Project 1

External Sorting

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.

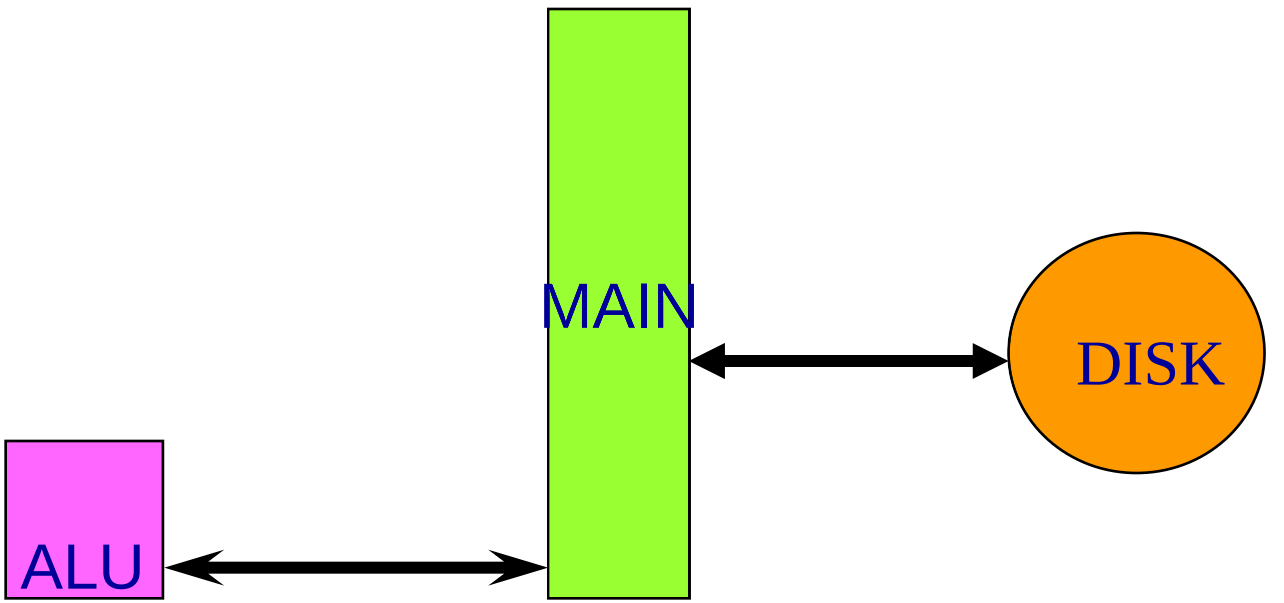
Small n But Large File

- 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.

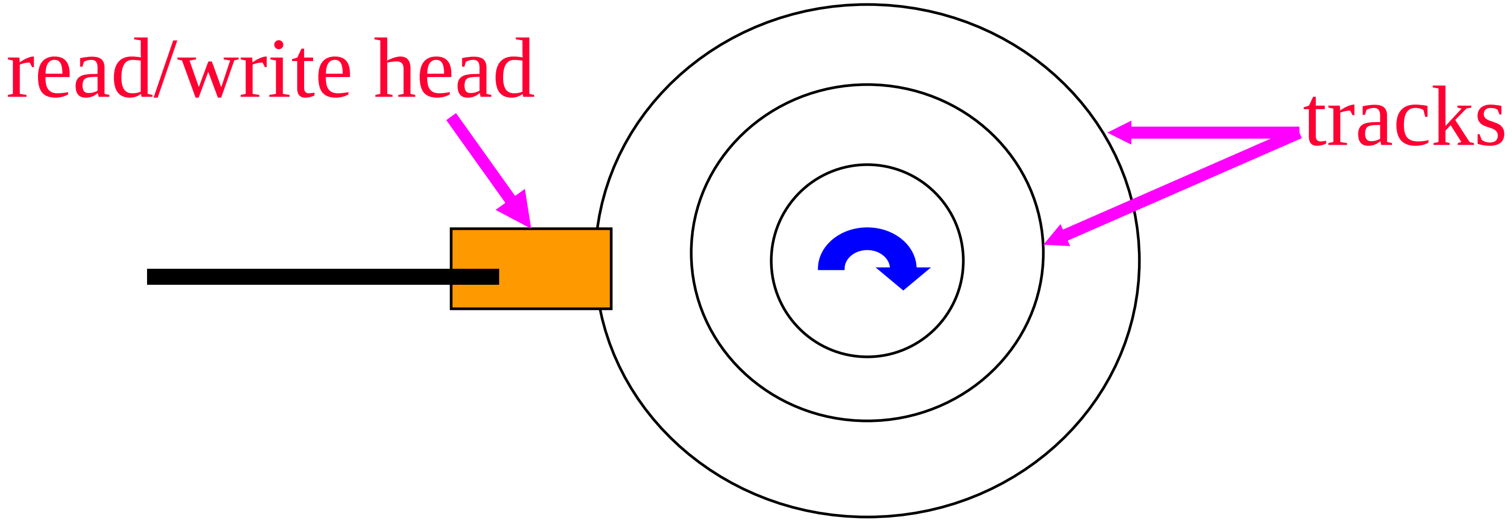
Data Structures/Concepts

- 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

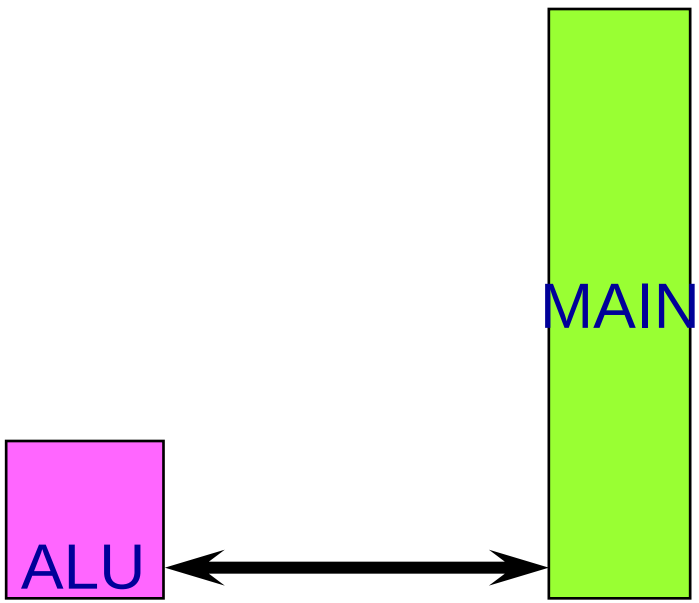


Disk Characteristics

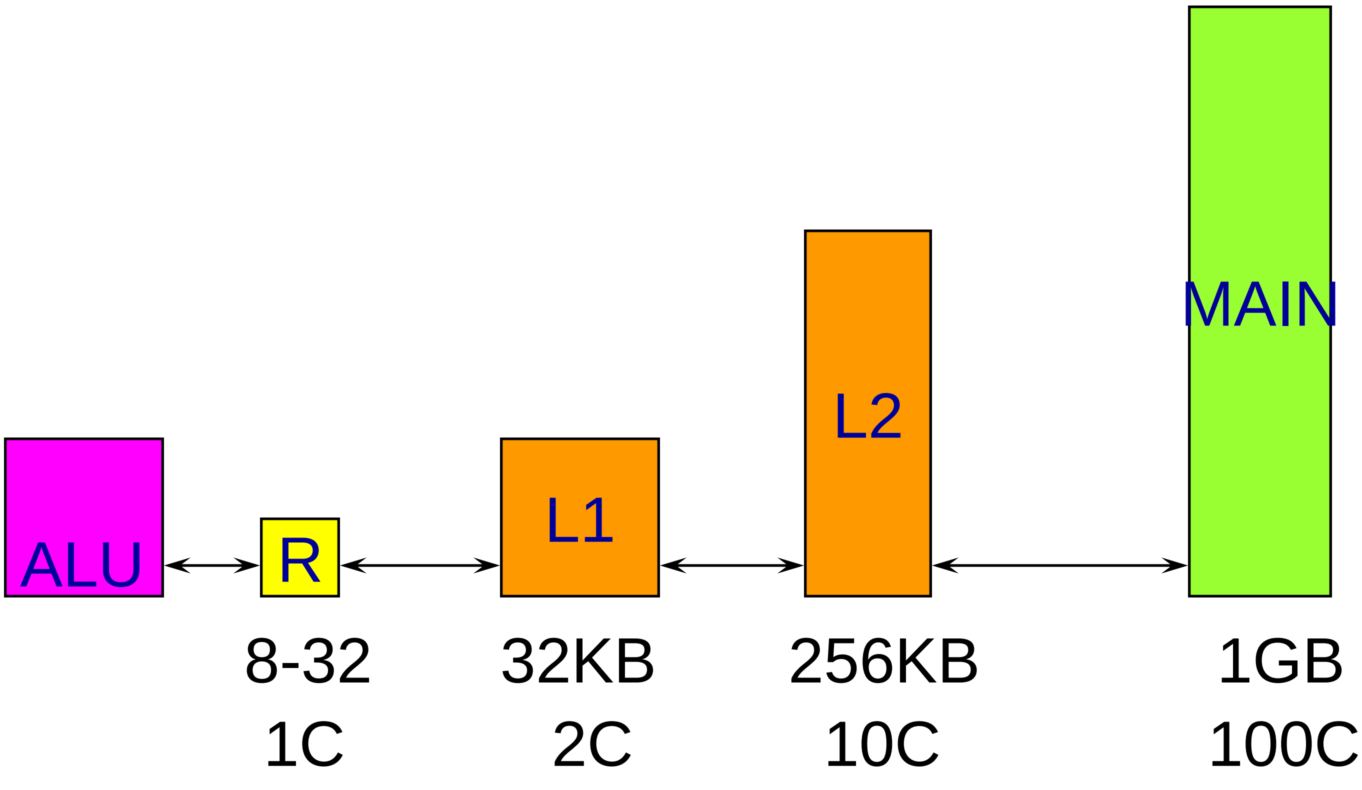


- Seek time
  - Approx. 100,000 arithmetics
- Latency time
  - Approx. 25,000 arithmetics
- Transfer time
- Data access by block

Traditional Internal Memory Model



More Accurate Memory Model



Phase 1

Warm up

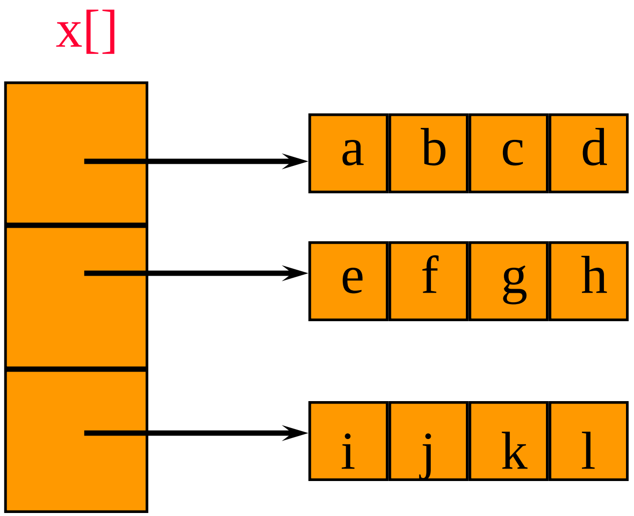
Matrix Multiplication

```
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!

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

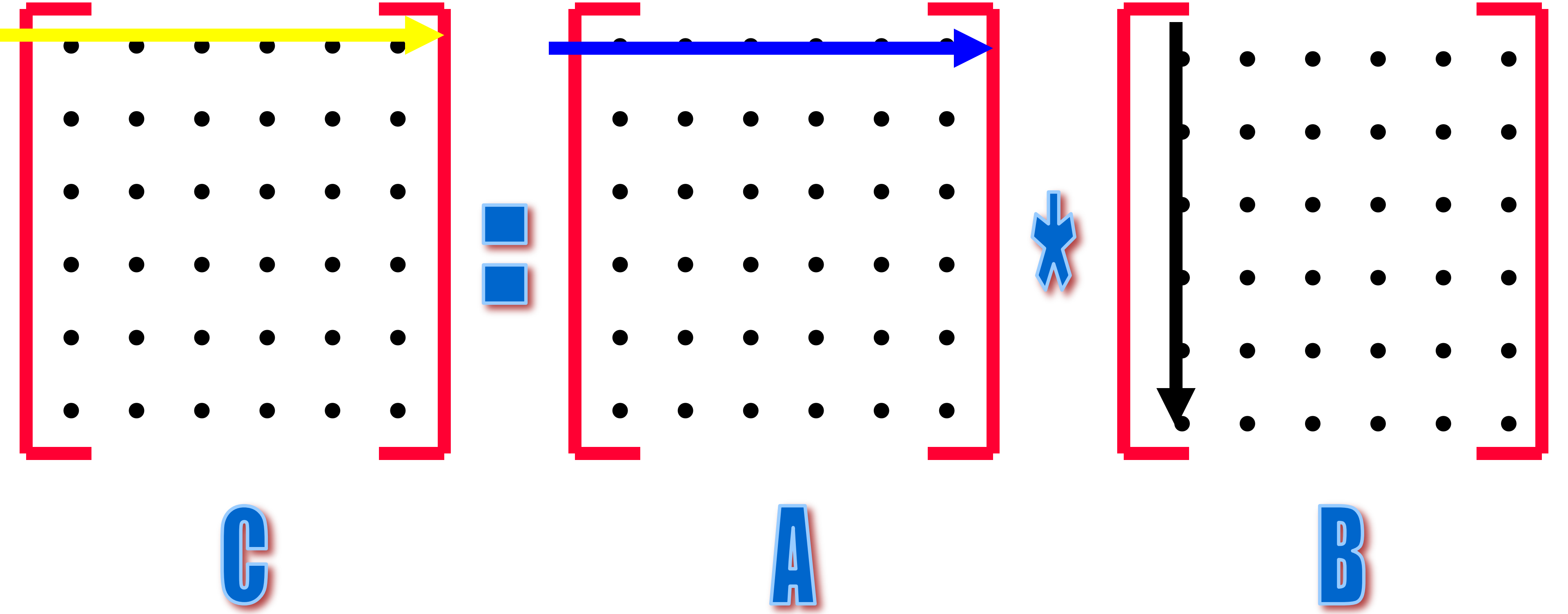
int x[3][4];



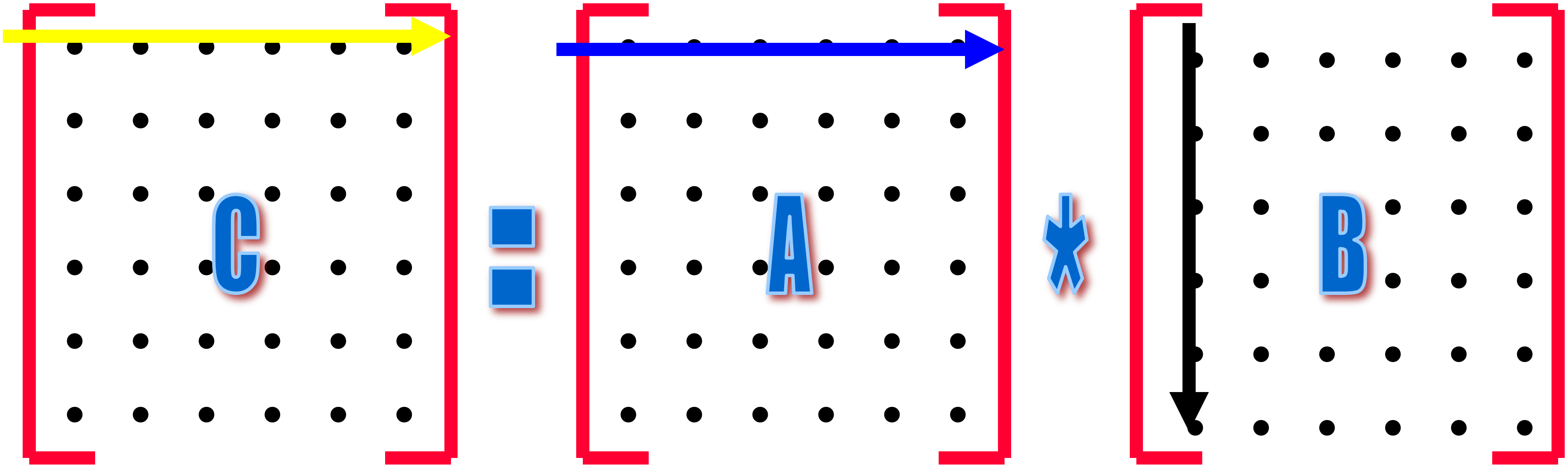
Array of Arrays Representation

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];
```



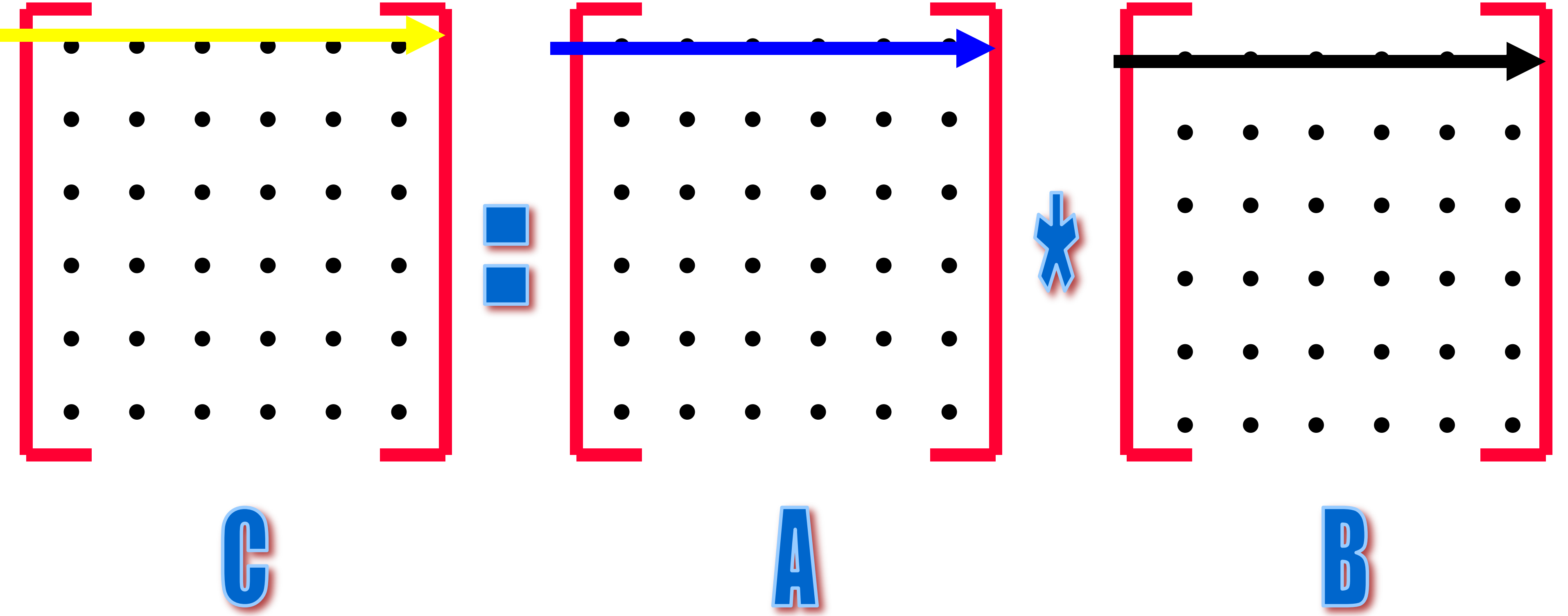
ijk Analysis



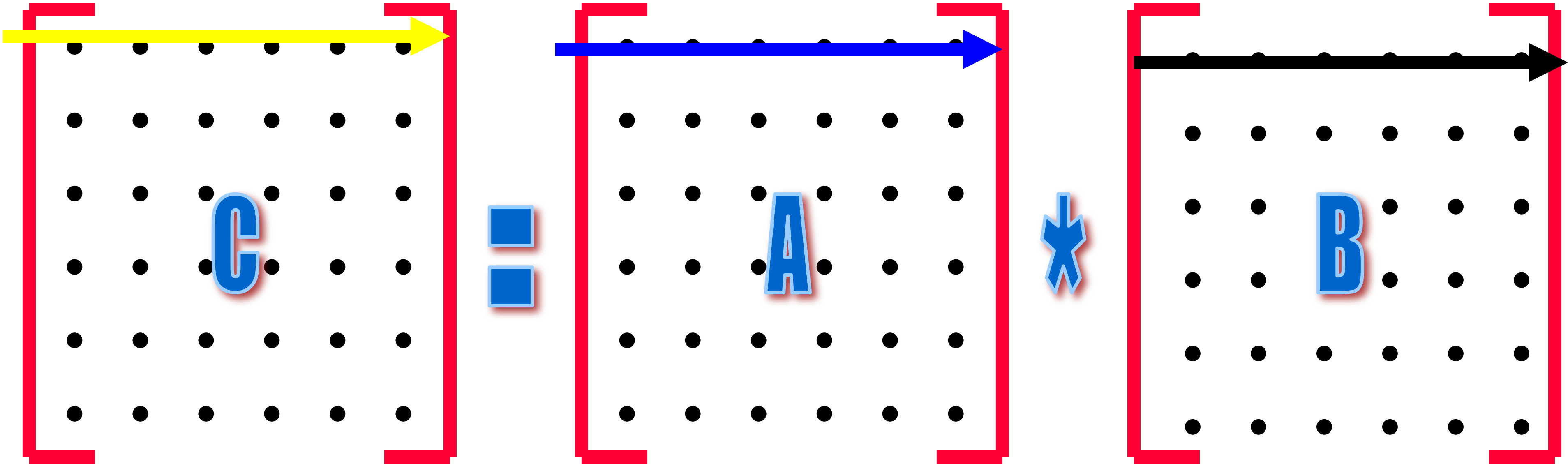
- Block size = width of cache line = w.
- Assume one-level cache.
- C => n^2/w cache misses.
- A => n^3/w cache misses, when n is large.
- B => n^3 cache misses, when n is large.
- Total cache misses = $n^3/w(1/n + 1 + w)$.

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 = $n^3/w(2 + 1/n)$.

Warm up project

- Cache miss simulation
  - Files
- Count the missed hits
- #mh vs cache line
- #mh vs data volumn
- Simulation vs Theoretical results
- Others …
- https://www.cplusplus.com

Phase 2

External Sort: Quick 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

- To sort a large instance, select a pivot element from out of the n elements.
- Partition the n elements into 3 groups left, middle and right.
- The middle group contains only the pivot element.
- All elements in the left group are <= pivot.
- All elements in the right group are >= pivot.
- Sort left and right groups recursively.
- Answer is sorted left group, followed by middle group followed by sorted right group.

Internal Quick Sort

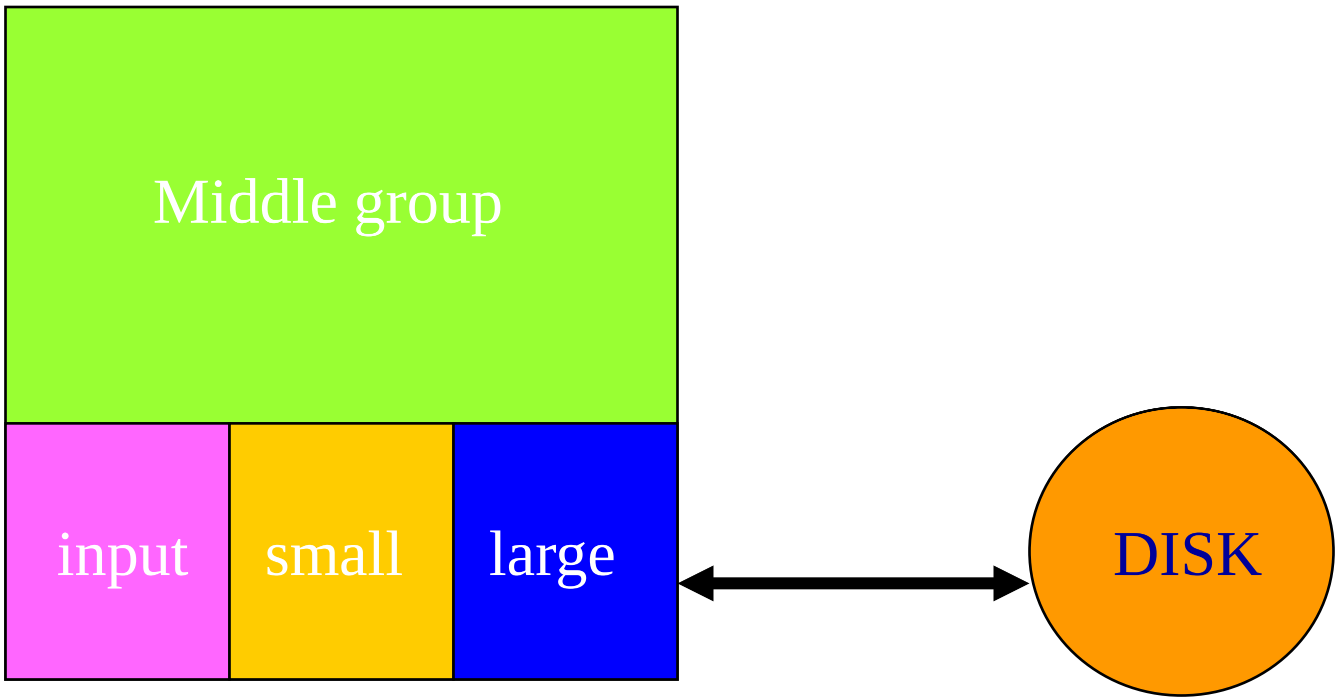
6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot.

2	5	4	1	3	6	7	9	10	11	8

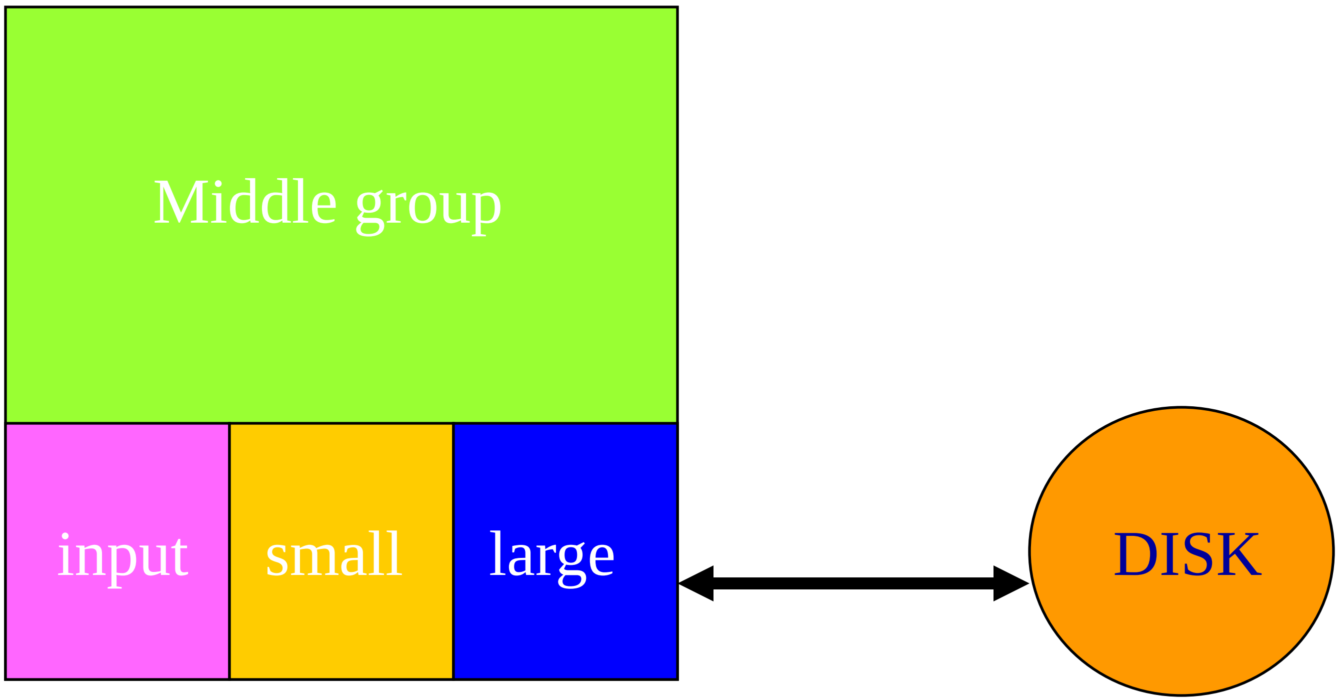
Sort left and right groups recursively.

Quick Sort – External Adaptation



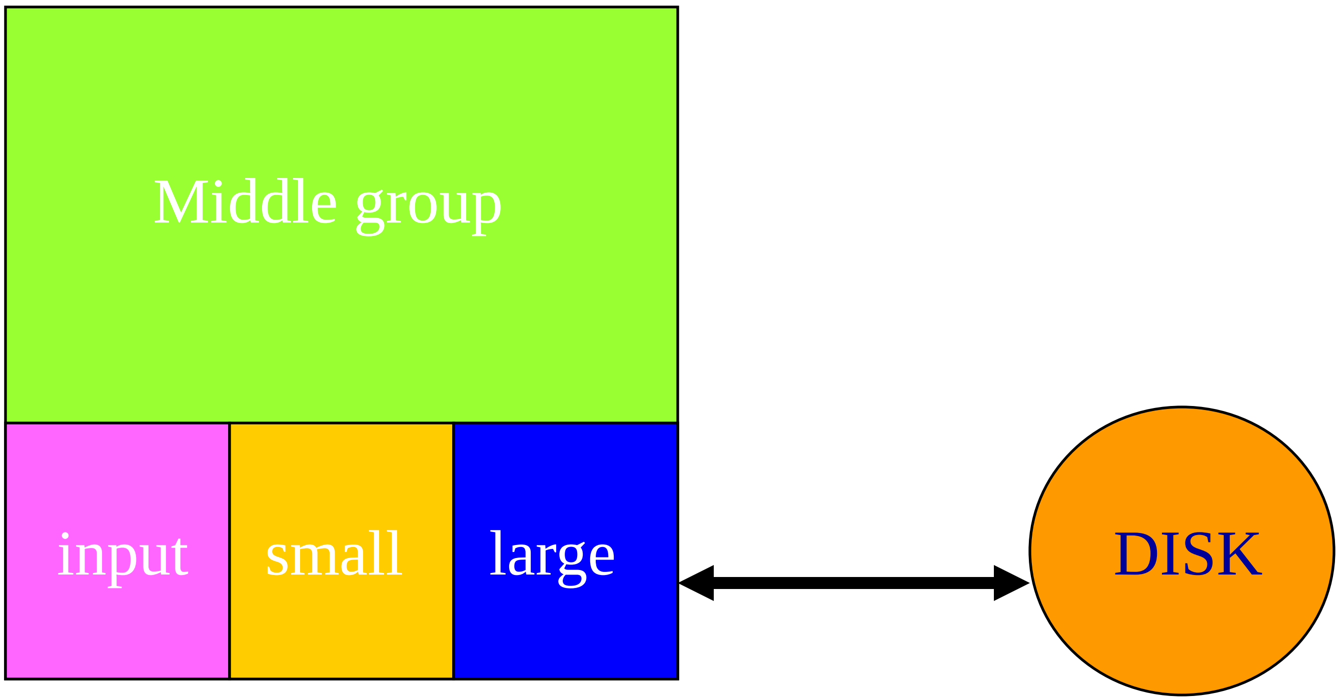
- 3 input/output buffers
  - input, small, large
- rest is used for middle group

Quick Sort – External Adaptation



- fill middle group from disk
- if next record <= middle_{min} send to small
- 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

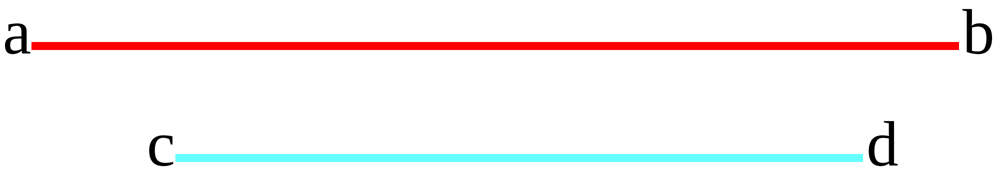


- Fill input buffer when it gets empty.
- Write small/large buffer when full.
- Write middle group in sorted order when done.
- Double-ended priority queue.

Double-ended Priority Queue: Interval Heaps

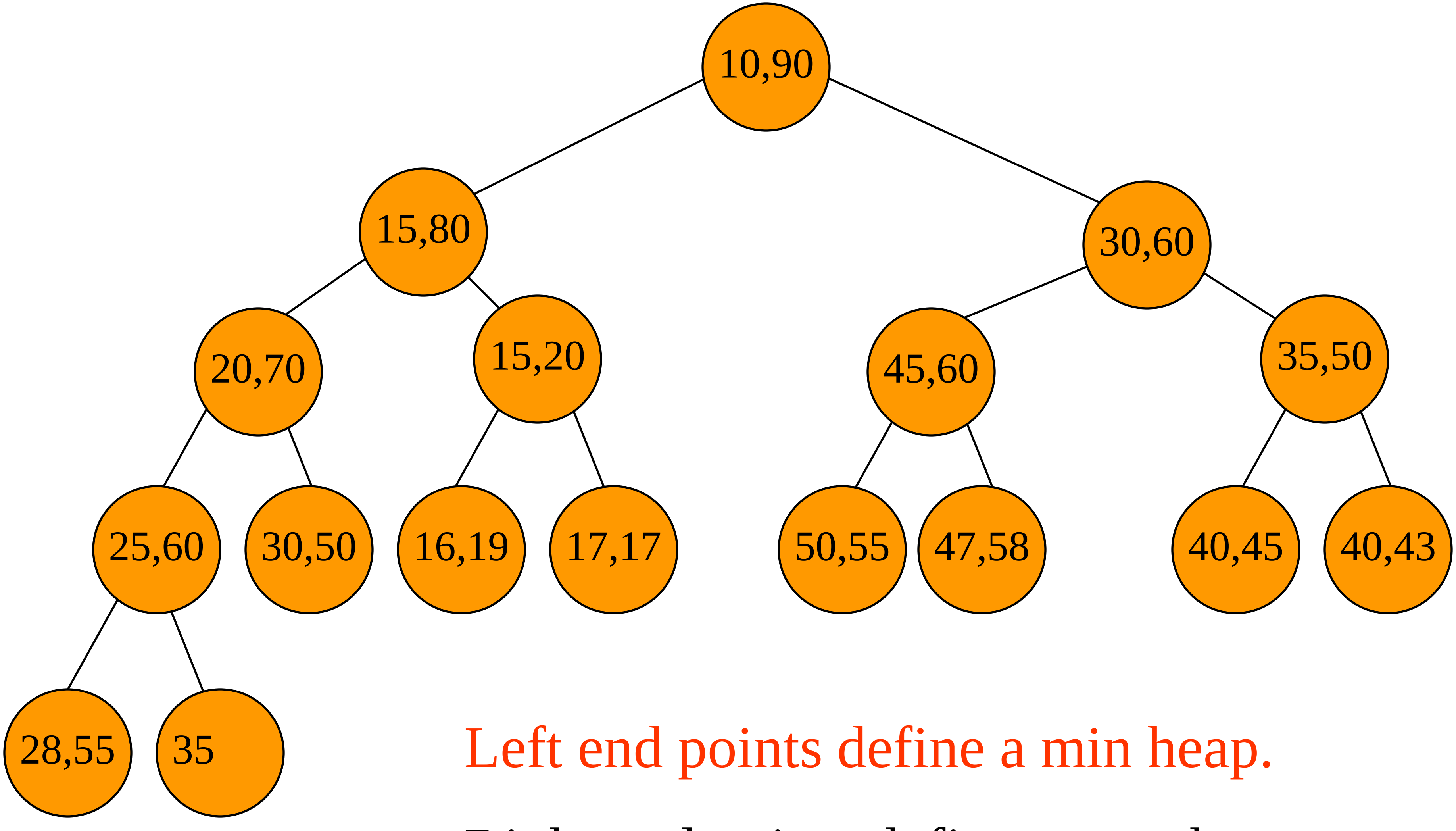
- Complete binary tree.
- Each node (except possibly last one) has 2 elements.
- Last node has 1 or 2 elements.
- Let a and b be the elements in a node P, a <= b.
- [a, b] is the interval represented by P.
- The interval represented by a node that has just one element a is [a, a].
- The interval [c,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

Interval



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

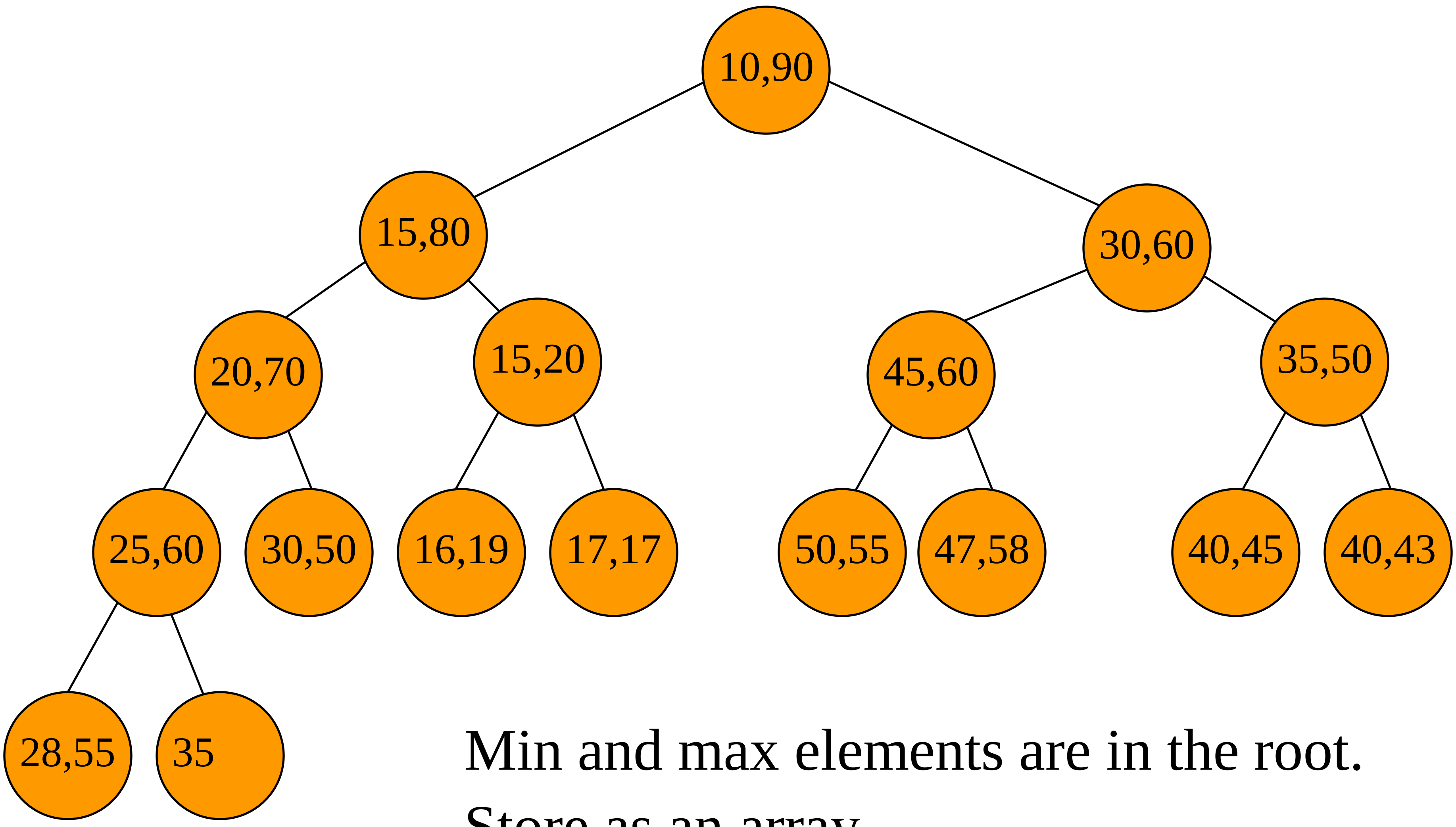
Example Interval Heap



Left end points define a min heap.

Right end points define a max heap.

Example Interval Heap

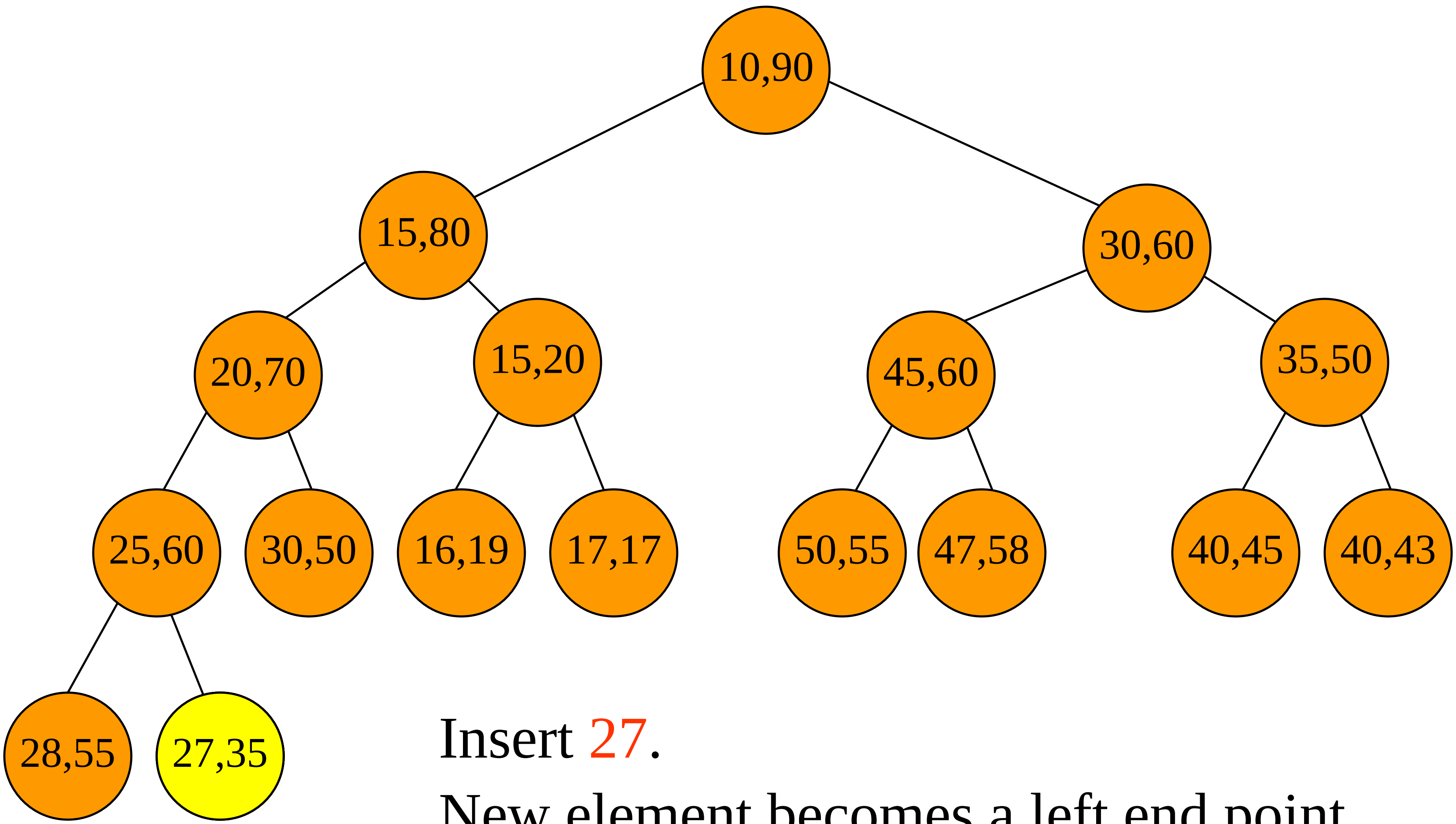


Min and max elements are in the root.

Store as an array.

Height is $\sim\log_2 n$.

Insert An Element

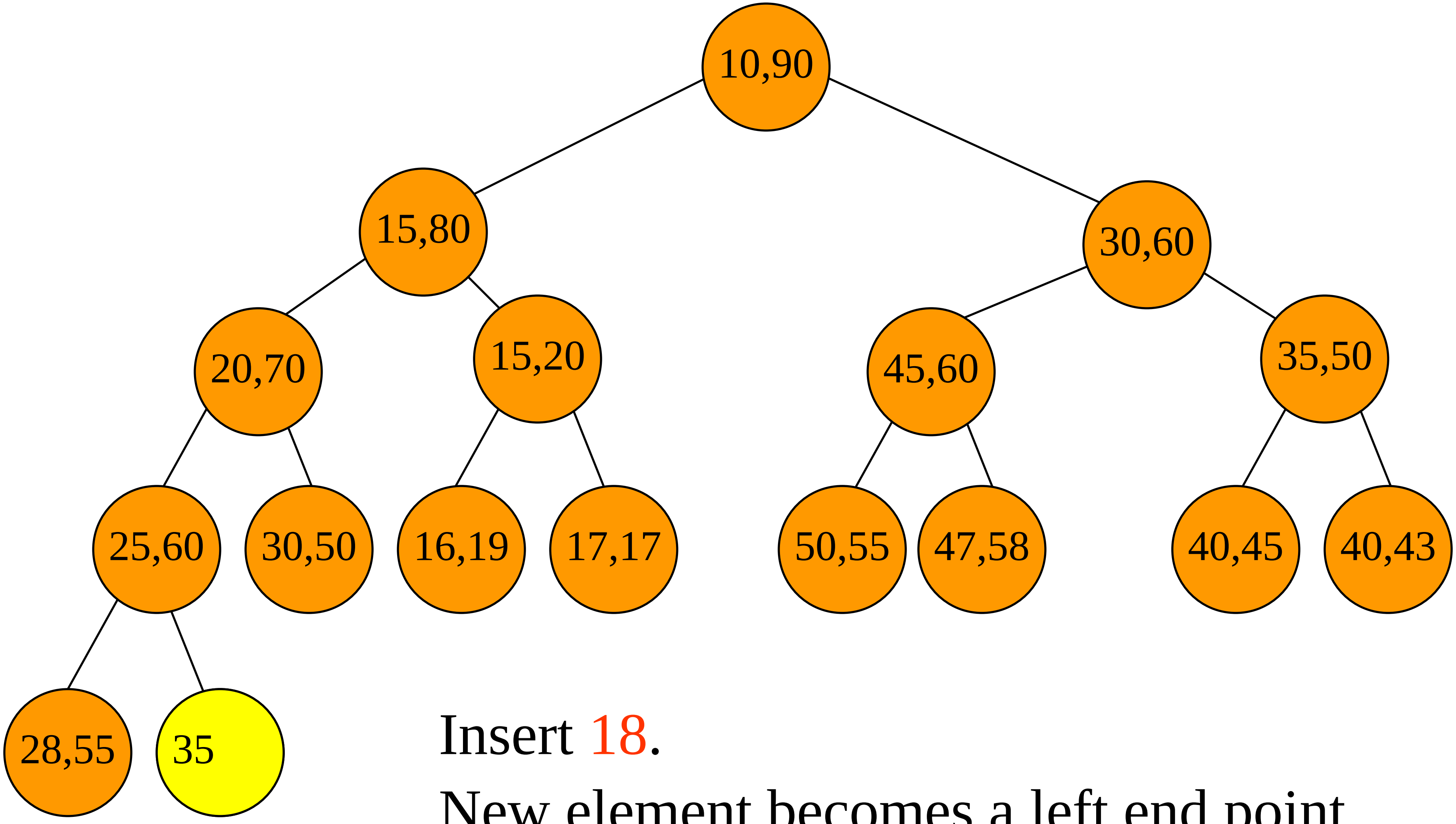


Insert 27.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

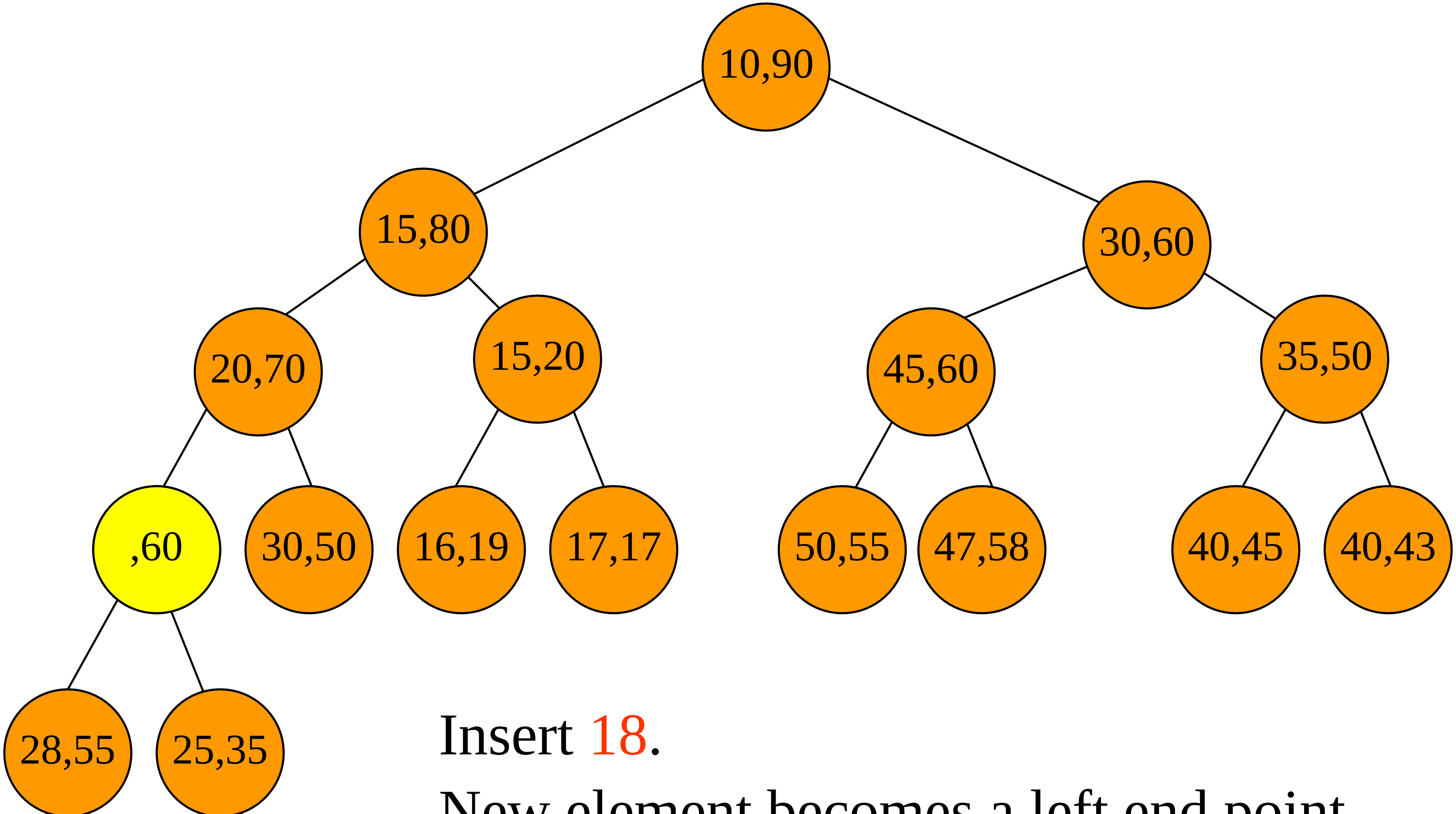


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

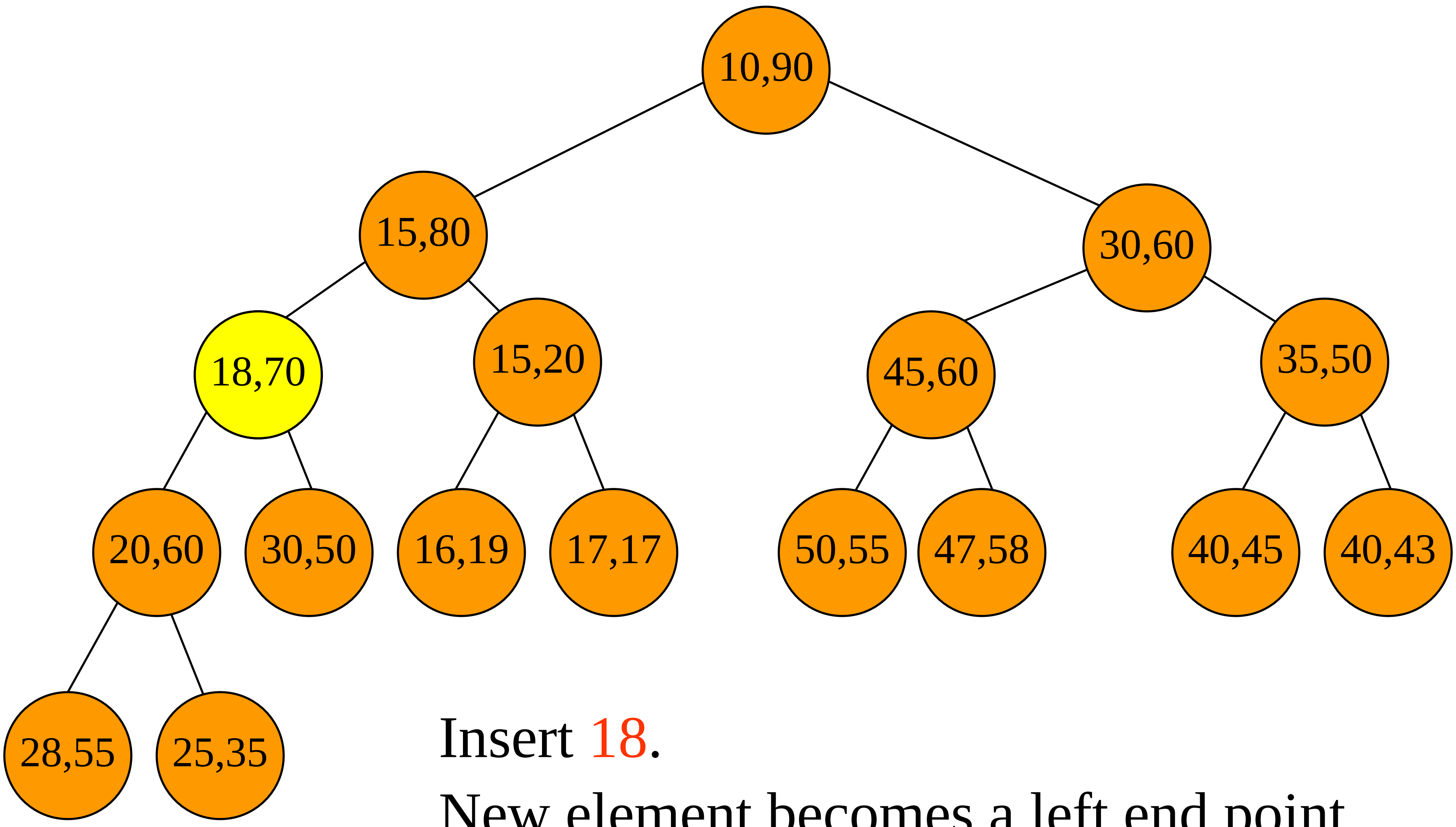


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

Another Insert

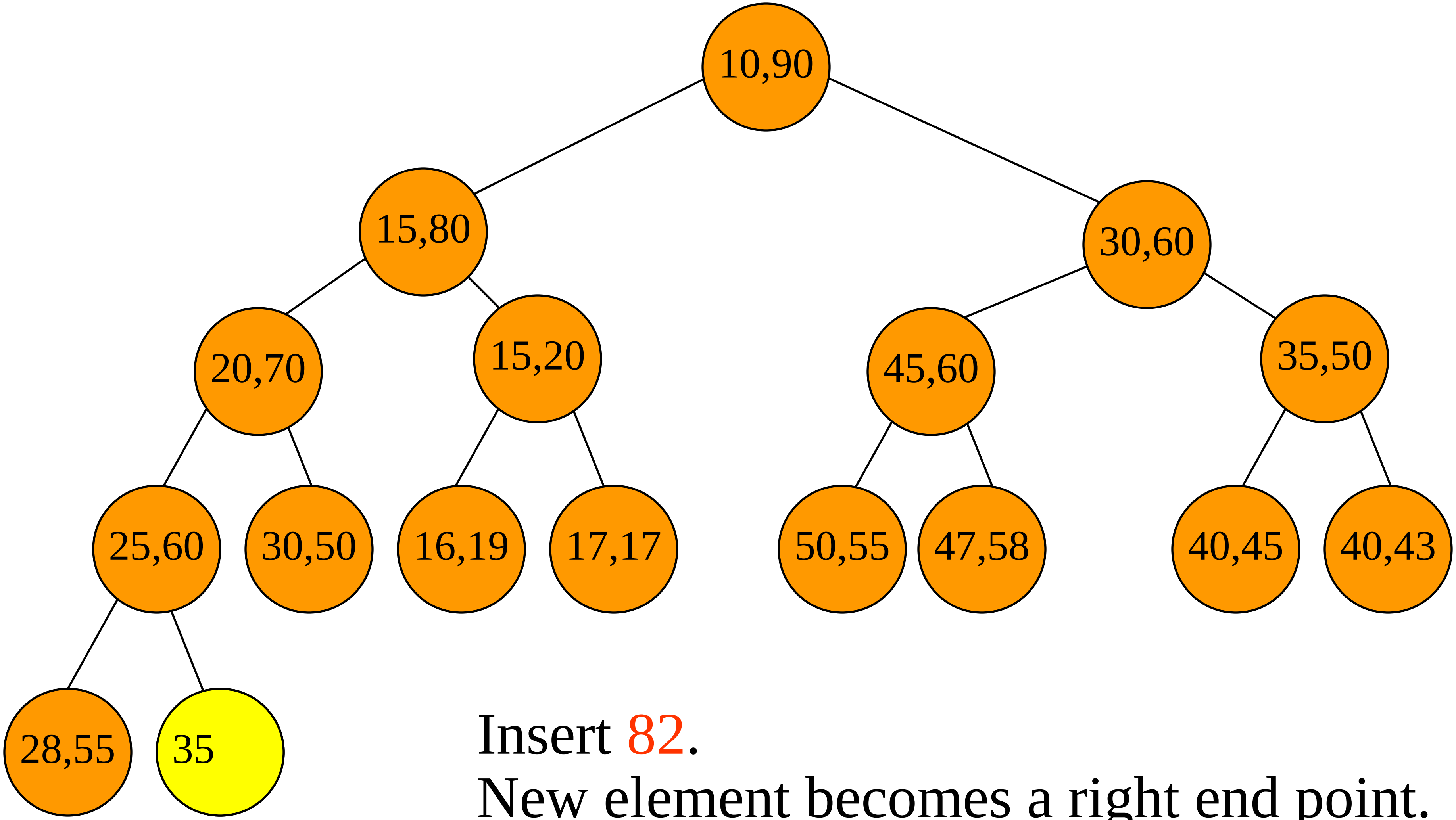


Insert 18.

New element becomes a left end point.

Insert new element into min heap.

Yet Another Insert

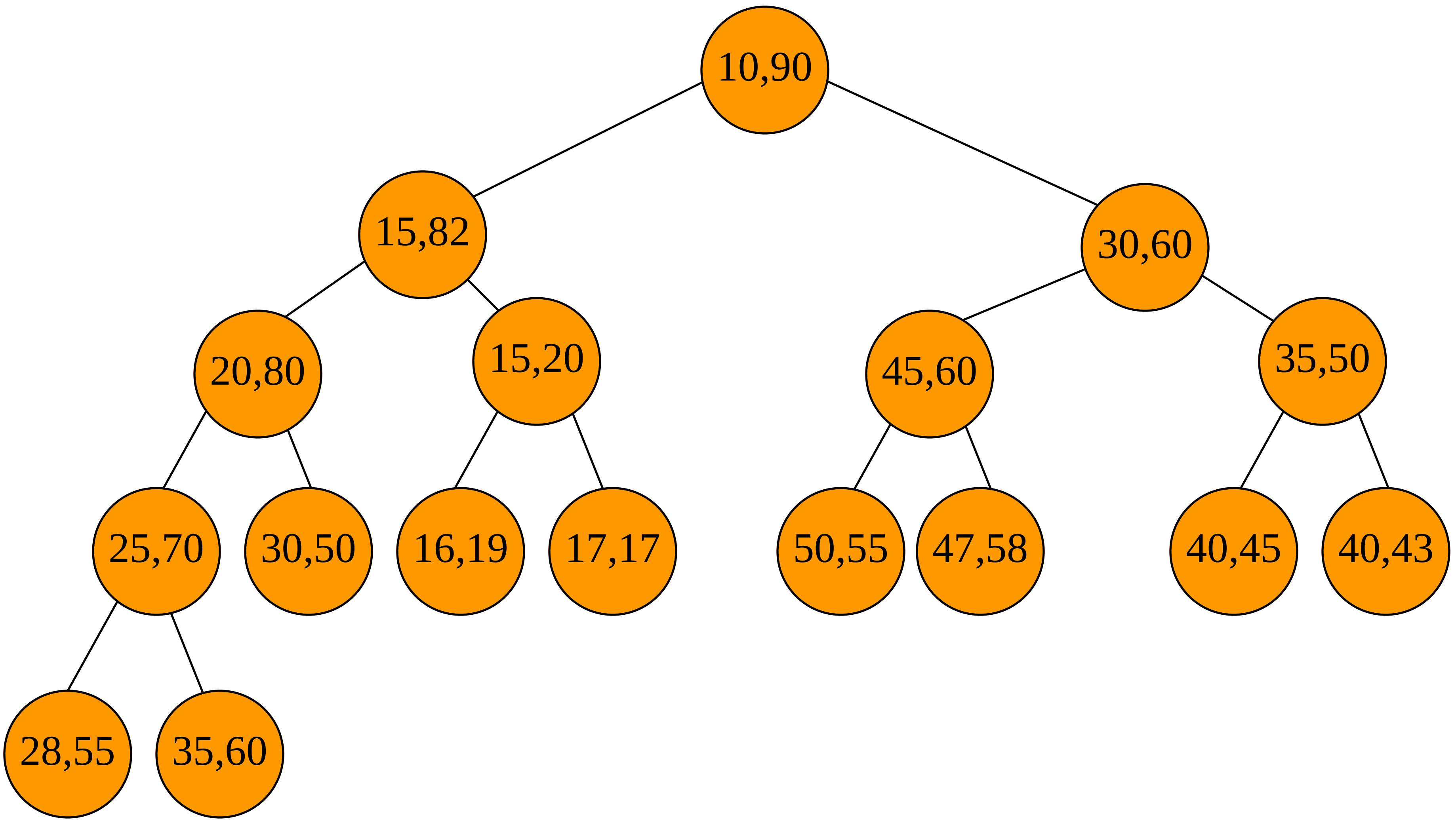


Insert 82.

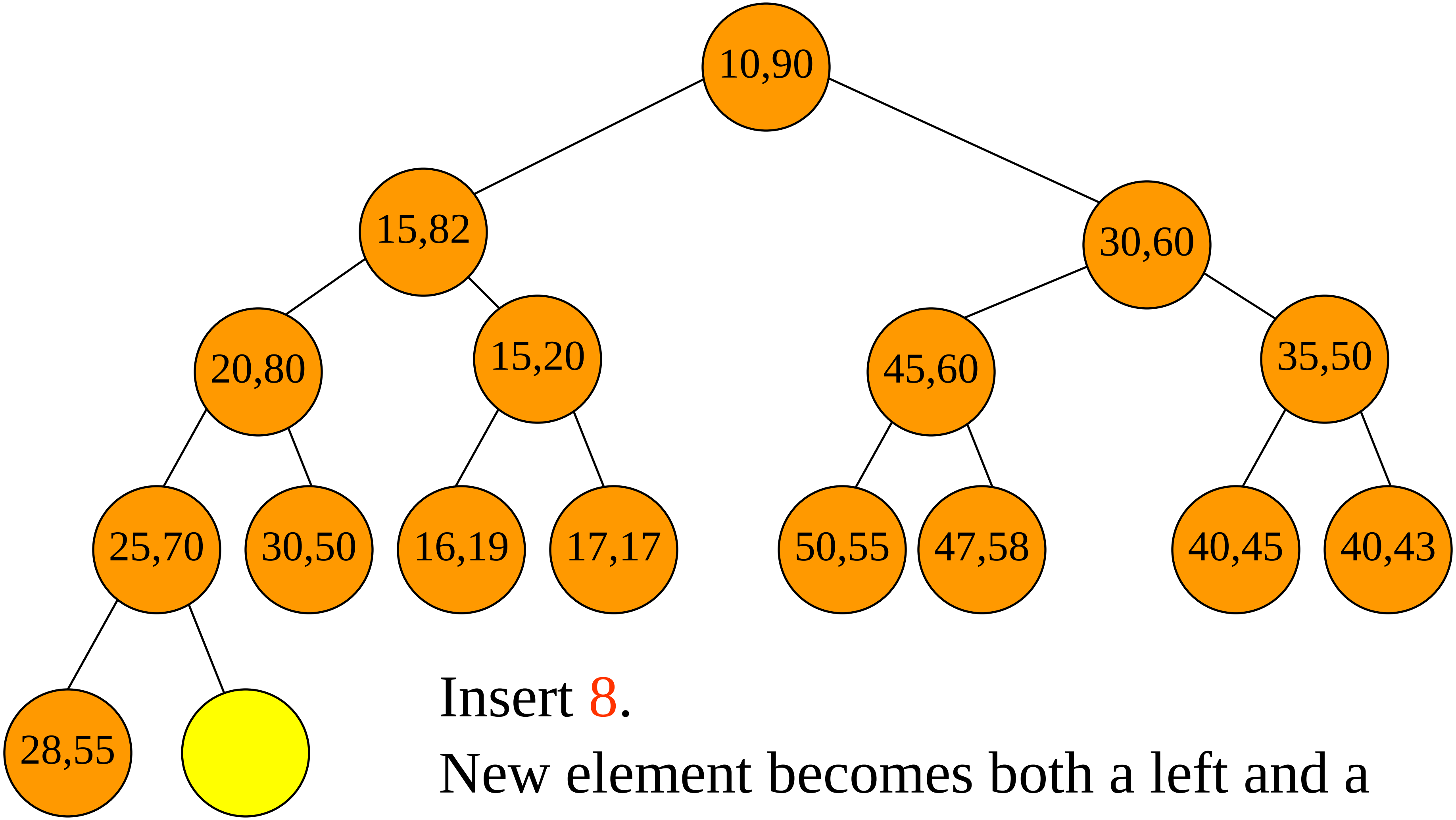
New element becomes a right end point.

Insert new element into max heap.

After 82 Inserted



One More Insert Example

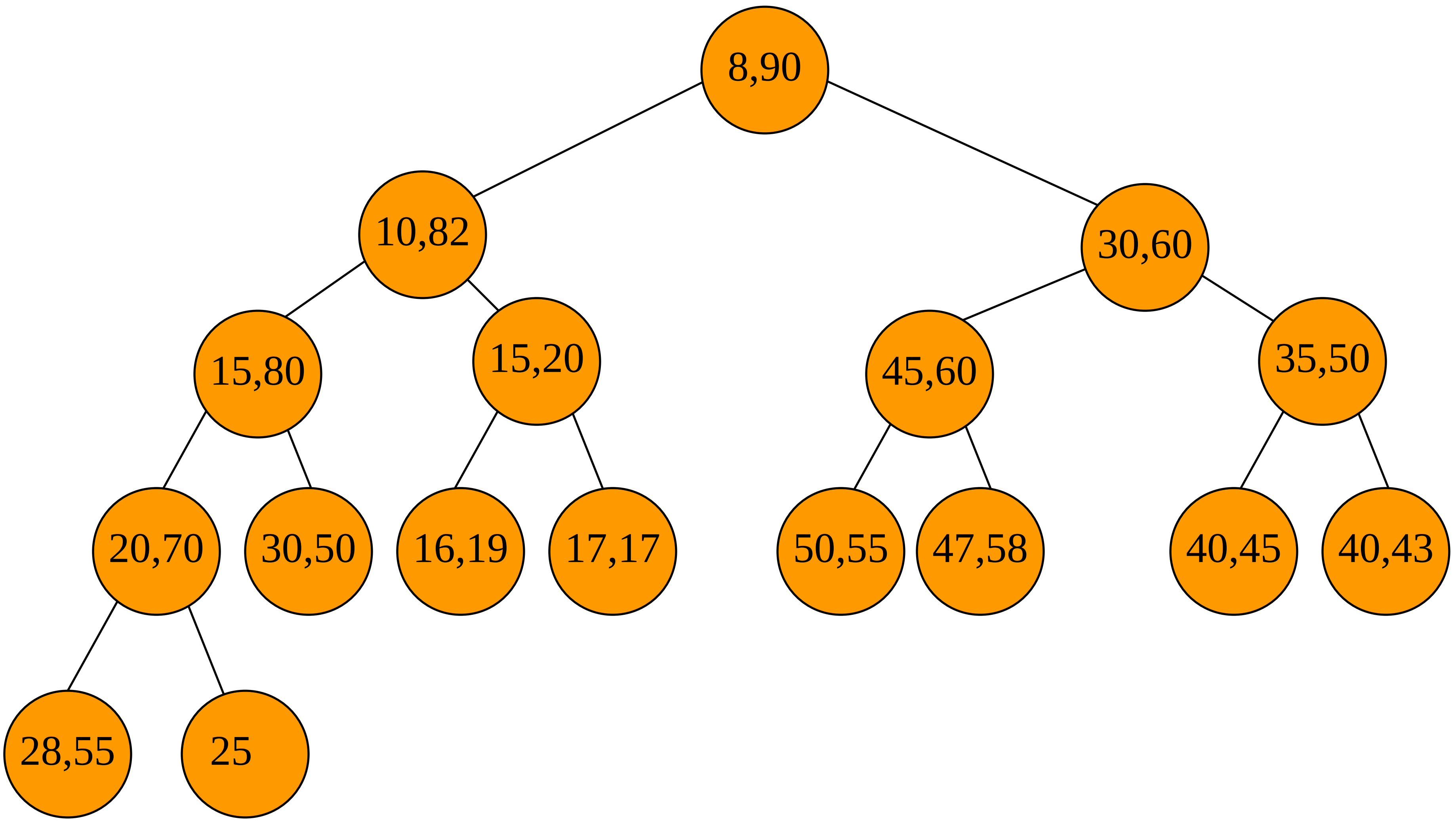


Insert 8.

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

Insert new element into min heap.

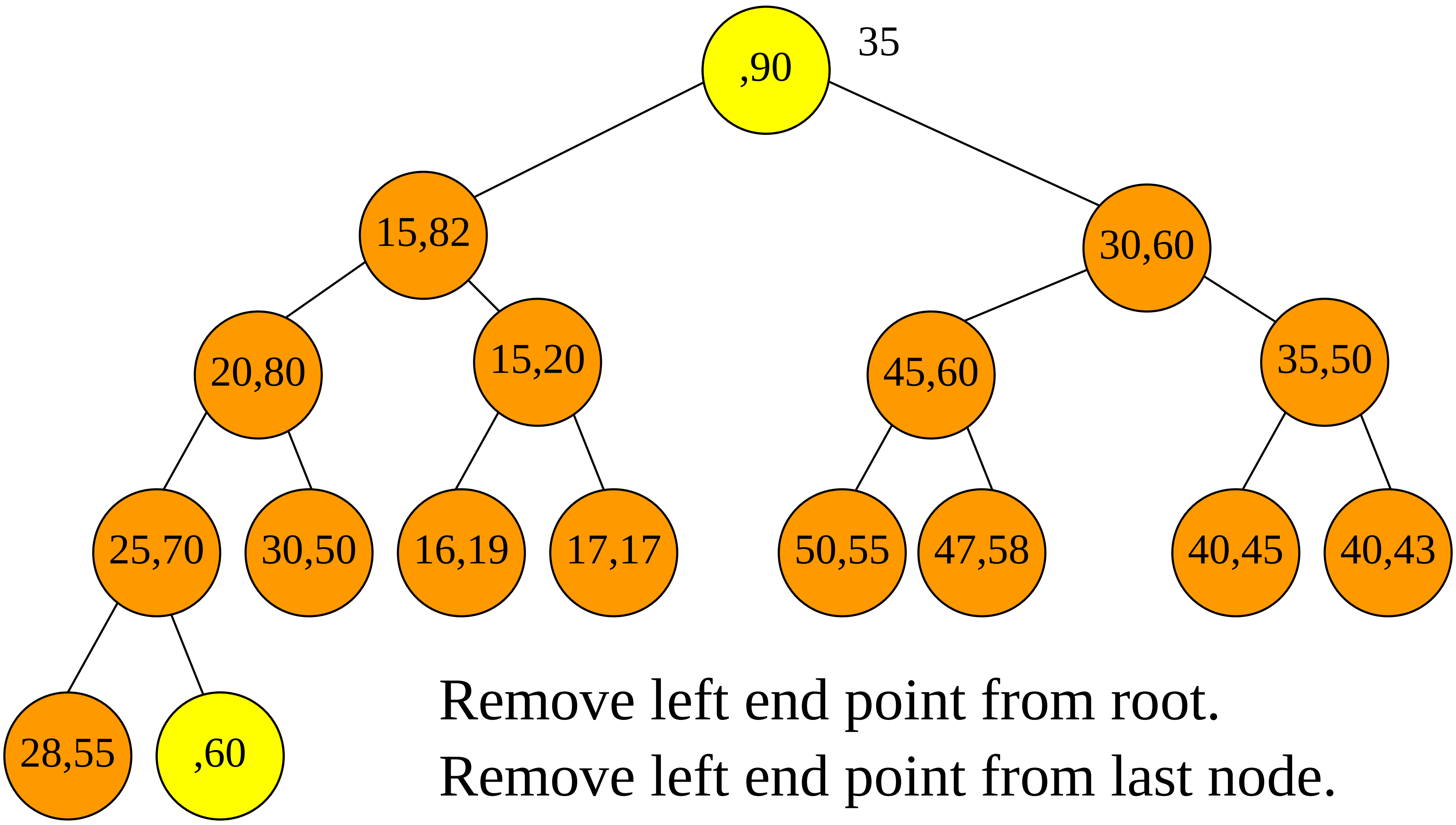
After 8 Is Inserted



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



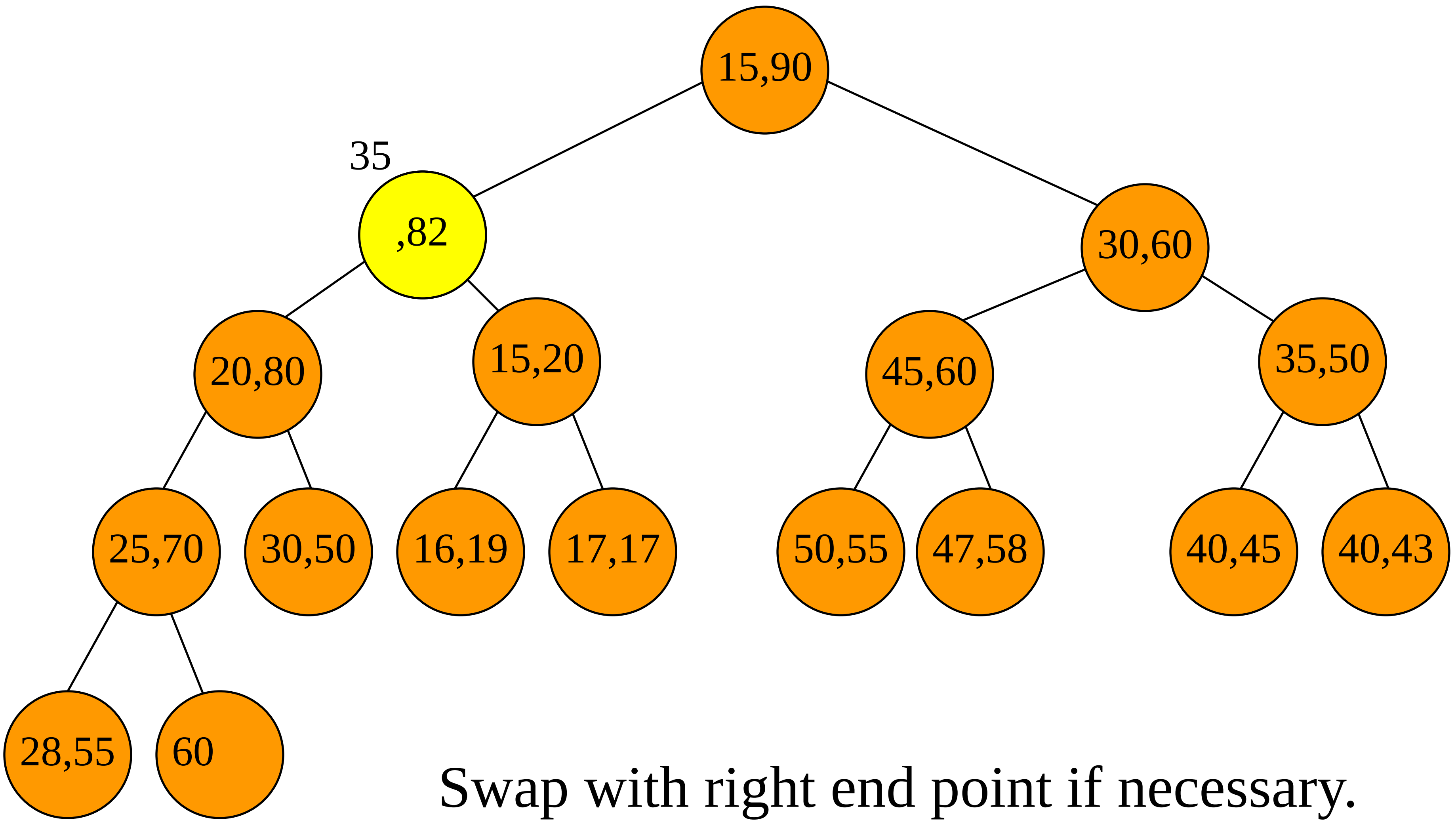
Remove left end point from root.

Remove left end point from last node.

Delete last node if now empty.

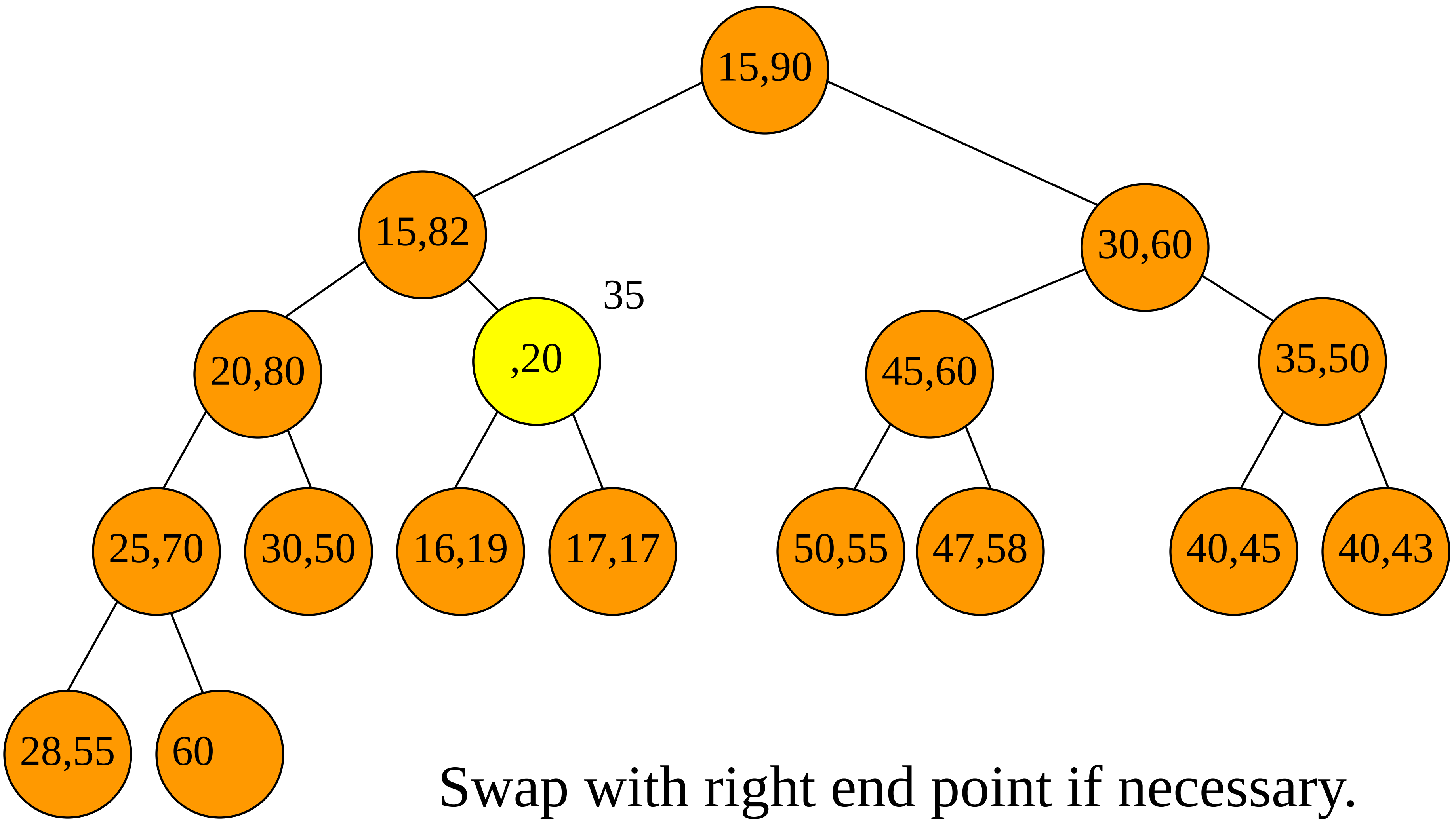
Reinsert into min heap, begin at root.

Remove Min Element Example



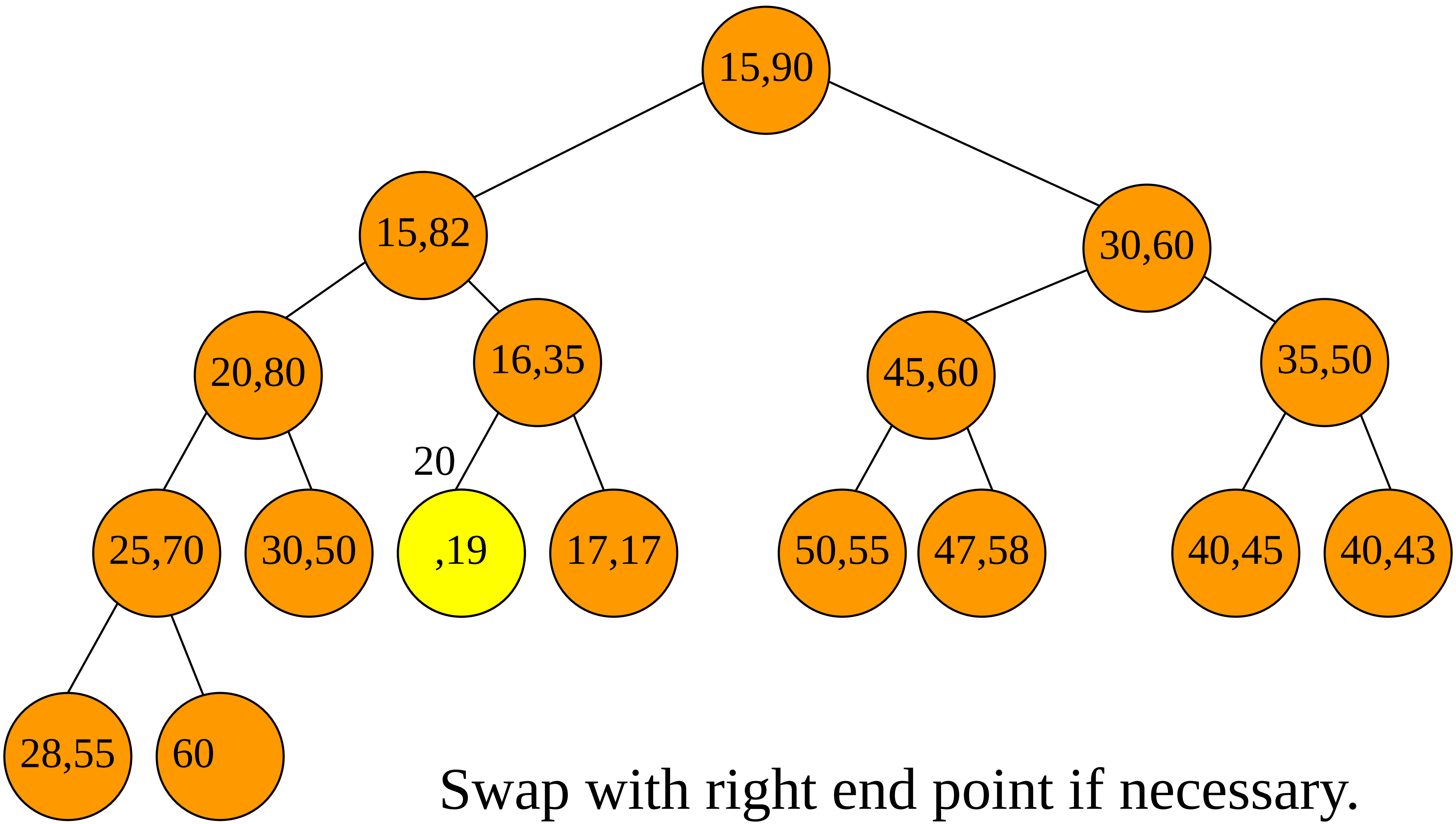
Swap with right end point if necessary.

Remove Min Element Example



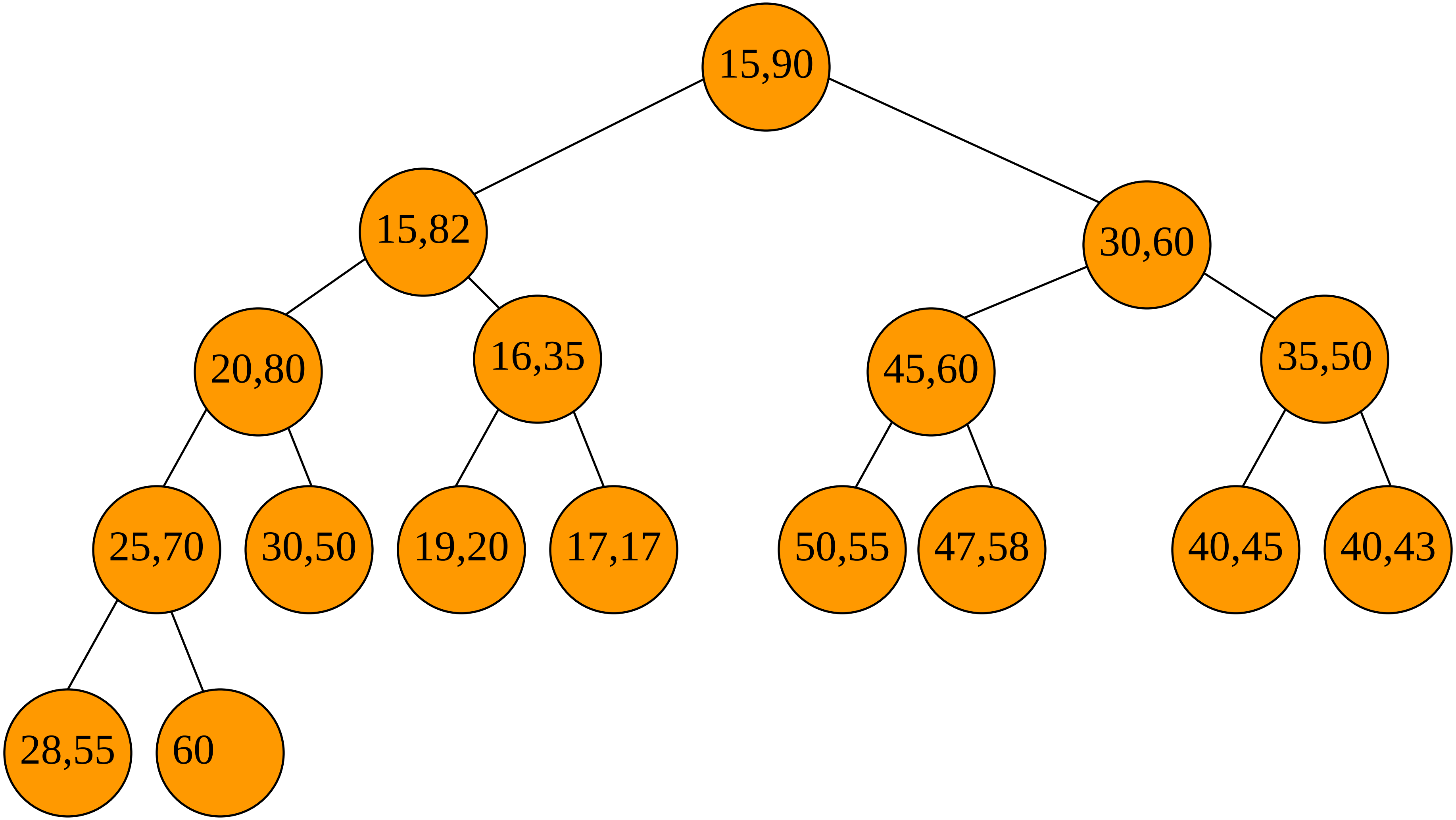
Swap with right end point if necessary.

Remove Min Element Example

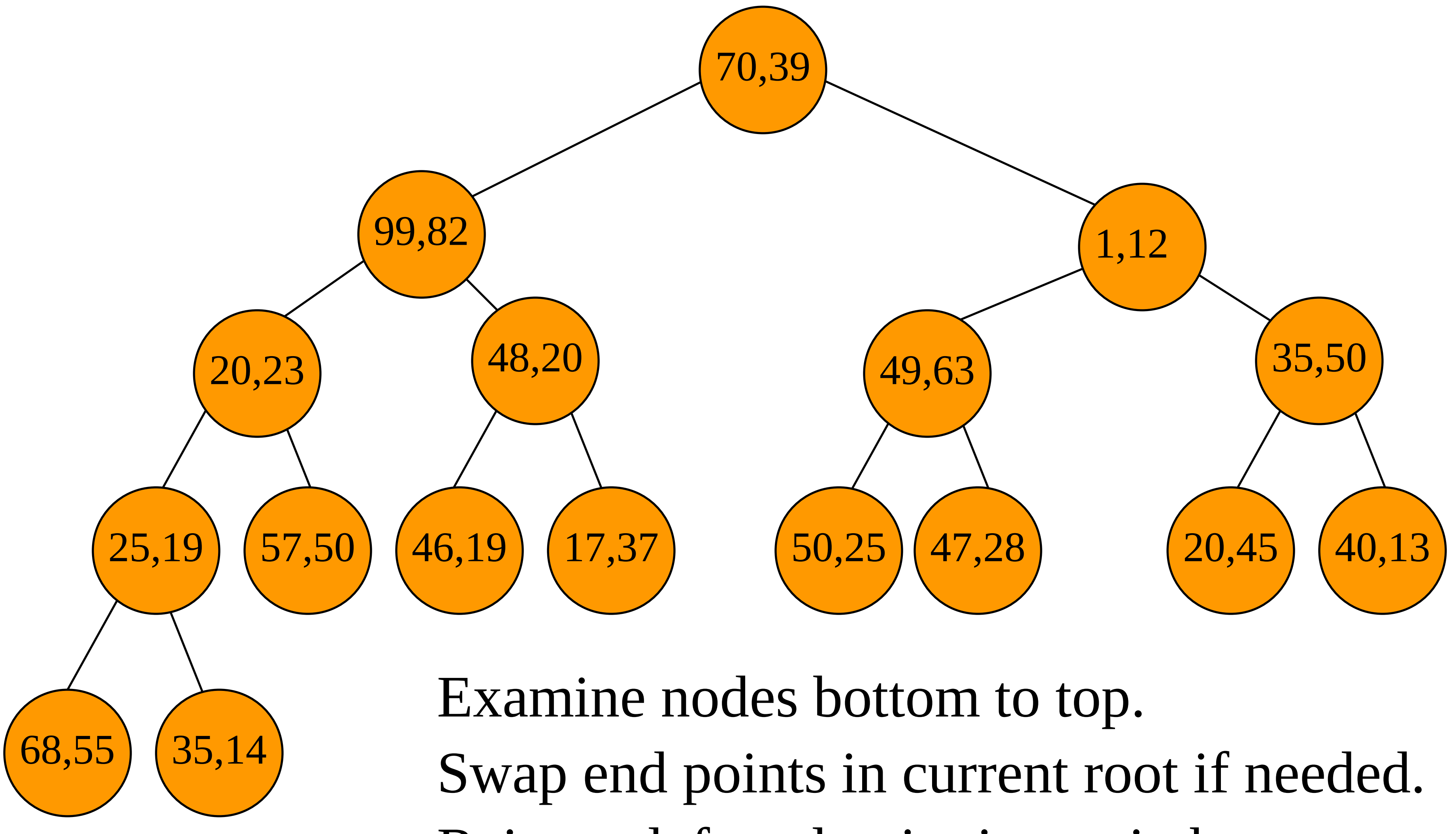


Swap with right end point if necessary.

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.

Hints

- You can use Min-max heap or Deap to implement Double-ended Priority Queue
- Count disk I/Os (File reads/writes)

Phase 3

External Sort: Merge Sort

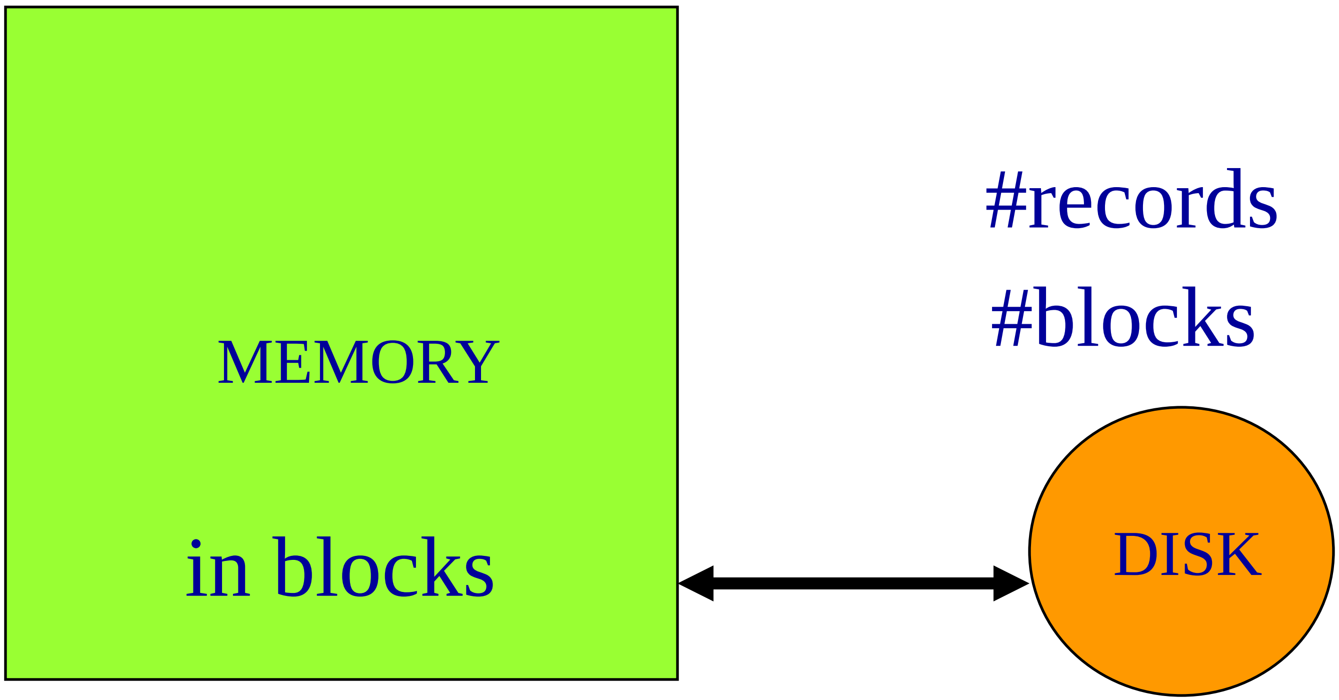
Internal Merge Sort Review

- 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

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

Run Generation

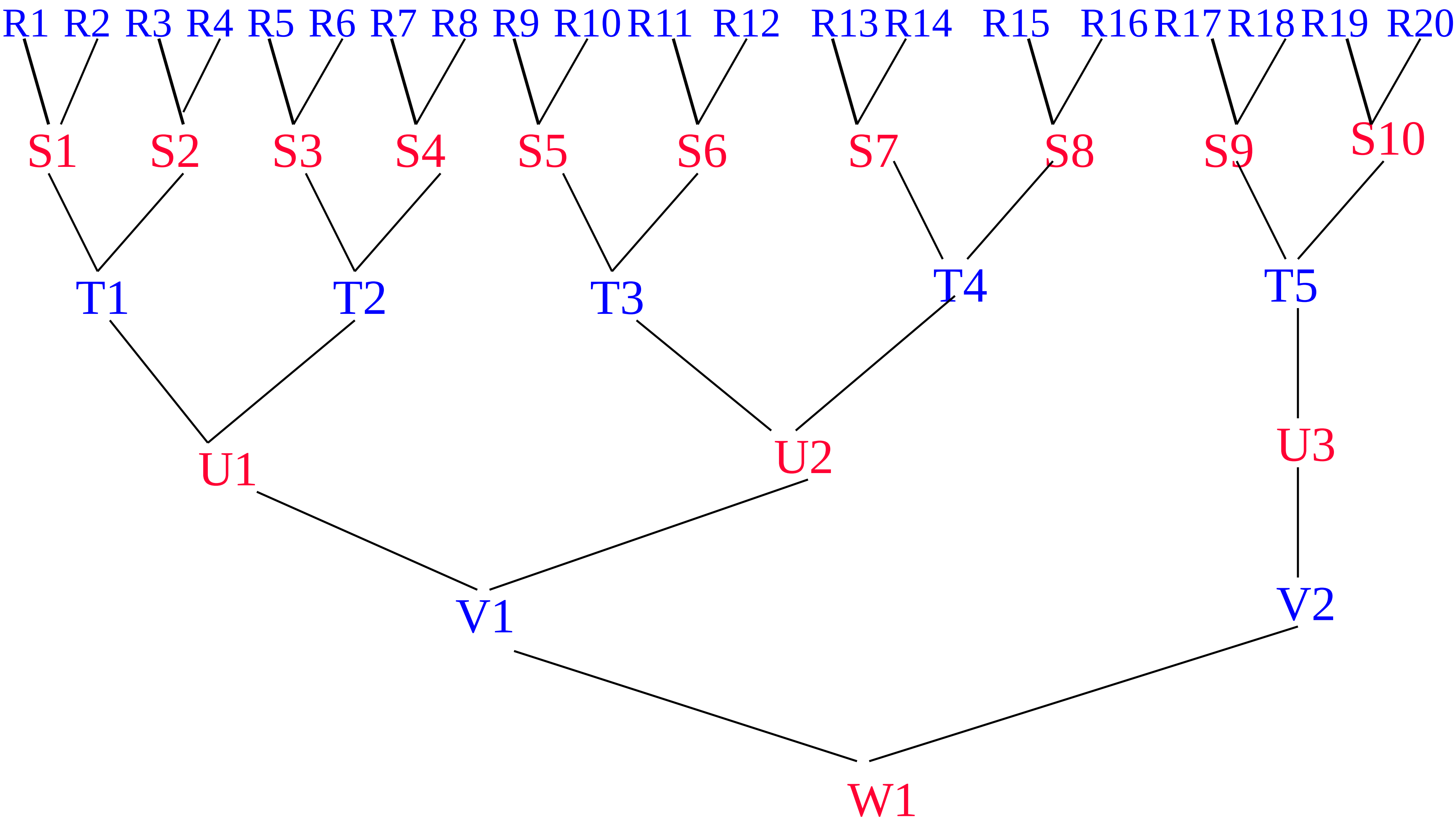


- Input blocks.
- Sort.
- Output as a run.

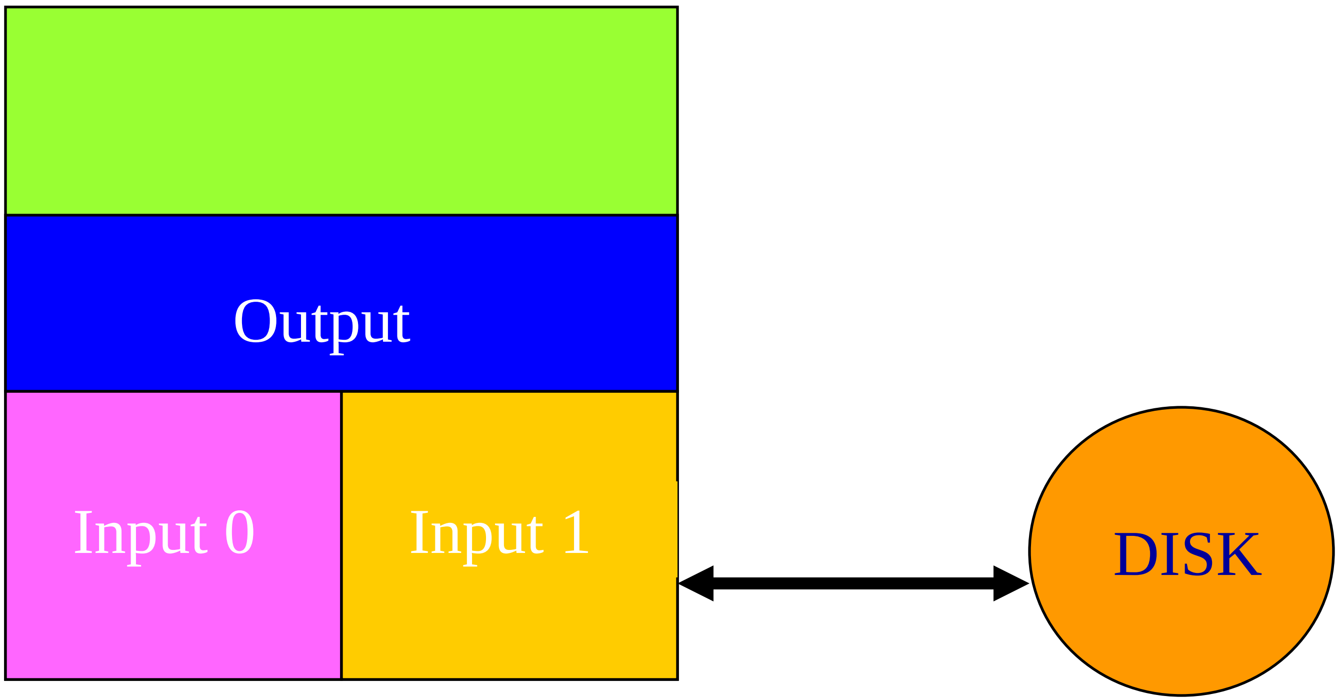
Run Merging

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

Merge 20 Runs



Merge R1 and R2



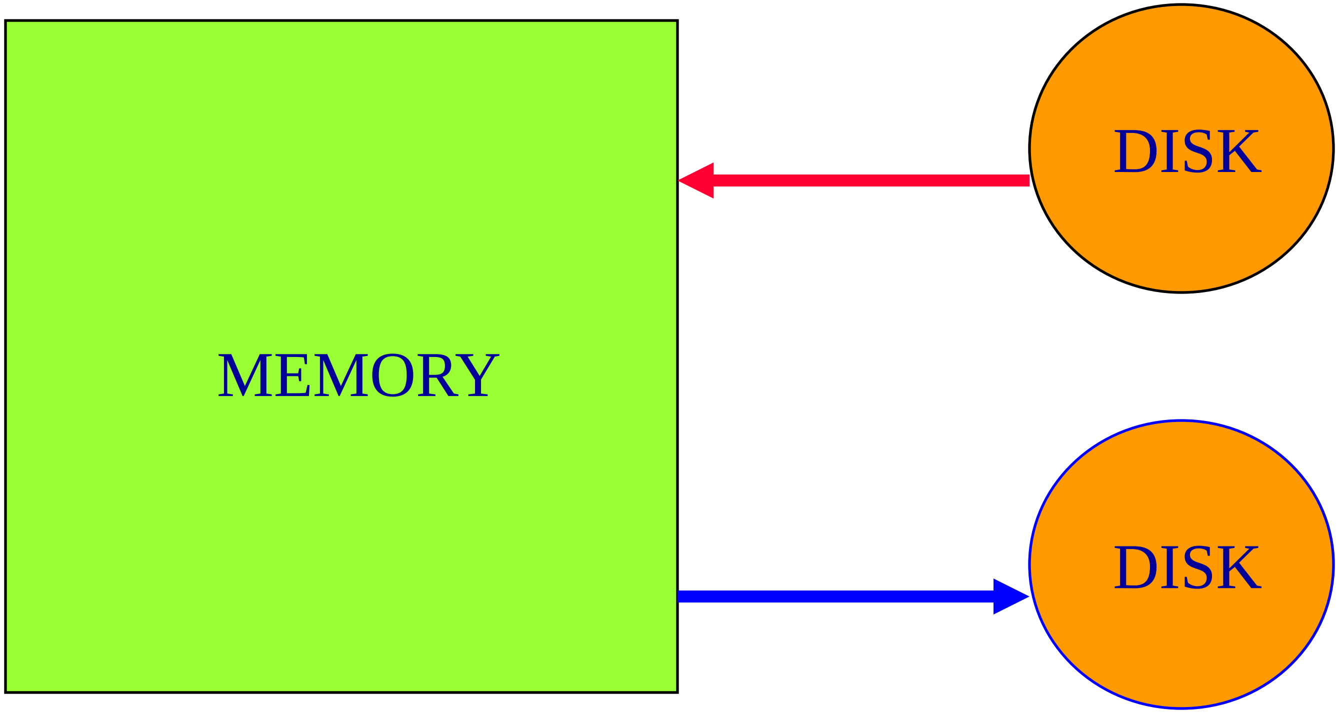
- Fill I0 (Input 0) from R1 and I1 from R2.
- Merge from I0 and I1 to output buffer.
- Write whenever output buffer full.
- Read whenever input buffer empty.

Phase 4

Merge Sort: Improve Run Generation

Improve Run Generation

- Overlap input, output, and internal sorting.

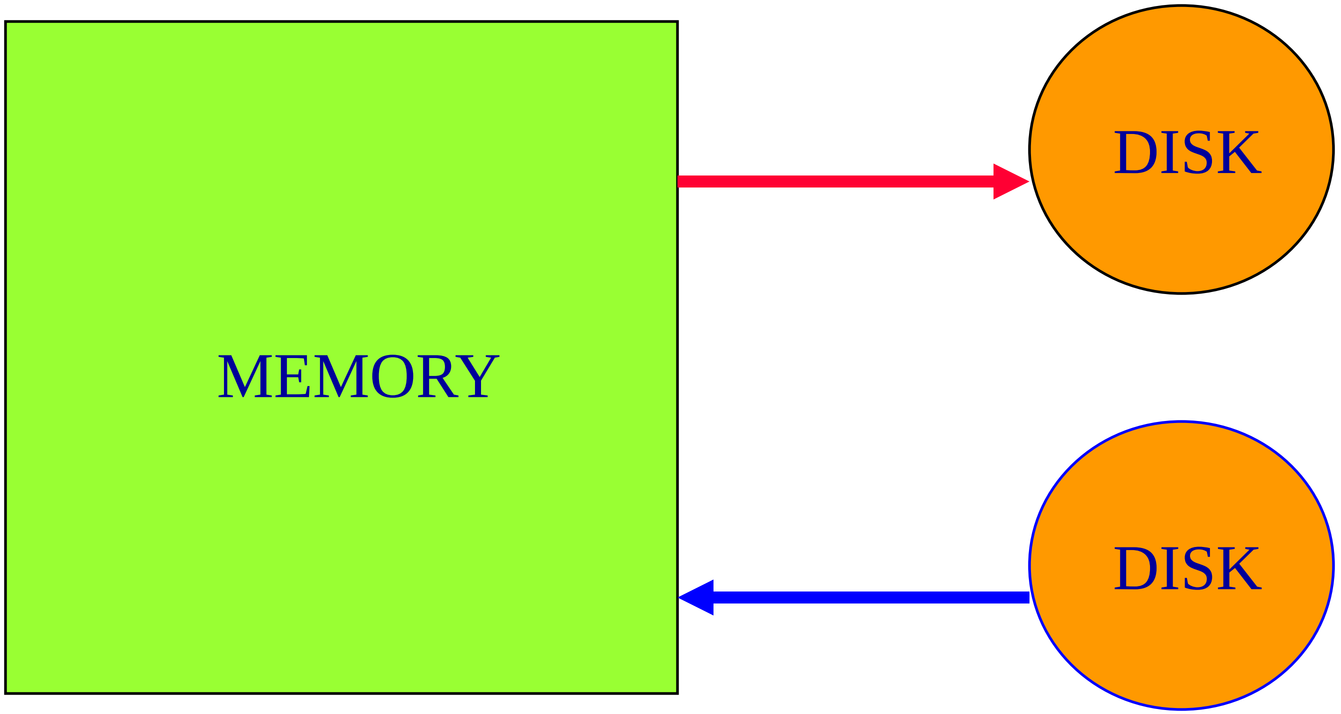


Improve Run Generation

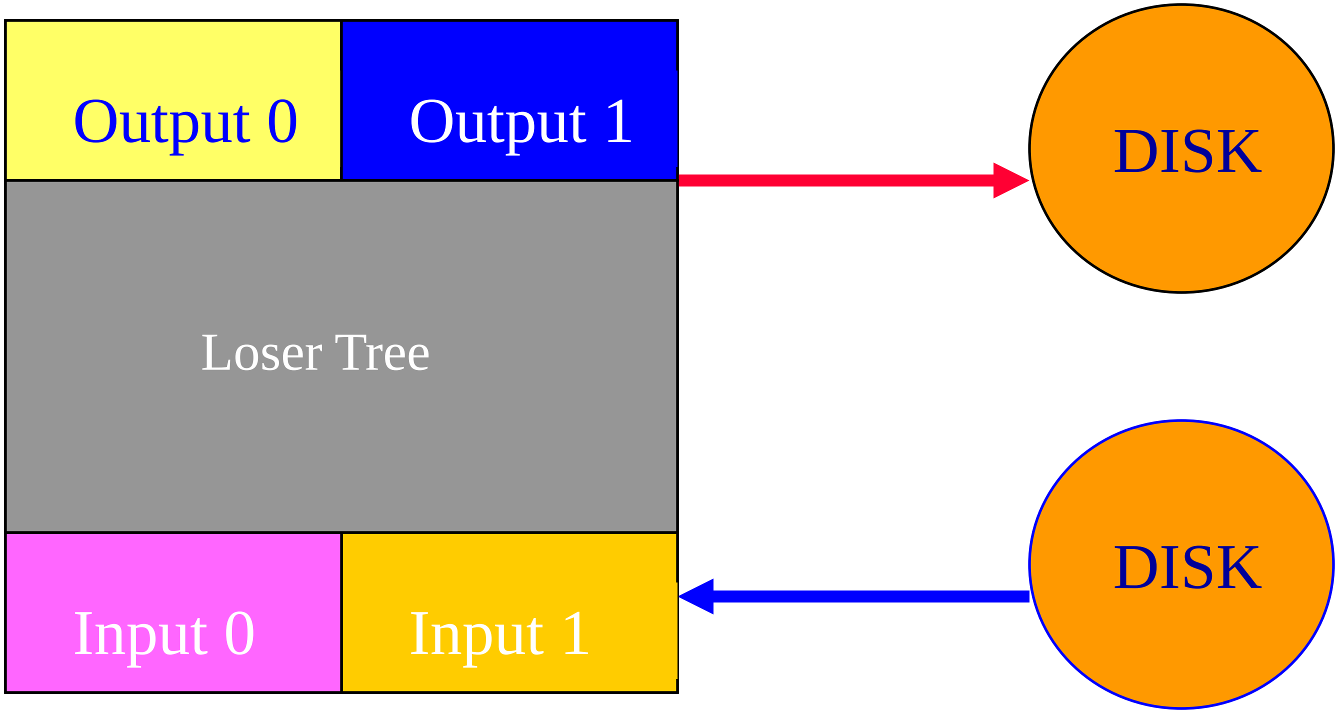
- Generate runs whose length (on average) exceeds memory size.
- Equivalent to reducing number of runs generated.

Improve Run Generation

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

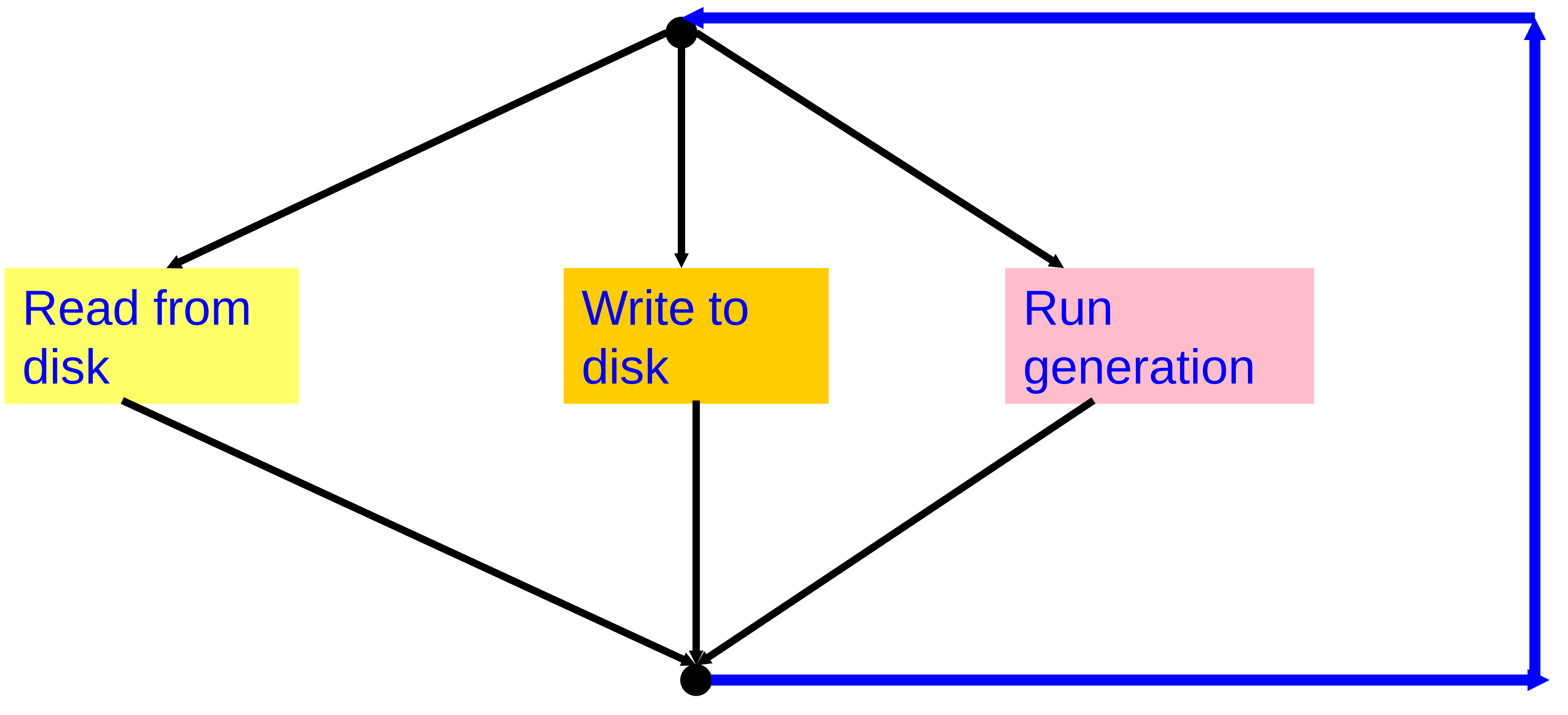


New Strategy



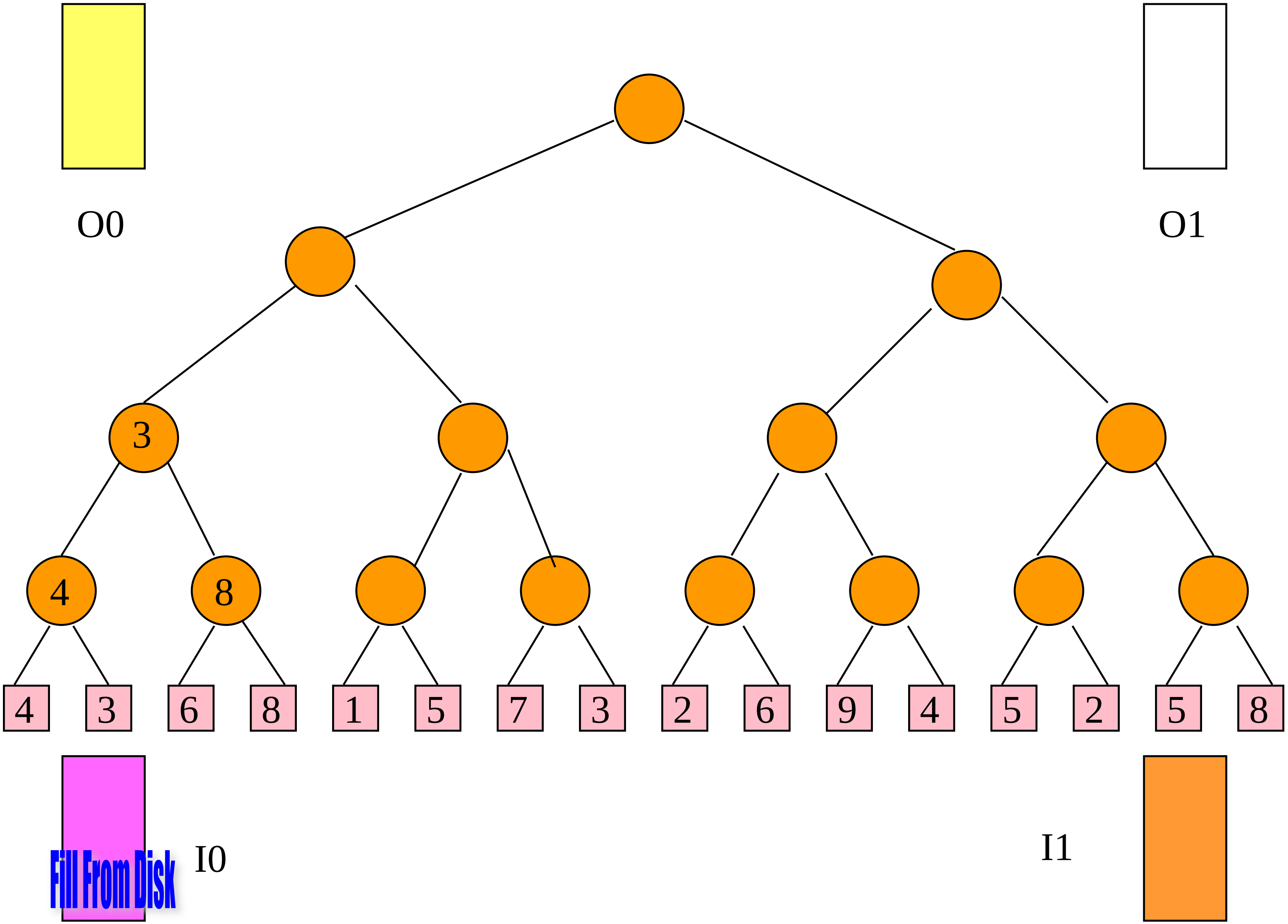
- Use 2 input and 2 output buffers.
- Rest of memory is used for a min loser tree.
- Actually, 3 buffers adequate.

Steady State Operation

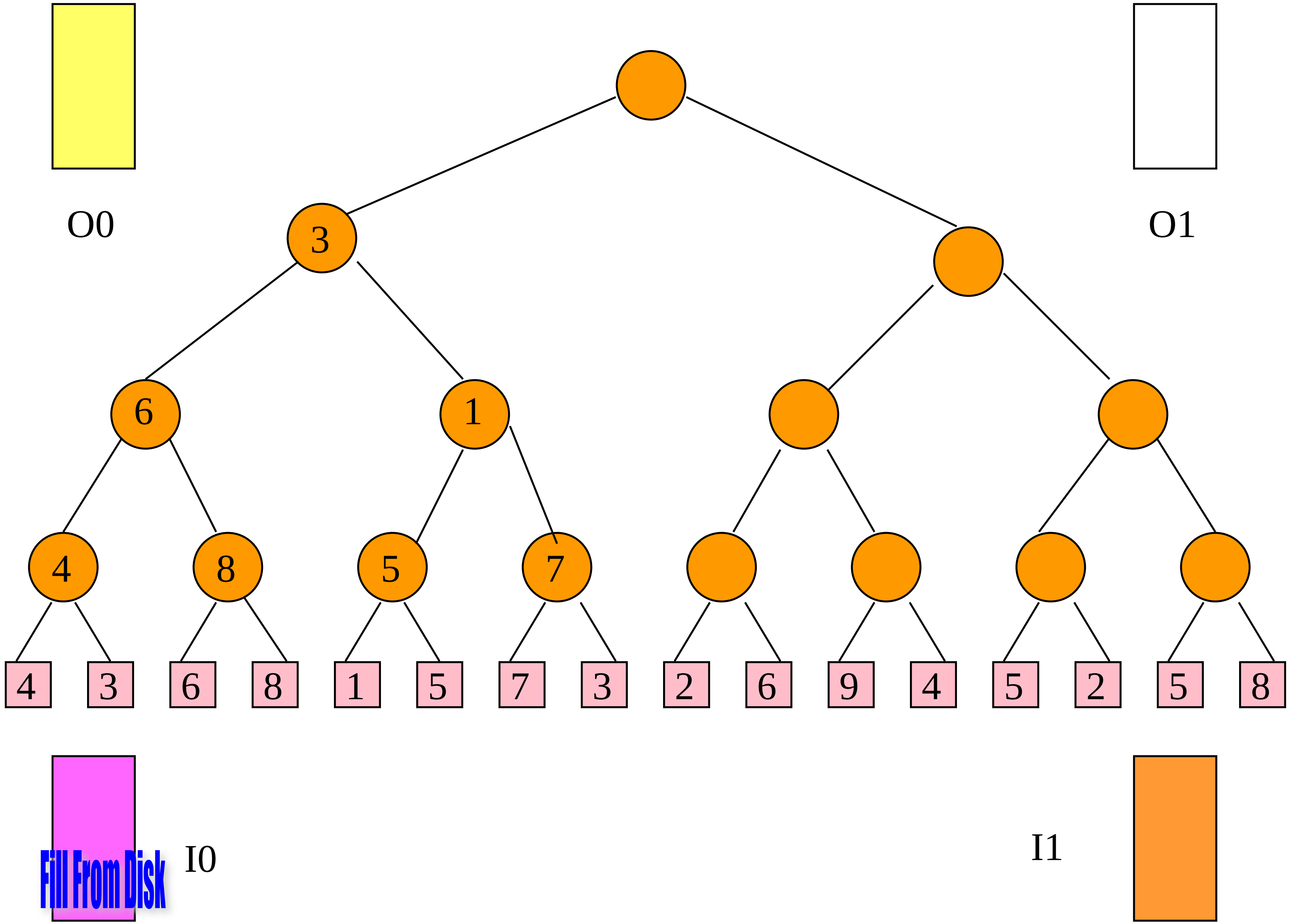


- Synchronization is done when the active input buffer gets empty (the active output buffer will be full at this time).

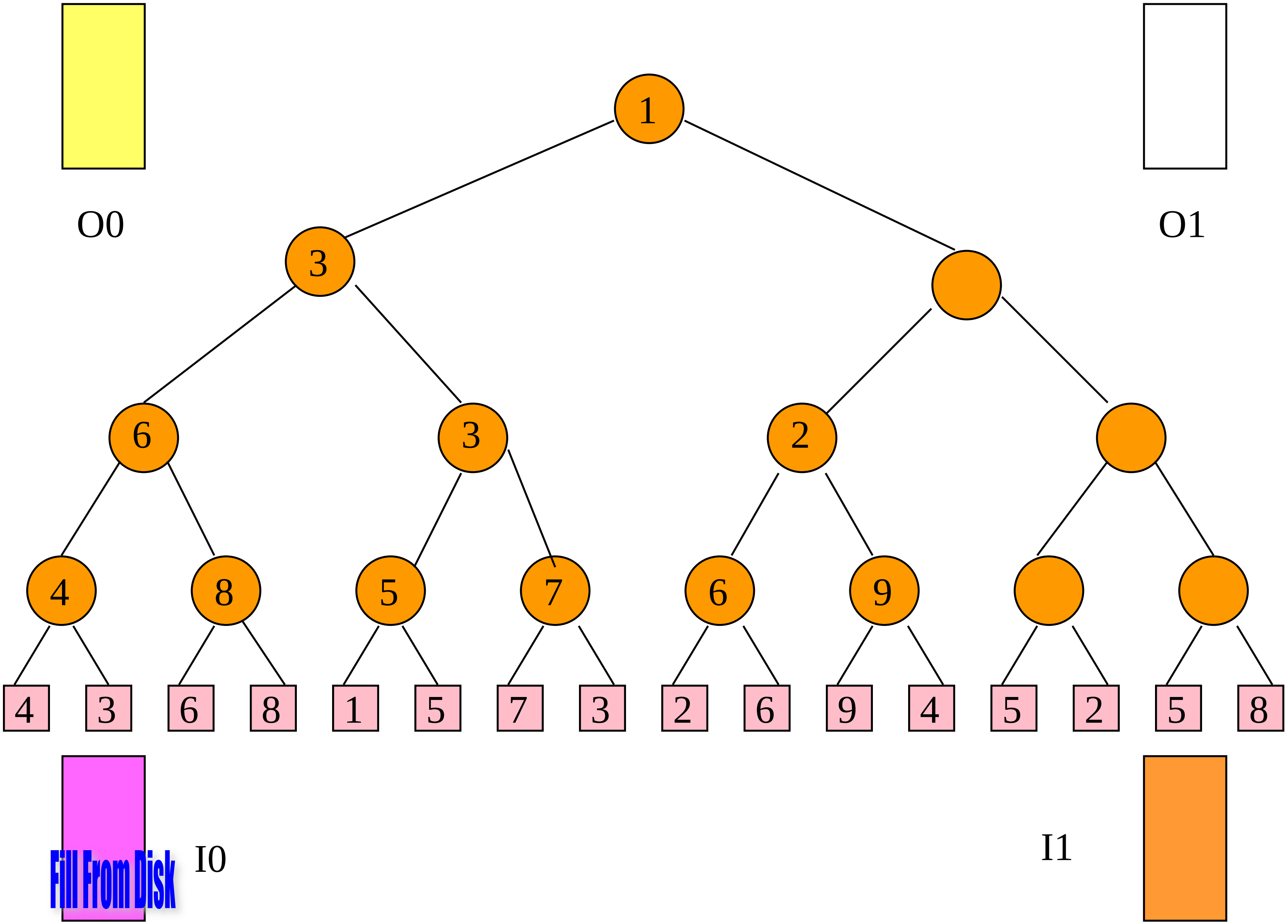
Initialize



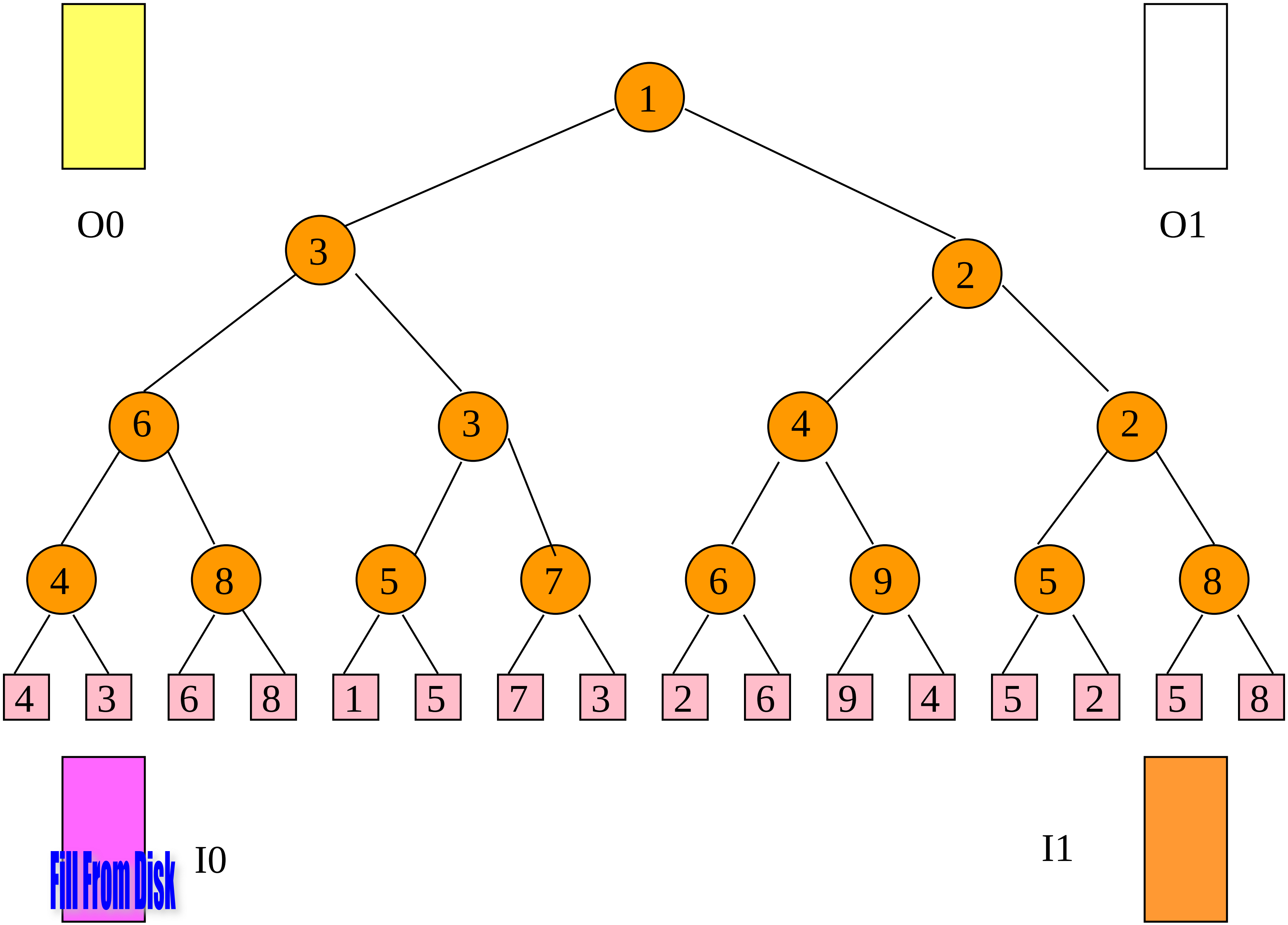
Initialize



Initialize

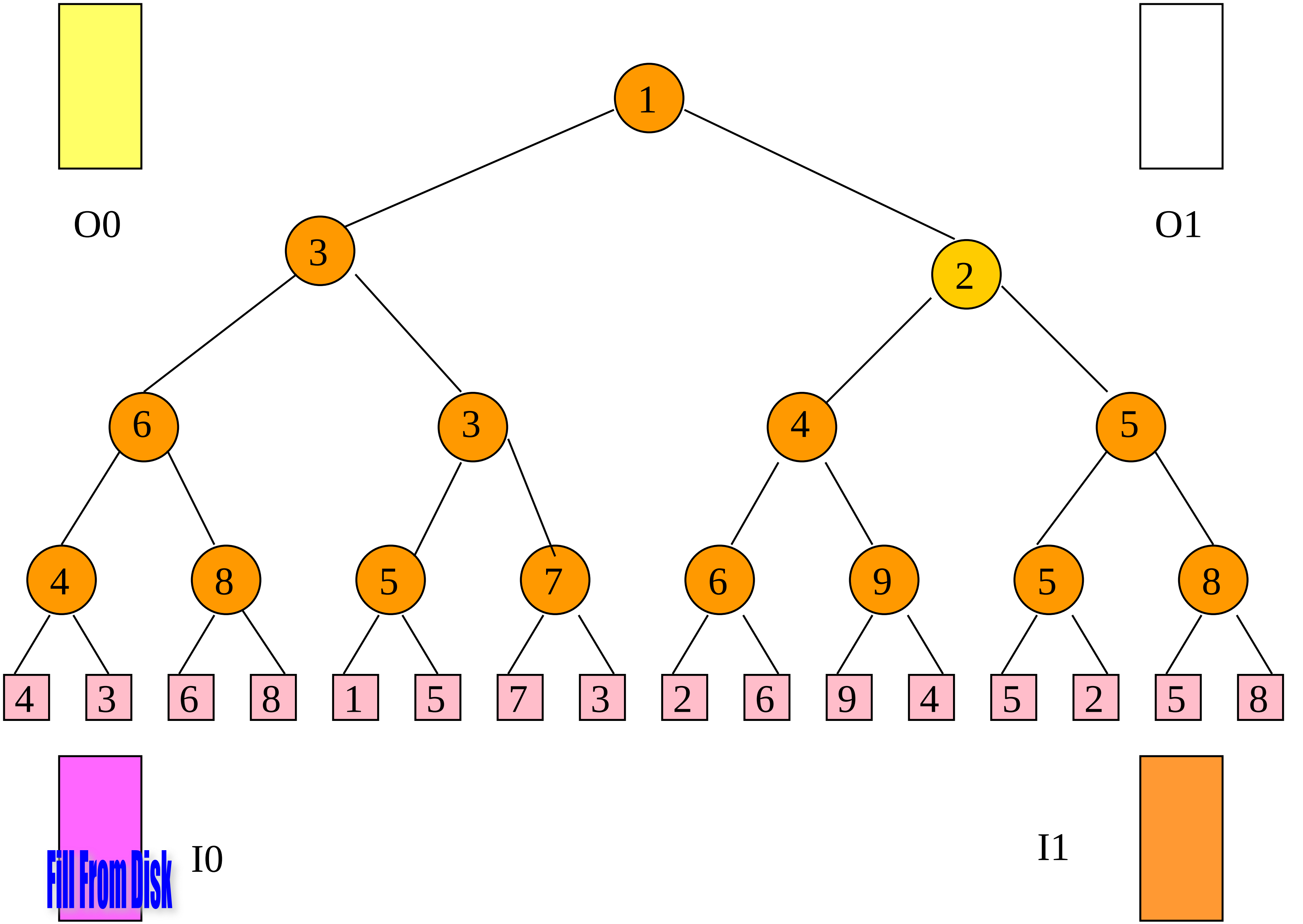


Initialize



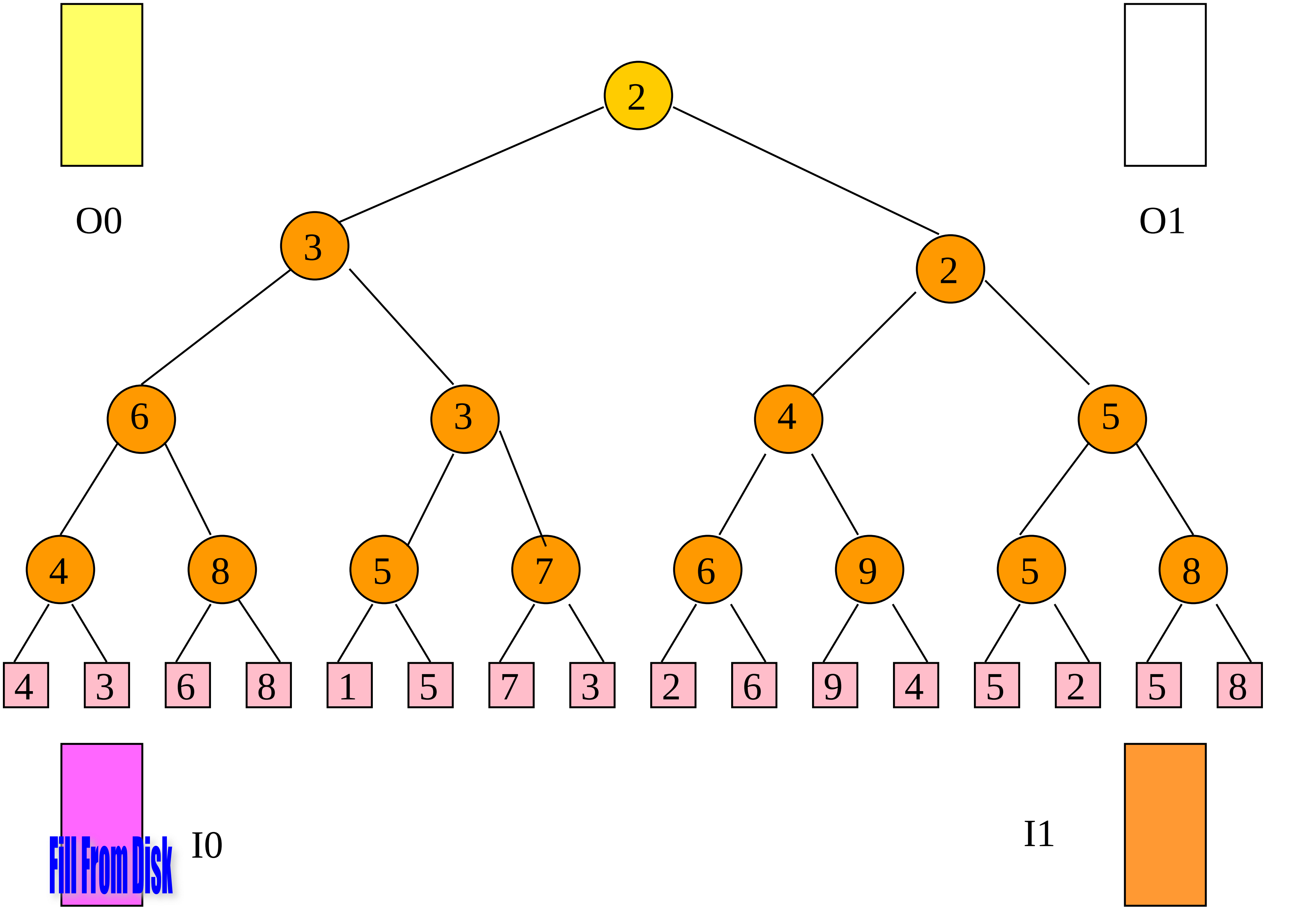
O0
O1
1
3
2
6
3
4
2
4
8
5
7
6
9
5
8
4 3 6 8 1 5 7 3 2 6 9 4 5 2 5 8
Fill From Disk
I0
I1

Initialize



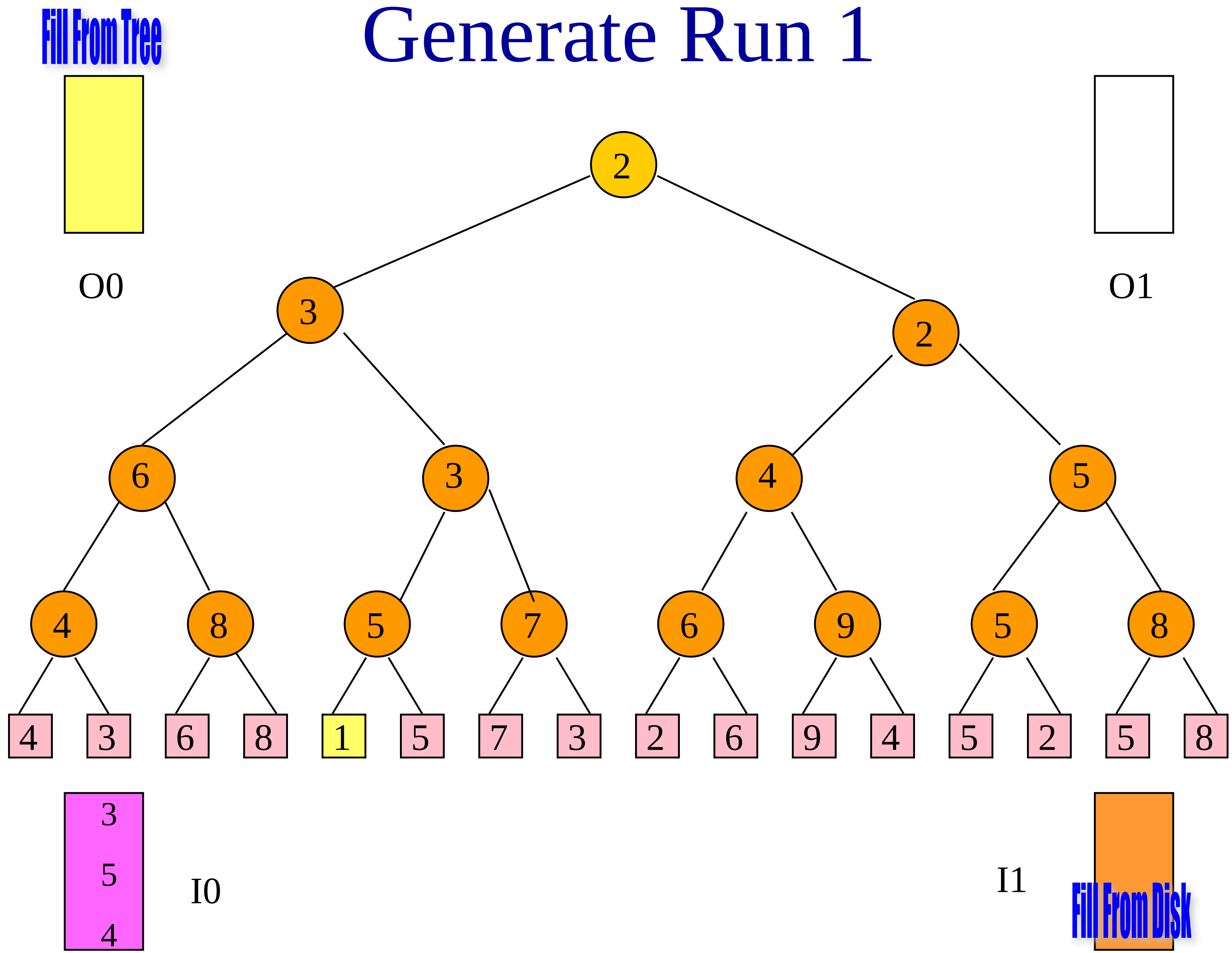
O0
O1
1
3
2
6
3
4
5
4
8
5
7
6
9
5
8
4
3
6
8
1
5
7
3
2
6
9
4
5
2
5
8
I0
Fill From Disk
I1

Initialize

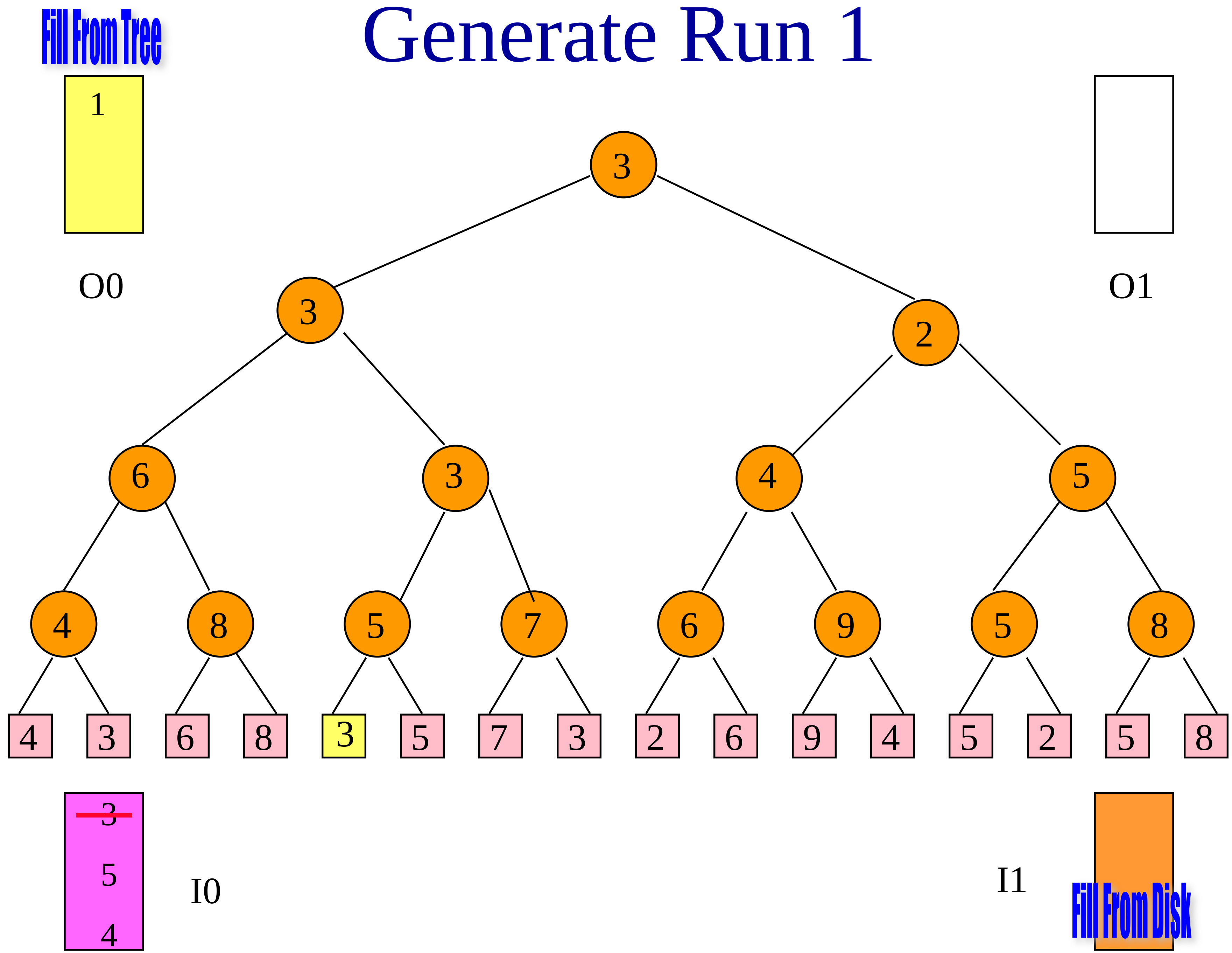


O0
O1
2
3
2
6
3
4
5
4
8
5
7
6
9
5
8
4
3
6
8
1
5
7
3
2
6
9
4
5
2
5
8
I0
Fill From Disk
I1

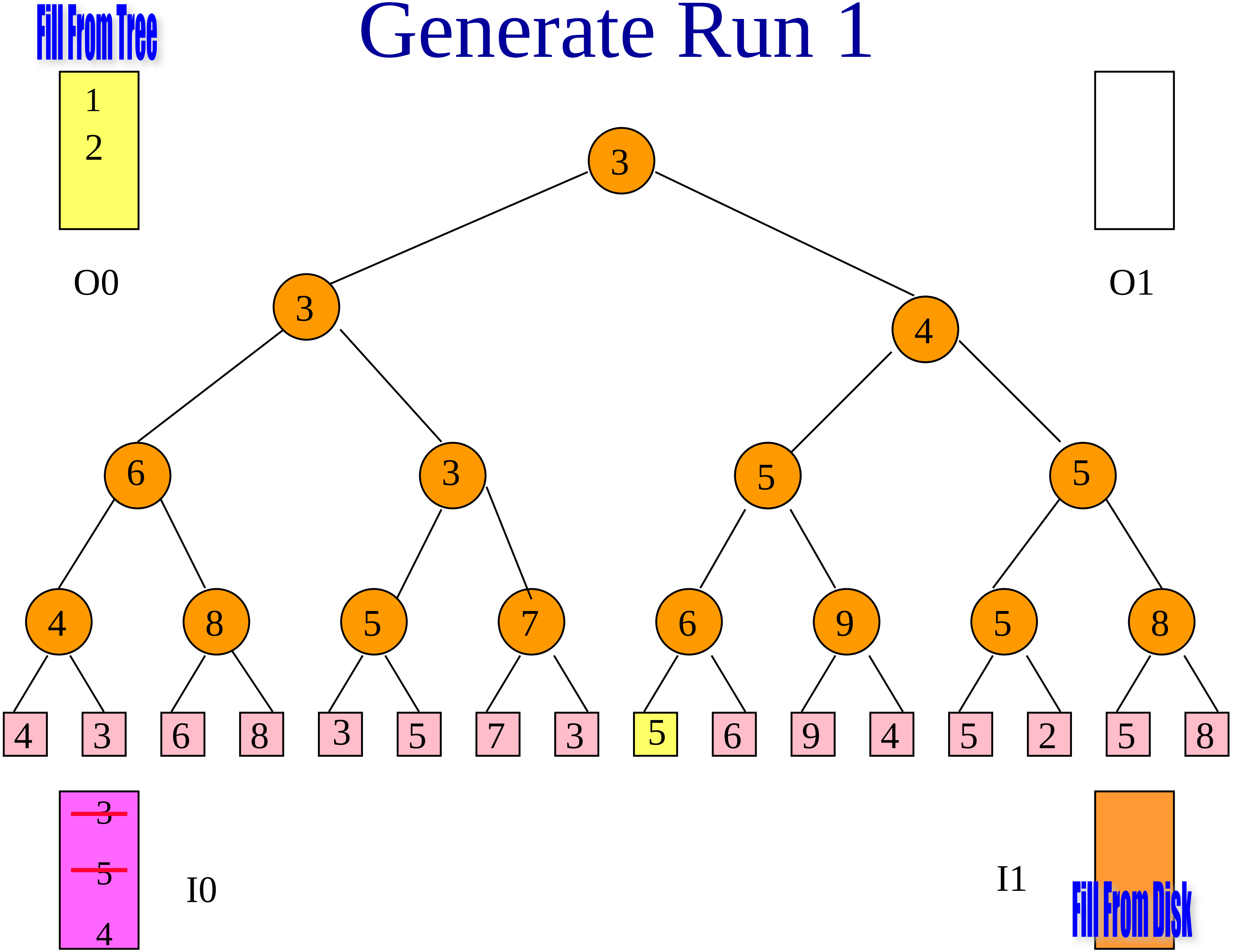
Generate Run 1



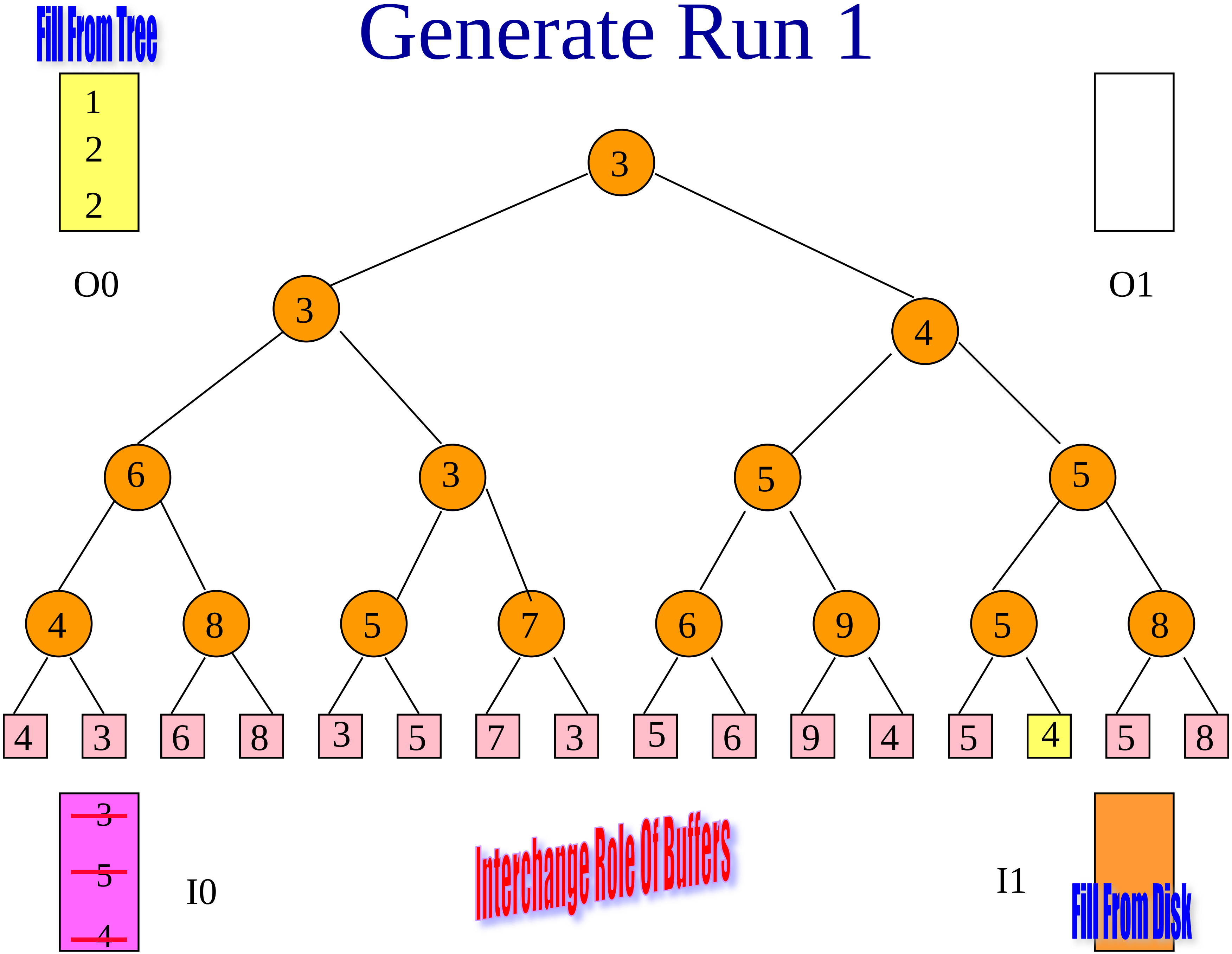
Generate Run 1



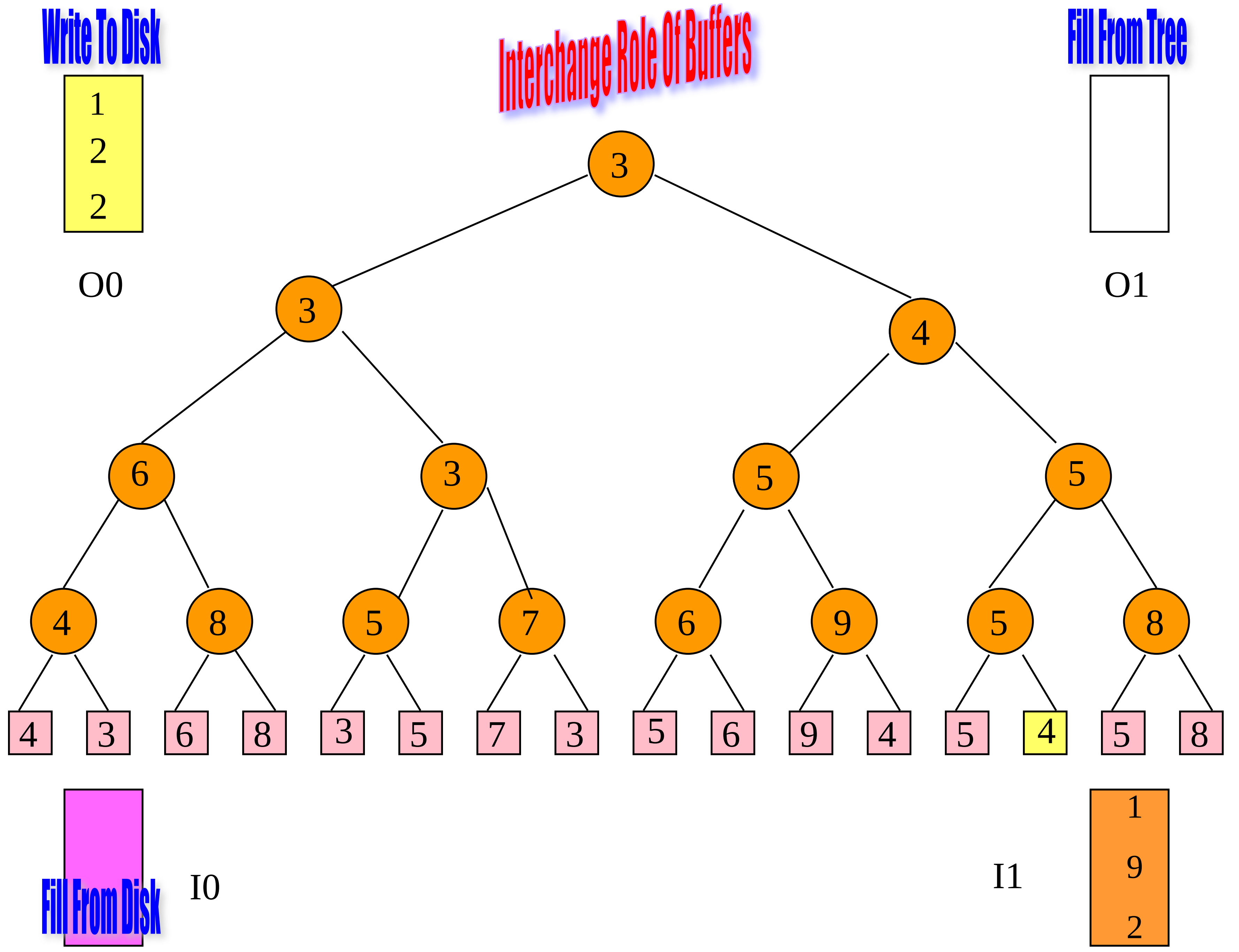
Generate Run 1



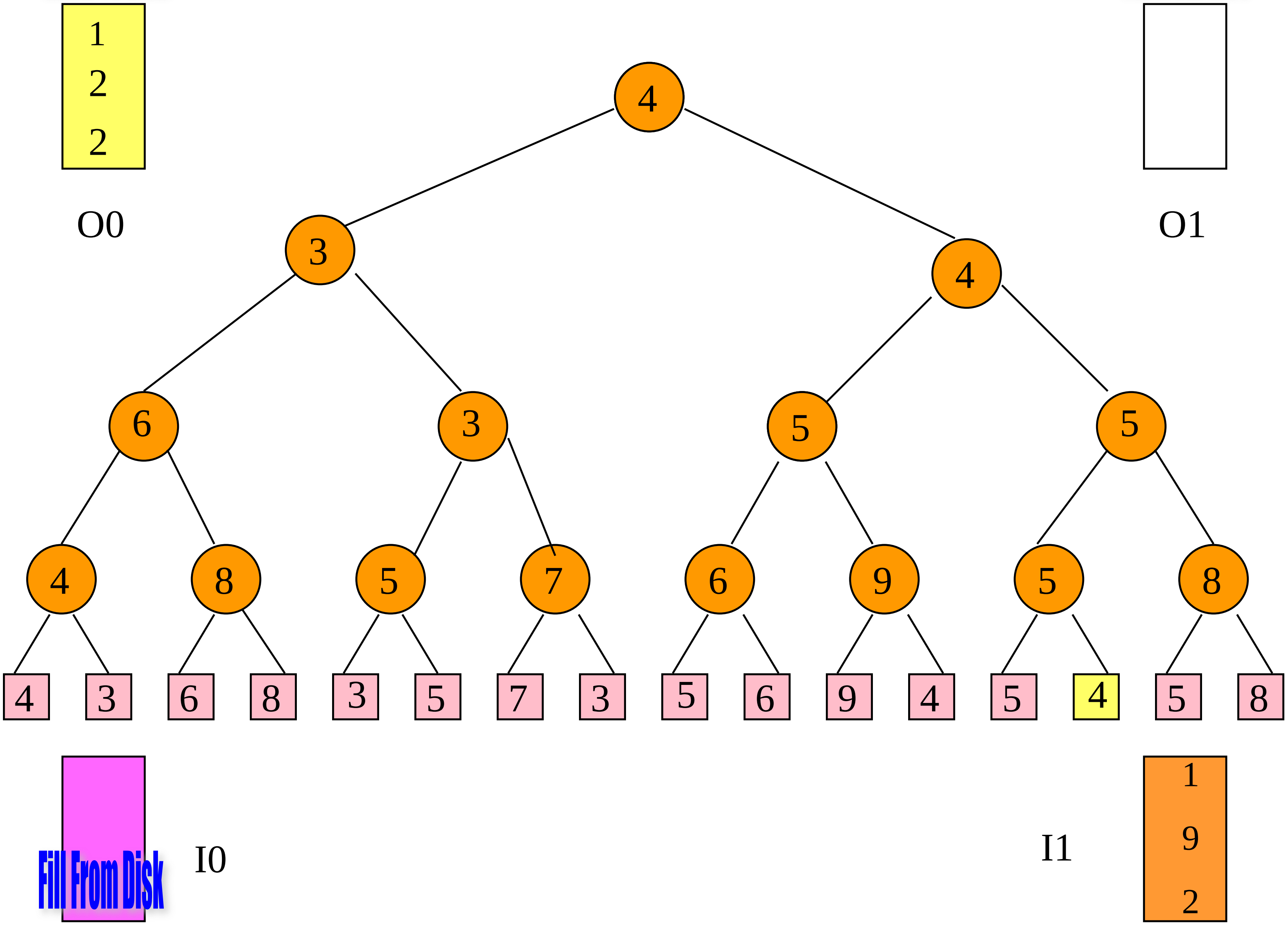
Generate Run 1



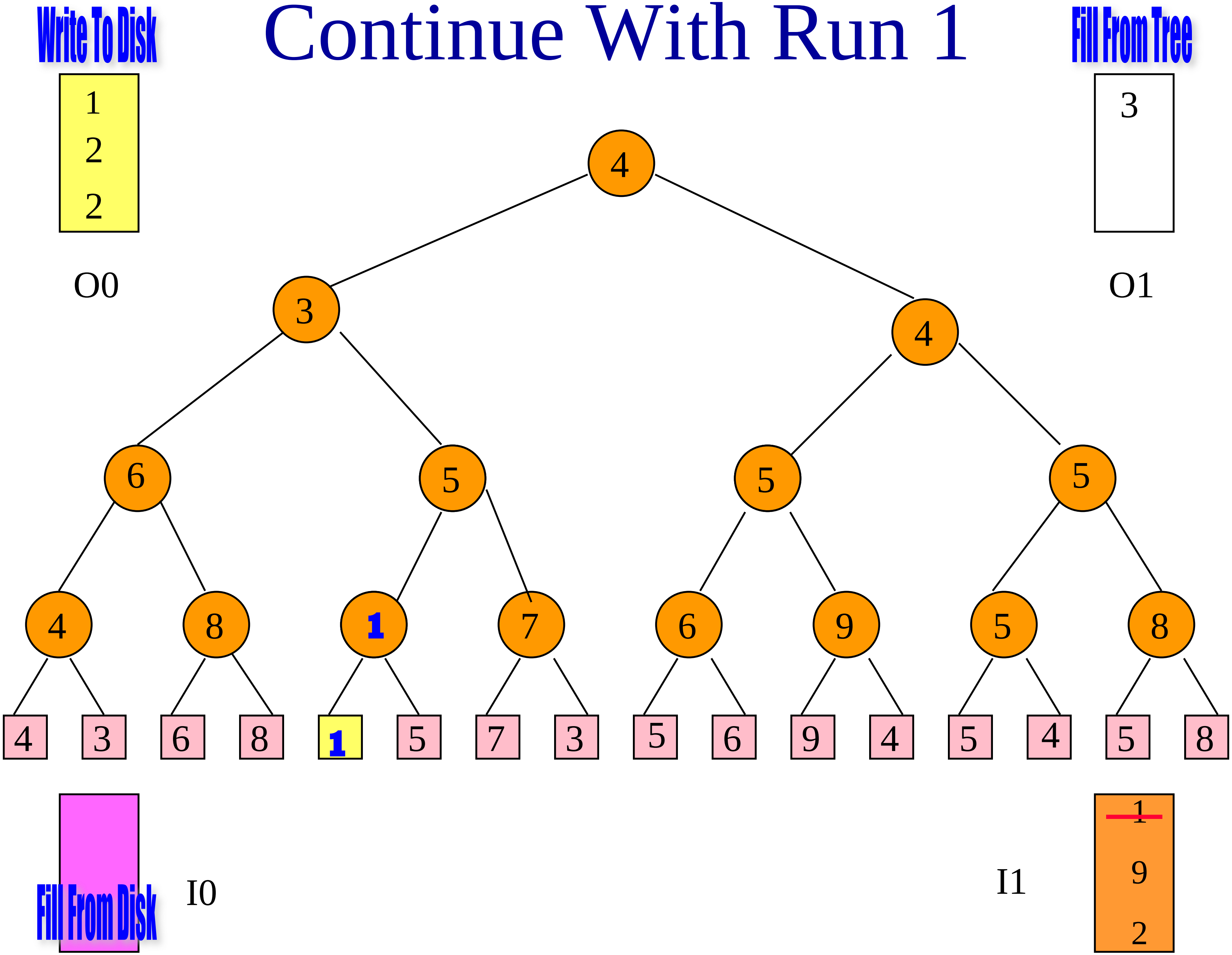
Interchange Role Of Buffers



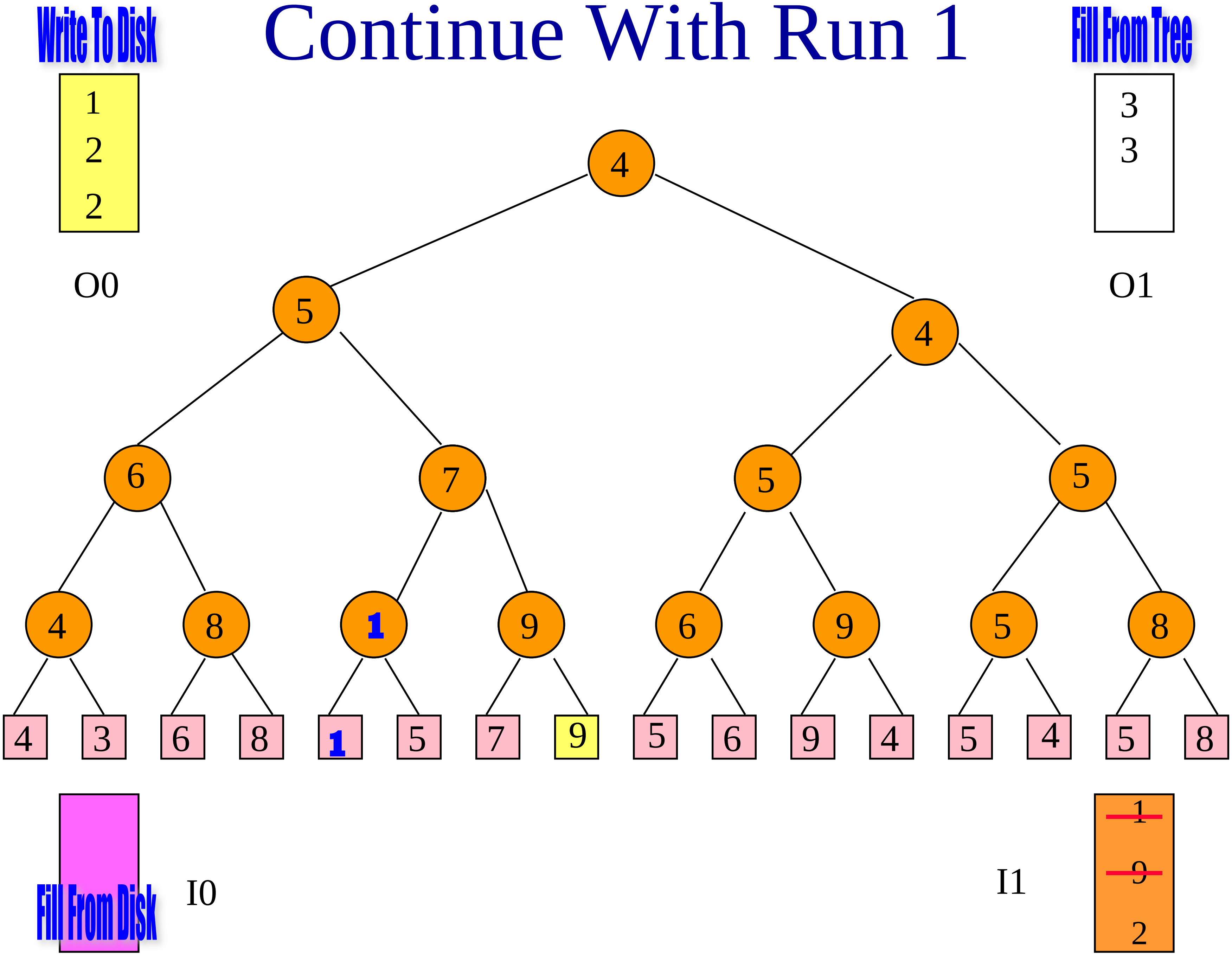
Continue With Run 1



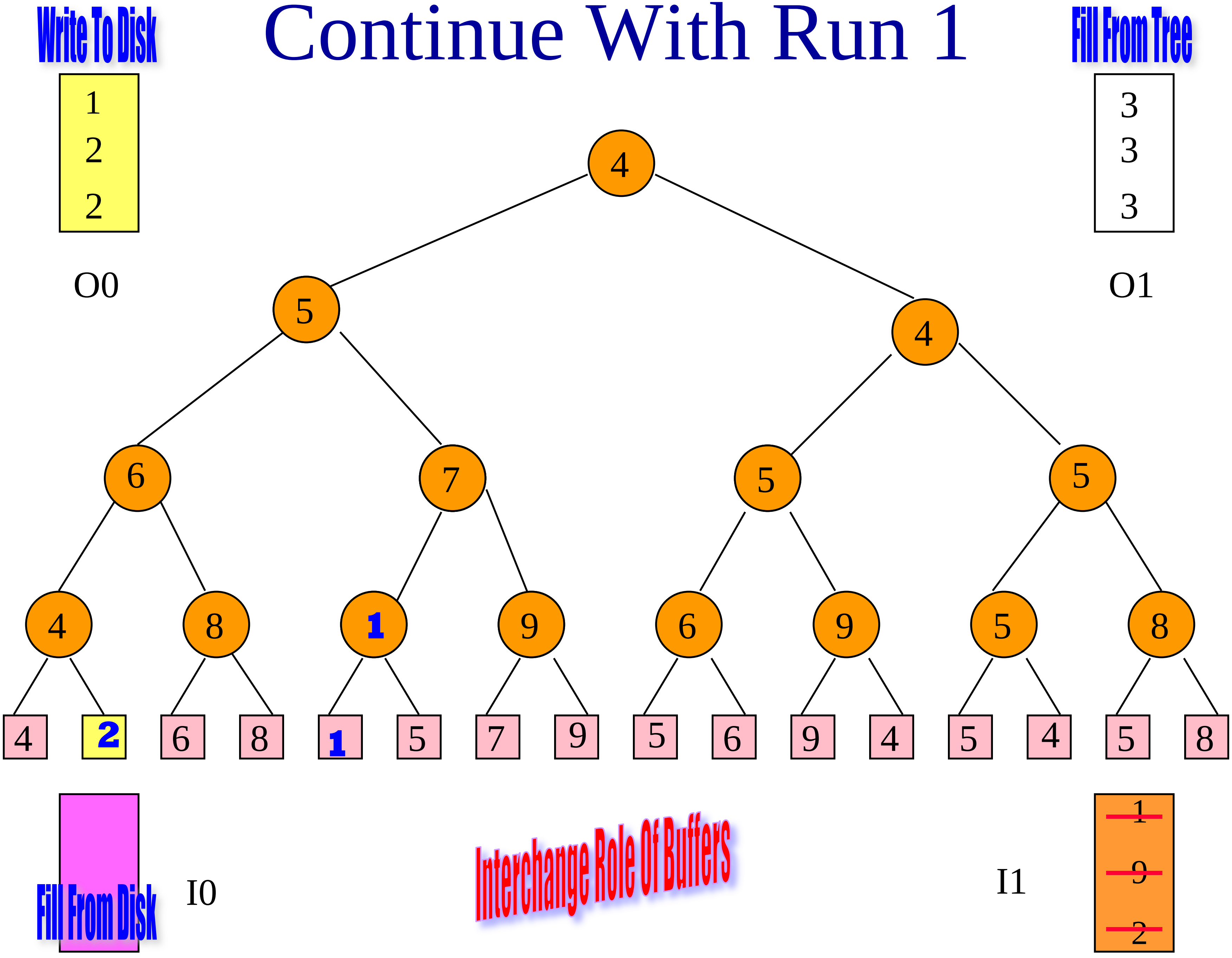
Continue With Run 1

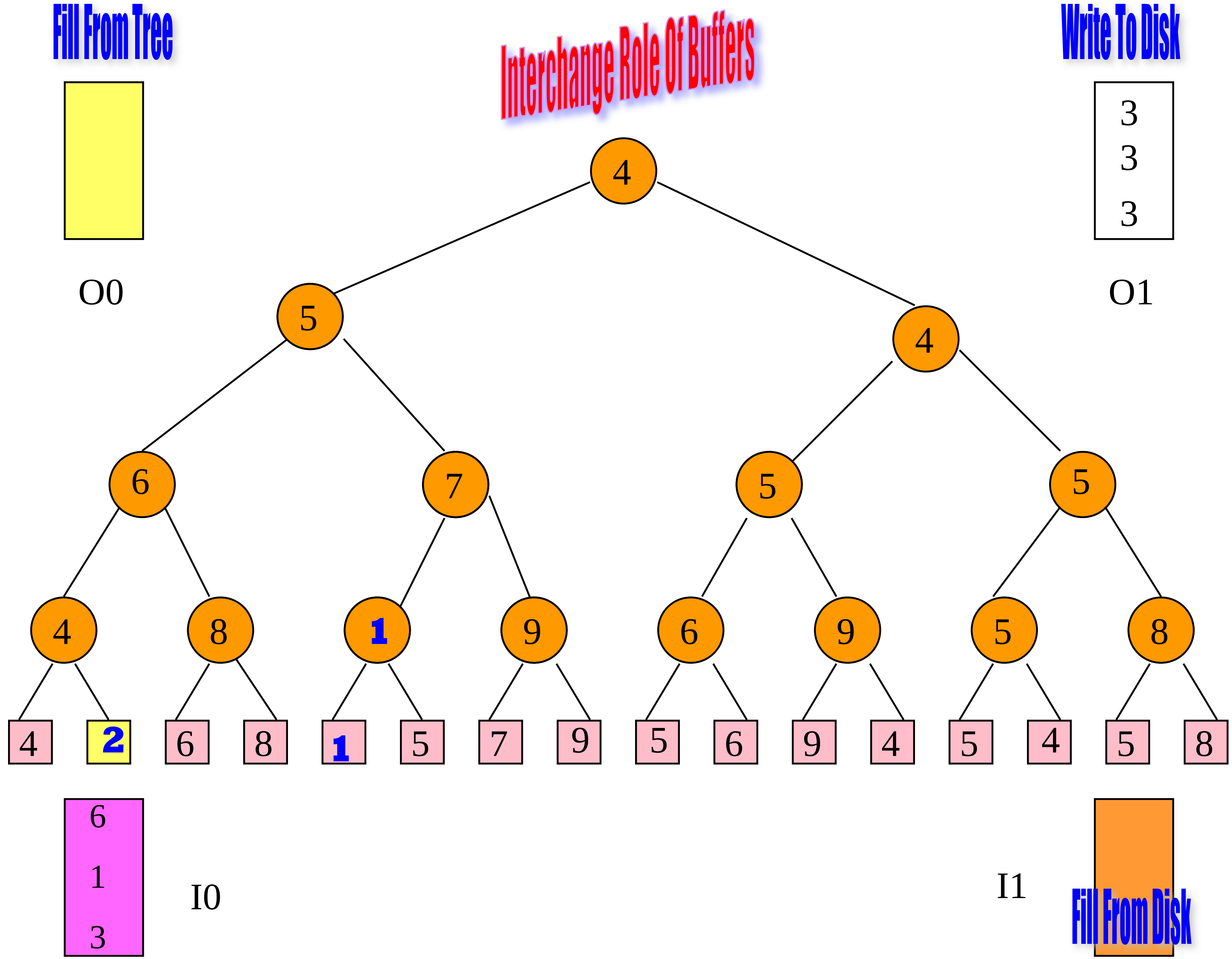


Continue With Run 1



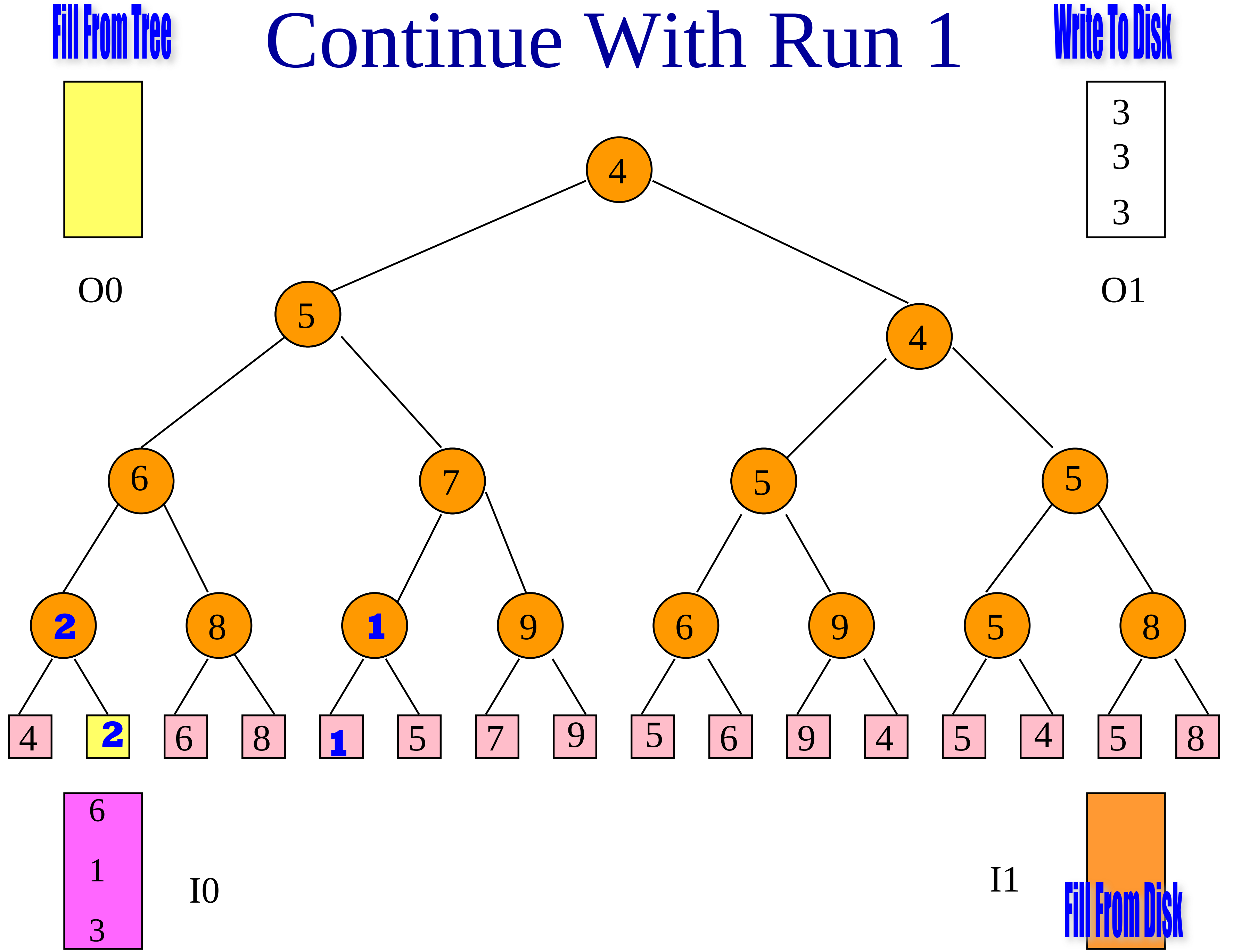
Continue With Run 1



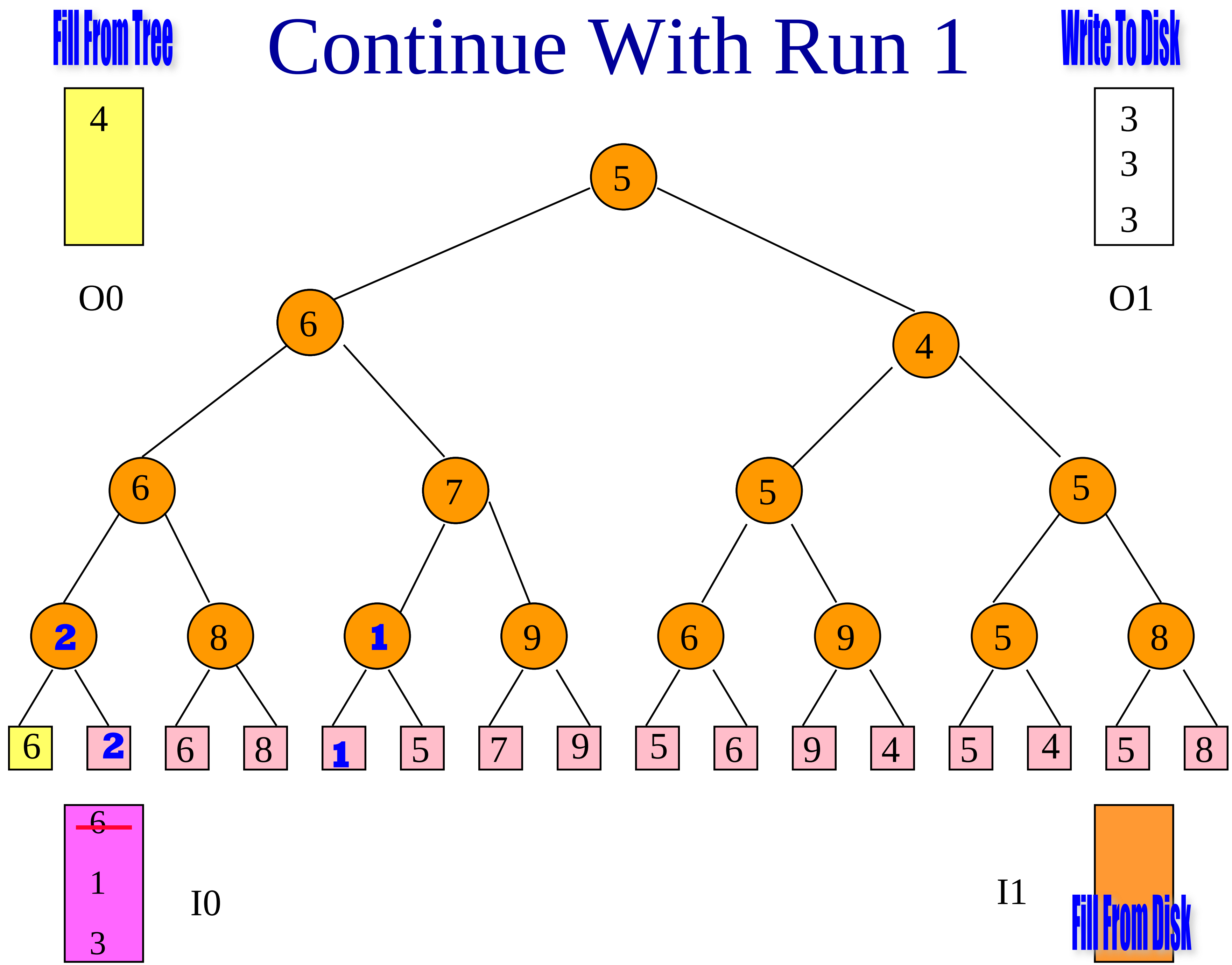


Fill From Tree
Interchange Role Of Buffers
Write To Disk
3
3
3
O0
O1
4
5
4
6
7
5
5
4
8
1
9
6
9
5
8
4
2
6
8
1
5
7
9
5
6
9
4
5
4
5
8
6
1
3
I0
I1
Fill From Disk

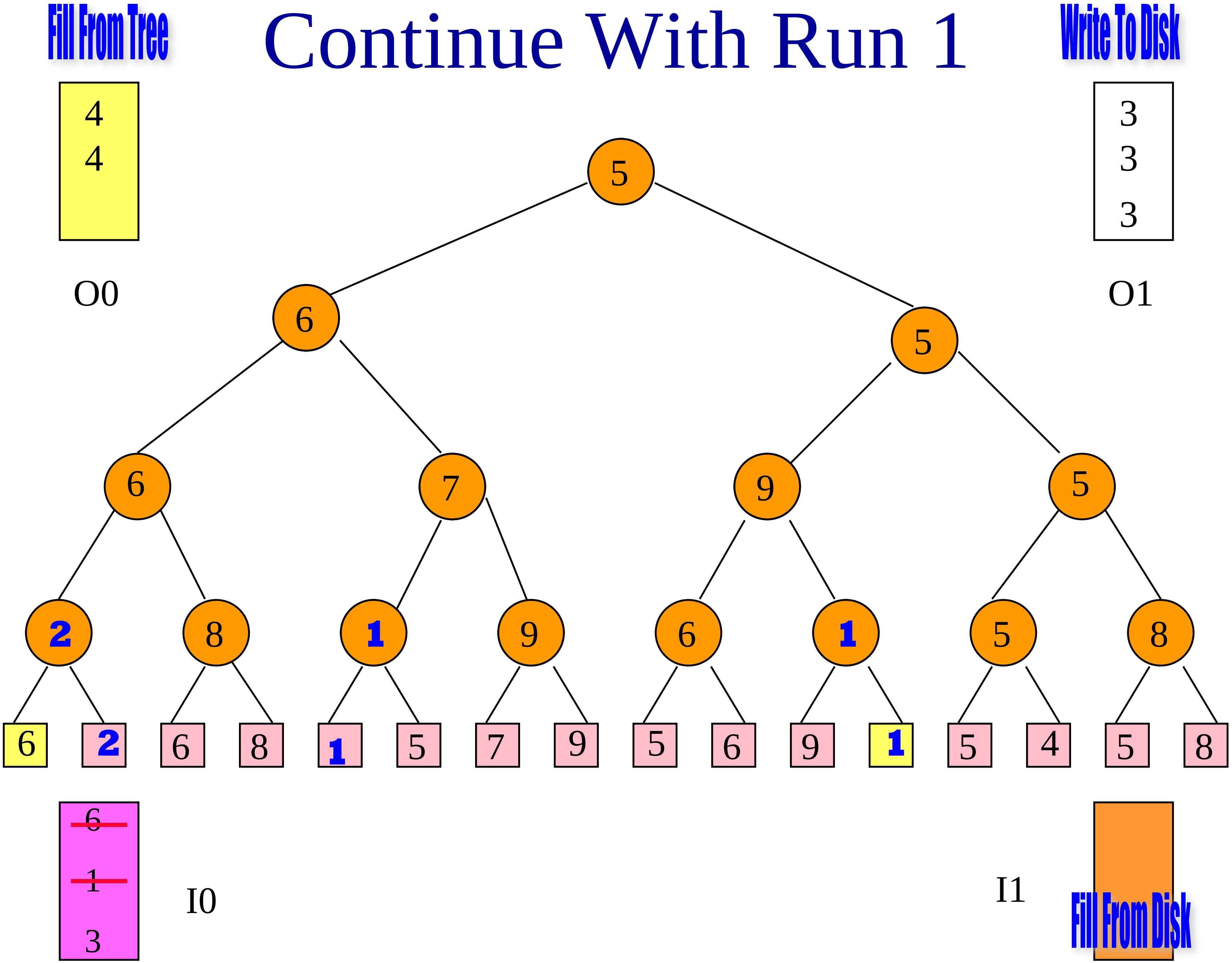
Continue With Run 1



Continue With Run 1



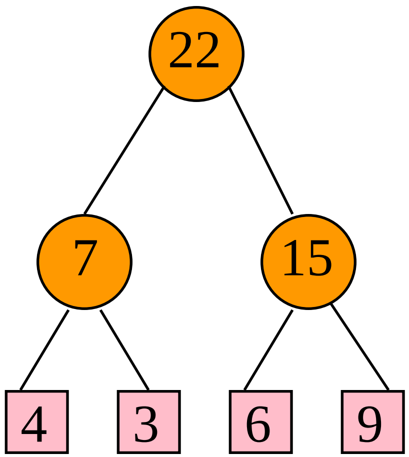
Continue With Run 1



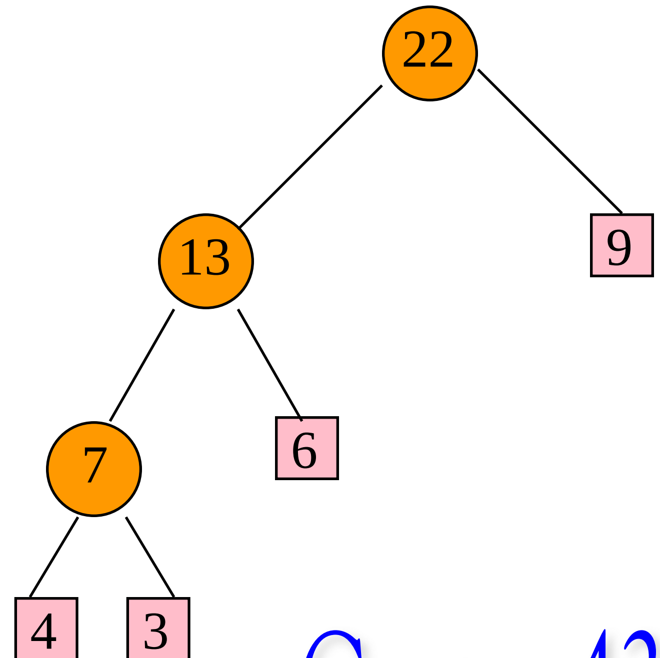
RUN SIZE

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

Merging Runs Of Different Length



Cost = 44



Cost = 42

Best merge sequence?

Requirements

- 3 buffers for improving run generation
- Run lengths and best merge sequence should be output
- Performance comparison

- #include <thread>

Phase 5

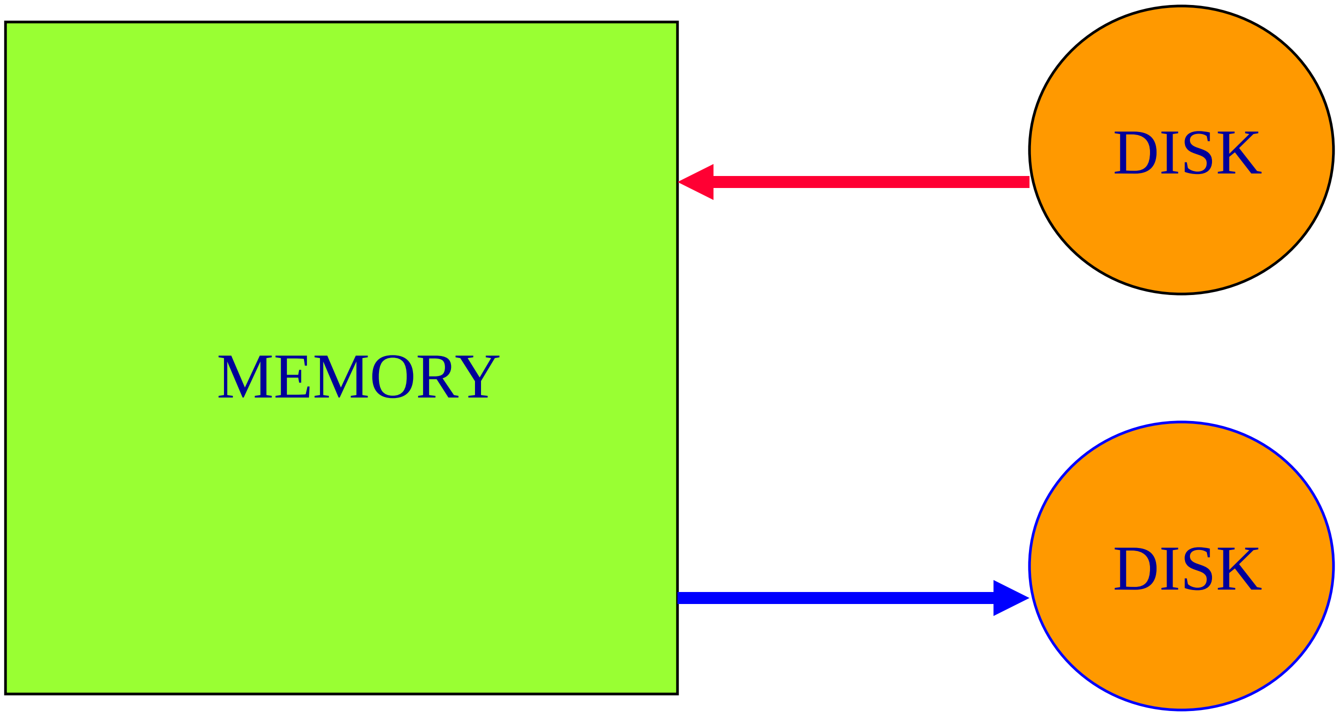
Merge Sort: Improve Run Merging

Improve Run Merging

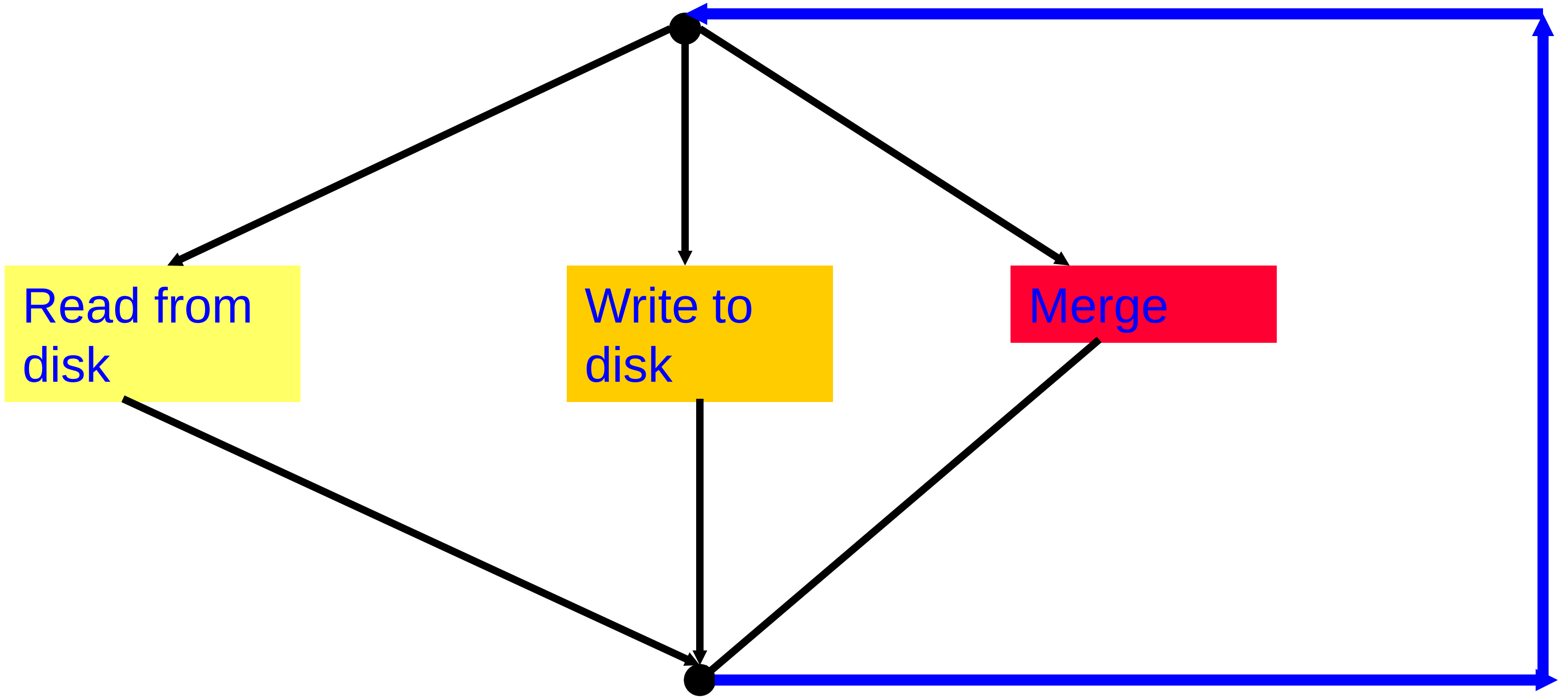
- Reduce number of merge passes.
  - Use higher order merge.
  - Number of passes
    $= \text{ceil}(\log_k(\text{number of initial runs}))$
    where k is the merge order.
- More generally, a higher-order merge reduces the cost of the optimal merge tree.

Improve Run Merging

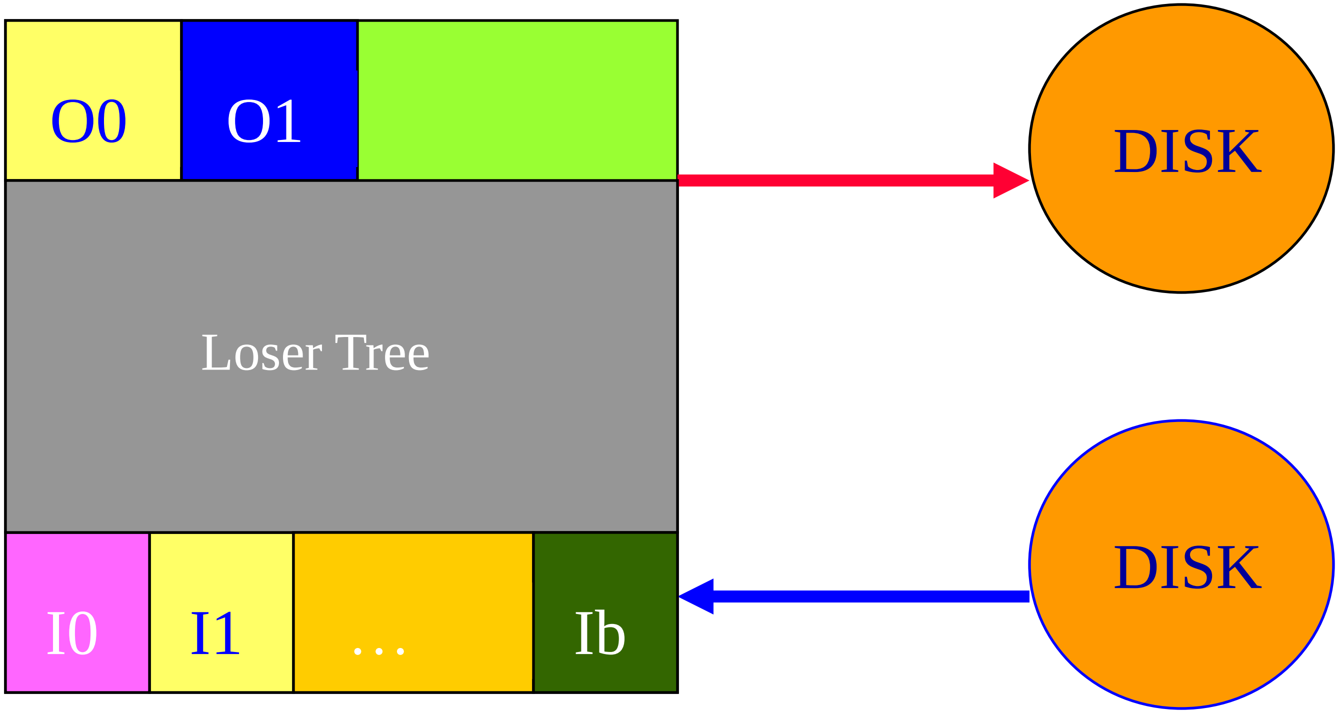
- Overlap input, output, and internal merging.



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

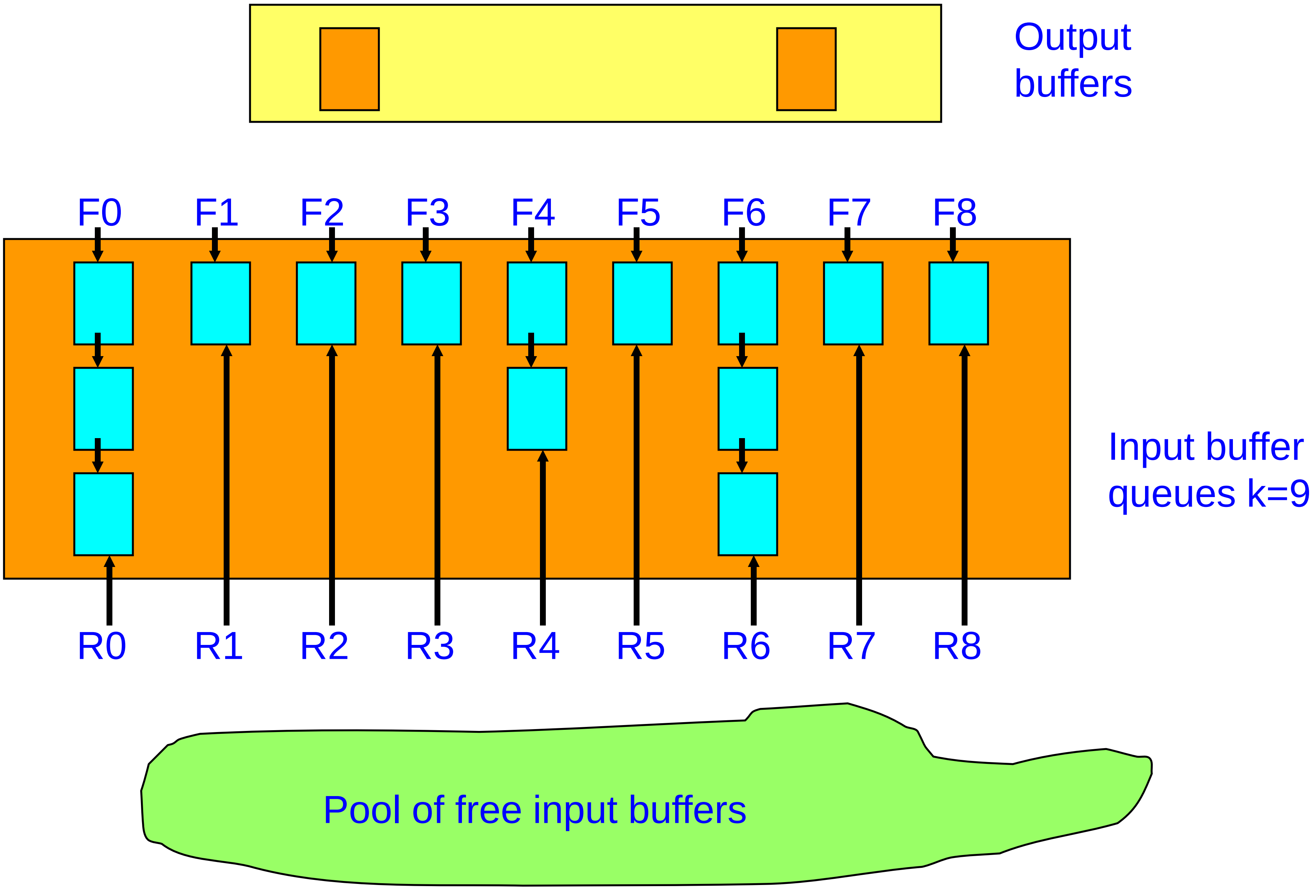
- When 2 input buffers are dedicated to each of the k runs being merged, 2k buffers are not enough!
- Input buffers must be allocated to runs on an as needed basis.

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.

- 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?

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.

There may be no next buffer in the queue.

What Can Go Wrong?

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)

$\boxed{\text{initiate read of next block for this run;}}$

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

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;
```

Requirements

- Allocate as needed strategy
- Performance comparison & analysis