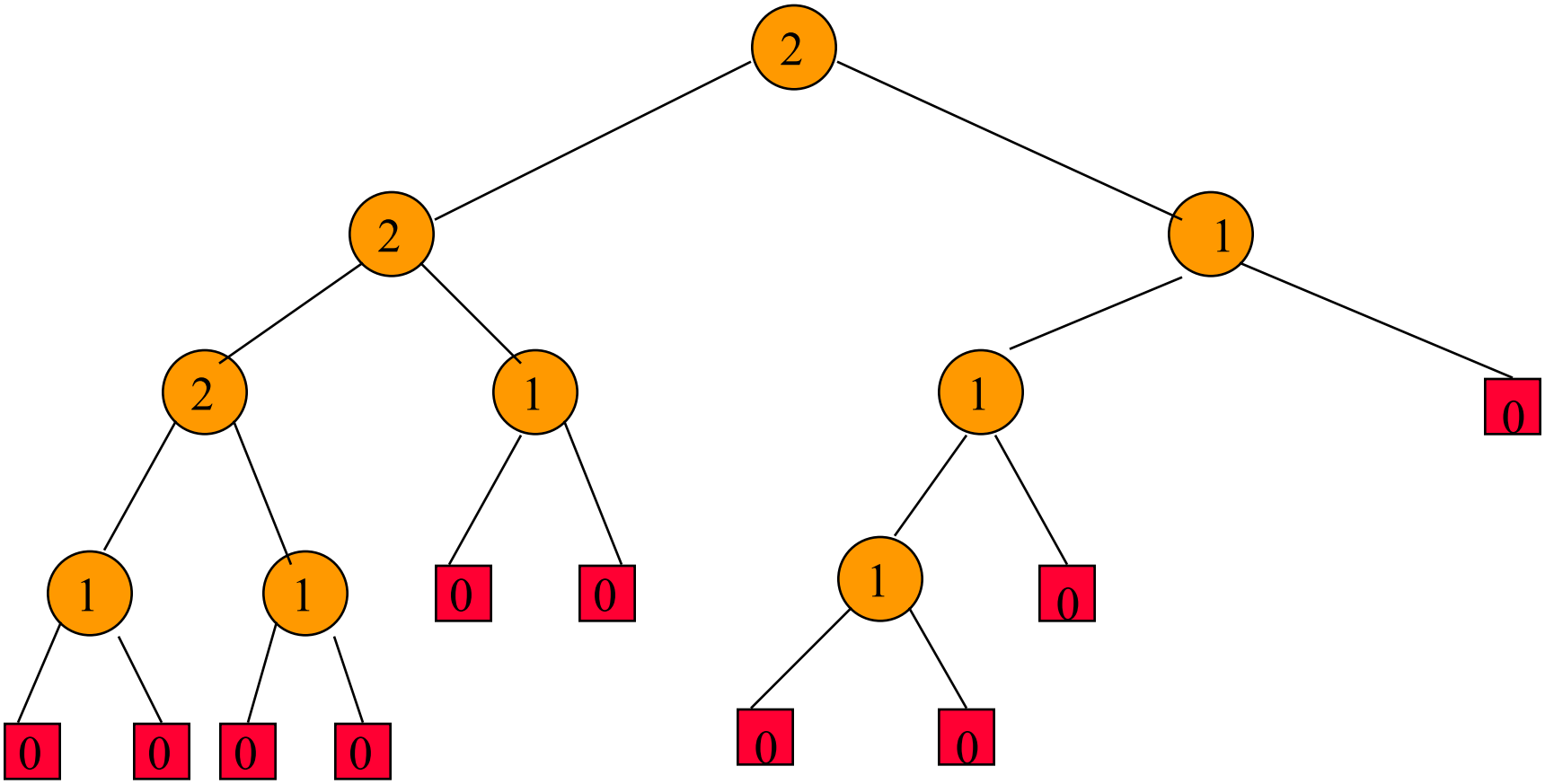
$$s(x) = \min \{s(\text{leftChild}(x)), s(\text{rightChild}(x))\} + 1$$

(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))

So, in a leftist tree, s(x) = s(rightChild) + 1.

A Leftist Tree



s() decreases by 1 each time you move to a right child

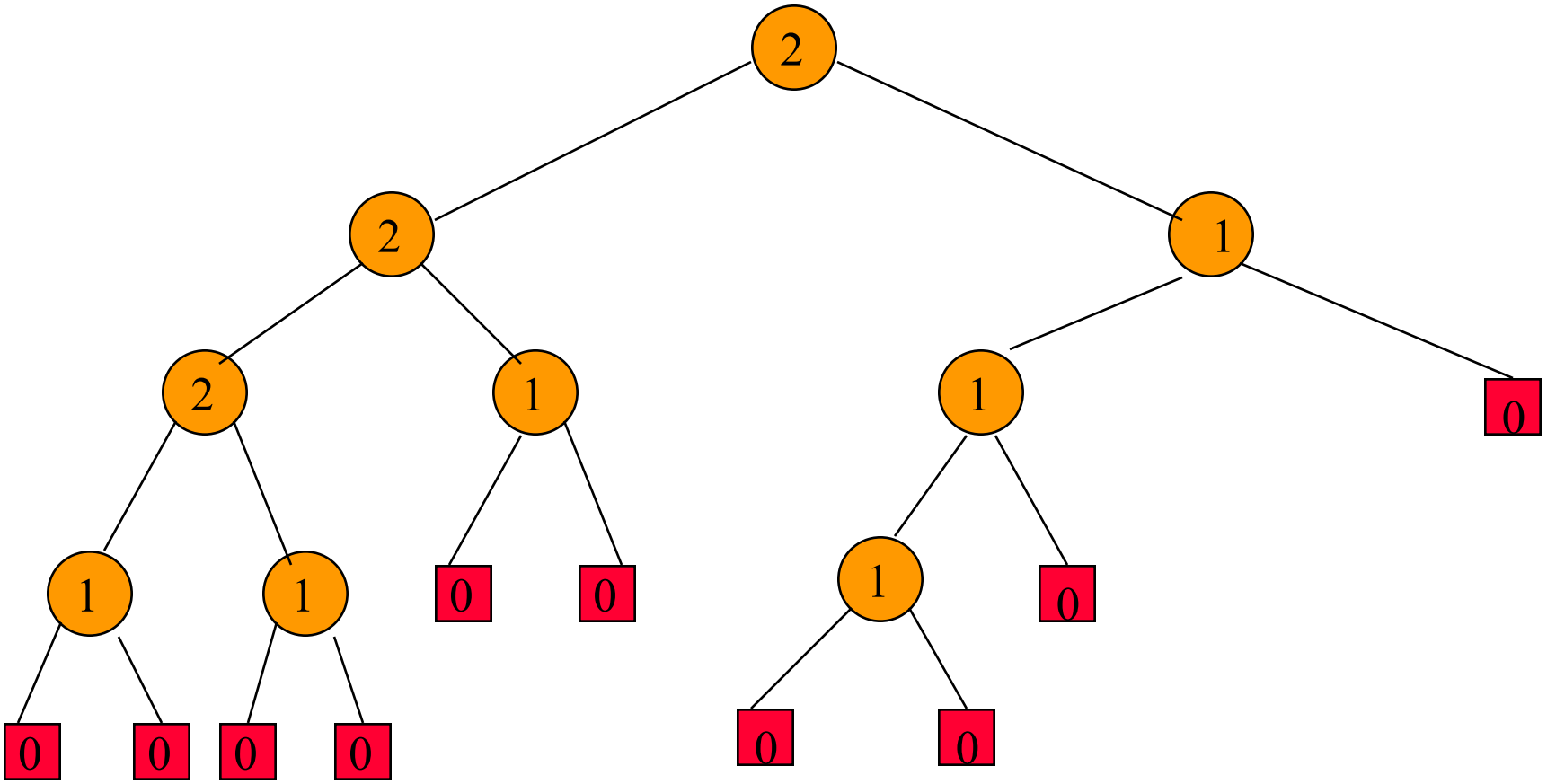
how about when you move to a left child?

subtrees are also leftist trees

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.

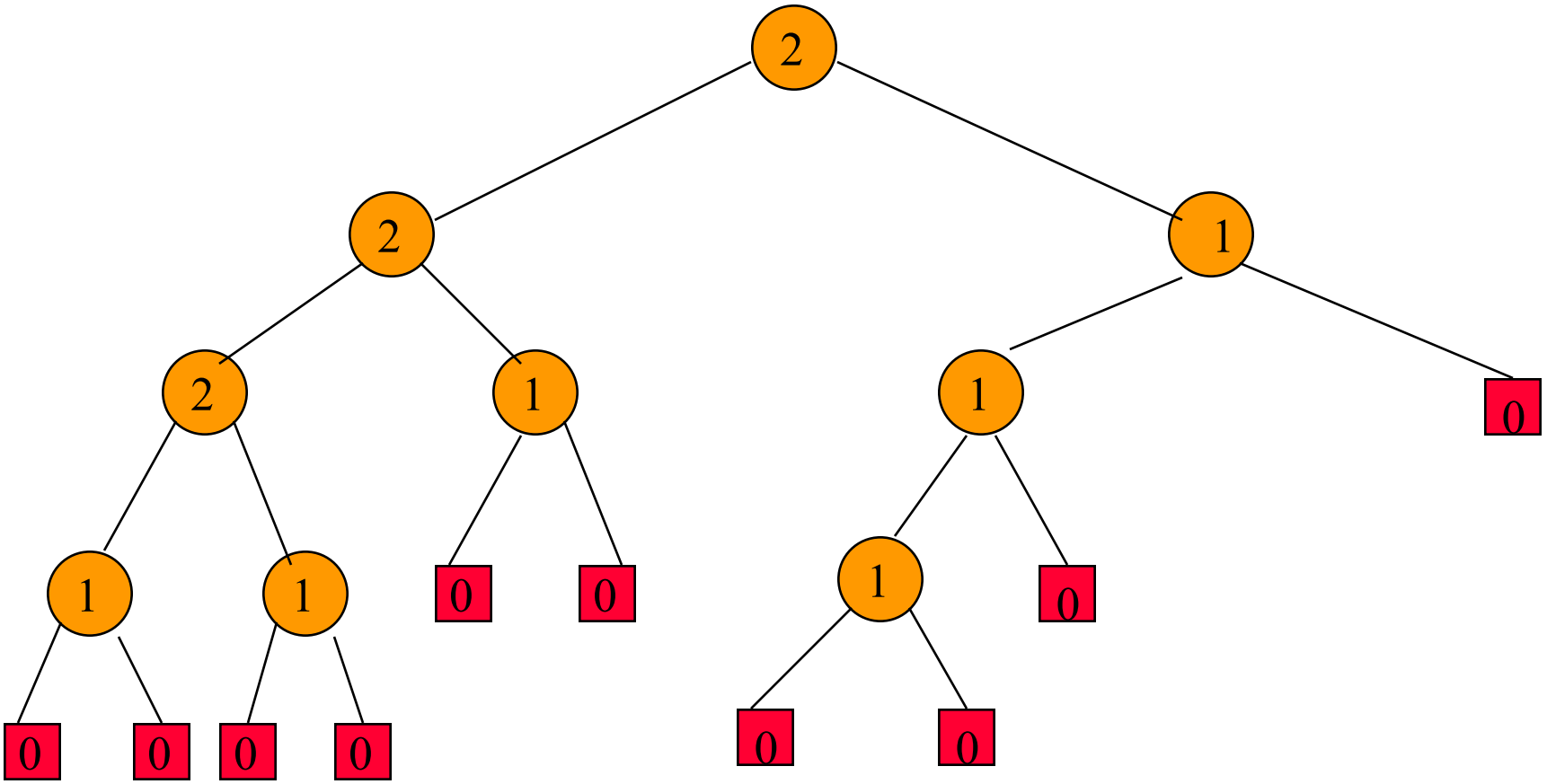
Leftist Trees—Property 2

The number of internal nodes is at least

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

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

A Leftist Tree



Levels 1 and 2 have no external nodes.

Max # of internal nodes?

Leftist Trees—Property 3

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

Property 2 =>

- $n >= 2^{s(root)} - 1 \Rightarrow s(root) <= \log_2(n+1)$

Property 1 => length of rightmost path is s(root).

Leftist Trees As Priority Queues

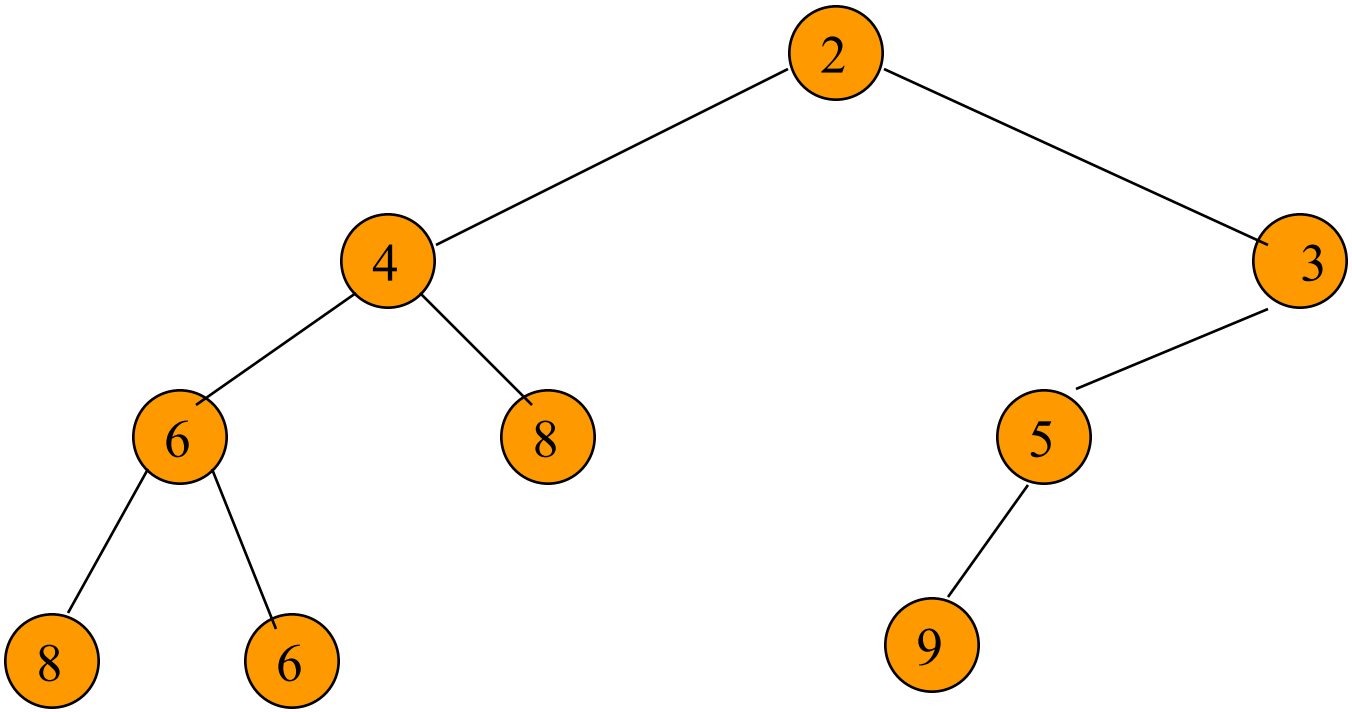
Min leftist tree … leftist tree that is a min tree.

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

findMin()

put()

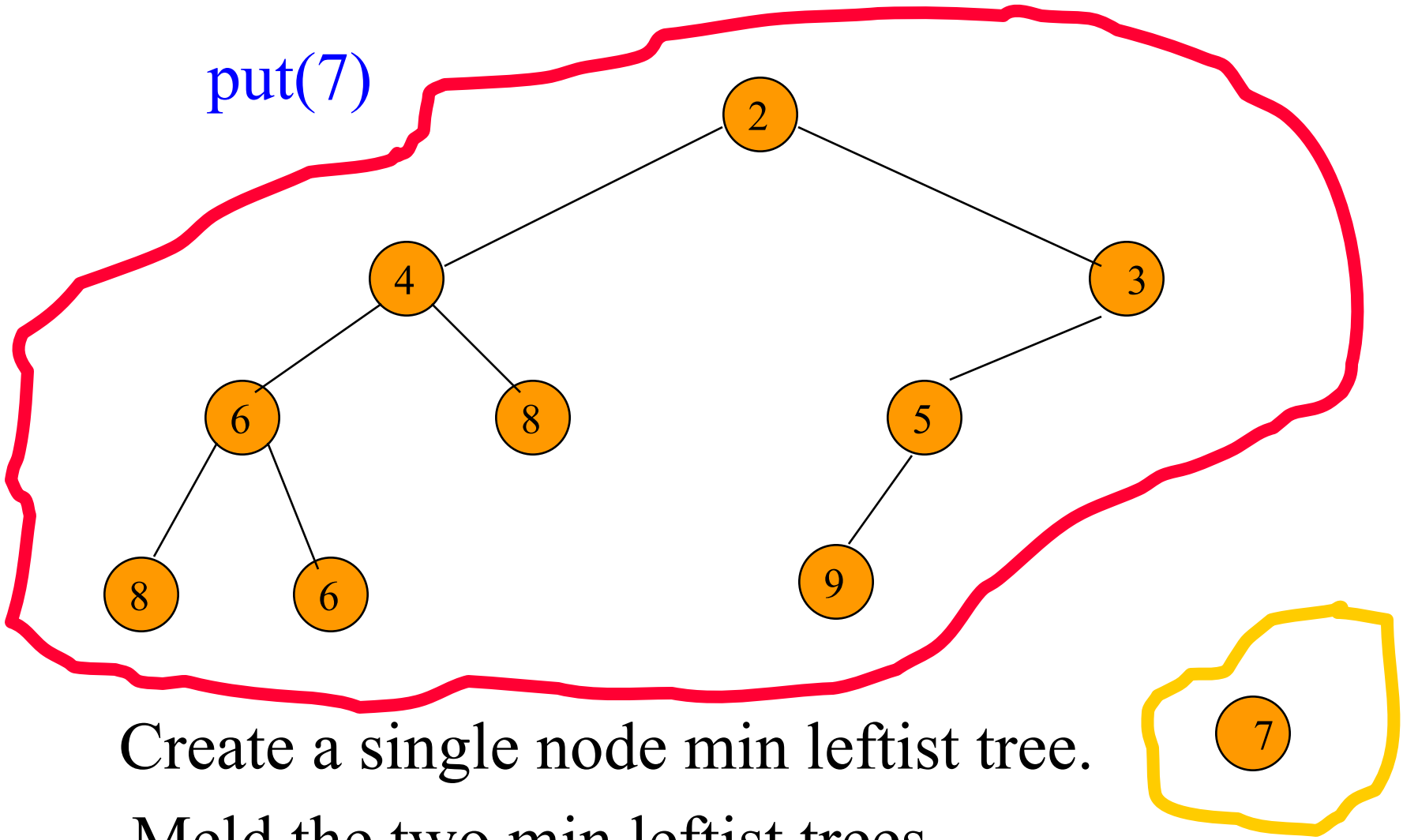
removeMin()

meld()

initialize()

put() and removeMin() use meld().

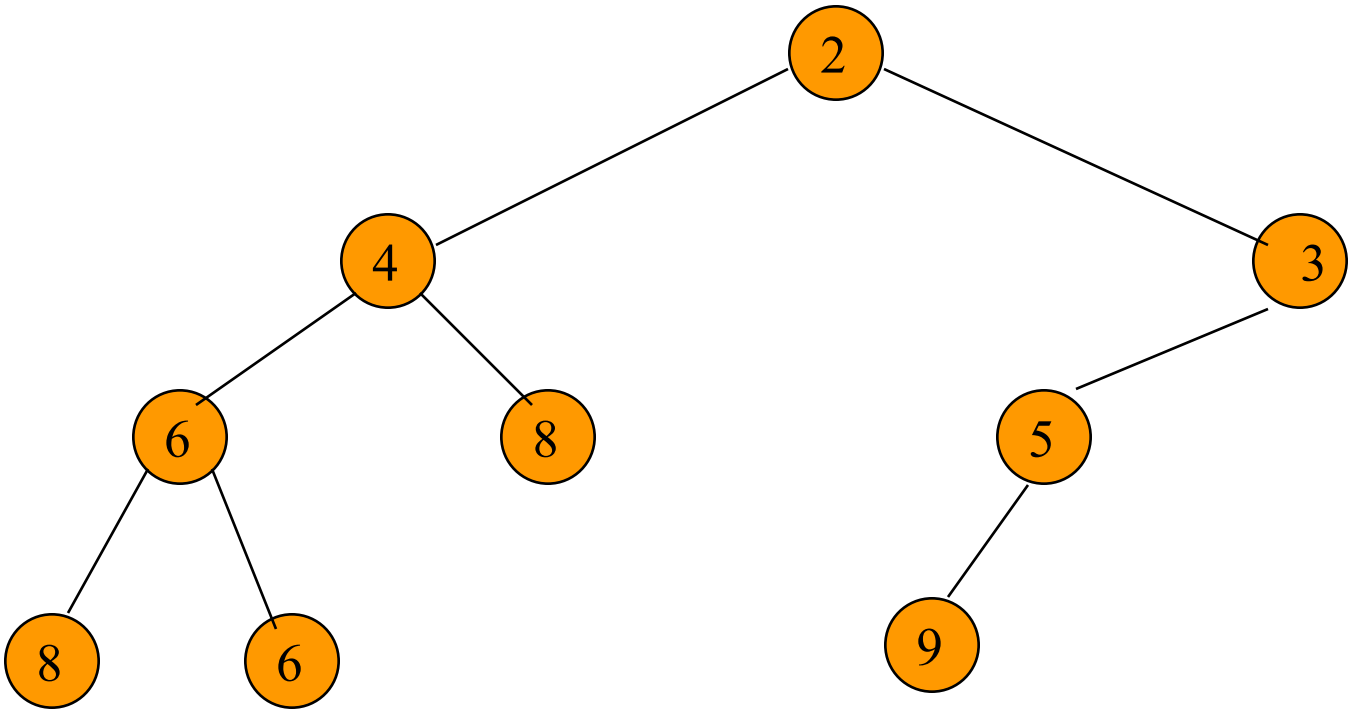
Put Operation



Create a single node min leftist tree.

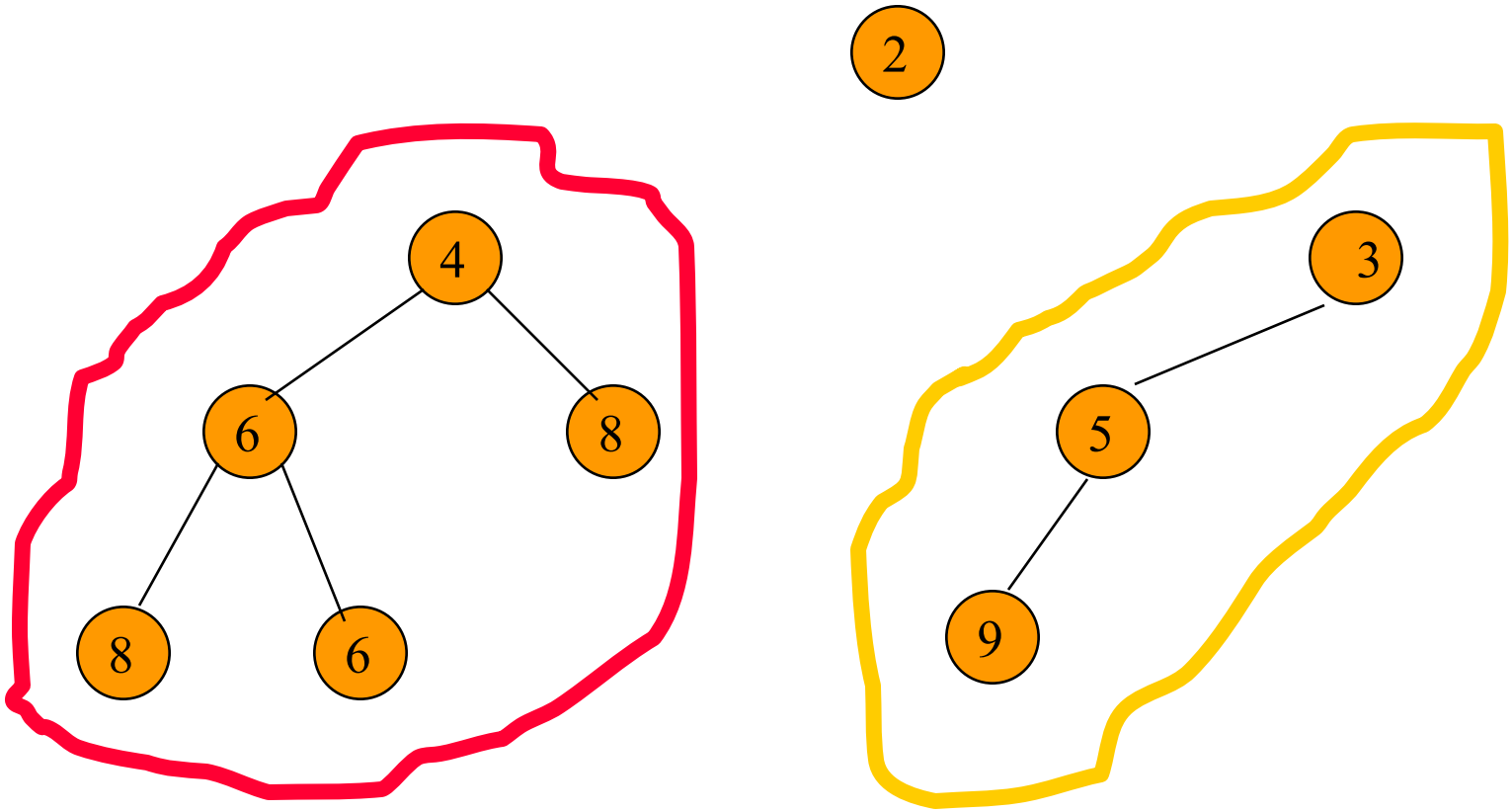
Meld the two min leftist trees.

Remove Min



Remove the root.

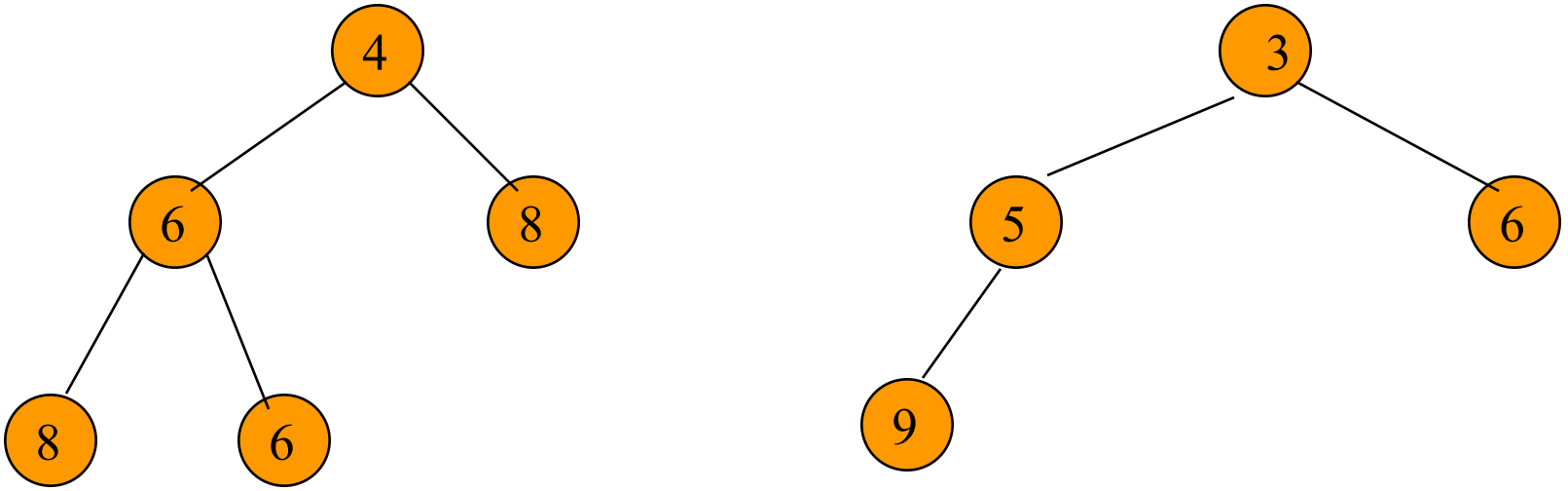
Remove Min



Remove the root.

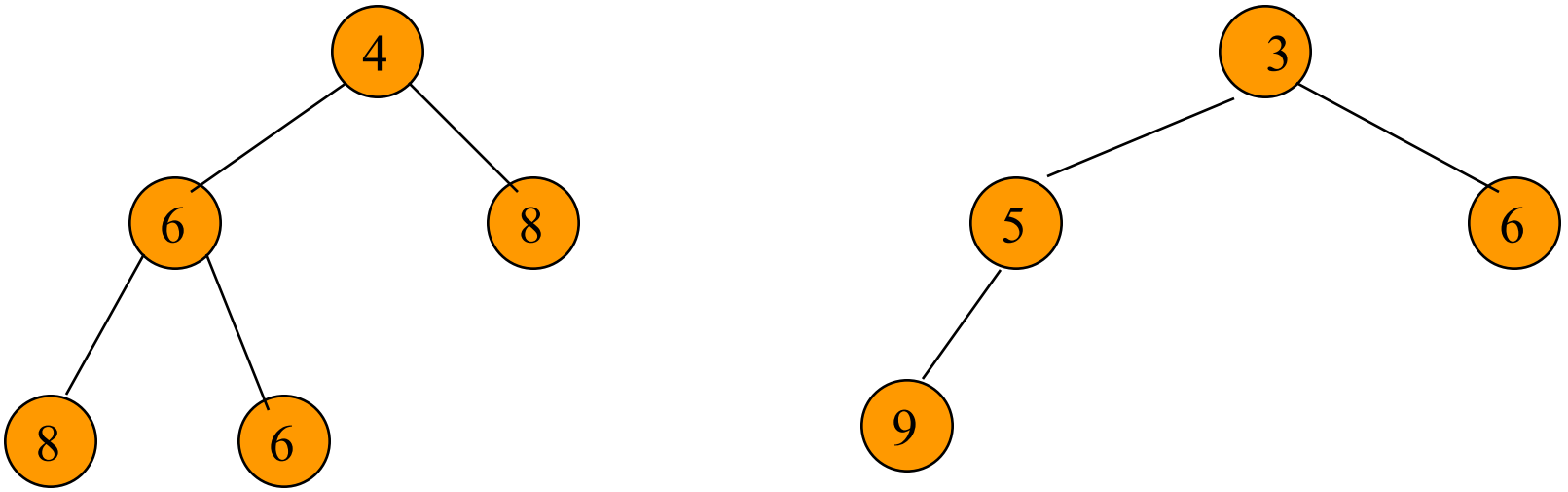
Meld the two subtrees.

Meld Two Min Leftist Trees



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

Meld Two Min Leftist Trees

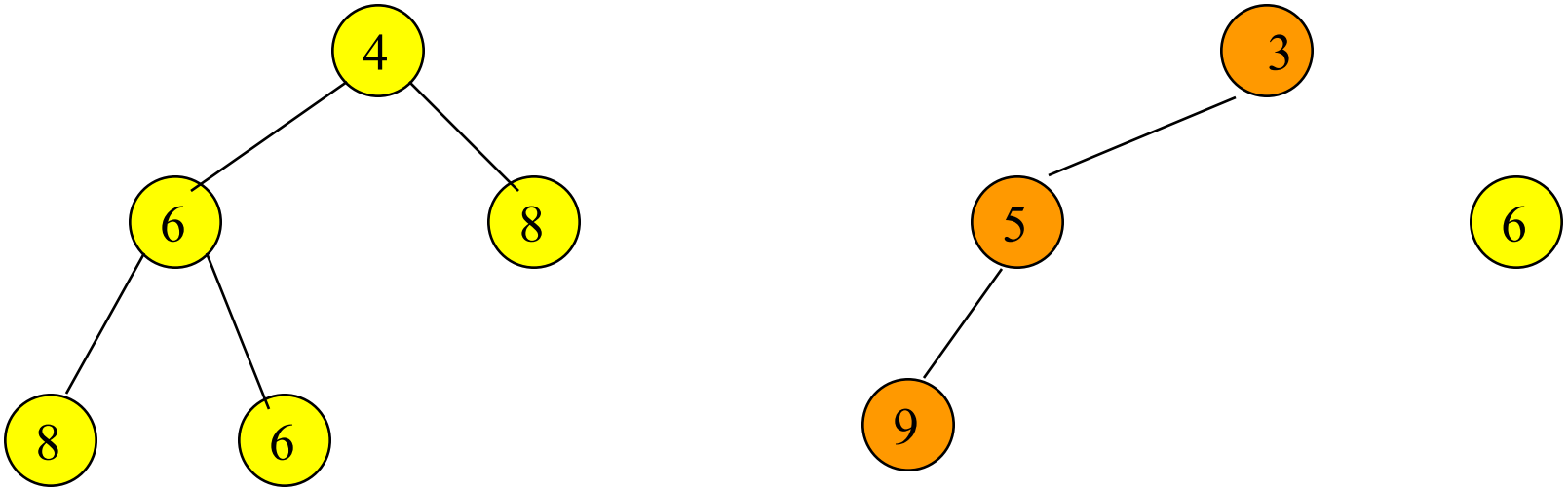


Meld right subtree of tree with smaller root and all of other tree.

Make the result the new right subtree of the smaller root.

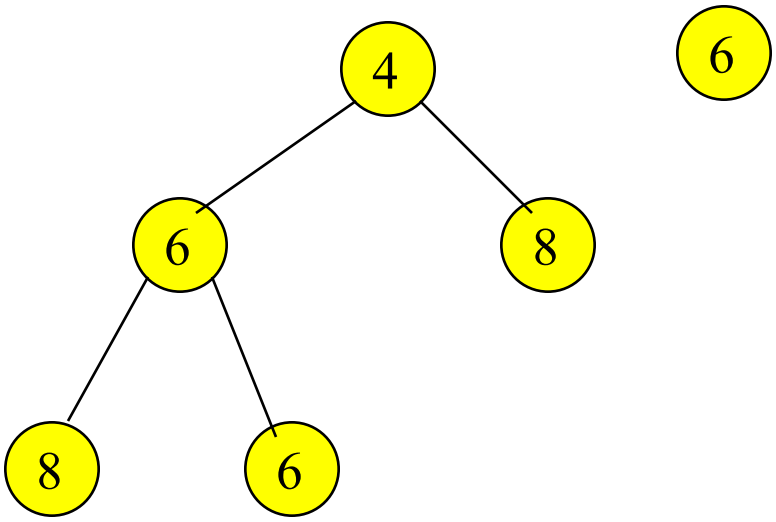
Swap left and right subtrees if needed.

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.

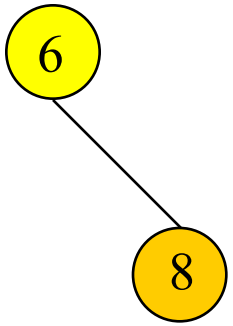
Meld Two Min Leftist Trees

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

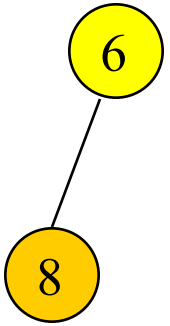
Right subtree of 6 is empty. So, result of melding right subtree of tree with smaller root and other tree is the other tree.

Meld Two Min Leftist Trees

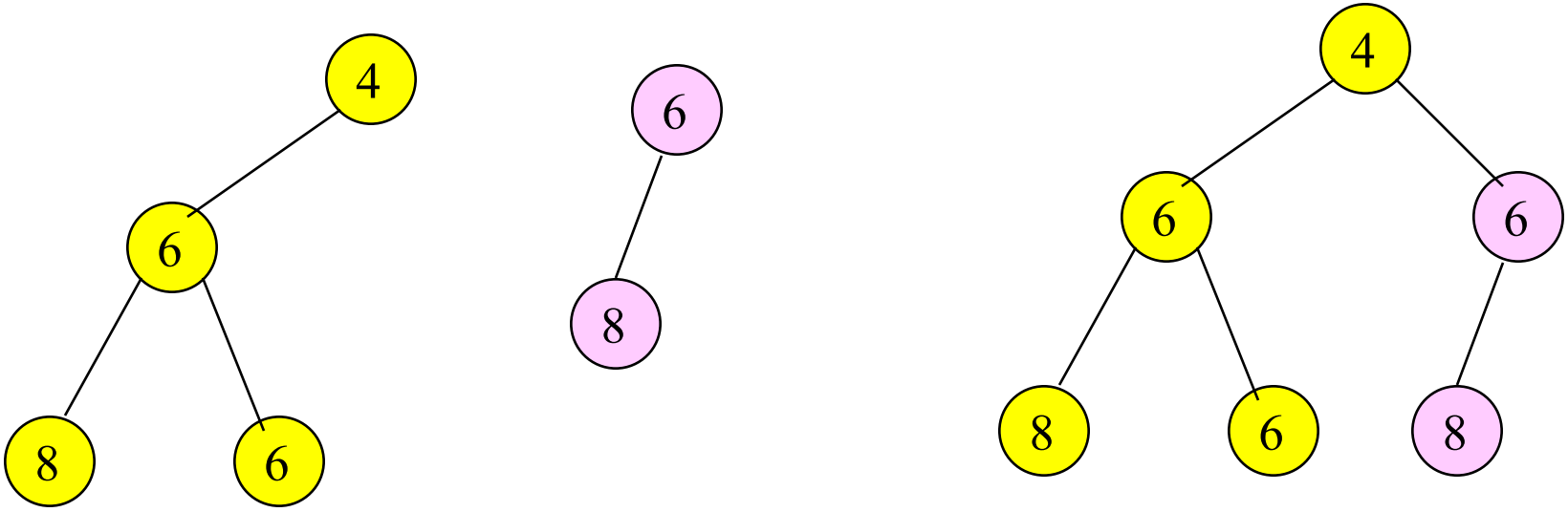
Make melded subtree right subtree of smaller root.



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



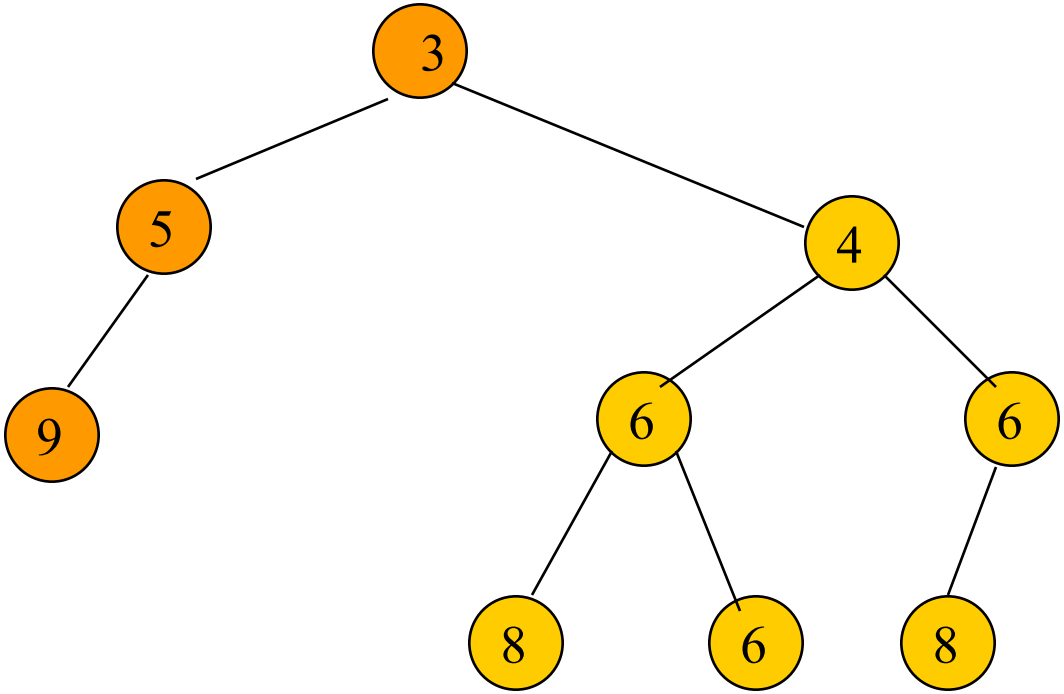
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

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

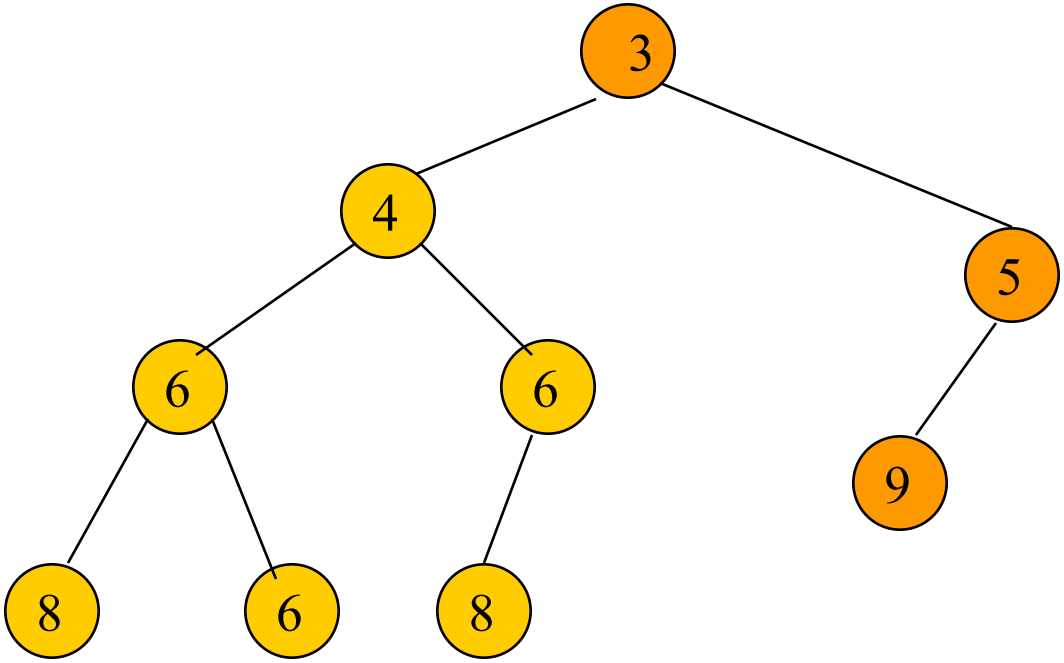
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

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

Meld Two Min Leftist Trees

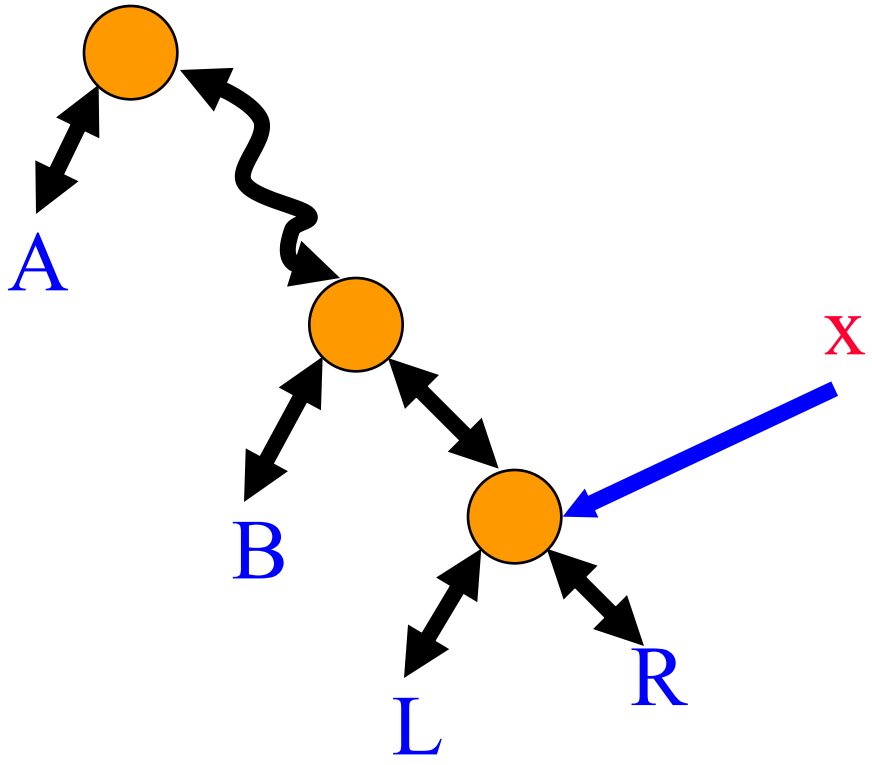


Initializing In O(n) Time

- 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.

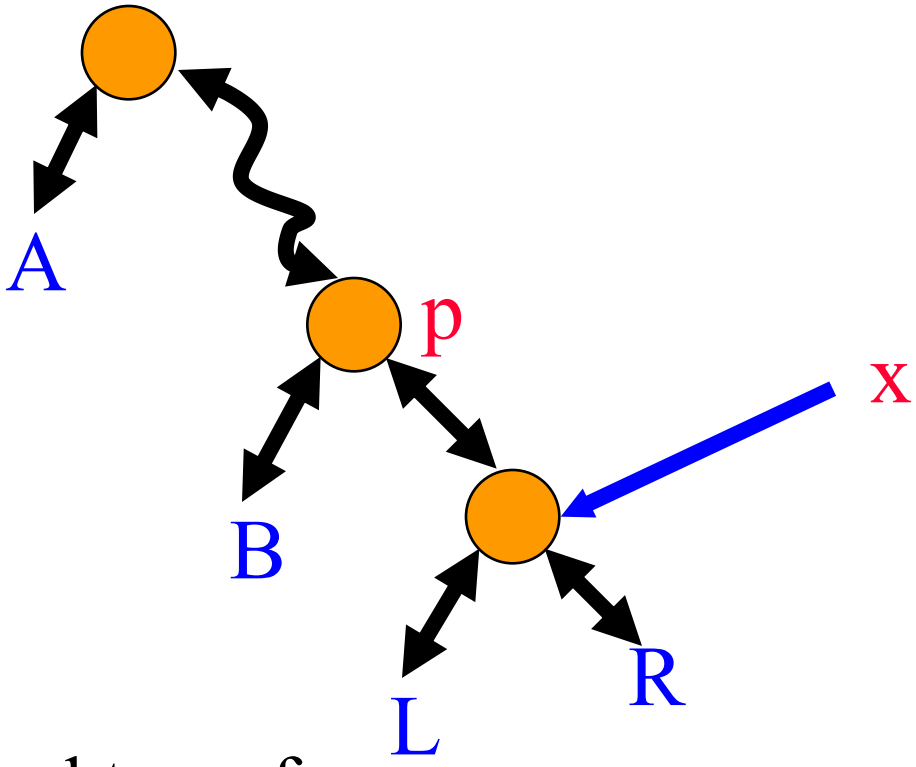
Arbitrary Remove

Remove element in node pointed at by x.



x = root => remove min.

Arbitrary Remove, x != root



Make L right subtree of p.

Adjust s and leftist property on path from p to root (stop at first node, if any, whose s value does not change).

Meld with R.

Skew Heap

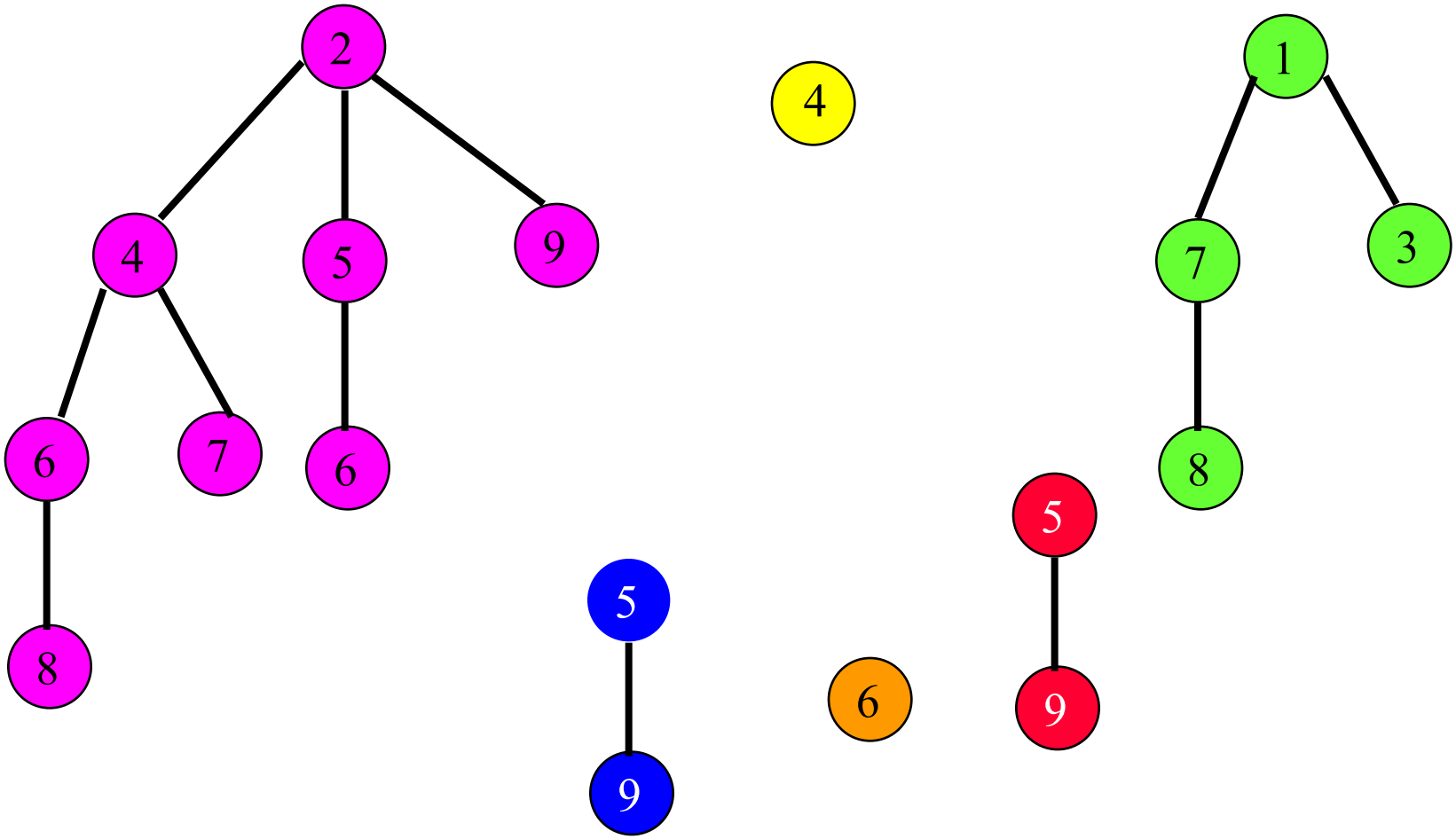
- Similar to leftist tree
- No s() values stored
- Swap left and right subtrees of all nodes on rightmost path rather than just when s(l(x)) < s(r(x))
- Amortized complexity of insert, remove min, and meld is O(log n)

Binomial Heaps

	Leftist trees	Binomial heaps	
		Actual	Amortized
Find min (or max)	O(1)	O(1)	O(1)
Insert	O(log n)	O(1)	O(1)
Remove 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.

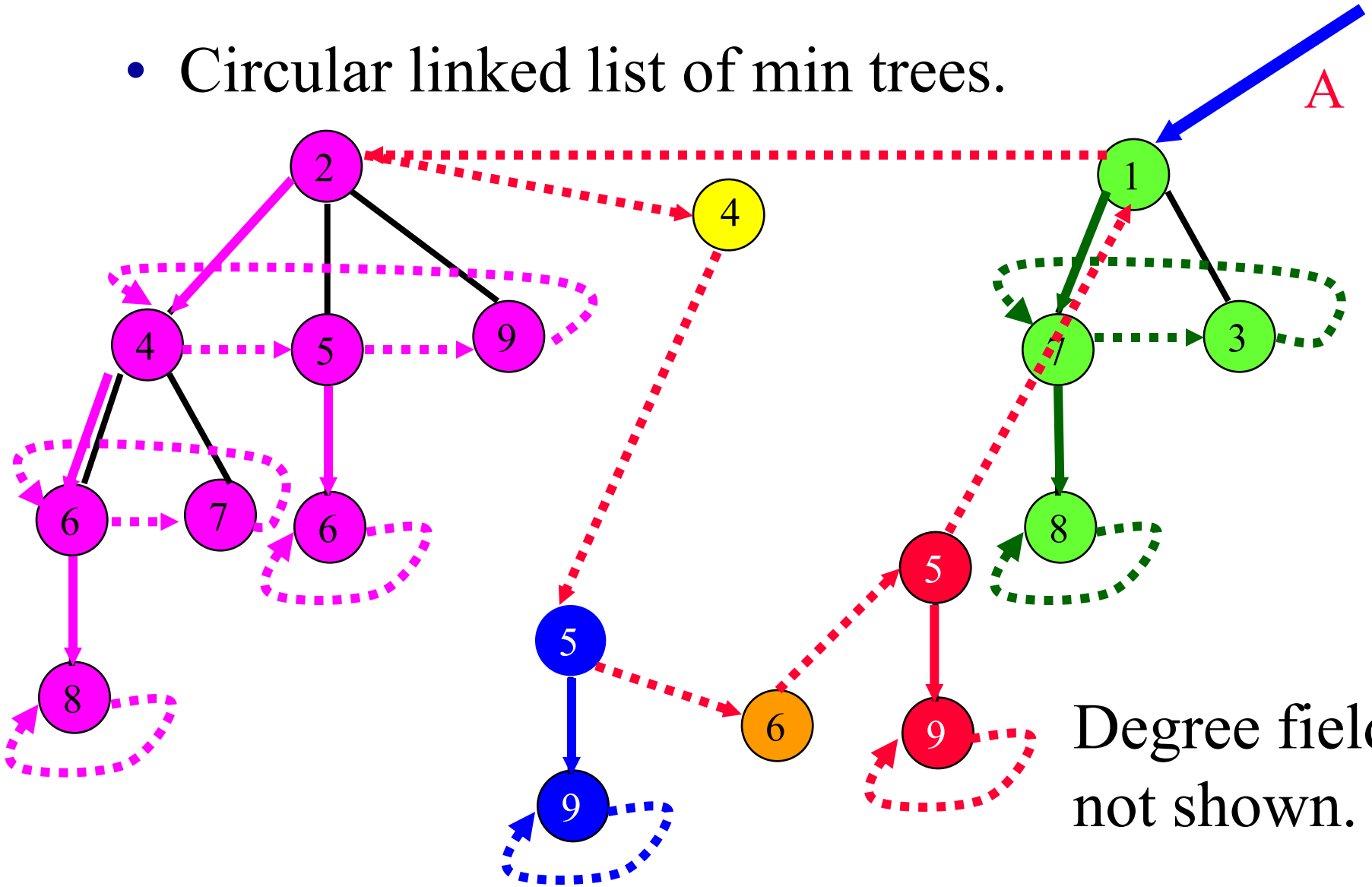


Node Structure

- 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

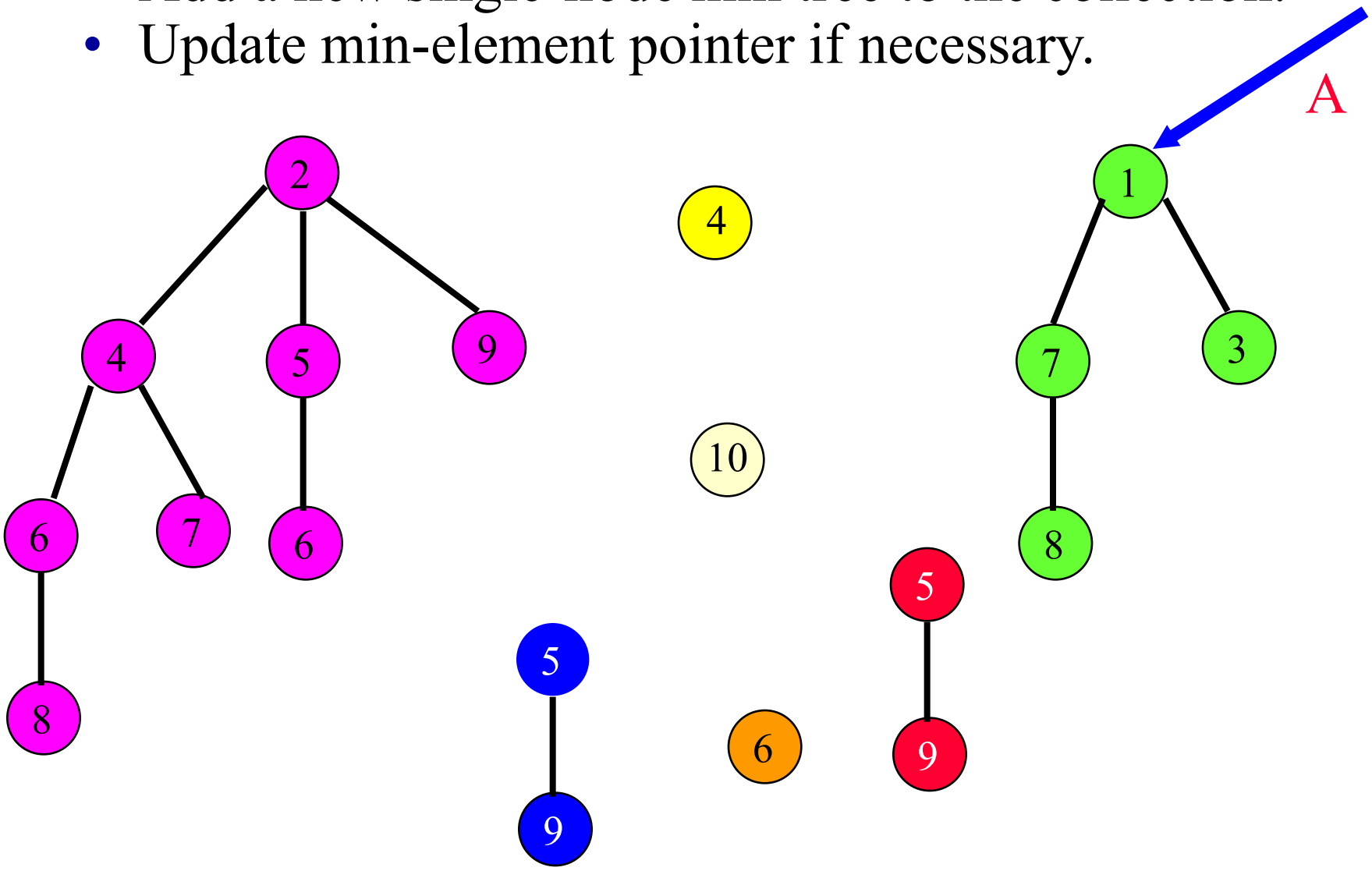
- Circular linked list of min trees.



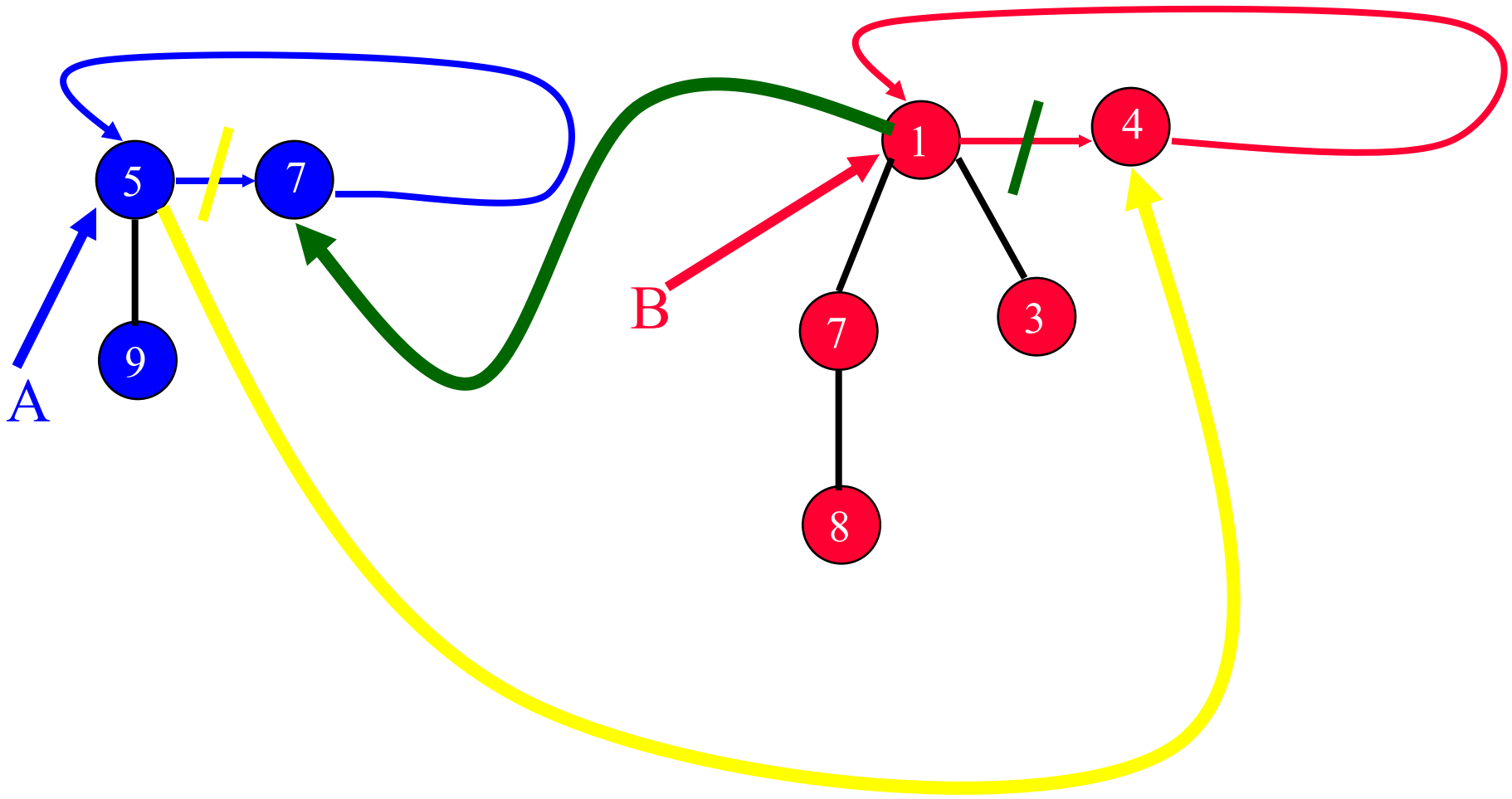
Degree fields not shown.

Insert 10

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

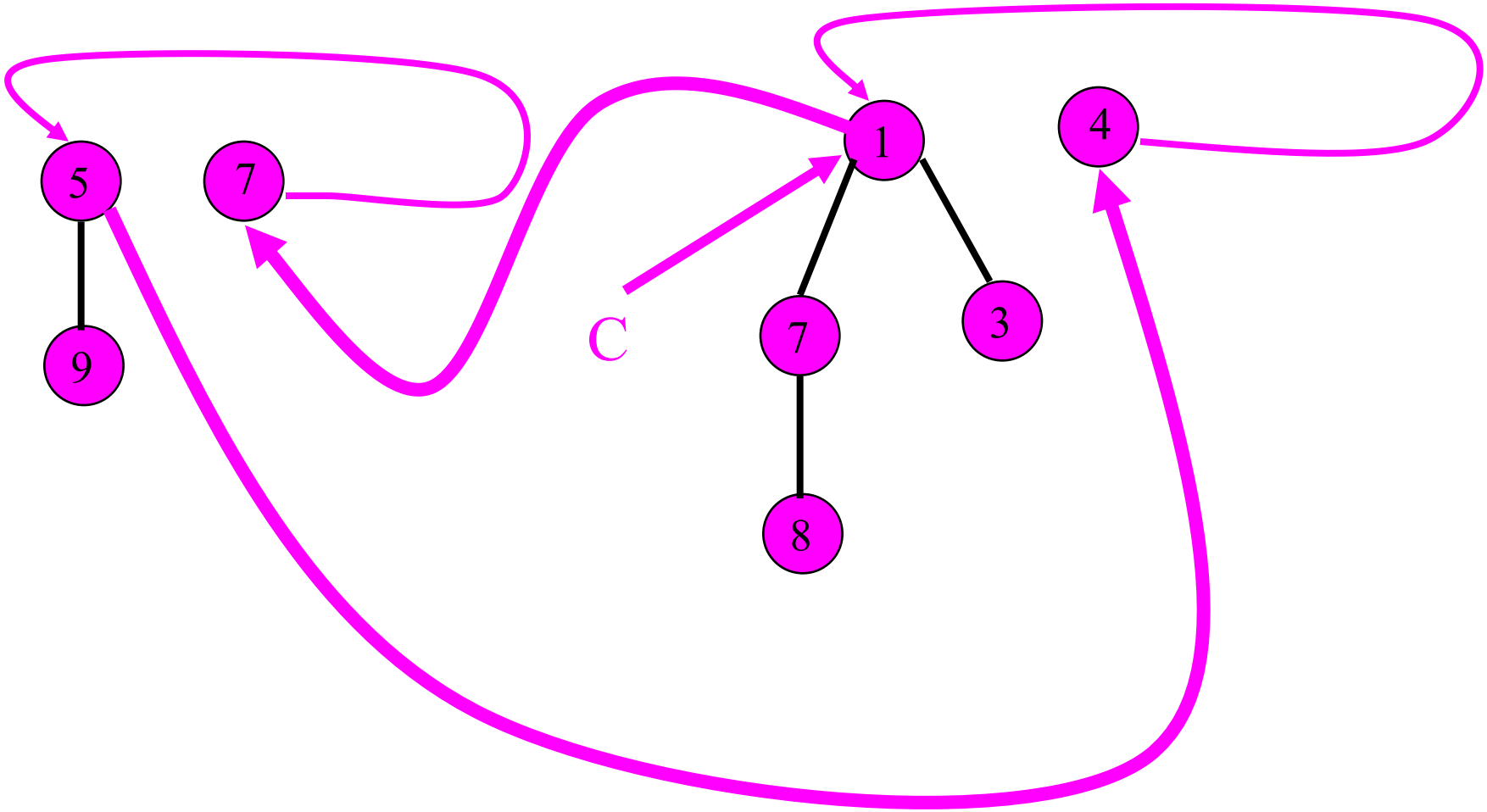


Meld



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

Meld

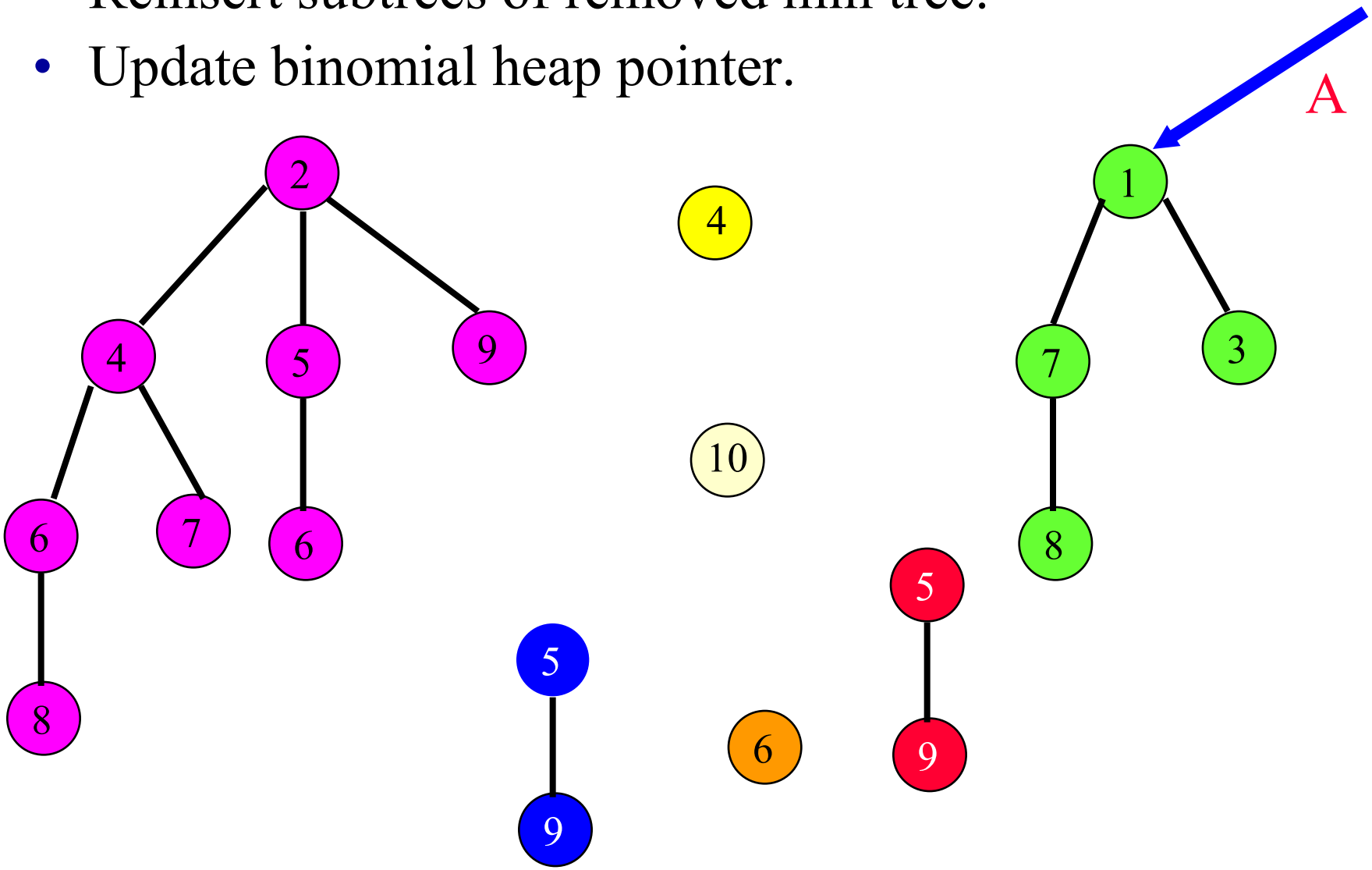


Remove Min

- Empty binomial heap => fail.

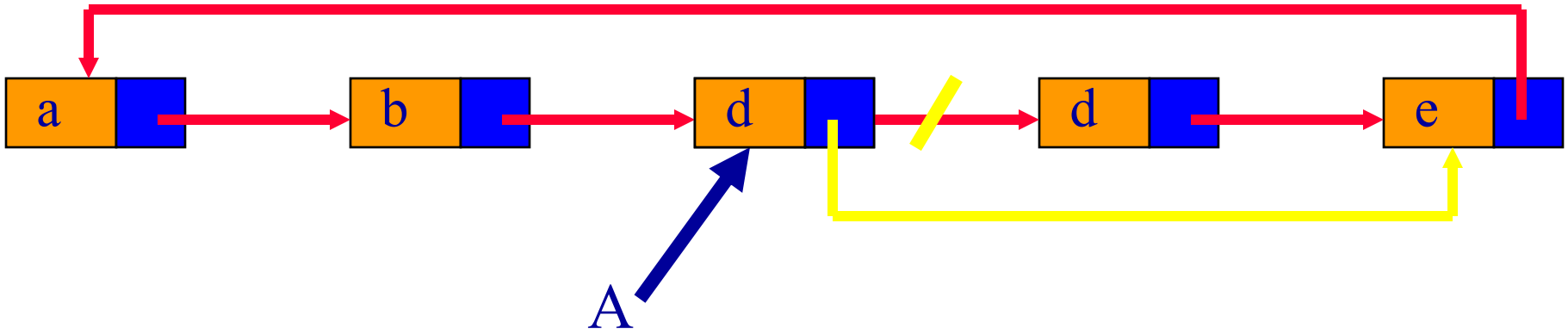
Nonempty Binomial Heap

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



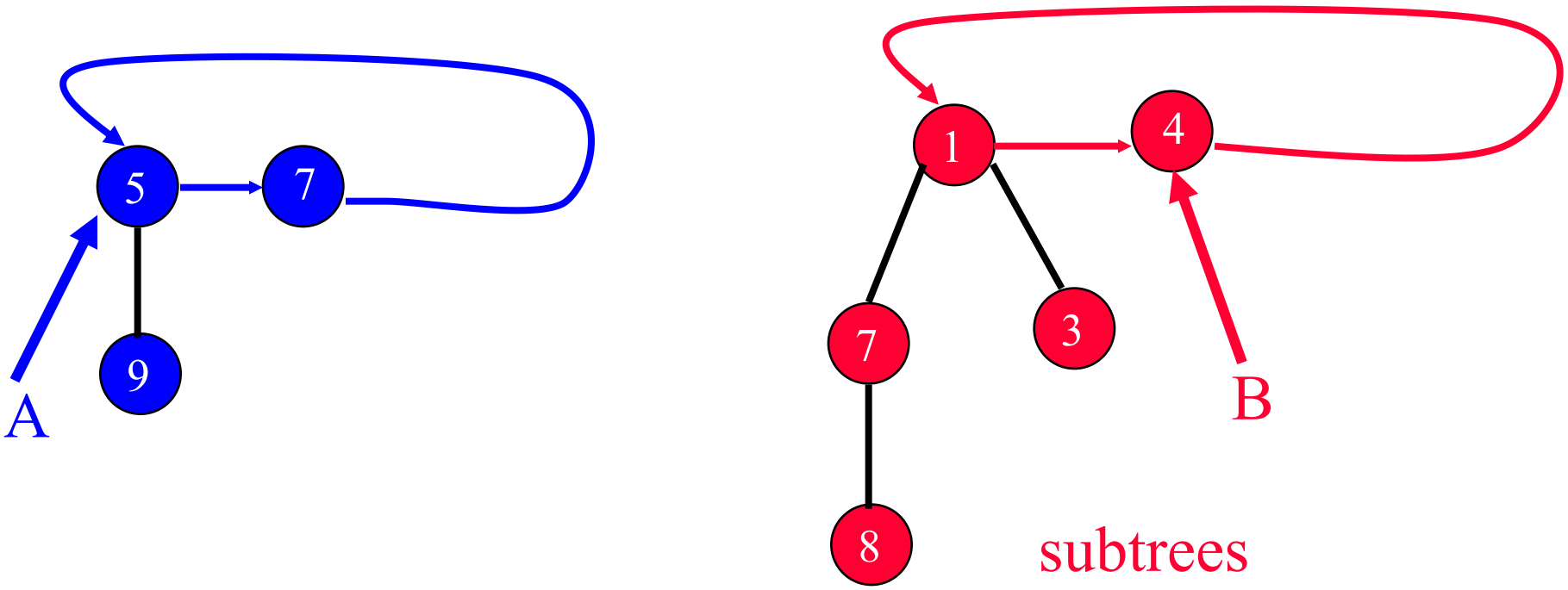
Remove Min Tree

- Same as remove an element 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.

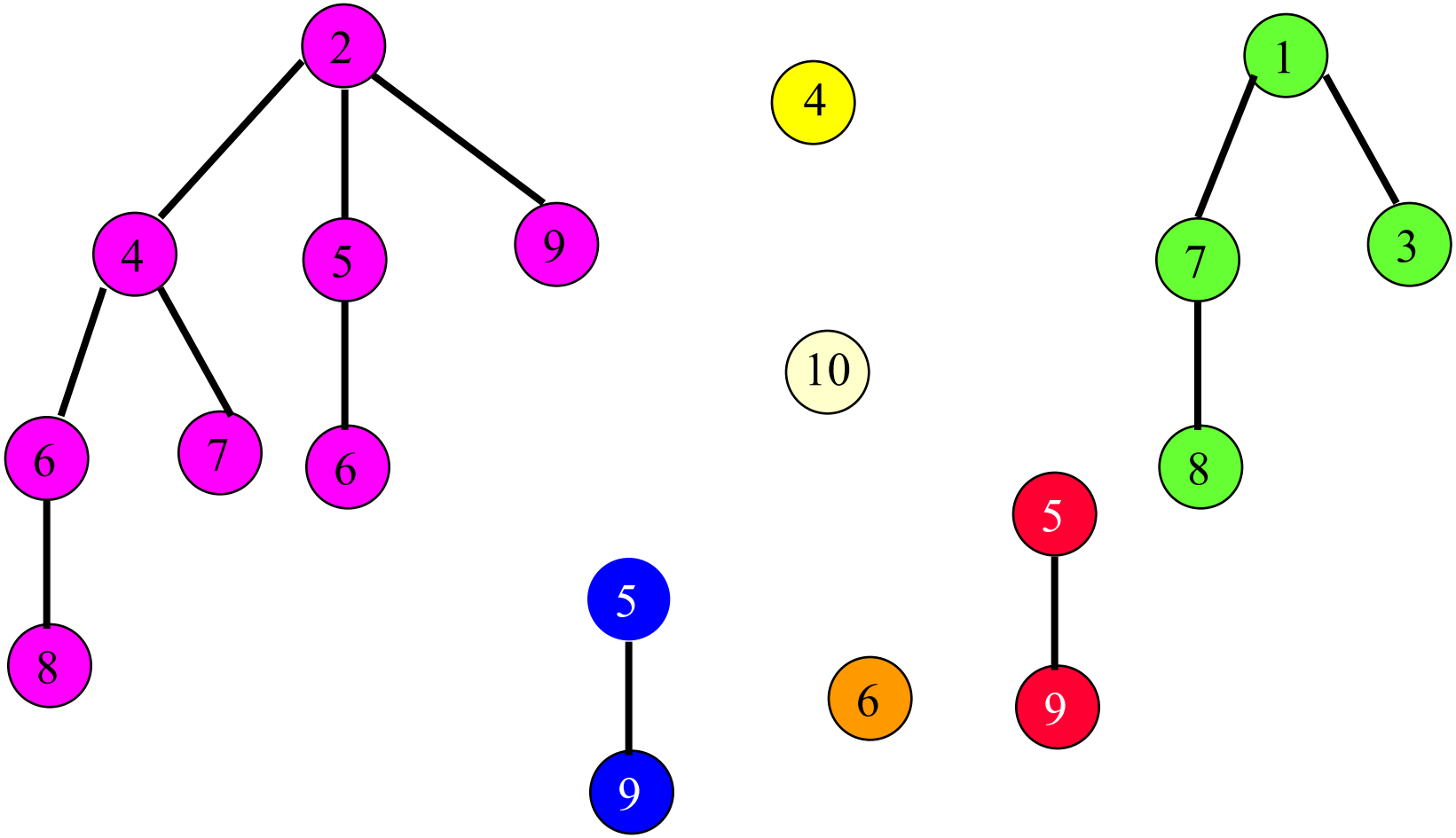
Complexity Of Remove Min

- 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).

Correct Remove Min

- During reinsert of subtrees, pairwise combine min trees whose roots have equal degree.
- This is essential to get stated amortized bounds and so is the correct way to remove from a Binomial heap.
- The simple remove min described earlier does not result in the desired amortized complexity and so is incorrect for Binomial heaps.

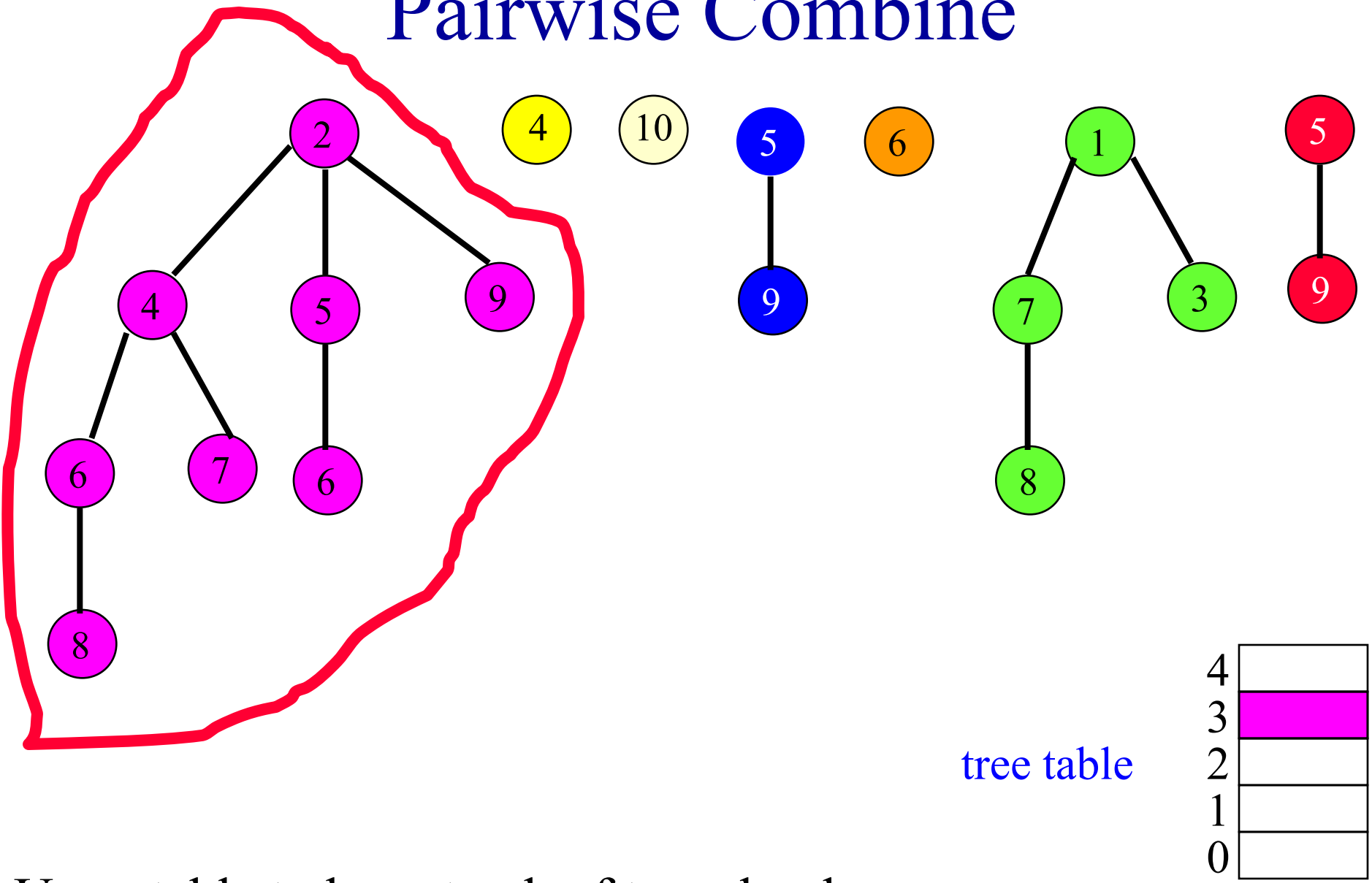
Pairwise Combine



Examine the s = 7 trees in some order.

Determined by the 2 top-level circular lists.

Pairwise Combine

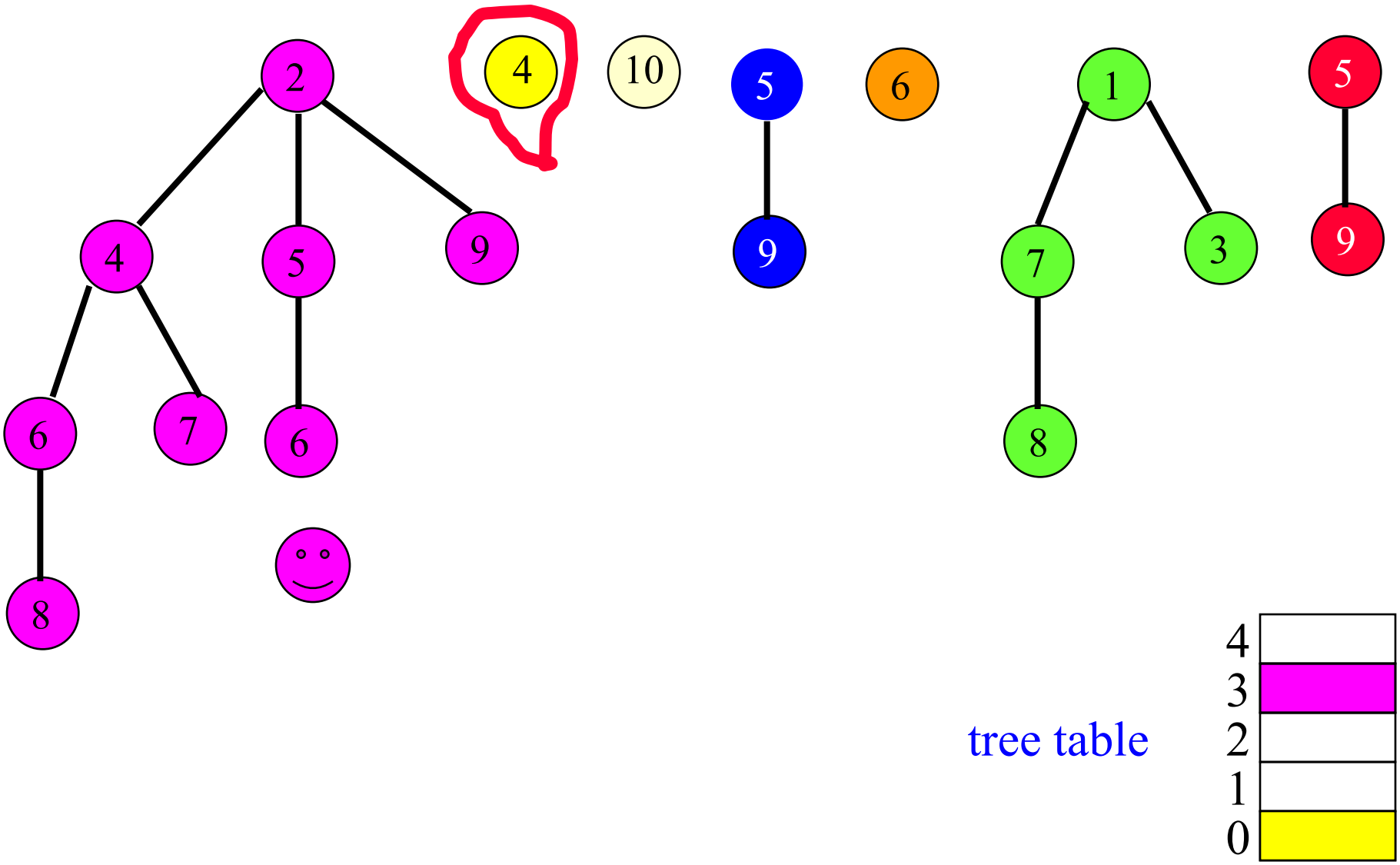


tree table

4	
3	
2	
1	
0	

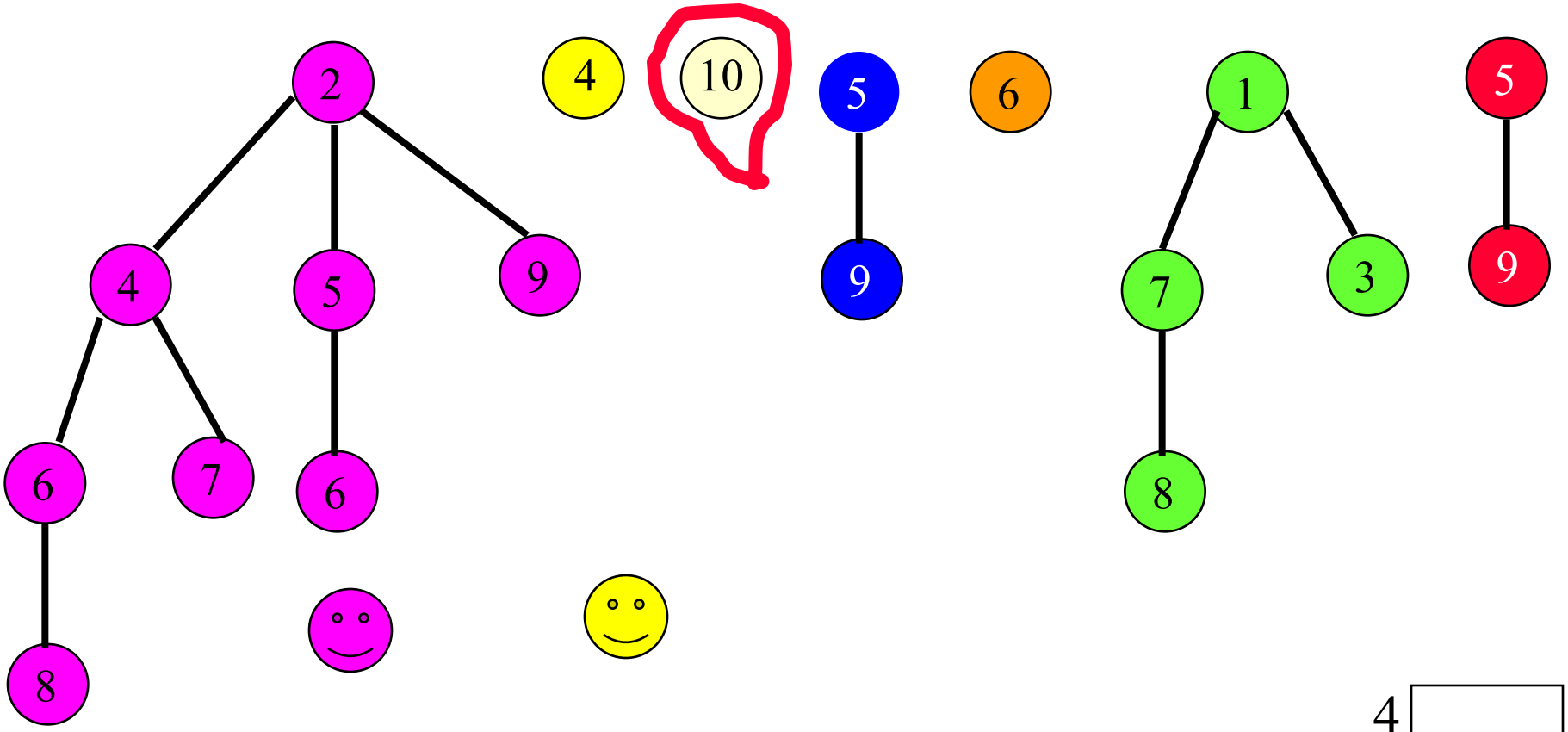
Use a table to keep track of trees by degree.

Pairwise Combine



tree table

Pairwise Combine



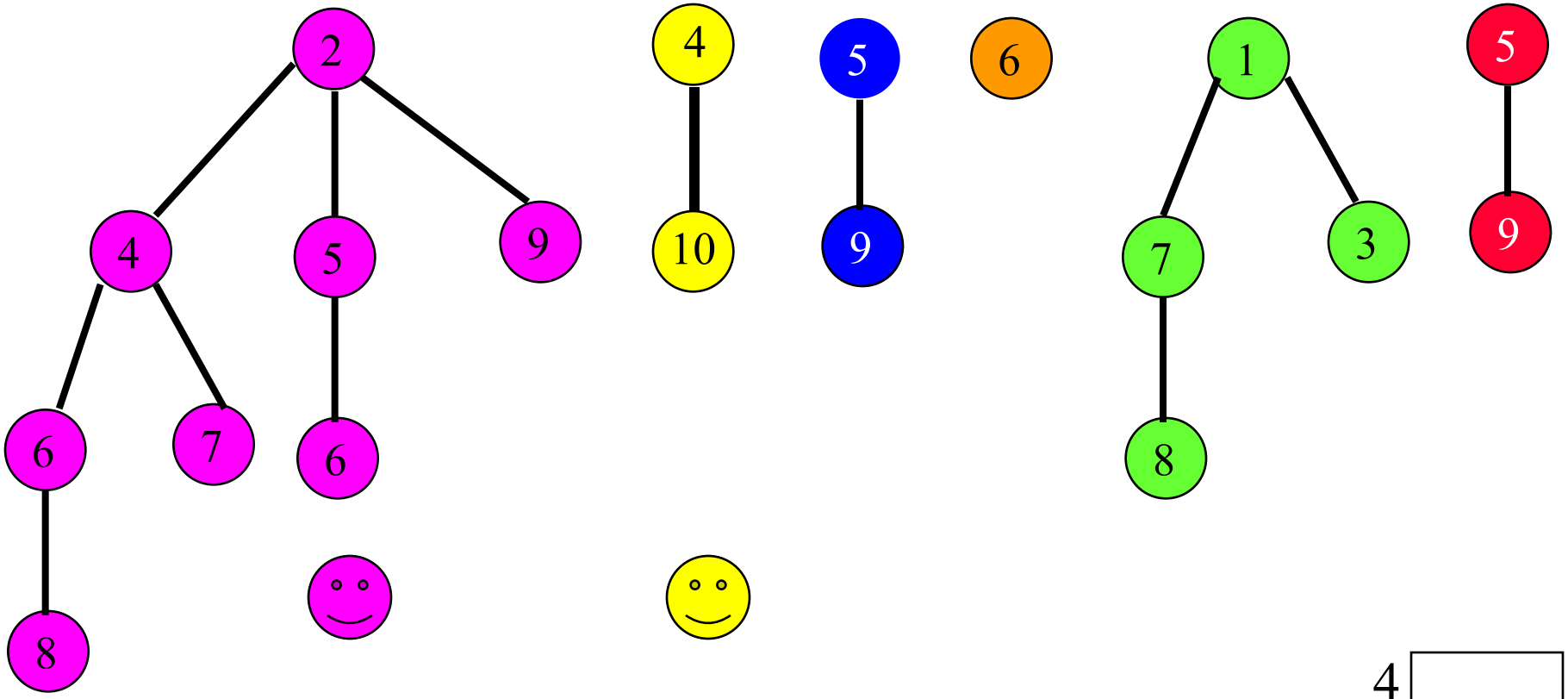
tree table

4	
3	
2	
1	
0	

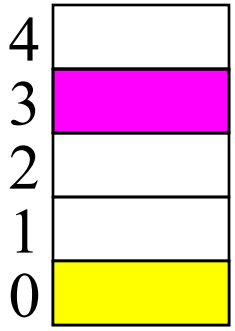
Combine 2 min trees of degree 0.

Make the one with larger root a subtree of other.

Pairwise Combine

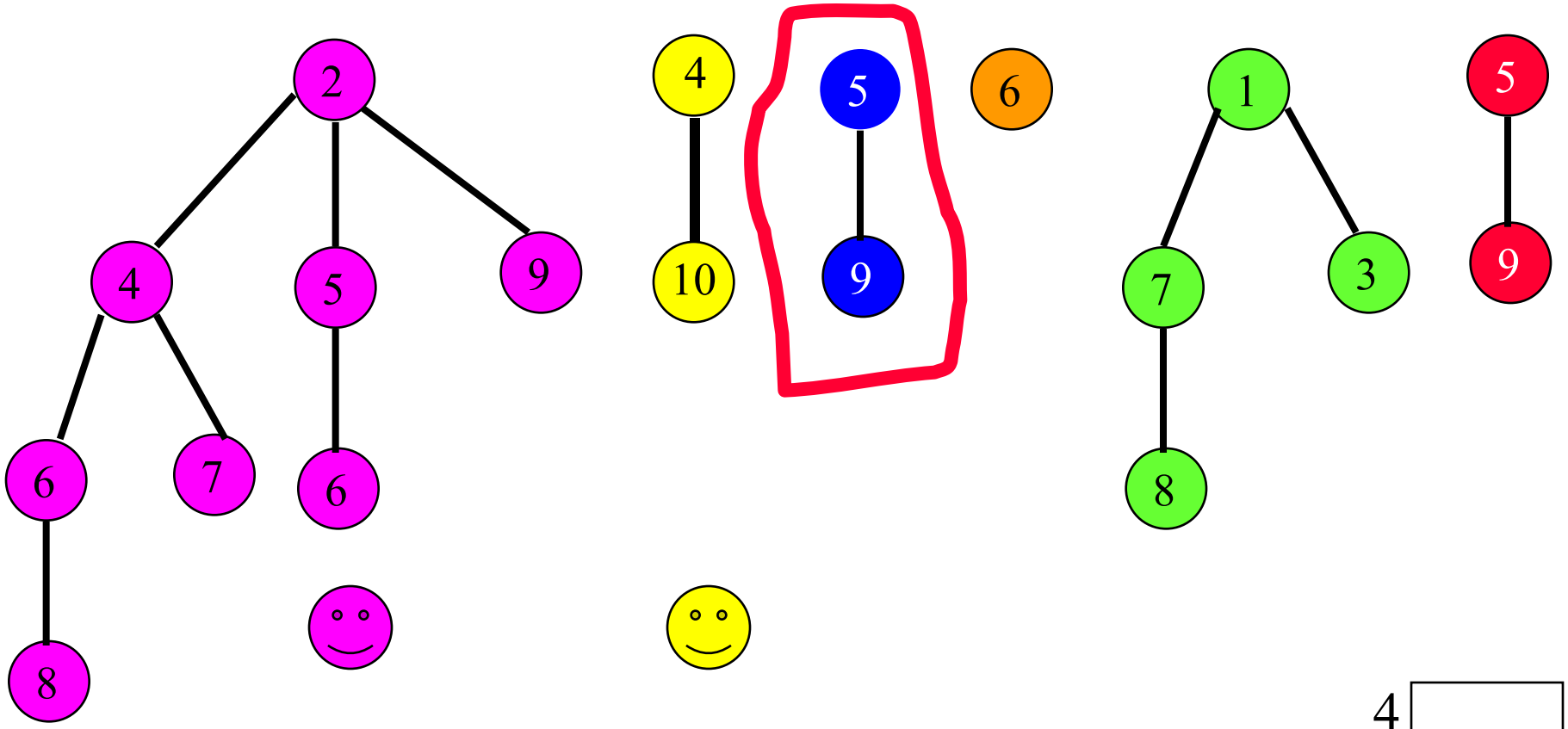


tree table

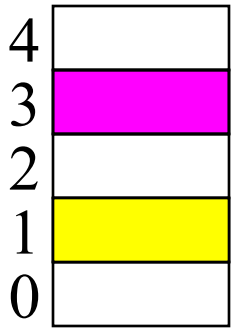


Update tree table.

Pairwise Combine



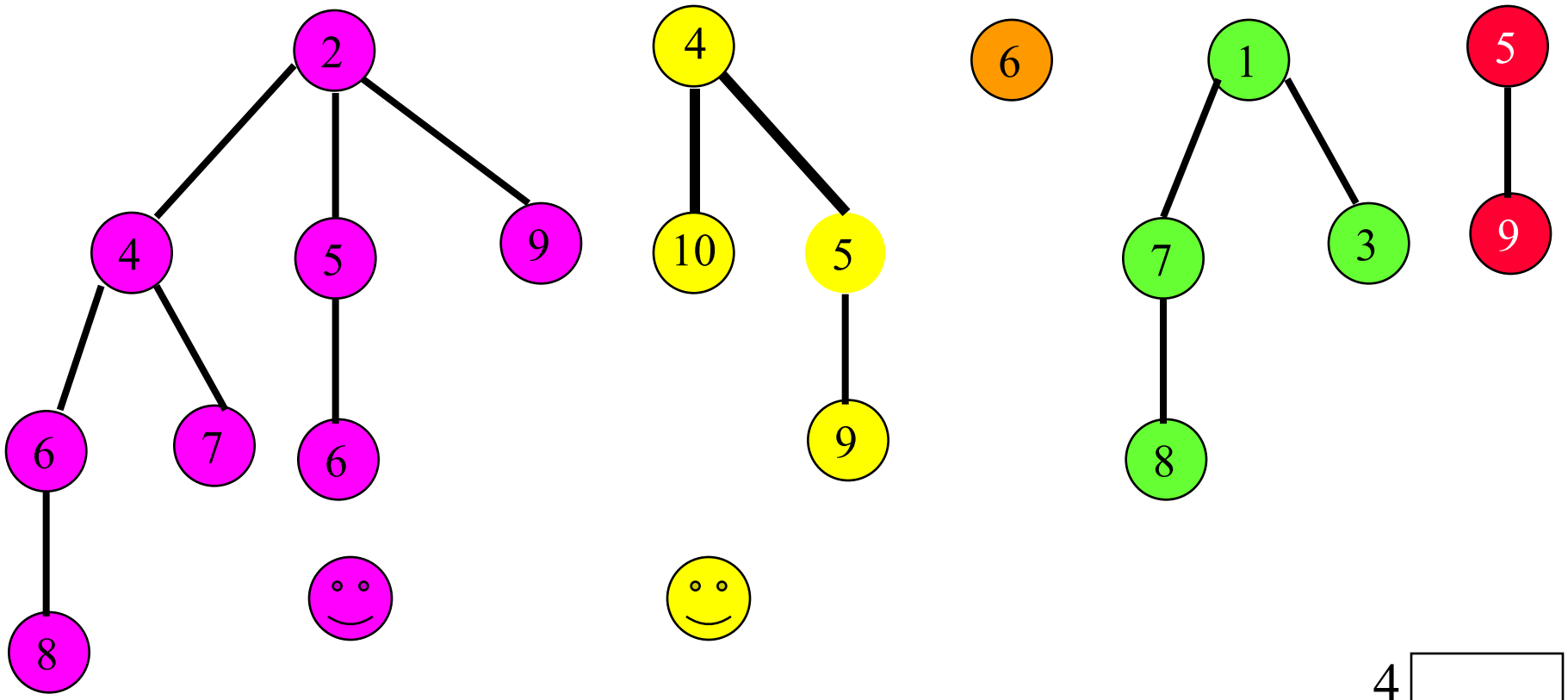
tree table



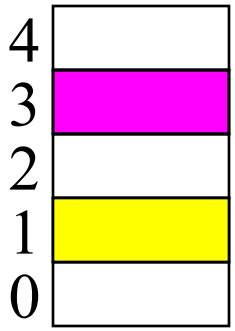
Combine 2 min trees of degree 1.

Make the one with larger root a subtree of other.

Pairwise Combine

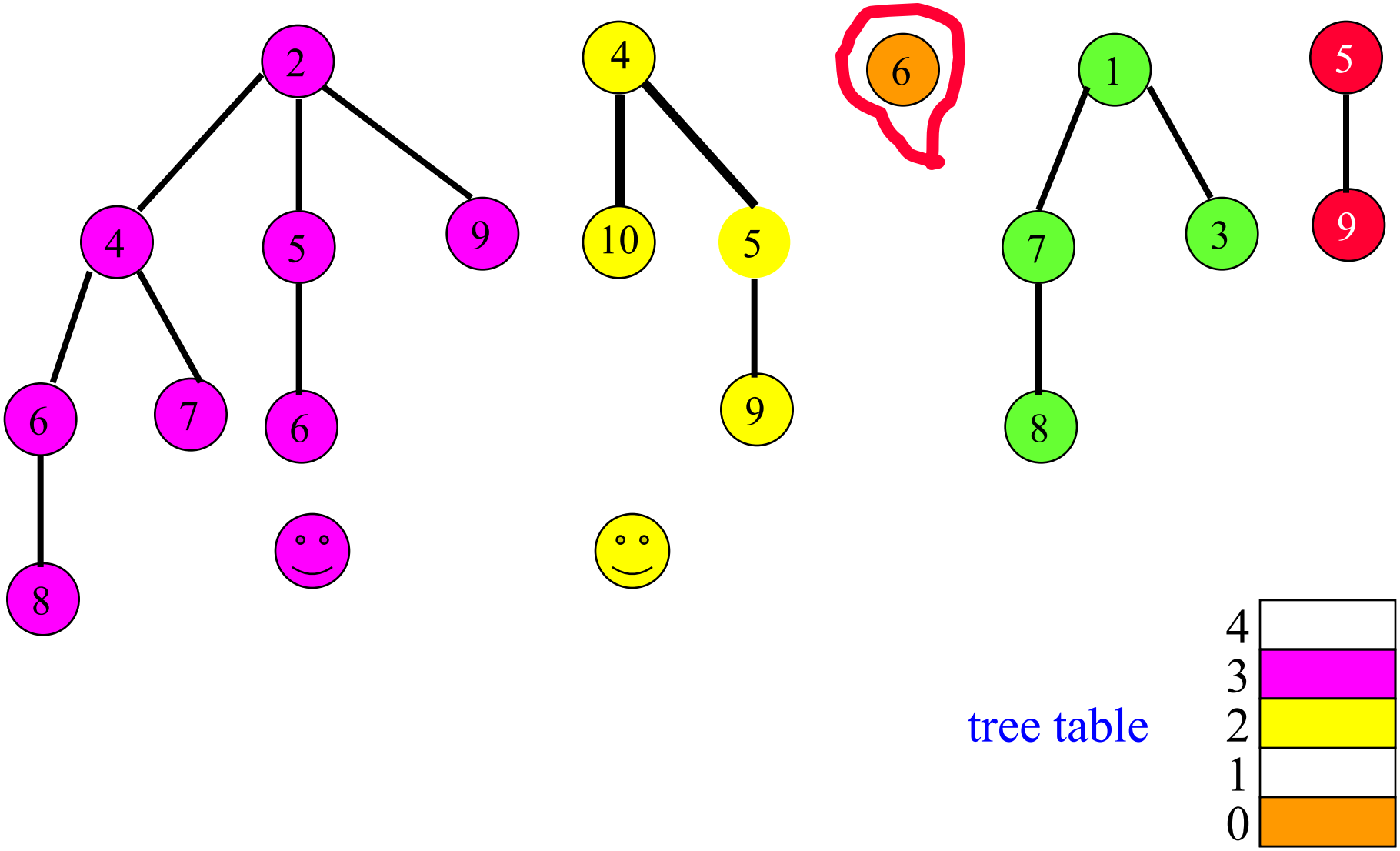


tree table

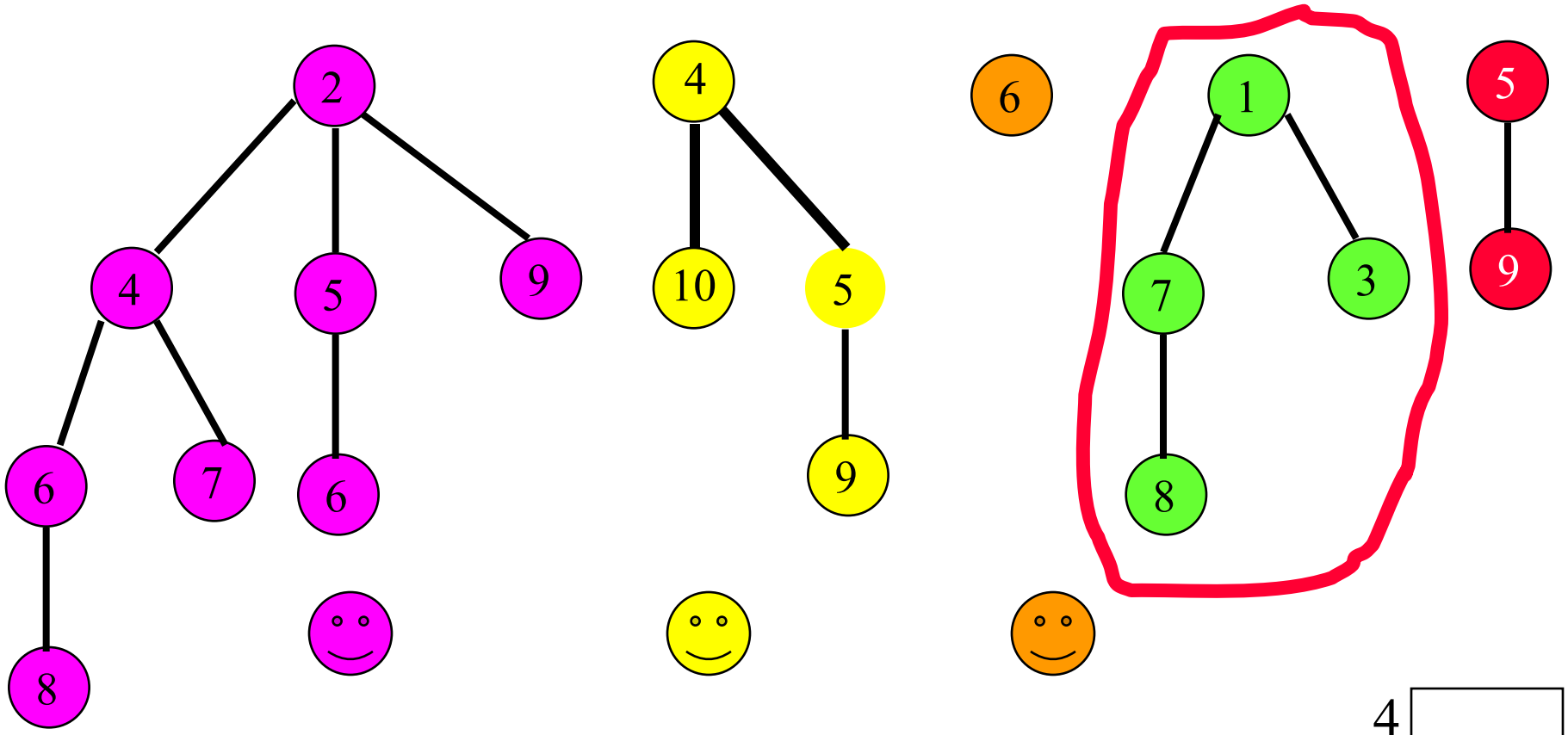


Update tree table.

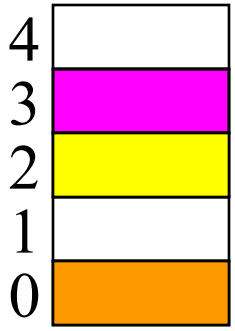
Pairwise Combine



Pairwise Combine



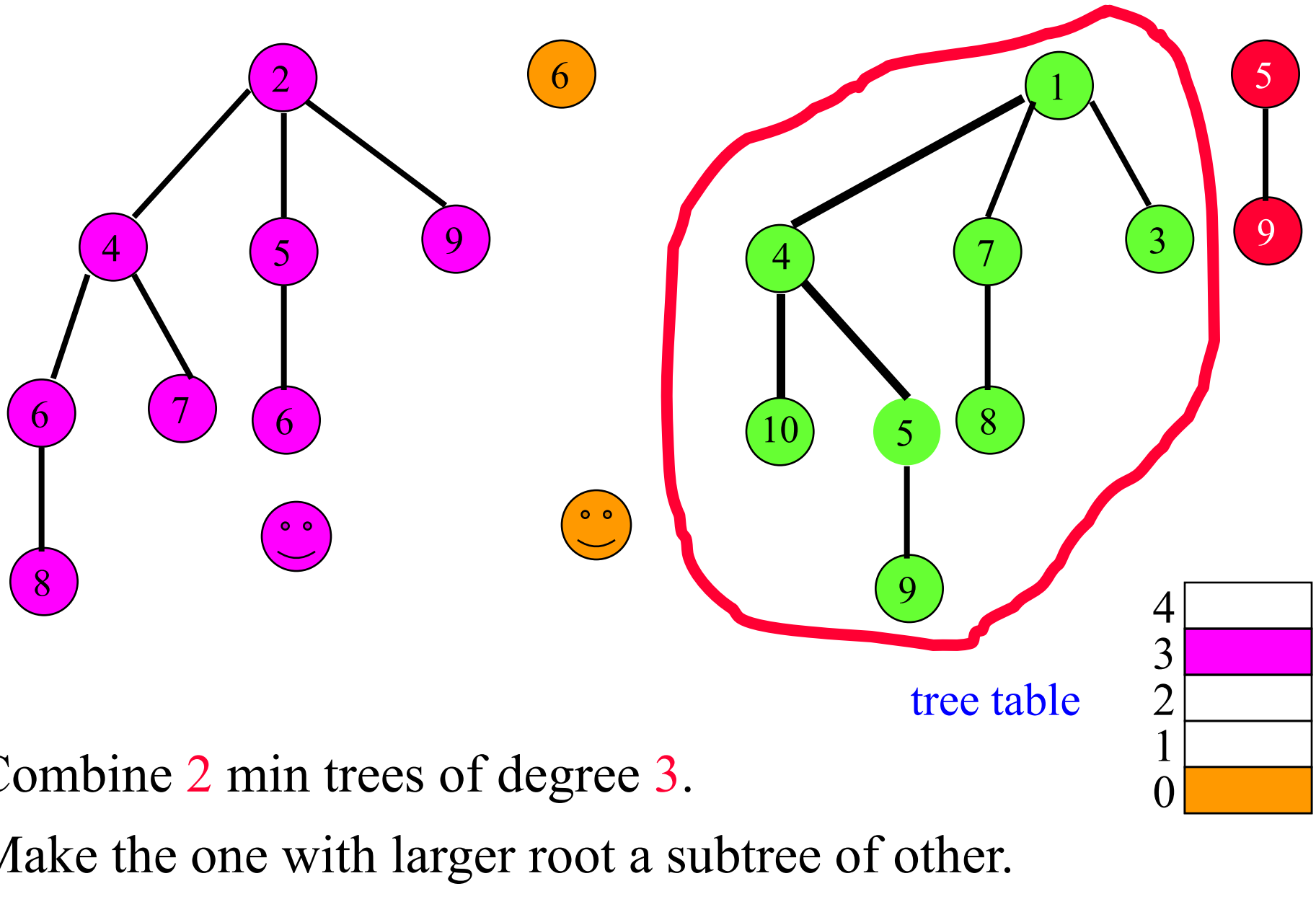
tree table



Combine 2 min trees of degree 2.

Make the one with larger root a subtree of other.

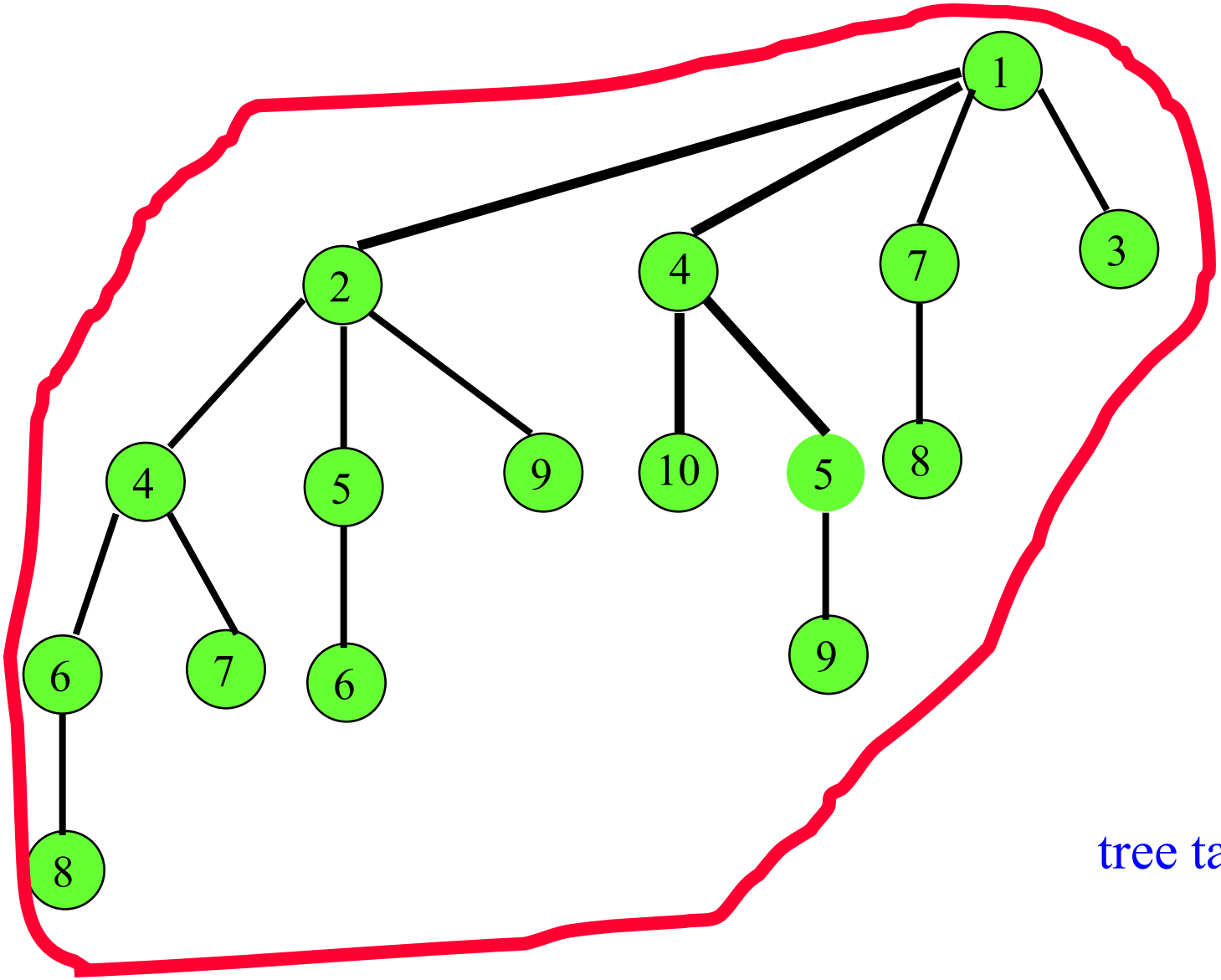
Pairwise Combine

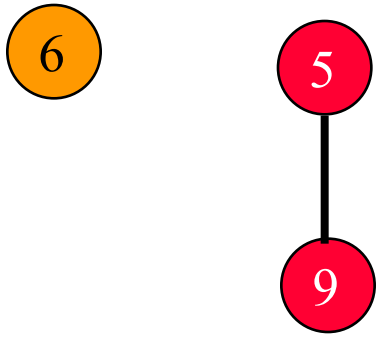


Combine 2 min trees of degree 3.

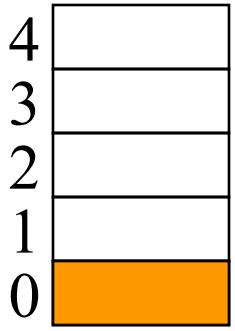
Make the one with larger root a subtree of other.

Pairwise Combine



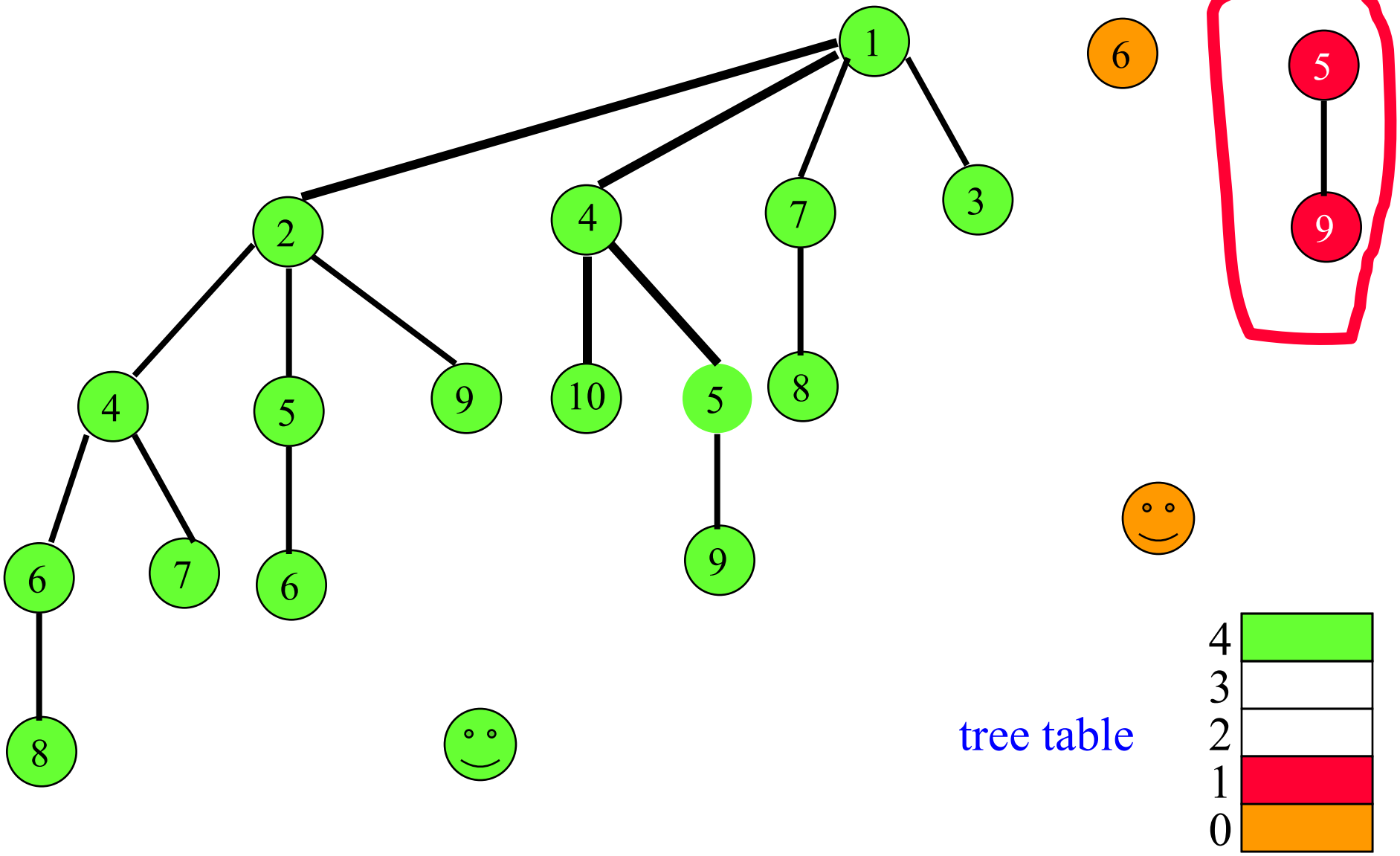


tree table

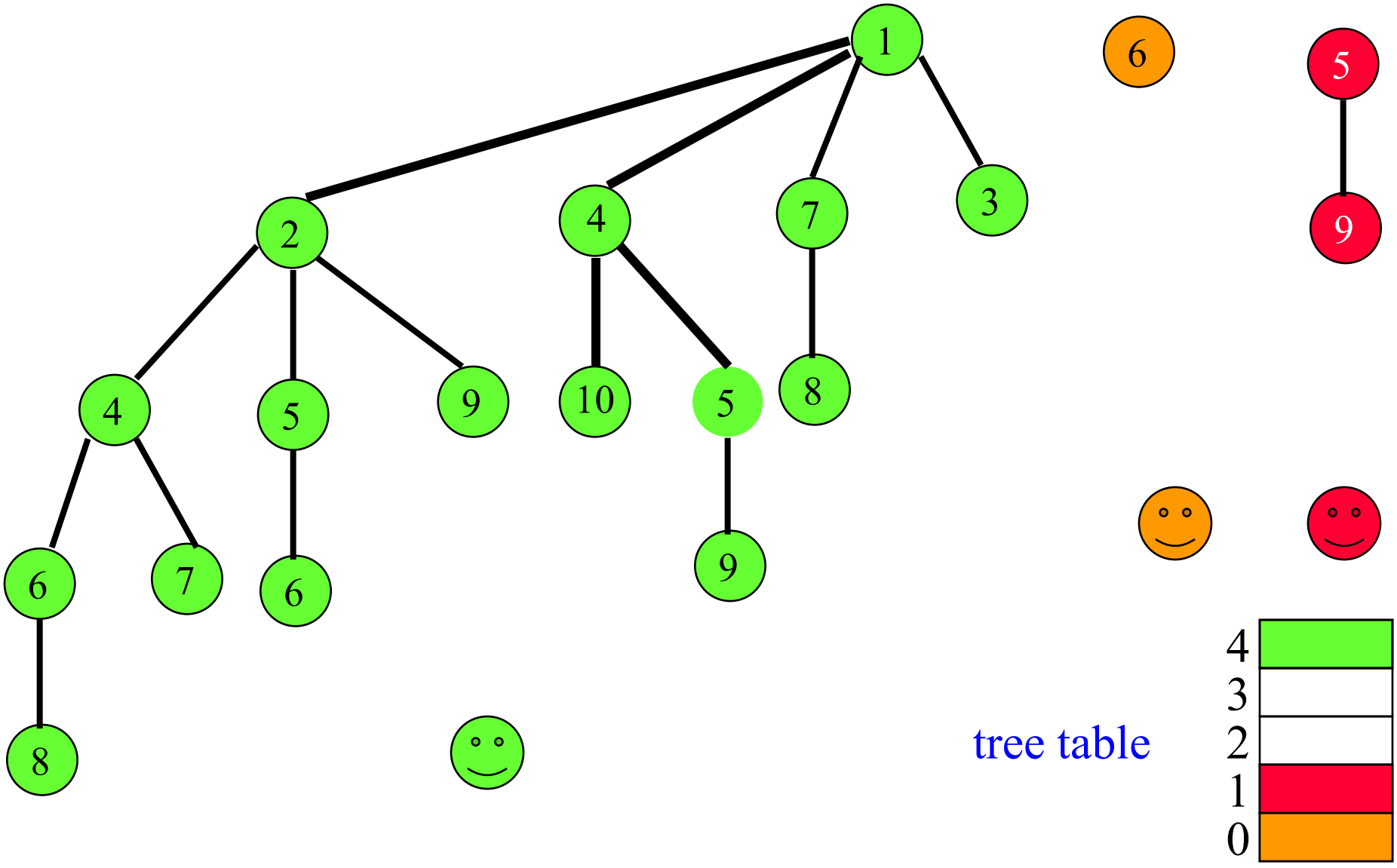


Update tree table.

Pairwise Combine



Pairwise Combine



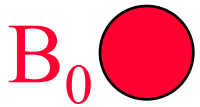
Create circular list of remaining trees.

Complexity Of Correct Remove Min

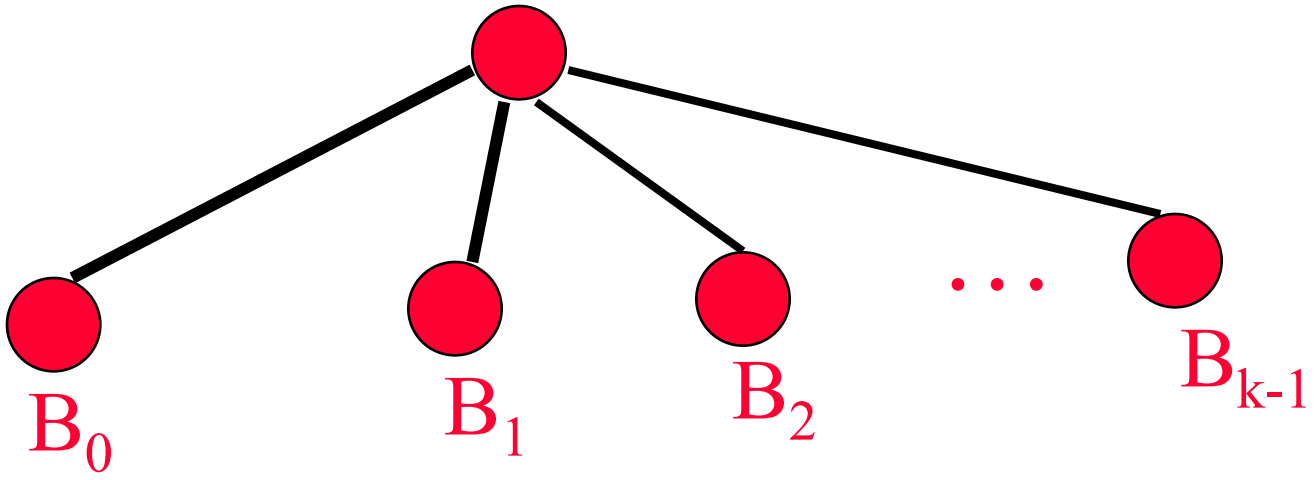
- Remove tree with min element. O(1)
- Reinsert its subtrees. O(1)
- Create and initialize tree table.
  - O(MaxDegree).
  - Done once only.
- Examine s min trees and pairwise combine.
  - O(s).
- Collect remaining trees from tree table, reset table entries to null, and set binomial heap pointer.
  - O(MaxDegree).
- Overall complexity of remove min.
  - O(MaxDegree + s).

Binomial Trees

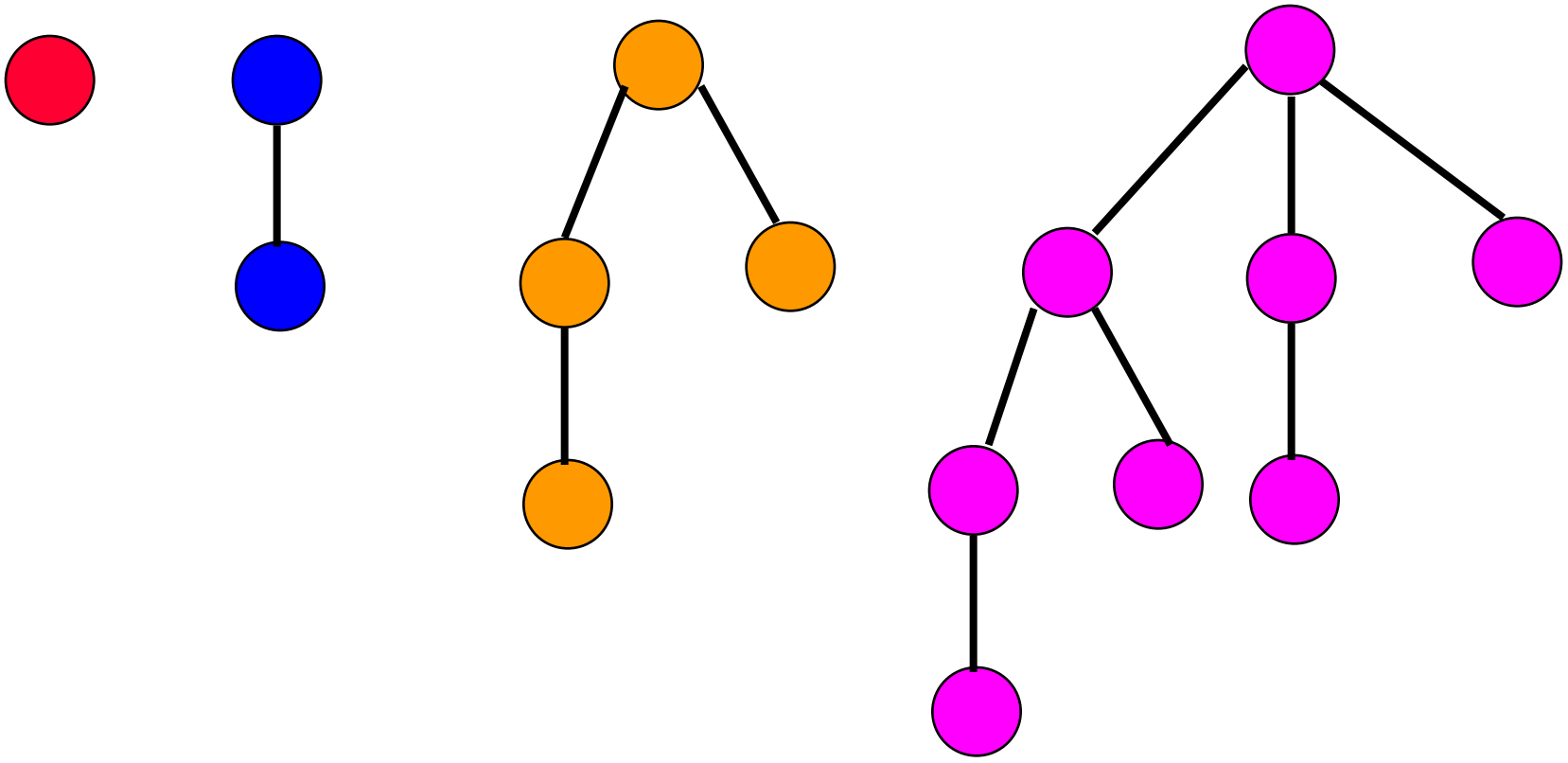
- B_k is degree k binomial tree.



- B_k , $k > 0$, is:

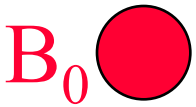


Examples



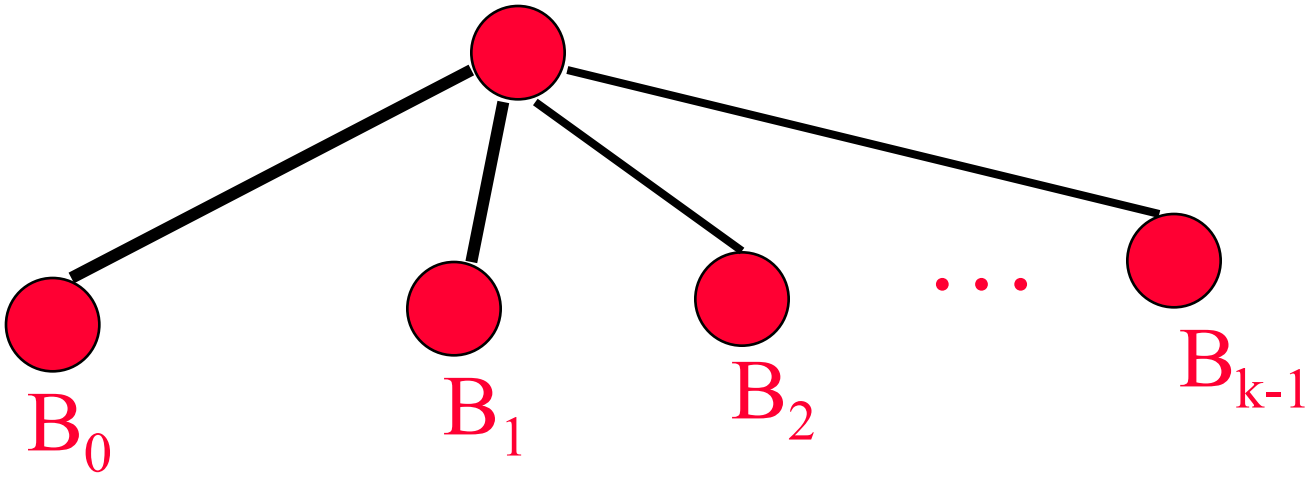
Number Of Nodes In B_k

- $N_k =$ number of nodes in B_k.



$N_0 = 1$

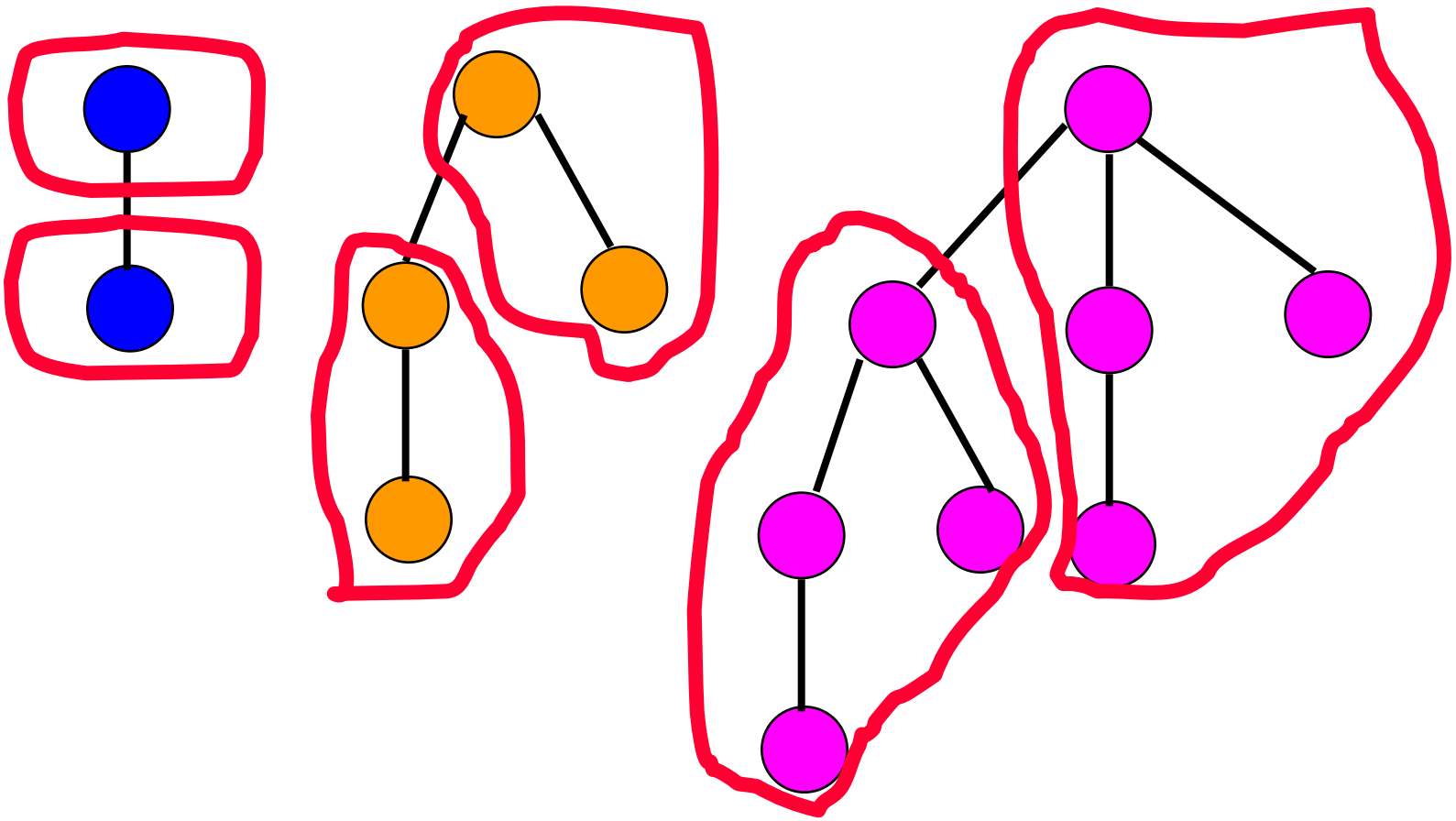
- B_k, $k > 0$, is:



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

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, MaxDegree = O(log n).

Analysis Of Binomial Heaps

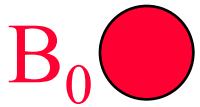
	Leftist trees	Binomial heaps	
		Actual	Amortized
Find min (or max)	O(1)	O(1)	O(1)
Insert	O(log n)	O(1)	O(1)
Remove min (or max)	O(log n)	O(n)	O(log n)
Meld	O(log n)	O(1)	O(1)

Operations

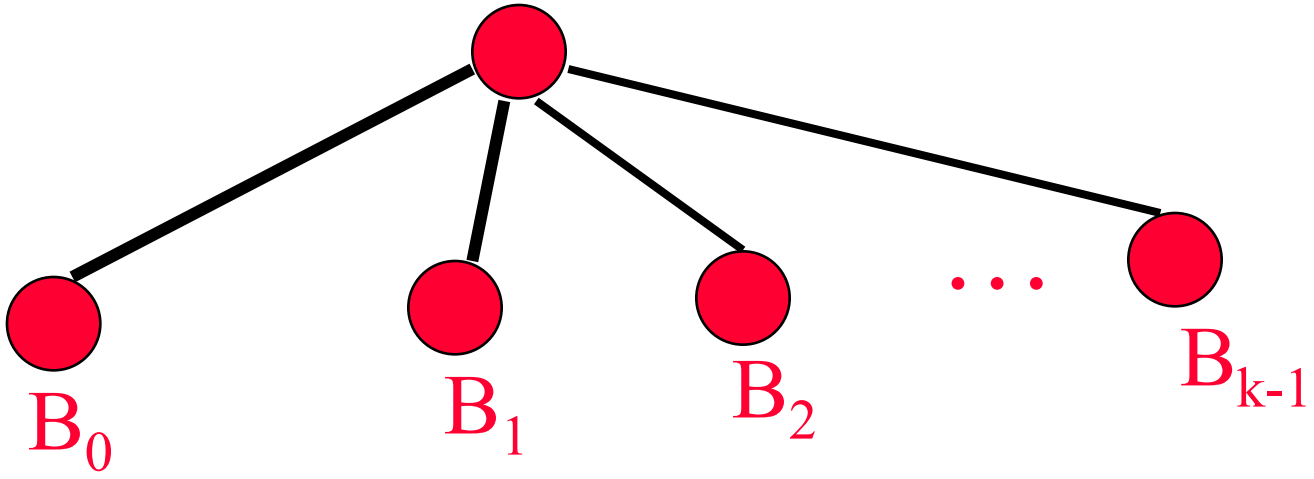
- Insert
  - Add a new min tree to top-level circular list.
- Meld
  - Combine two circular lists.
- Remove min
  - Pairwise combine min trees whose roots have equal degree.
  - O(MaxDegree + s), where s is number of min trees following removal of min element but before pairwise combining.

Binomial Trees

- B_k is degree k binomial tree.

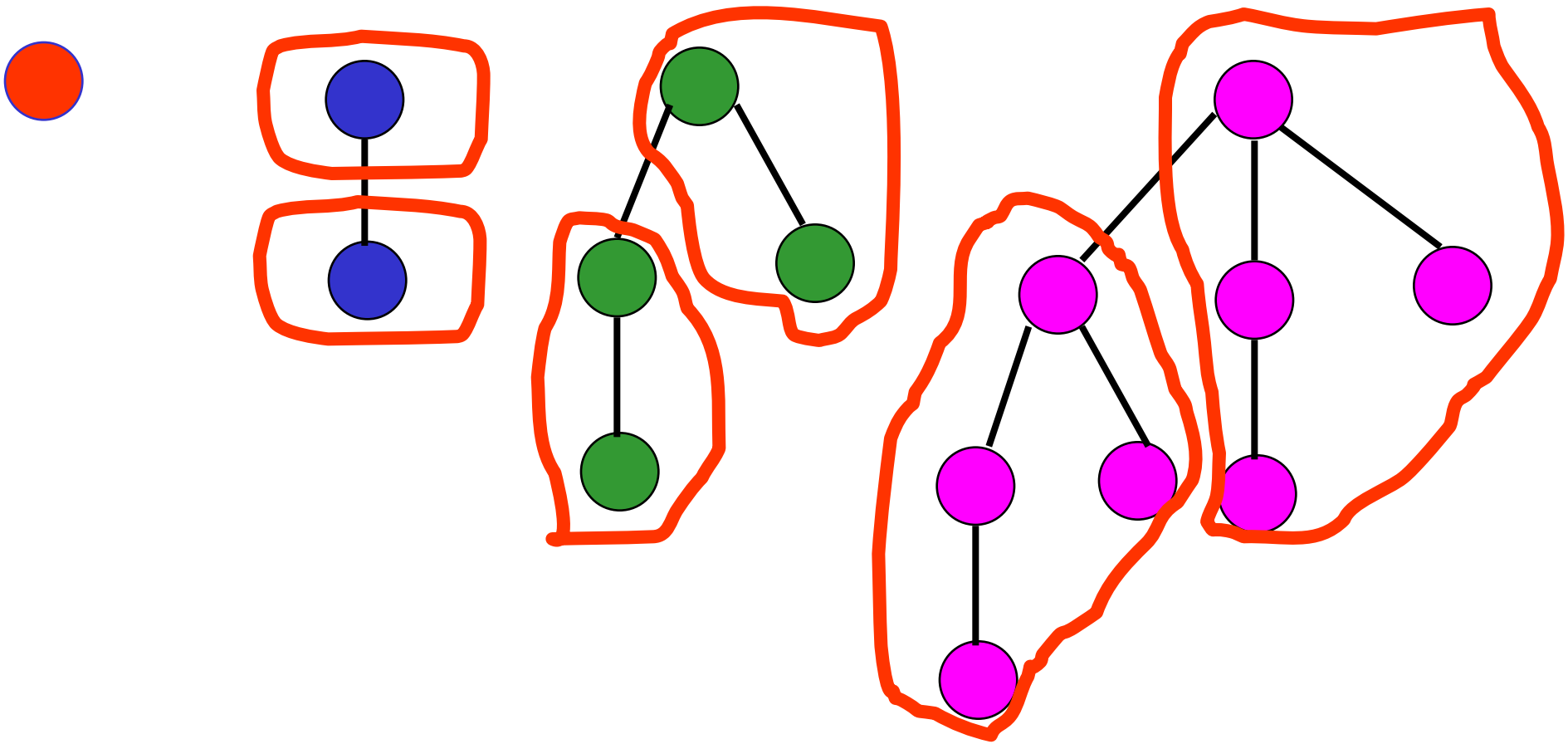


- B_k , $k > 0$, is:



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

- Initially, all trees in system are Binomial trees (actually, there are no trees initially).
- Assume true before an operation, show true after the operation.
- Insert creates a B_0.
- Meld does not create new trees.
- Remove Min
  - Reinserted subtrees are binomial trees.
  - Pairwise combine takes two trees of equal degree and makes one a subtree of the other.

Complexity of Remove Min

- Let n be the number of inserts.
  - 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 establish 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 remove mins and divide by the number of remove mins.
- Consider the sequence insert, insert, …, insert, remove min.
  - The cost of the remove min is O(n), where n is the number of inserts in the sequence.
  - So, amortized cost of a remove min is O(n/1) = O(n).

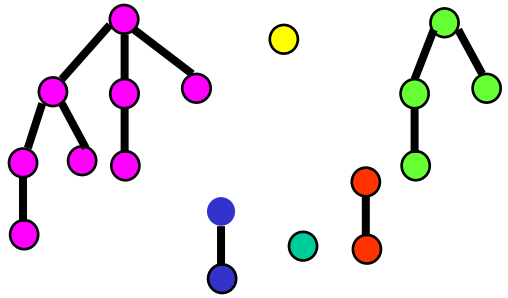
Accounting Method

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

Potential Function

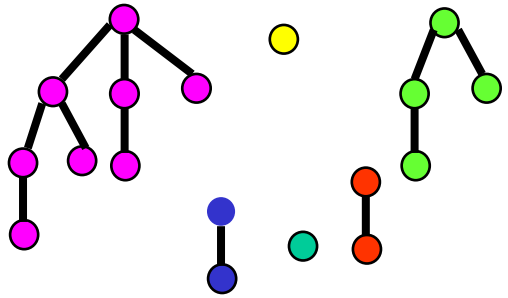
- P(i) = amortizedCost(i) – actualCost(i) + P(i – 1)
- P(i) – P(0) is the amount by which the first i operations have been over charged.
- We shall use a credit scheme to keep track of (some of) the over charge.
- There will be 1 credit on each min tree.
- Initially, #trees = 0 and so total credits and P(0) = 0.
- Since number of trees cannot be <0, the total credits is always >= 0 and hence P(i) >= 0 for all i.

Insert



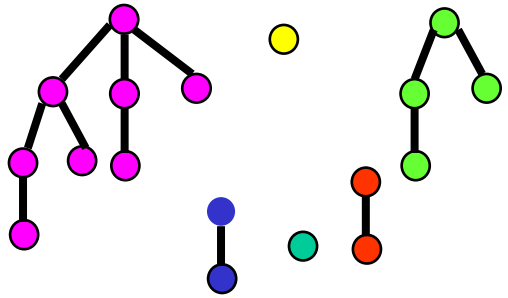
- Guessed amortized cost = 2.
- Use 1 unit to pay for the actual cost of the insert.
- Keep the remaining 1 unit as a credit.
- Keep this credit with the min tree that is created by the insert operation.
- Potential increases by 1, because there is an overcharge of 1.

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.

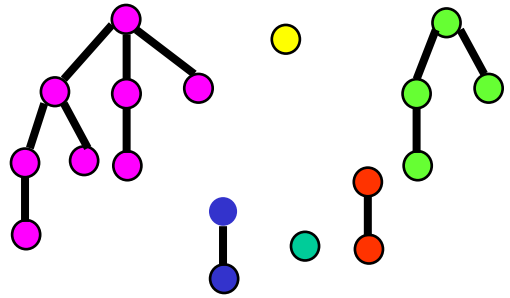
Remove Min



- Let MinTrees be the set of min trees in the binomial heap just before remove 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 remove min is <= MaxDegree + s

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

Remove Min



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

Paying Actual Cost Of A Remove 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 remove min operation.
  - #MinTrees comes from the min trees themselves, at the rate of 1 unit per min tree, using up their credits.
  - Potential may increase or decrease but remains nonnegative as each remaining tree has a credit.

Potential Method

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

Potential Function

- P(i) = Σ#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
  - ΔP = 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 Remove Min

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

ith Operation Is A Remove Min

- Actual cost of remove 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 remove min = actual cost + ΔP

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

  $<= 3\log_2 n$.

Actual Cost Of Any Operation Sequence

- Start with empty Binomial heaps
- Do i inserts, m melds, and r remove mins
- Actual cost is $O(i + m + r \log i)$