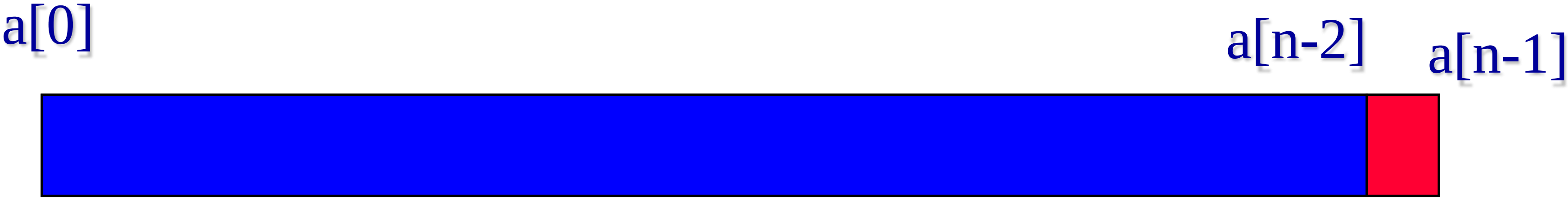
Chapter Ten

Further Consideration for Sorting

Sorting

- Rearrange n elements into ascending order.
- 7, 3, 6, 2, 1 ➔ 1, 2, 3, 6, 7

Insertion Sort



- n <= 1 → already sorted. So, assume n > 1.
- a[0:n-2] is sorted recursively.
- a[n-1] is inserted into the sorted a[0:n-2].
- Complexity is O(n^2).
- Usually implemented nonrecursively (see text).

Quick Sort

- When n <= 1, the list is sorted.
- When n > 1, select a pivot element from out of the n elements.
- Partition the n elements into 3 segments left, middle and right.
- The middle segment contains only the pivot element.
- All elements in the left segment are <= pivot.
- All elements in the right segment are >= pivot.
- Sort left and right segments recursively.
- Answer is sorted left segment, followed by middle segment followed by sorted right segment.

Example

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 segments recursively.

Choice Of Pivot

- Pivot is leftmost element in list that is to be sorted.
  - When sorting a[6:20], use a[6] as the pivot.
  - Text implementation does this.
- Randomly select one of the elements to be sorted as the pivot.
  - When sorting a[6:20], generate a random number r in the range [6, 20]. Use a[r] as the pivot.

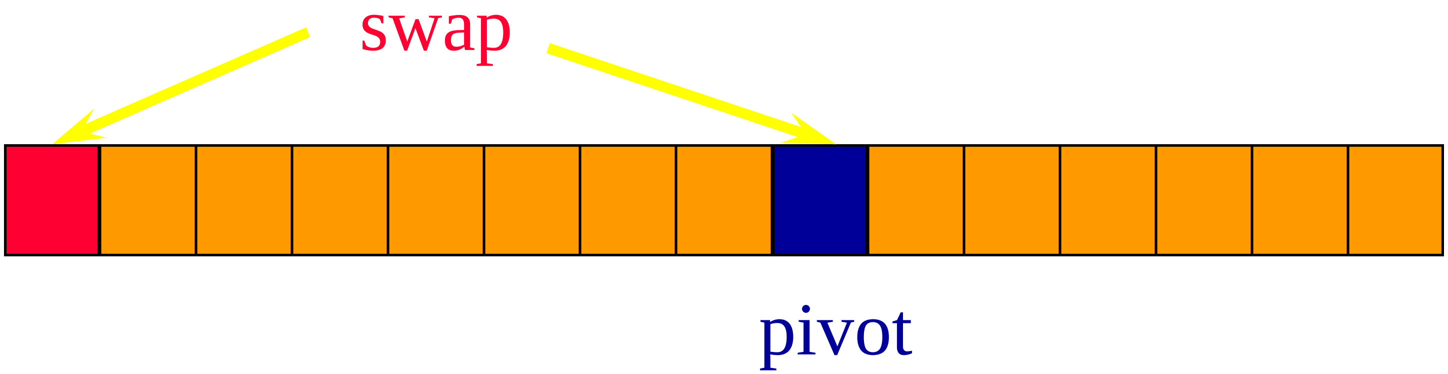
Choice Of Pivot

- Median-of-Three rule. From the leftmost, middle, and rightmost elements of the list to be sorted, select the one with median key as the pivot.
  - When sorting a[6:20], examine a[6], a[13] ((6+20)/2), and a[20]. Select the element with median (i.e., middle) key.
  - If a[6].key = 30, a[13].key = 2, and a[20].key = 10, a[20] becomes the pivot.
  - If a[6].key = 3, a[13].key = 2, and a[20].key = 10, a[6] becomes the pivot.

Choice Of Pivot

  - If a[6].key = 30, a[13].key = 25, and a[20].key = 10, a[13] becomes the pivot.

- When the pivot is picked at random or when the median-of-three rule is used, we can use the quick sort code of the text provided we first swap the leftmost element and the chosen pivot.



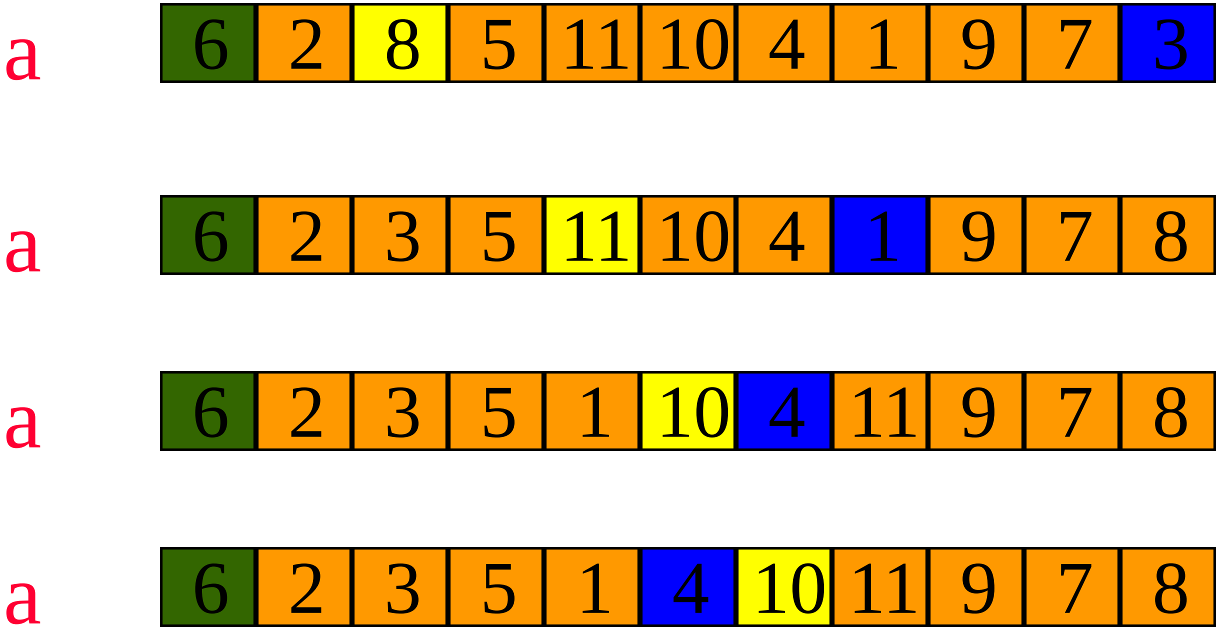
Partitioning Example Using Additional Array

a	6	2	8	5	11	10	4	1	9	7	3

b	2	5	4	1	3	6	7	9	10	11	8

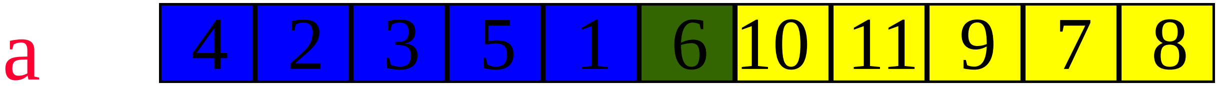
Sort left and right segments recursively.

In-Place Partitioning Example



a	6	2	8	5	11	10	4	1	9	7	3
a	6	2	3	5	11	10	4	1	9	7	8
a	6	2	3	5	1	10	4	11	9	7	8
a	6	2	3	5	1	4	10	11	9	7	8

bigElement is not to left of smallElement, terminate process. Swap pivot and smallElement.



a	4	2	3	5	1	6	10	11	9	7	8

Complexity

- $O(n)$ time to partition an array of n elements.
- Let $t(n)$ be the time needed to sort n elements.
- $t(0) = t(1) = c$, where c is a constant.
- When $t > 1$,

  $t(n) = t(|left|) + t(|right|) + dn$,

  where d is a constant.
- $t(n)$ is maximum when either $|left| = 0$ or $|right| = 0$ following each partitioning.

Complexity

- This happens, for example, when the pivot is always the smallest element.
- For the worst-case time,

  $t(n) = t(n-1) + dn, n > 1$
- Use repeated substitution to get $t(n) = O(n^2)$.
- The best case arises when |left| and |right| are equal (or differ by 1) following each partitioning.

Complexity Of Quick Sort

- So the best-case complexity is O(n log n).
- Average complexity is also O(n log n).
- To help get partitions with almost equal size, change in-place swap rule to:
  - Find leftmost element (bigElement) >= pivot.
  - Find rightmost element (smallElement) <= pivot.
  - Swap bigElement and smallElement provided bigElement is to the left of smallElement.
- O(n) space is needed for the recursion stack. May be reduced to O(log n).

Complexity Of Quick Sort

- To improve performance, stop recursion when segment size is <= 15 (say) and sort these small segments using insertion sort.

C++ STL sort Function

- Quick sort.
  - Switch to heap sort when number of subdivisions exceeds some constant times $\log_2 n$.
  - Switch to insertion sort when segment size becomes small.

Merge Sort

- Partition the n > 1 elements into two smaller instances.
- First ceil(n/2) elements define one of the smaller instances; remaining floor(n/2) elements define the second smaller instance.
- Each of the two smaller instances is sorted recursively.
- The sorted smaller instances are combined using a process called merge.
- Complexity is O(n log n).
- Usually implemented nonrecursively.

Merge Two Sorted Lists

- A = (2, 5, 6)

  B = (1, 3, 8, 9, 10)

  C = ()
- Compare smallest elements of A and B and merge smaller into C.
- A = (2, 5, 6)

  B = (3, 8, 9, 10)

  C = (1)

Merge Two Sorted Lists

- A = (5, 6)
  B = (3, 8, 9, 10)
  C = (1, 2)
- A = (5, 6)
  B = (8, 9, 10)
  C = (1, 2, 3)
- A = (6)
  B = (8, 9, 10)
  C = (1, 2, 3, 5)

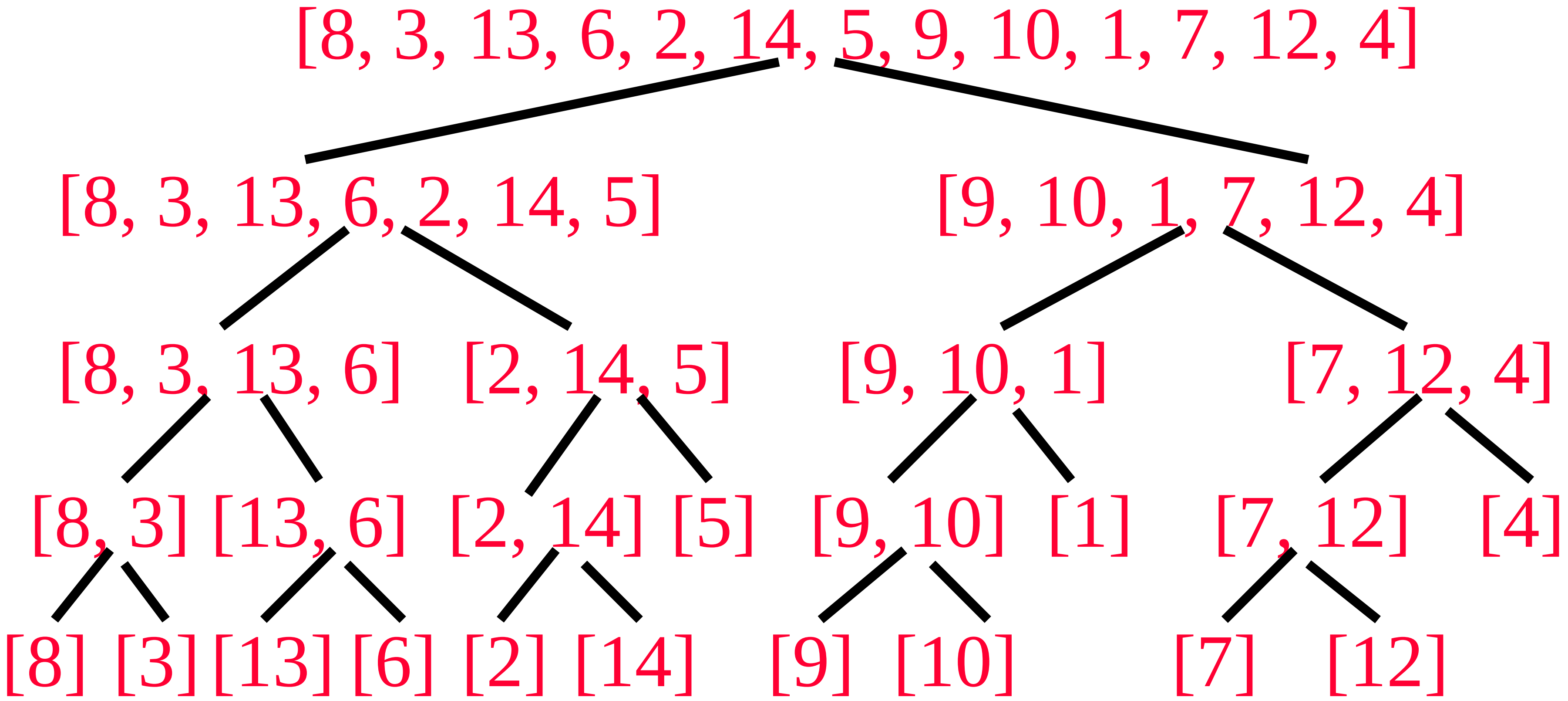
Merge Two Sorted Lists

- A = ()

  B = (8, 9, 10)

  C = (1, 2, 3, 5, 6)
- When one of A and B becomes empty, append the other list to C.
- O(1) time needed to move an element into C.
- Total time is O(n + m), where n and m are, respectively, the number of elements initially in A and B.

Merge Sort



[8, 3, 13, 6, 2, 14, 5, 9, 10, 1, 7, 12, 4]

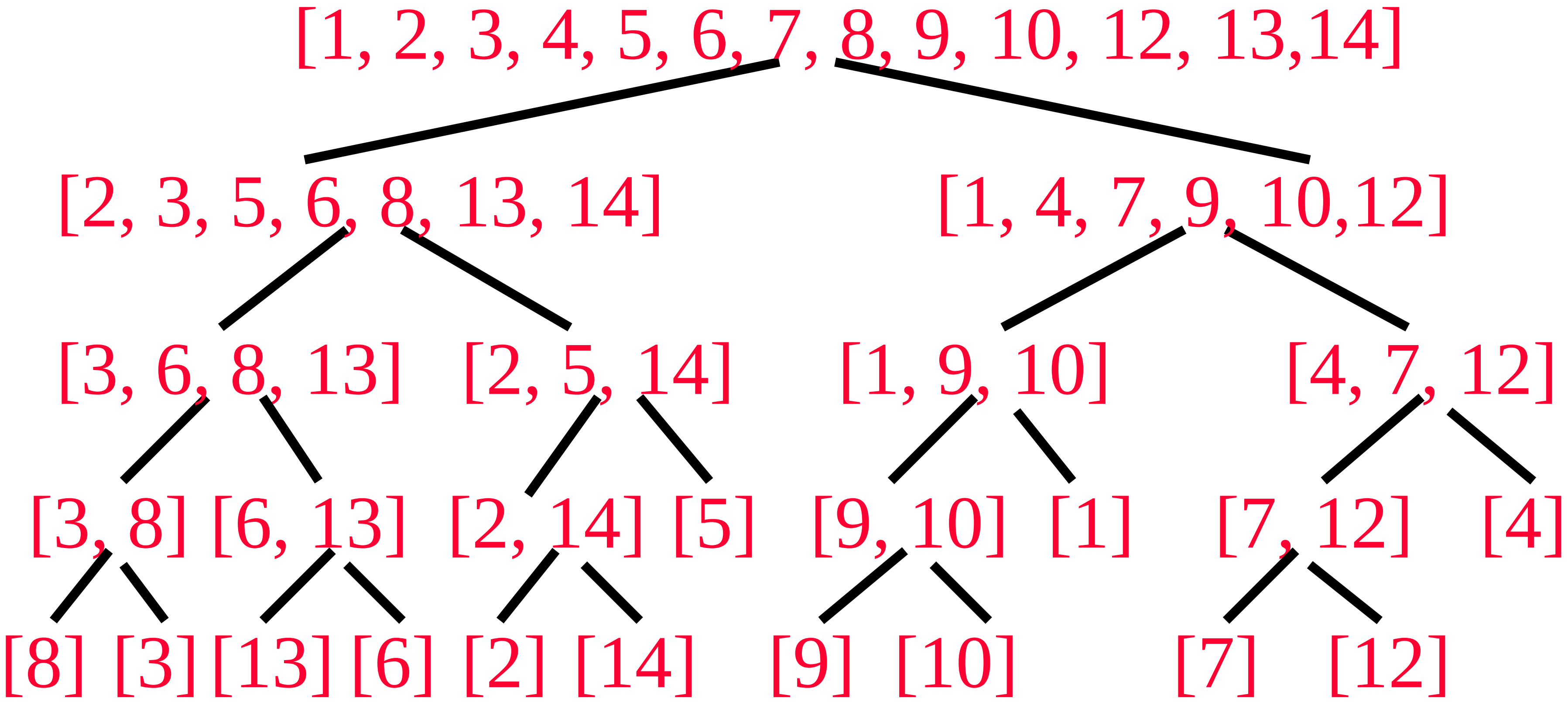
[8, 3, 13, 6, 2, 14, 5] [9, 10, 1, 7, 12, 4]

[8, 3, 13, 6] [2, 14, 5] [9, 10, 1] [7, 12, 4]

[8, 3] [13, 6] [2, 14] [5] [9, 10] [1] [7, 12] [4]

[8] [3] [13] [6] [2] [14] [9] [10] [7] [12]

Merge Sort



[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13,14]

[2, 3, 5, 6, 8, 13, 14] [1, 4, 7, 9, 10,12]

[3, 6, 8, 13] [2, 5, 14] [1, 9, 10] [4, 7, 12]

[3, 8] [6, 13] [2, 14] [5] [9, 10] [1] [7, 12] [4]

[8] [3] [13] [6] [2] [14] [9] [10] [7] [12]

Time Complexity

- Let $t(n)$ be the time required to sort n elements.
- $t(0) = t(1) = c$, where c is a constant.
- When $n > 1$,

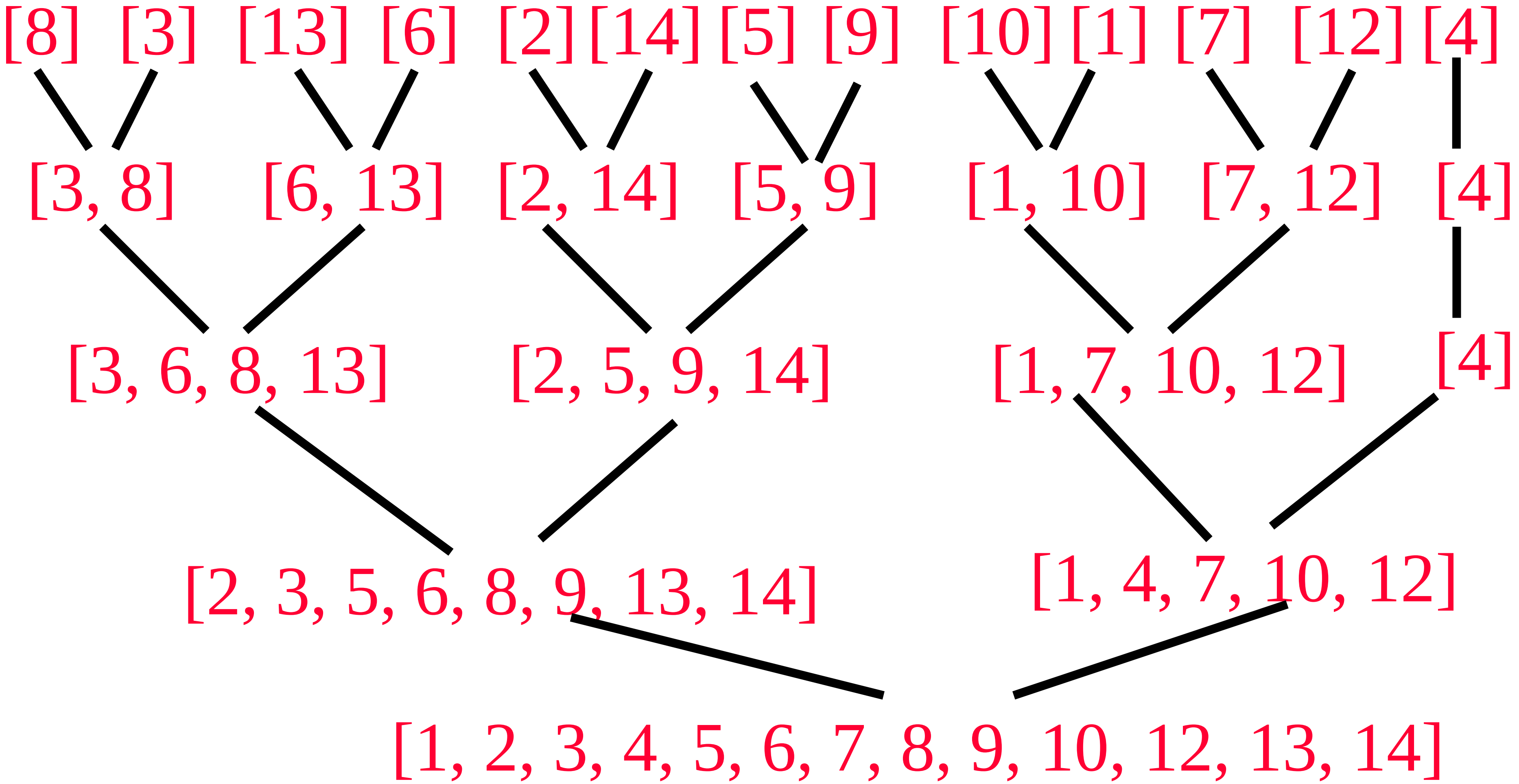
  $t(n) = t(ceil(n/2)) + t(floor(n/2)) + dn$,

  where d is a constant.
- To solve the recurrence, assume n is a power of 2 and use repeated substitution.
- $t(n) = O(n \log n)$.

Nonrecursive Version

- Eliminate downward pass.
- Start with sorted lists of size 1 and do pairwise merging of these sorted lists as in the upward pass.

Nonrecursive Merge Sort



[8] [3] [13] [6] [2] [14] [5] [9] [10] [1] [7] [12] [4]

[3, 8] [6, 13] [2, 14] [5, 9] [1, 10] [7, 12] [4]

[3, 6, 8, 13] [2, 5, 9, 14] [1, 7, 10, 12] [4]

[2, 3, 5, 6, 8, 9, 13, 14] [1, 4, 7, 10, 12]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14]

Complexity

- Sorted segment size is 1, 2, 4, 8, …
- Number of merge passes is ceil($\log_2 n$).
- Each merge pass takes O(n) time.
- Total time is O(n log n).
- Need O(n) additional space for the merge.
- Merge sort is slower than insertion sort when n <= 15 (approximately). So define a small instance to be an instance with n <= 15.
- Sort small instances using insertion sort.
- Start with segment size = 15.

Natural Merge Sort

- Initial sorted segments are the naturally ocurring sorted segments in the input.
- Input = [8, 9, 10, 2, 5, 7, 9, 11, 13, 15, 6, 12, 14].
- Initial segments are:

  [8, 9, 10] [2, 5, 7, 9, 11, 13, 15] [6, 12, 14]
- 2 (instead of 4) merge passes suffice.
- Segment boundaries have a[i] > a[i+1].

C++ STL stable_sort Function

- Merge sort is stable (relative order of elements with equal keys is not changed).
- Quick sort is not stable.
- STL's stable_sort is a merge sort that switches to insertion sort when segment size is small.

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.

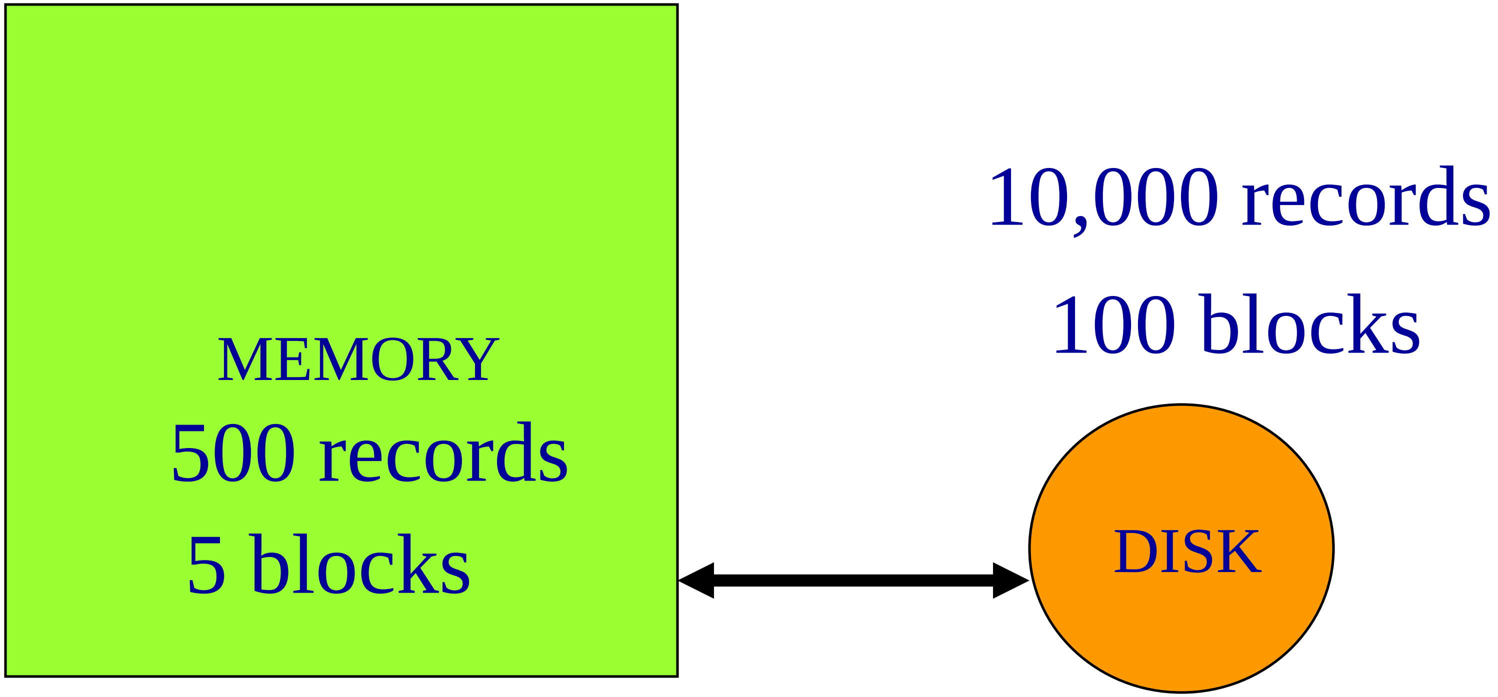
External Merge Sort

- 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



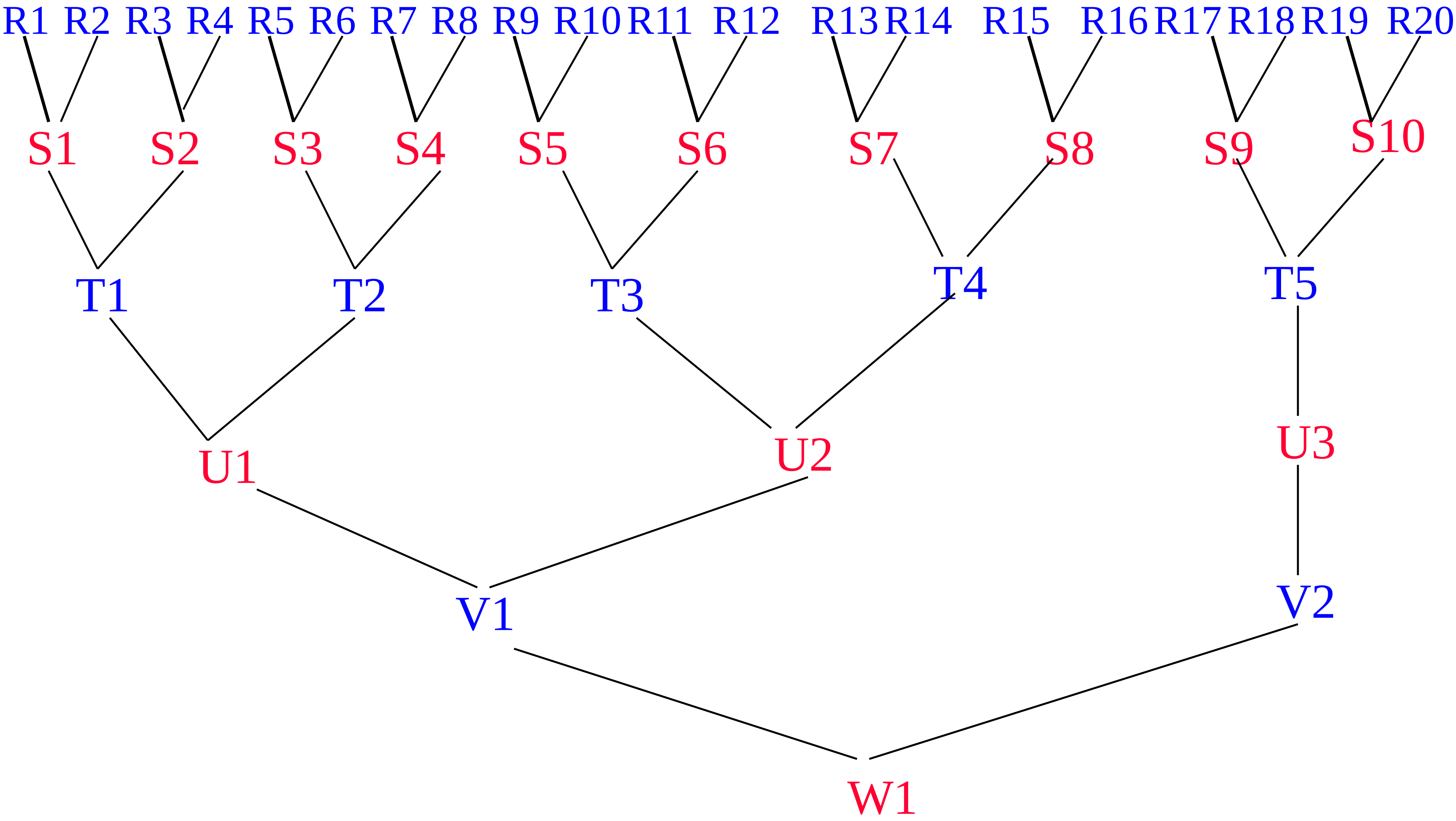
- Input 5 blocks.
- Sort.
- Output as a run.
- Do 20 times.

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

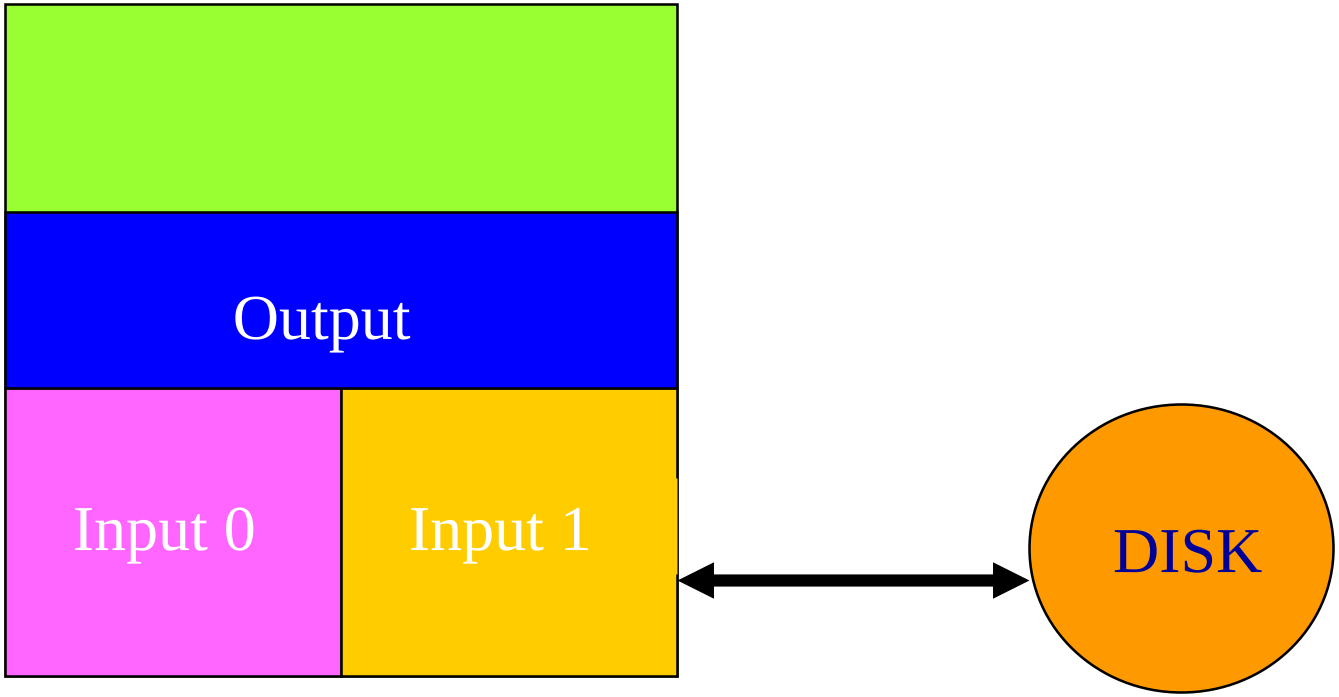
Run Merging

- 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


R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 R16 R17 R18 R19 R20
S1 S2 S3 S4 S5 S6 S7 S8 S9 S10
T1 T2 T3 T4 T5
U1 U2 U3
V1 V2
W1

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)

  $\quad$ * ceil($\log_2$(number of initial runs))

  = $(200t_{IO} + 100t_{IM})$ * ceil($\log_2(20)$)

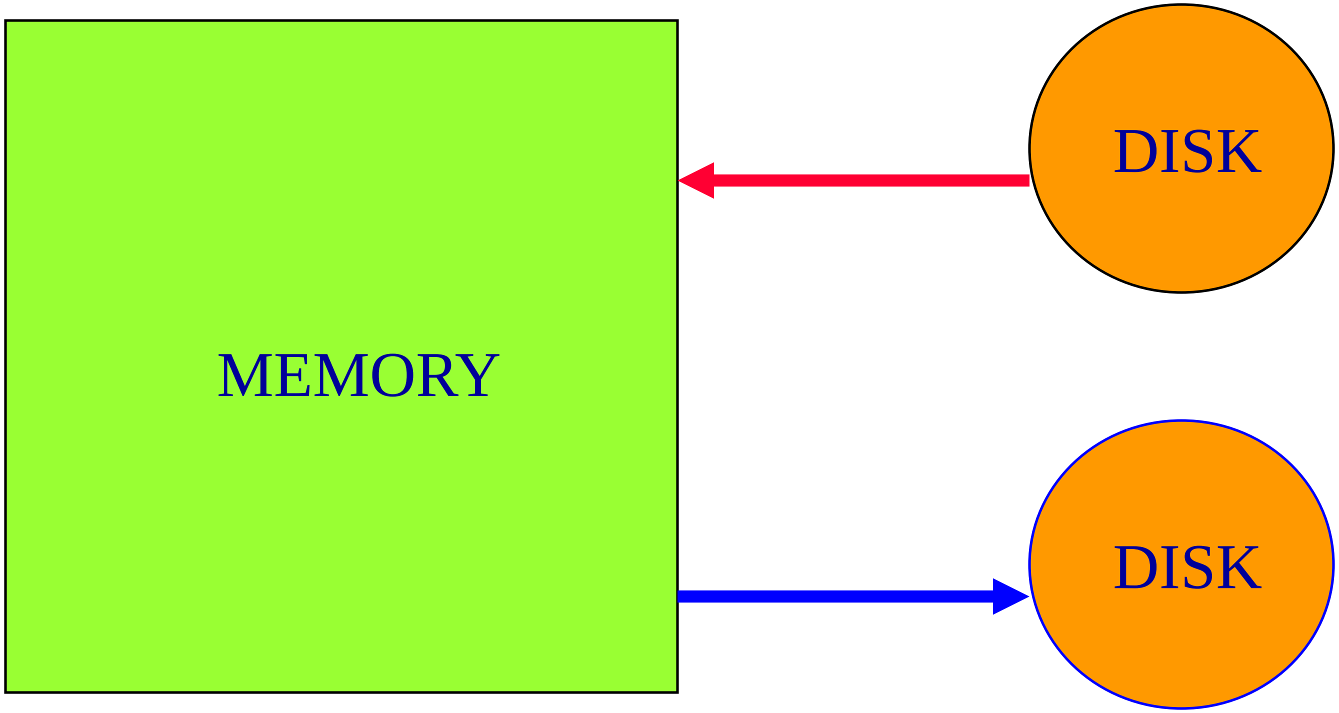
  = $(200t_{IO} + 100t_{IM})$ * 5

Factors In Overall Run Time

- Run generation. $200t_{IO} + 20t_{IS}$
  - Internal sort time.
  - Input and output time.
- Run merging. $(200t_{IO} + 100t_{IM}) * \mathrm{ceil}(\log_2(20))$
  - 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.

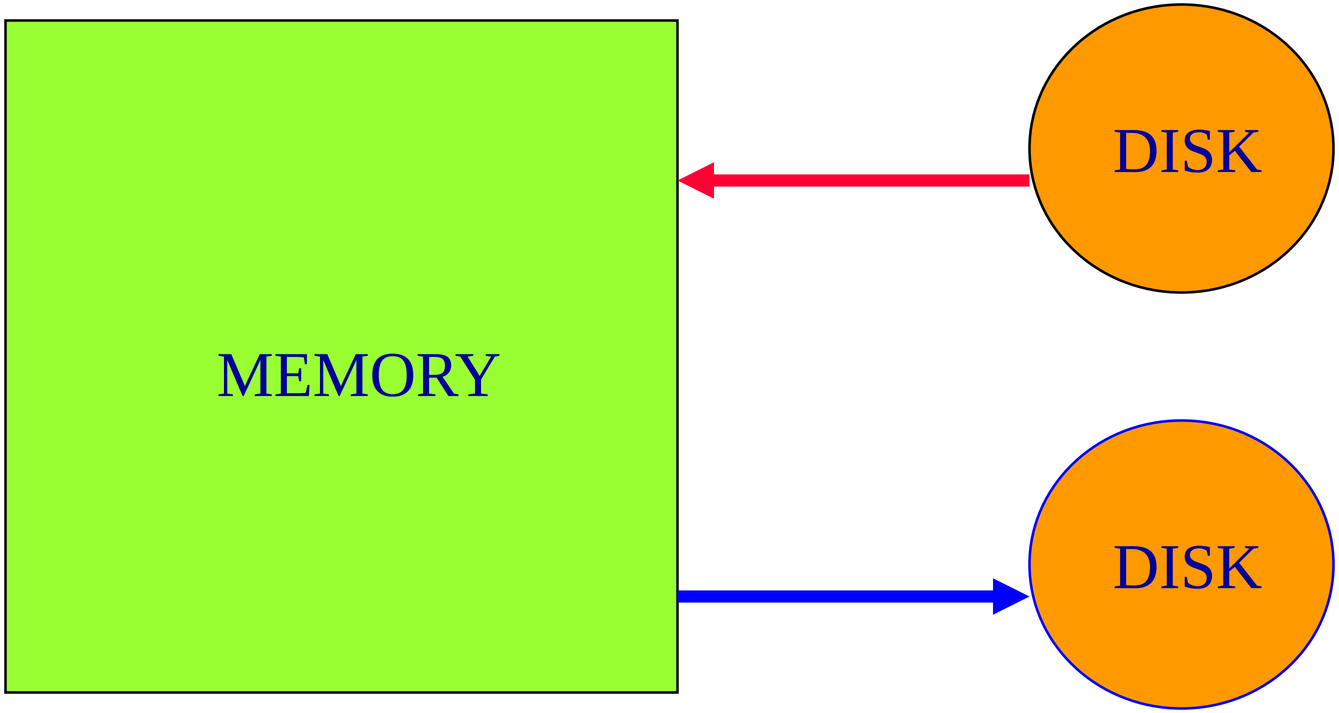


Improve Run Generation

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

Improve Run Merging

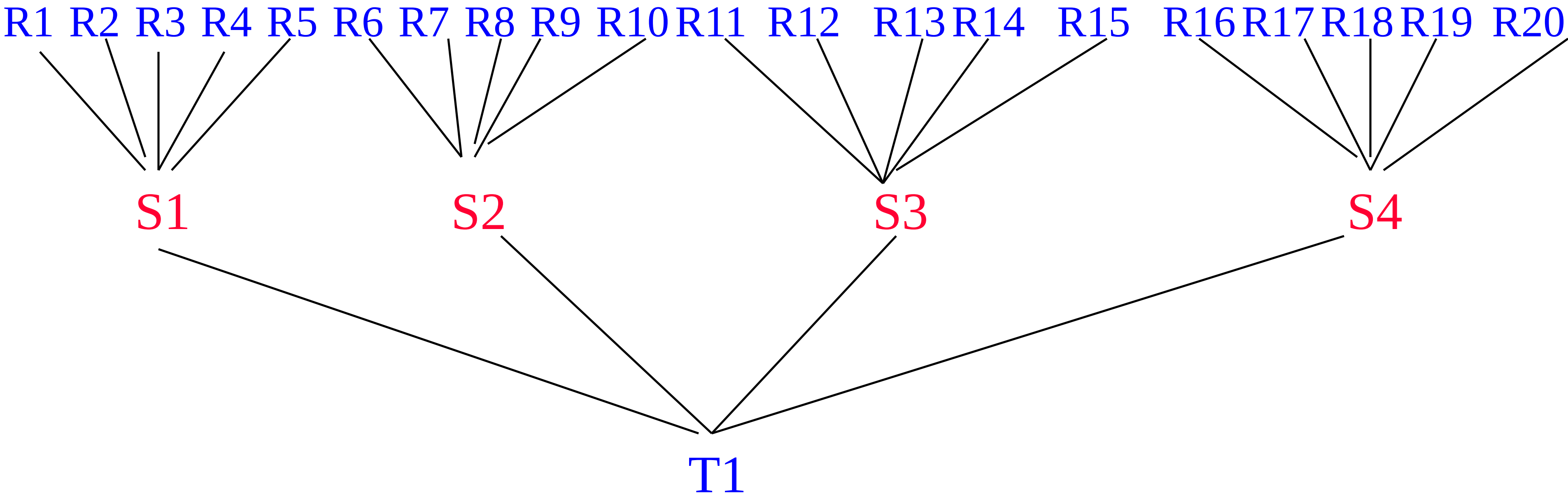
- Overlap input, output, and internal merging.



Improve Run Merging

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

Merge 20 Runs Using 5-Way Merging

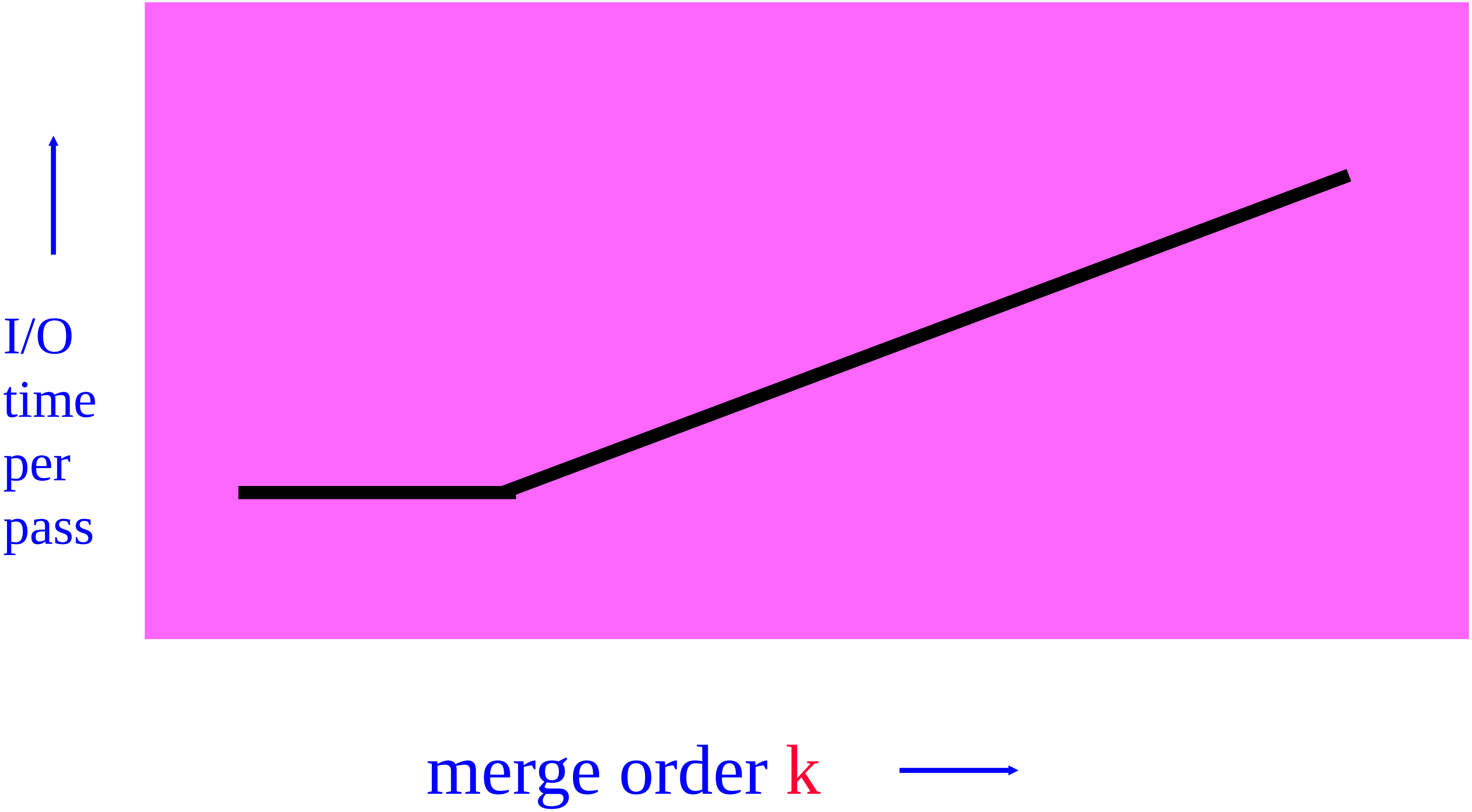


Number of passes = 2

I/O Time Per Merge Pass

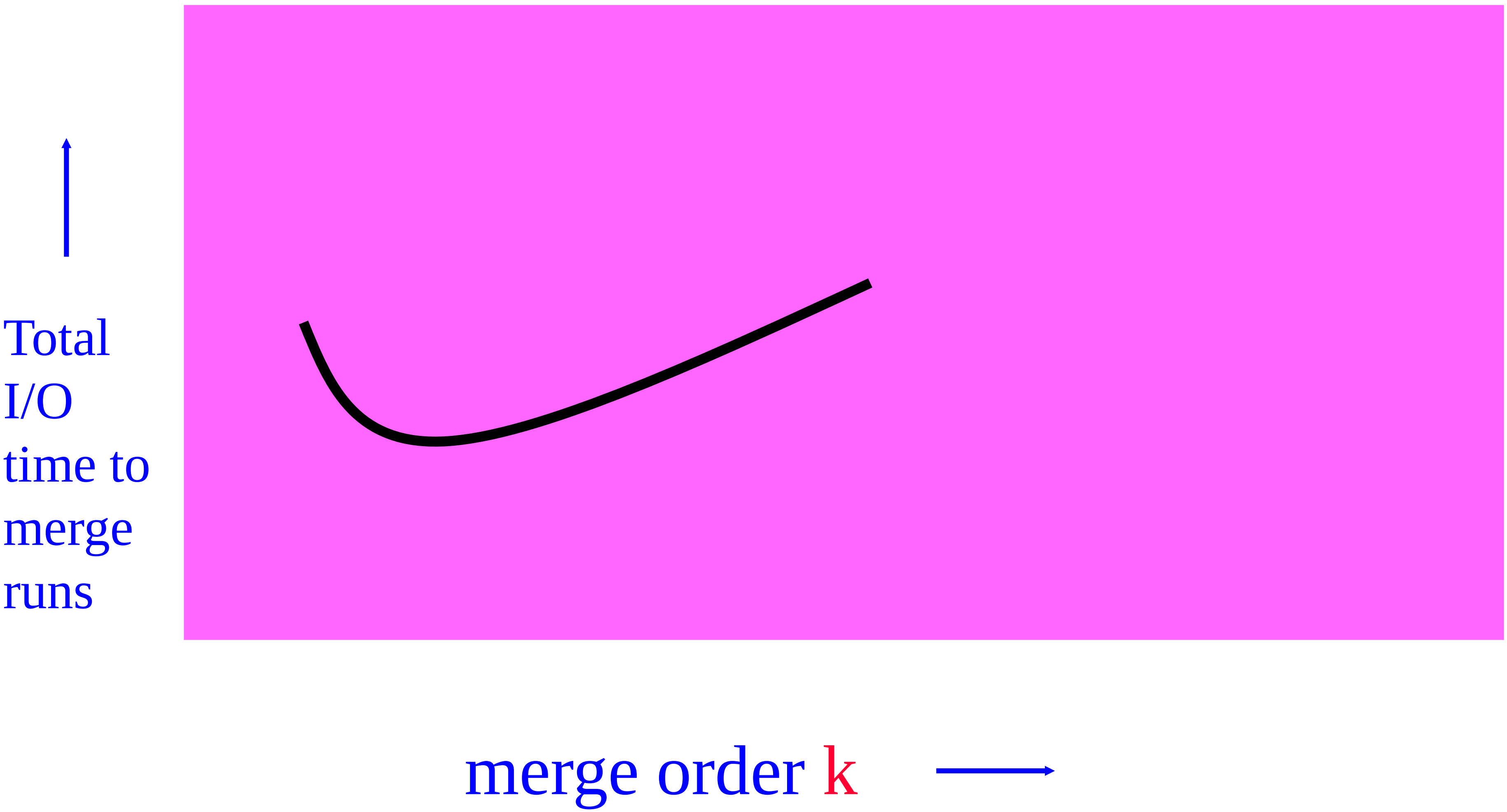
- Number of input buffers needed is linear in merge order k.
- Since memory size is fixed, block size decreases as k increases (after a certain k).
- So, number of blocks increases.
- So, number of seek and latency delays per pass increases.

I/O Time Per Merge Pass

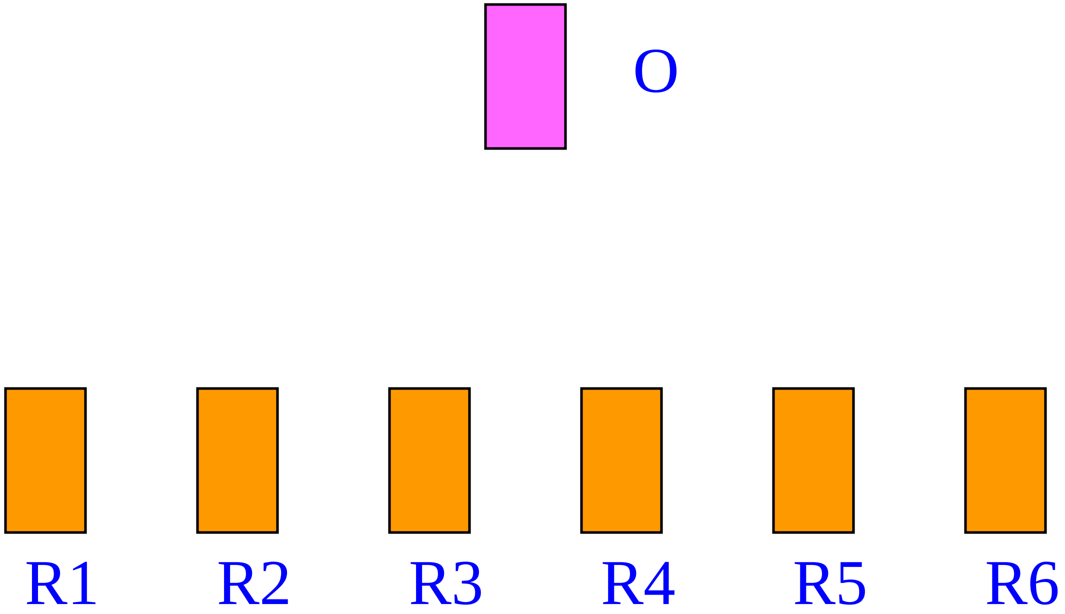


Total I/O Time To Merge Runs

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

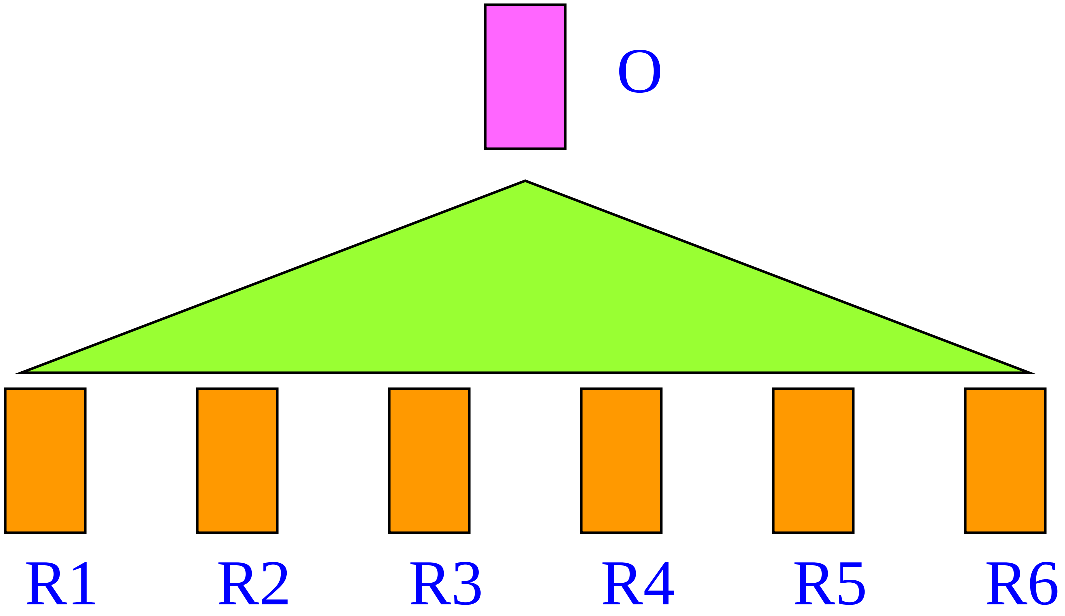


Internal Merge Time



- 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 Selection Tree



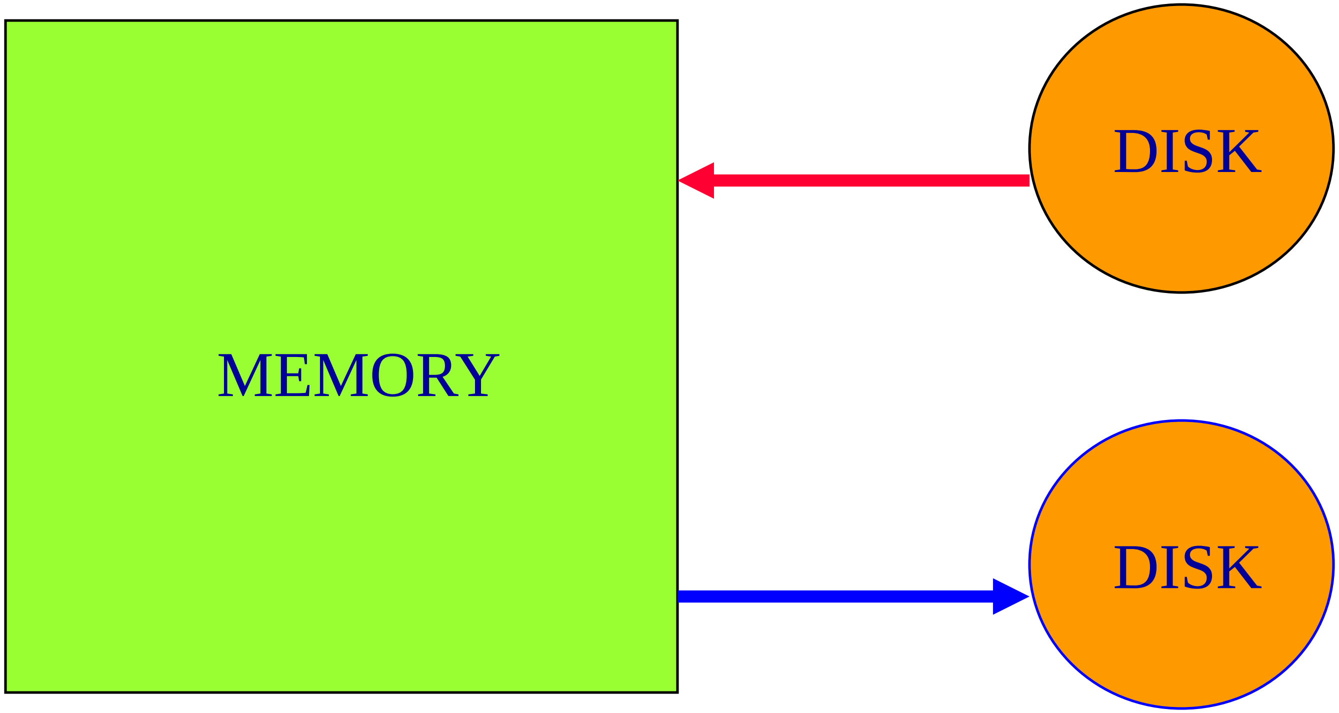
- Time to merge n records is $dn\log_2 k$, where d is a constant.
- Merge time per pass is $dn\log_2 k$.
- Total merge time is $(dn\log_2 k) \log_k r = dn\log_2 r$.

Improve Run Merging

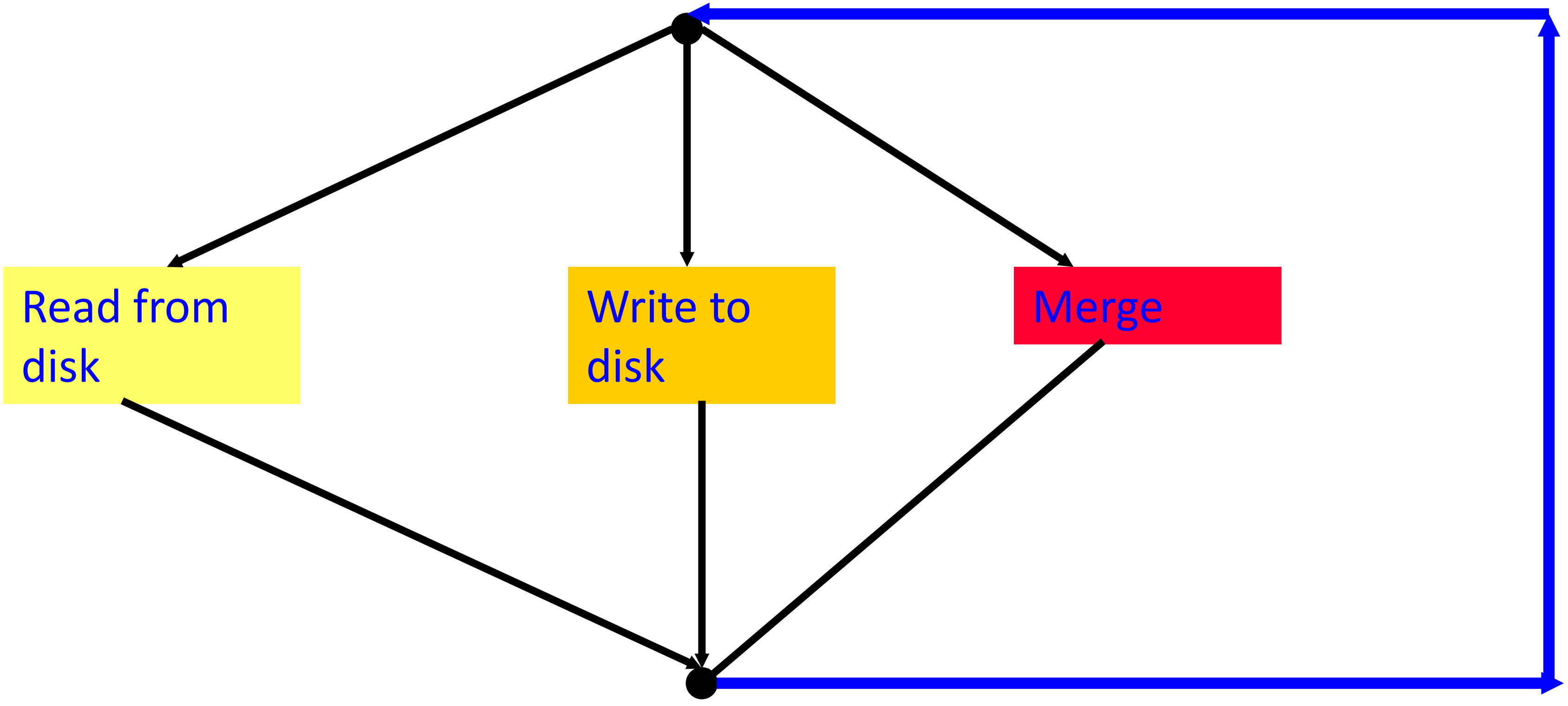
- Reduce number of merge passes.
  - Use higher order merge.
  - Number of passes
    = ceil($\log_k$(number of initial runs))
    where k is the merge order.
- More generally, a higher-order merge reduces the cost of the optimal merge tree.

Improve Run Merging

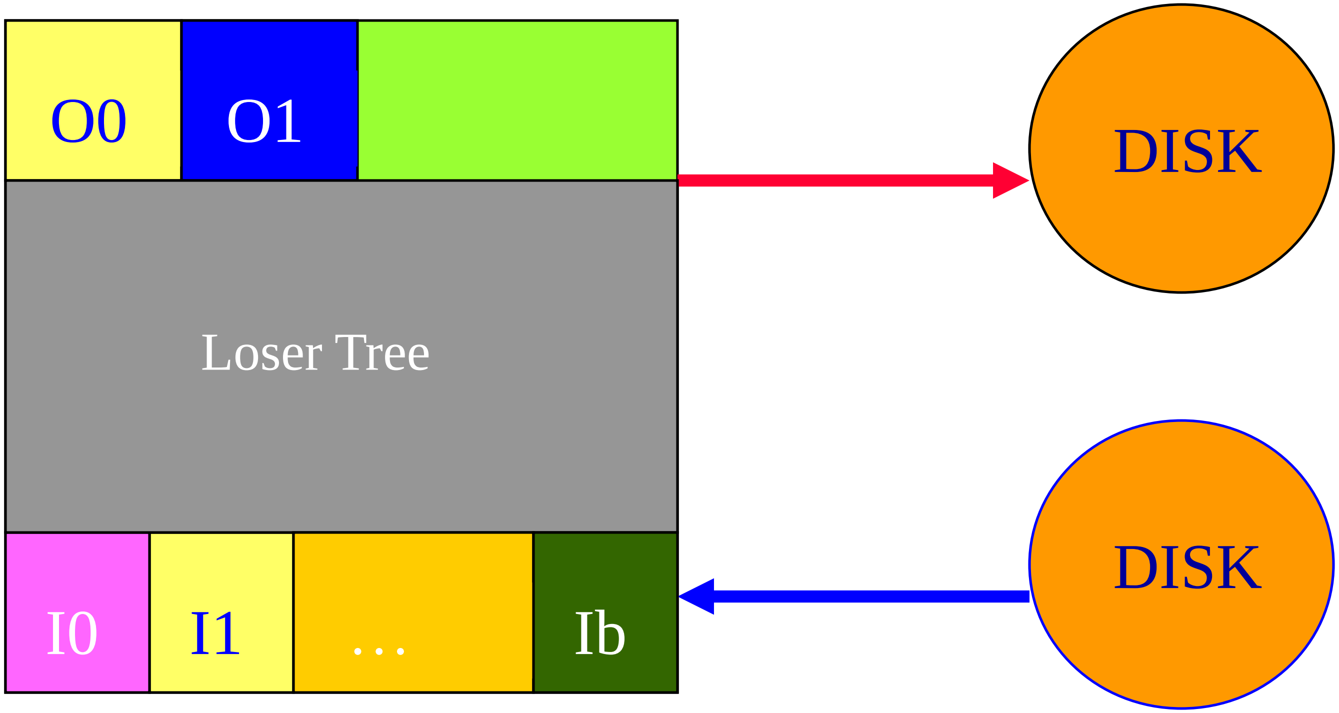
- Overlap input, output, and internal merging.



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.

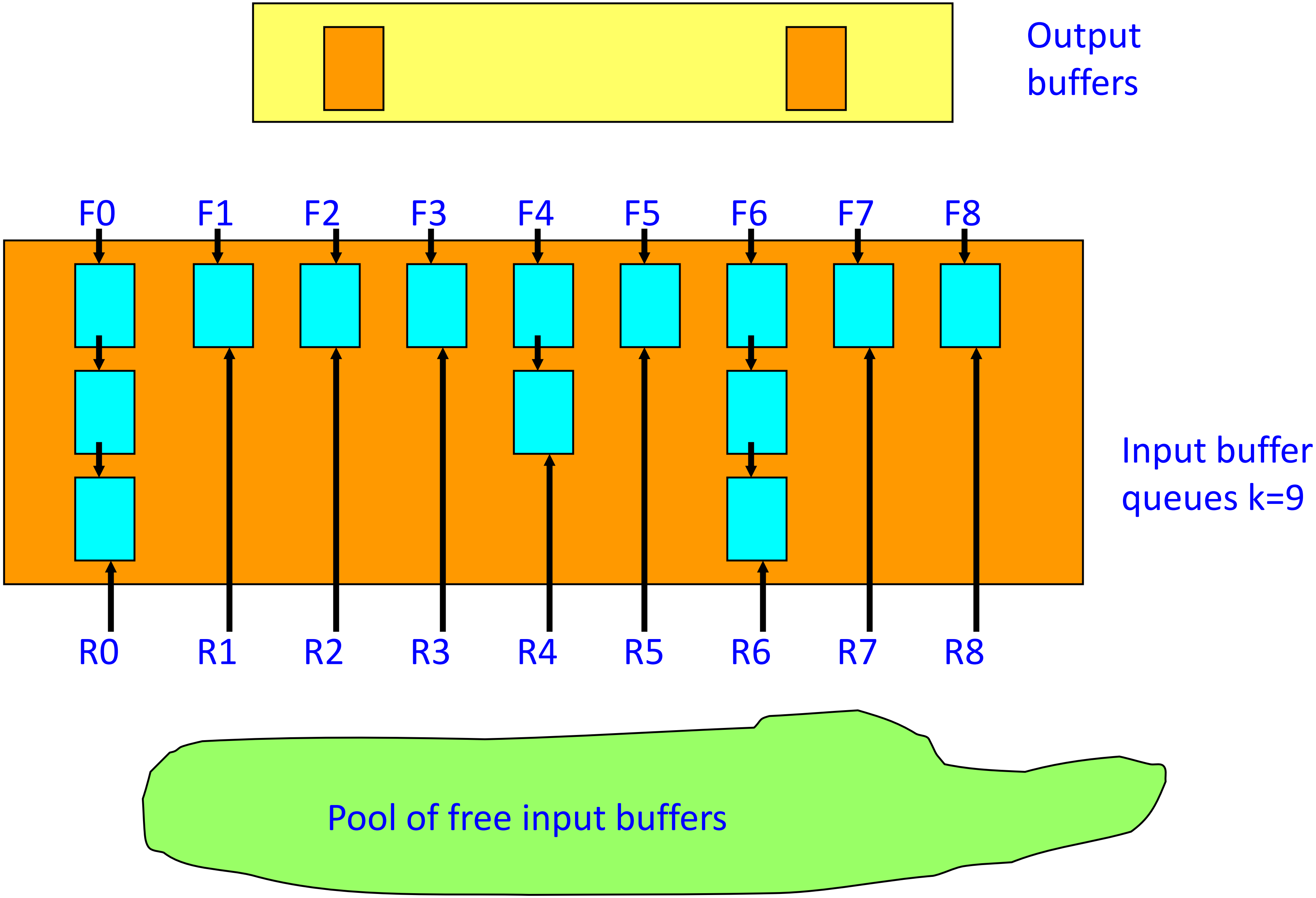
Number Of Input Buffers

- Want to use minimum number of buffers because buffer size (and hence block size) decreases as we increase number of 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



Output buffers

F0 F1 F2 F3 F4 F5 F6 F7 F8

Input buffer queues k=9

R0 R1 R2 R3 R4 R5 R6 R7 R8

Pool of free input buffers

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 the output buffer is not full 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 the output buffer is not full 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)
      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 the output buffer is not full 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 the output buffer is not full 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

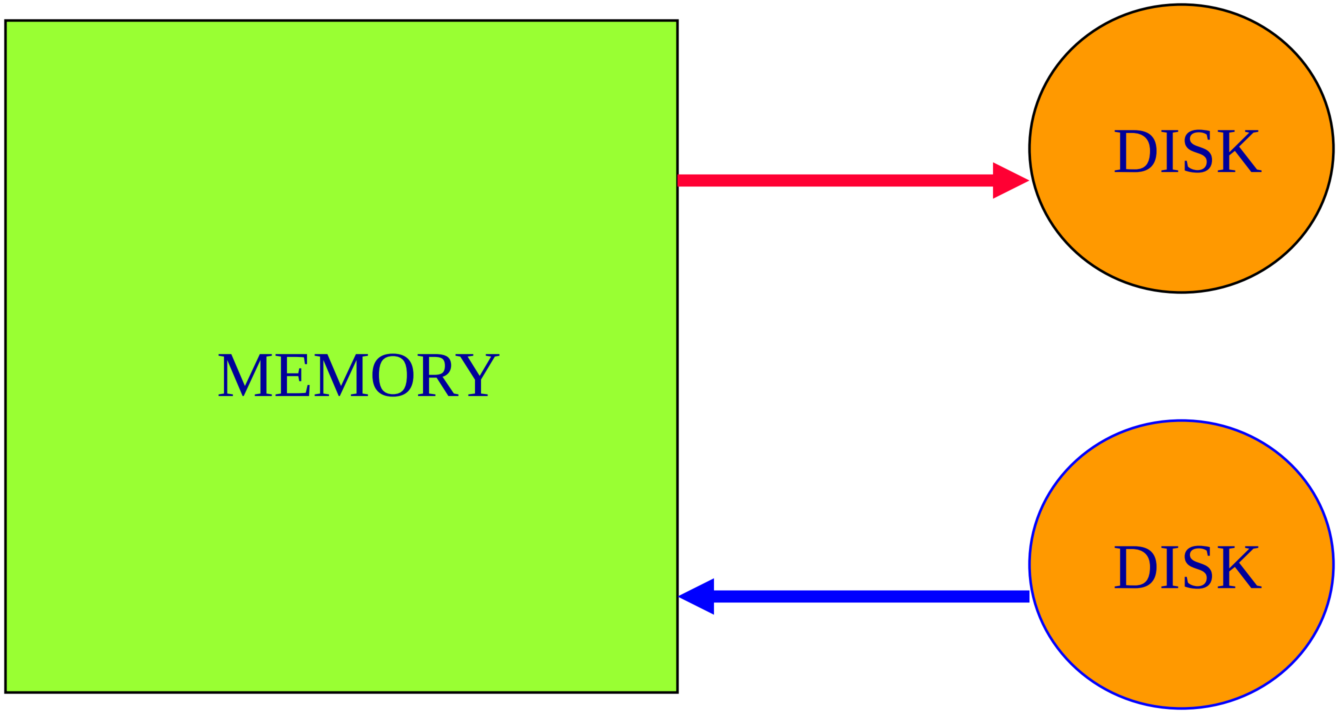
```
if (there is more input from this run)
    initiate read of next block for this run;
```

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;
```

Improve Run Generation

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



Internal Quick Sort

6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot (median of 3).

Input first, middle, and last blocks first.

In-place partitioning.

4	2	3	5	1	6	10	11	9	7	8

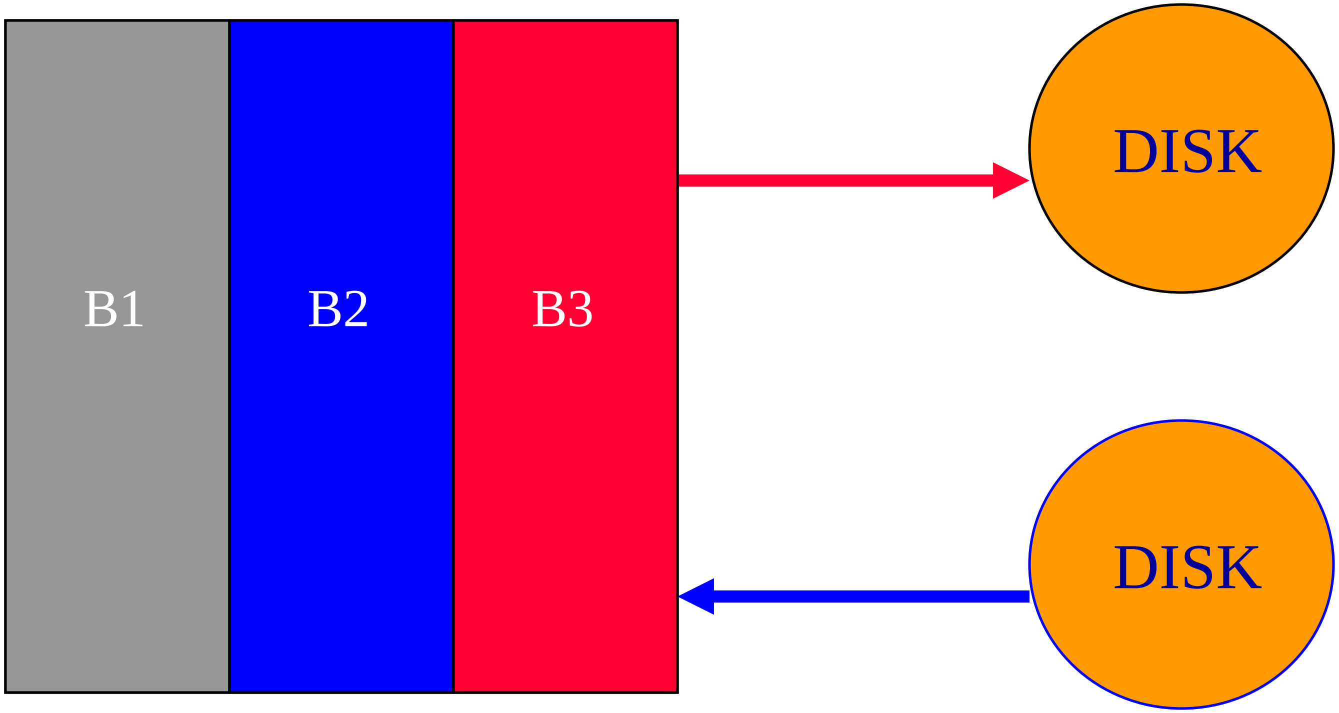
Input blocks from the ends toward the middle.

Sort left and right groups recursively.

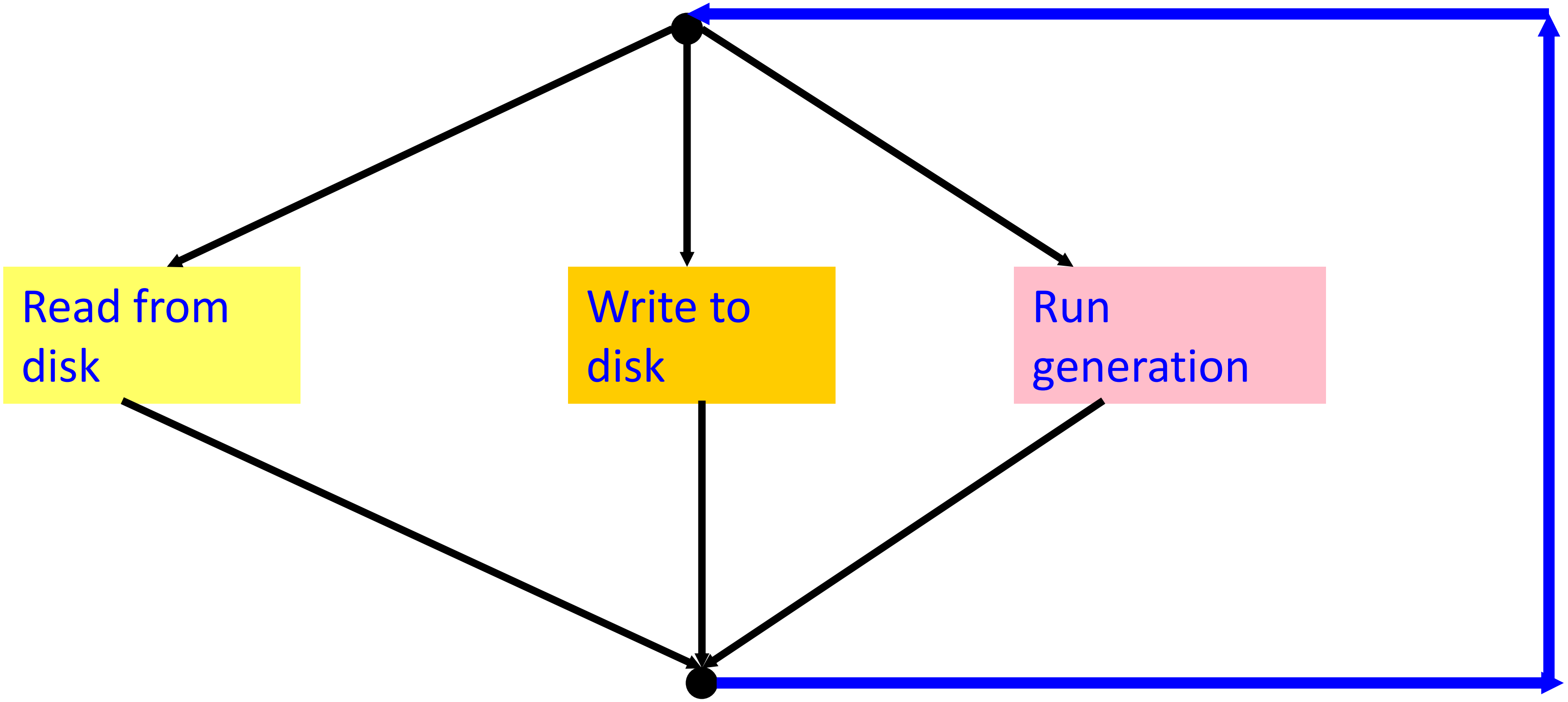
Can begin output as soon as left most block is ready.

Alternative Internal Sort Scheme

Partition into 3 buffers.

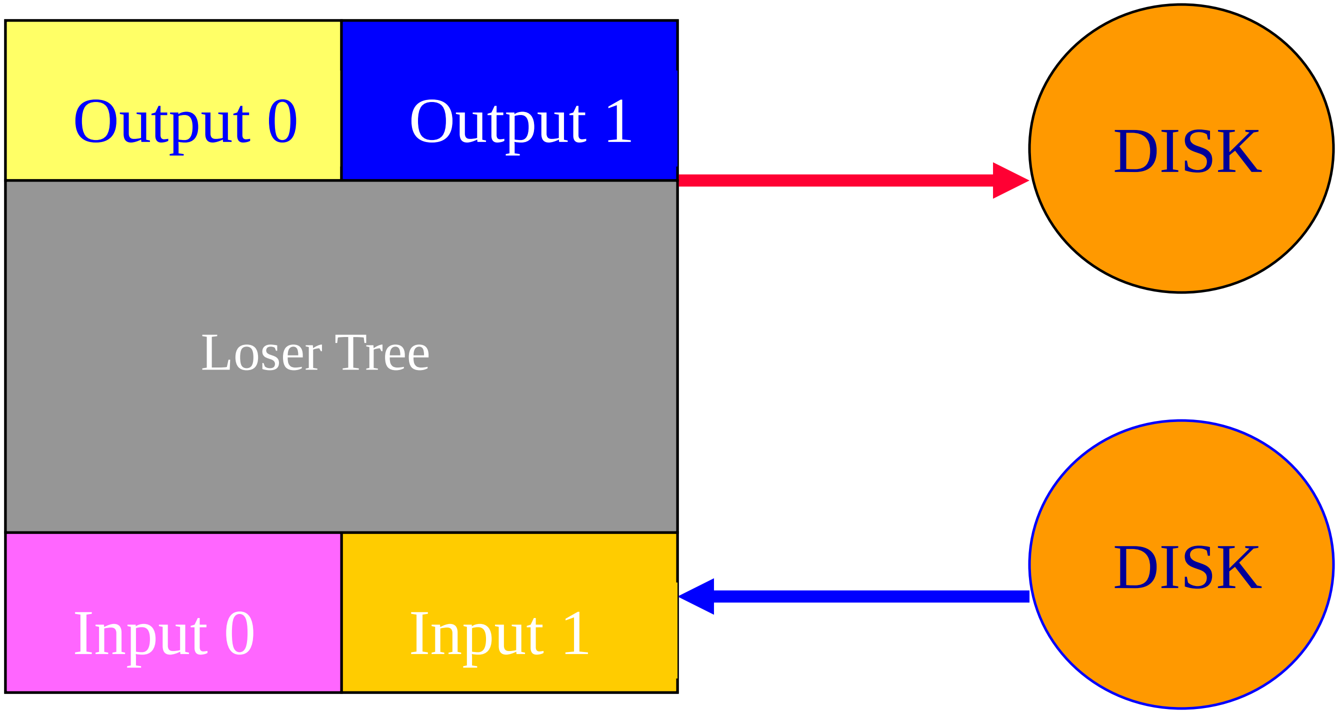


Steady State Operation



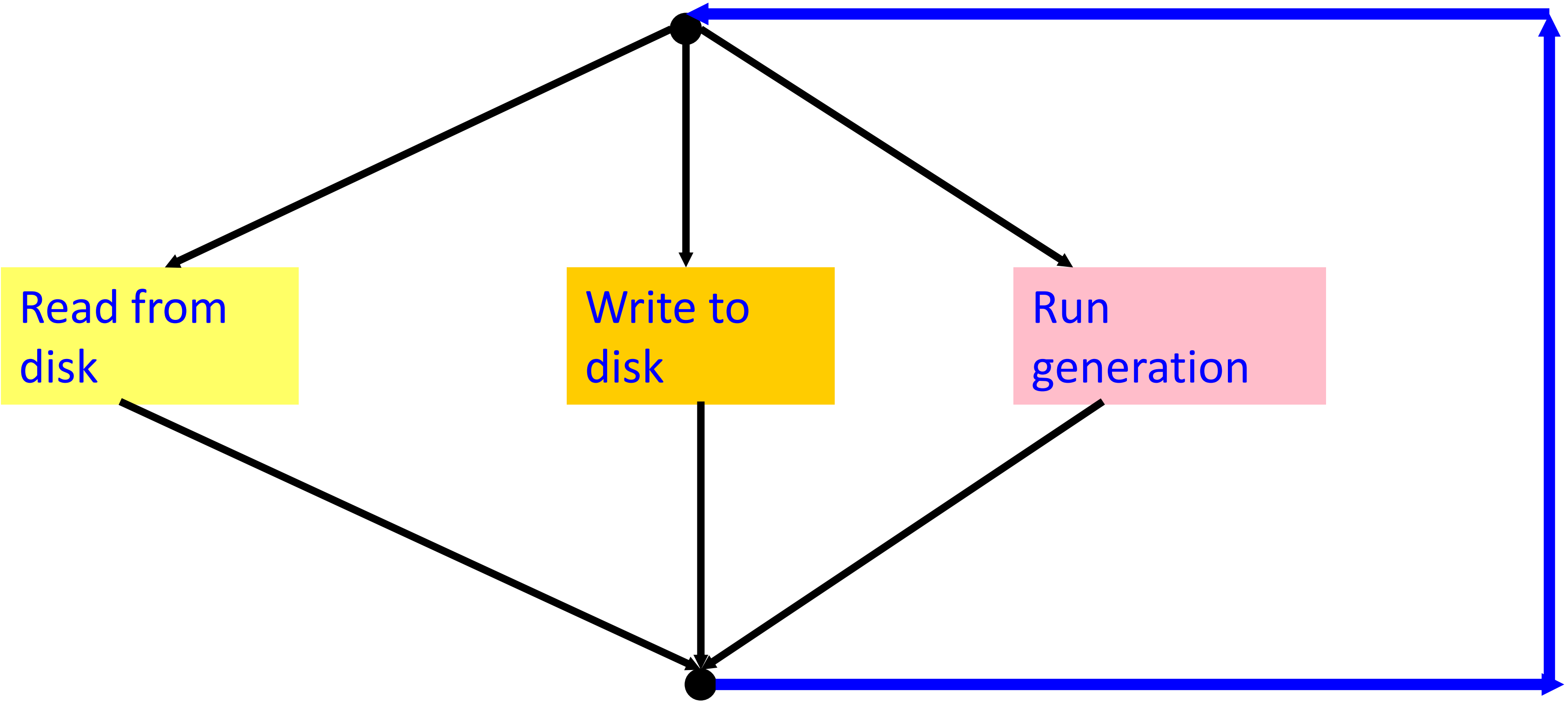
- Synchronization is done when the active input buffer gets empty (the active output buffer will be full at this time).

New Strategy



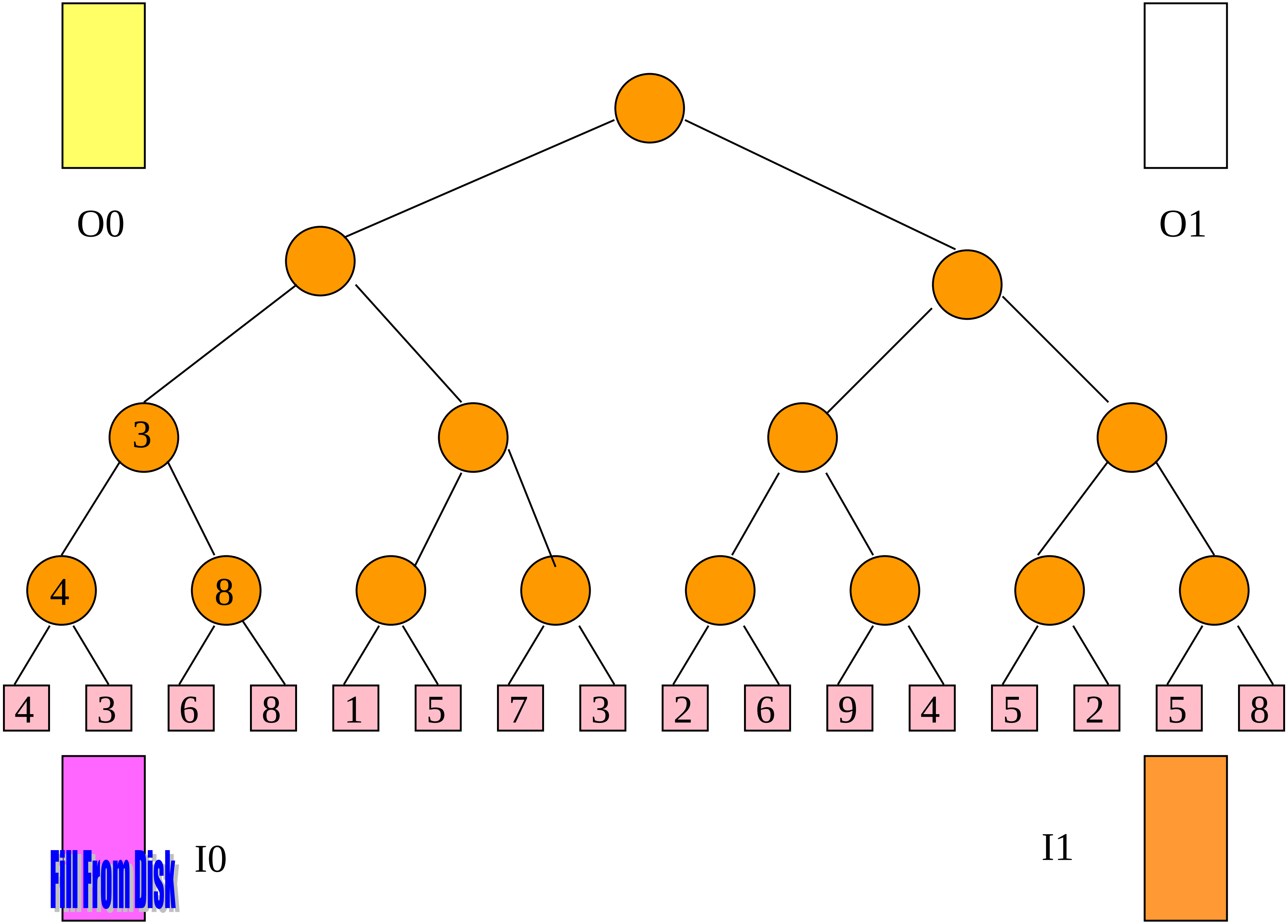
- Use 2 input and 2 output buffers.
- Rest of memory is used for a min loser tree.

Steady State Operation

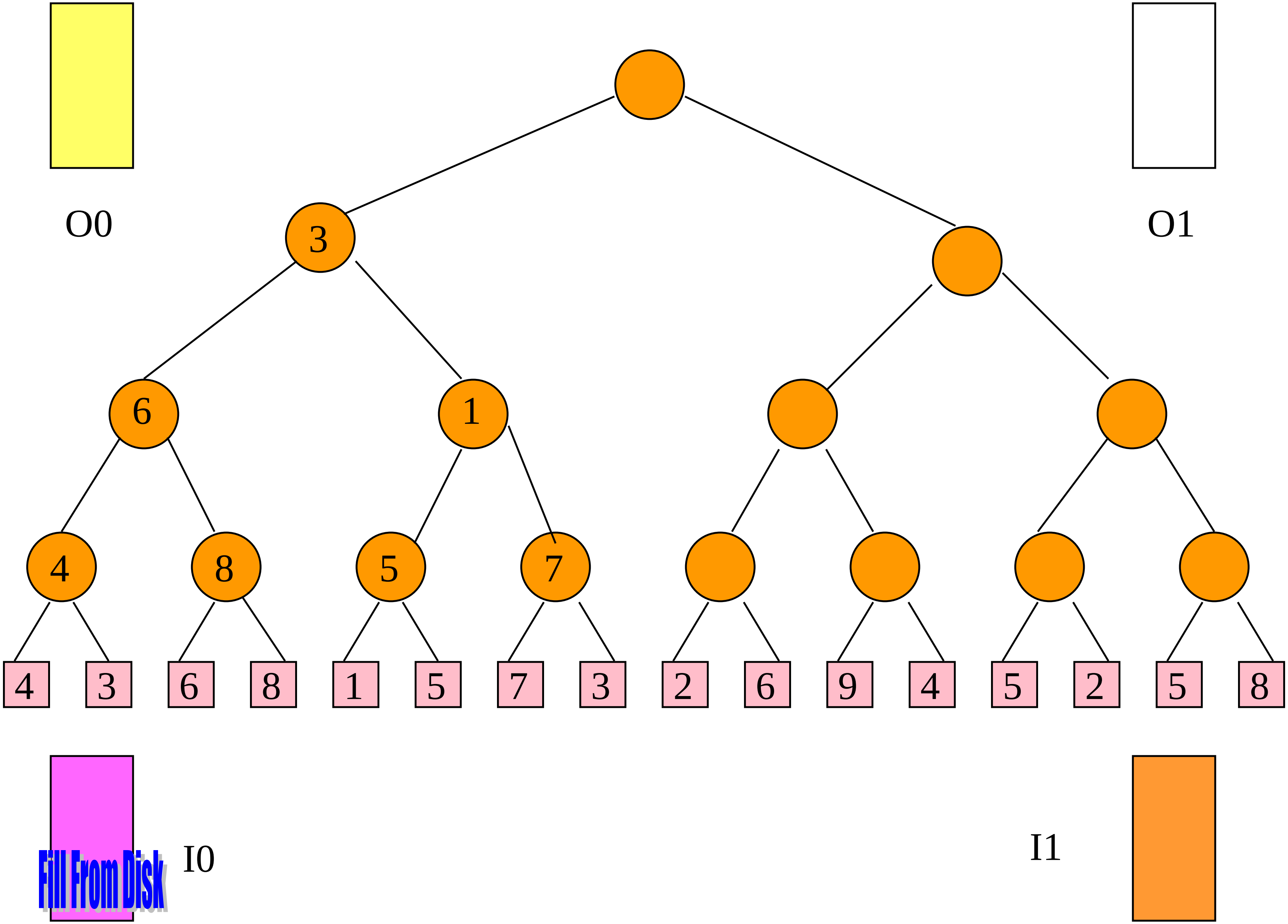


- Synchronization is done when the active input buffer gets empty (the active output buffer will be full at this time).

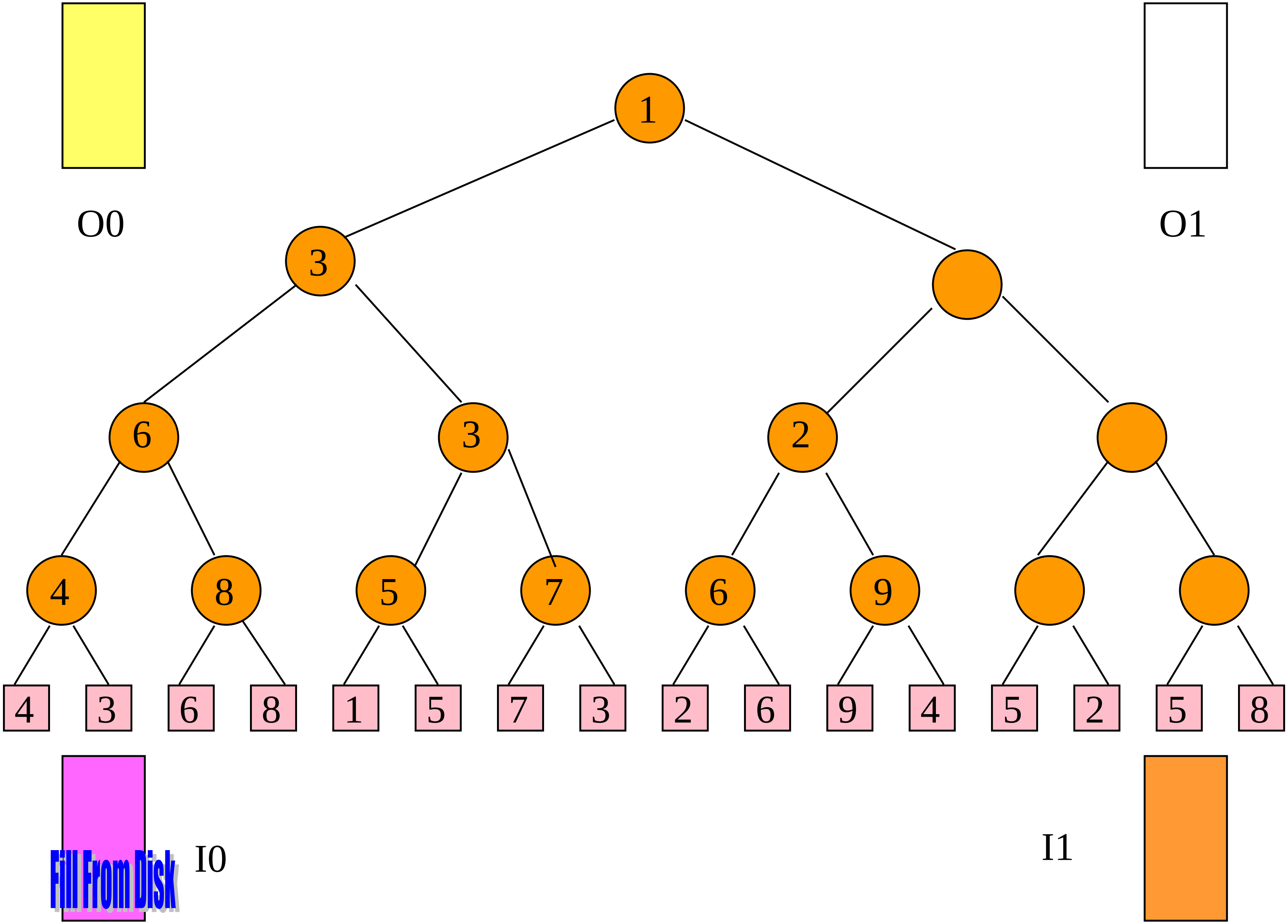
Initialize



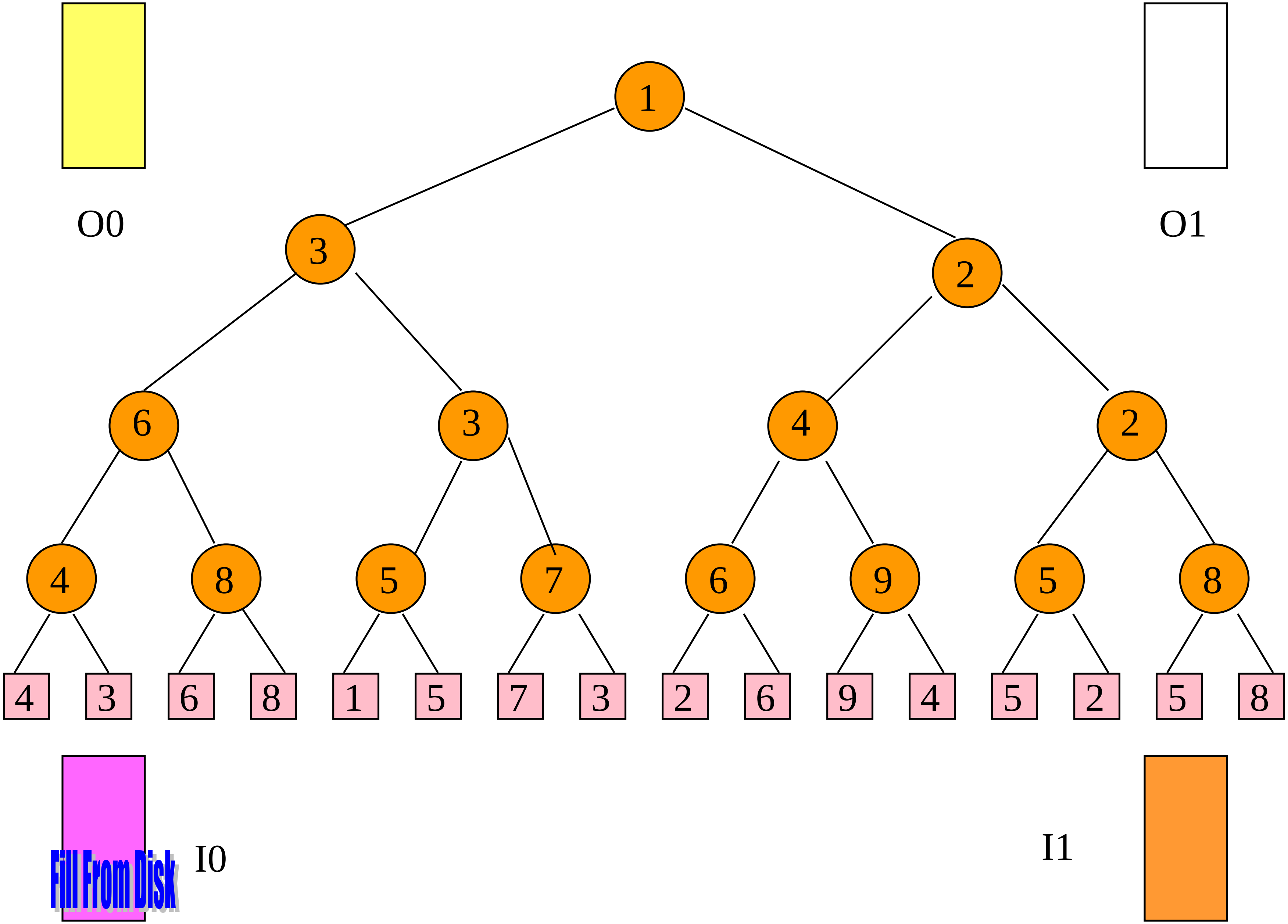
Initialize



Initialize

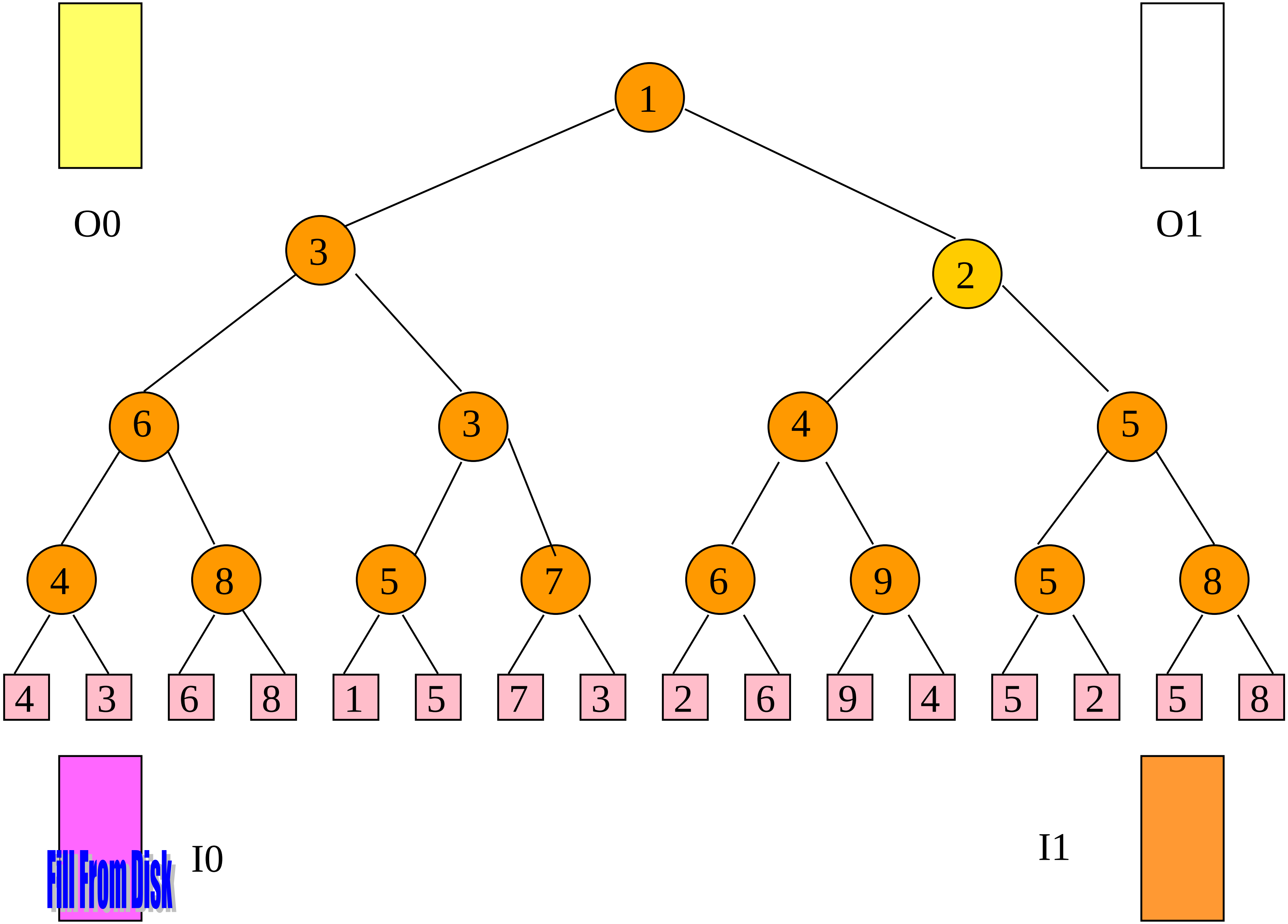


Initialize

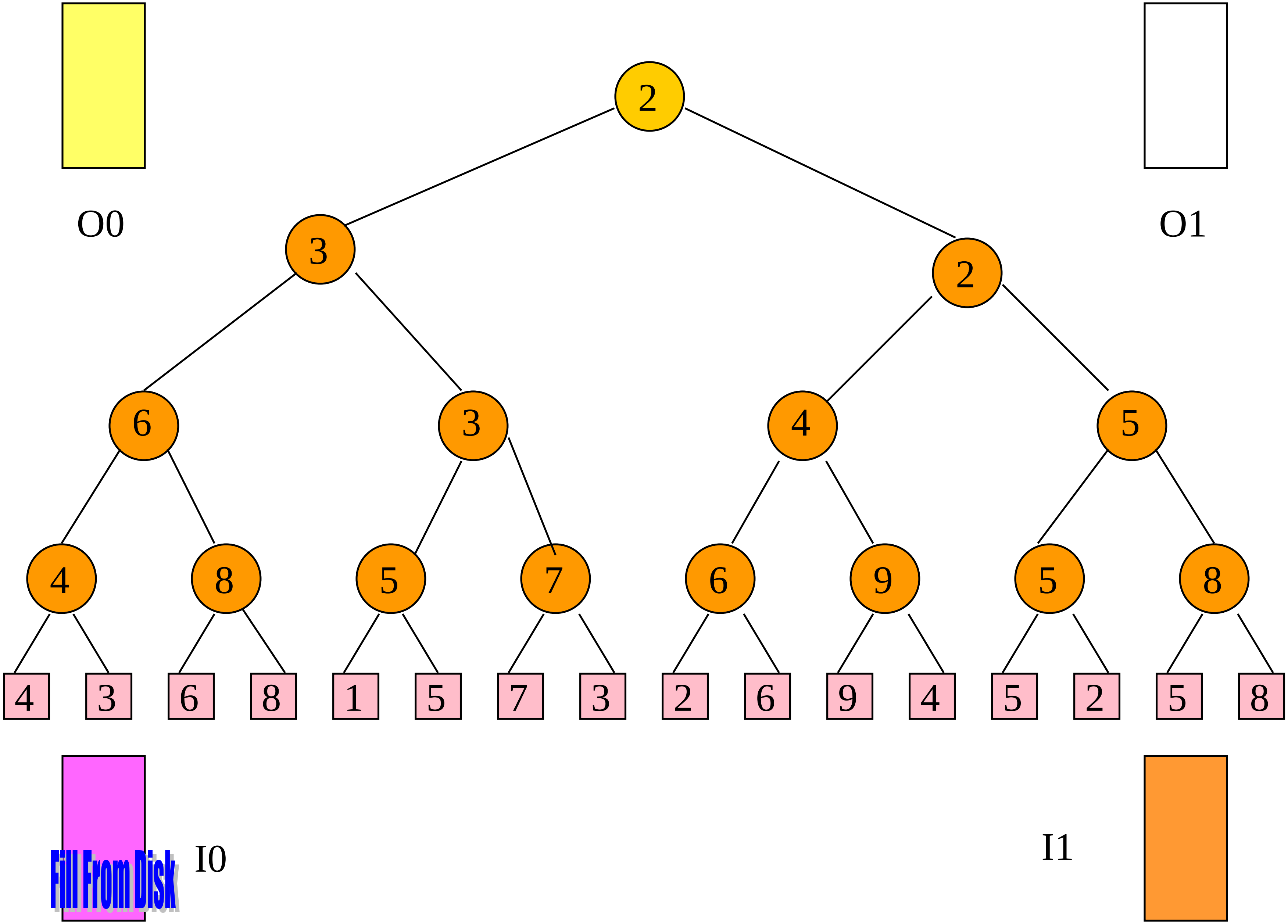


O0
O1
1
3
2
6
3
4
2
4
8
5
7
6
9
5
8
4
3
6
8
1
5
7
3
2
6
9
4
5
2
5
8
Fill From Disk
I0
I1

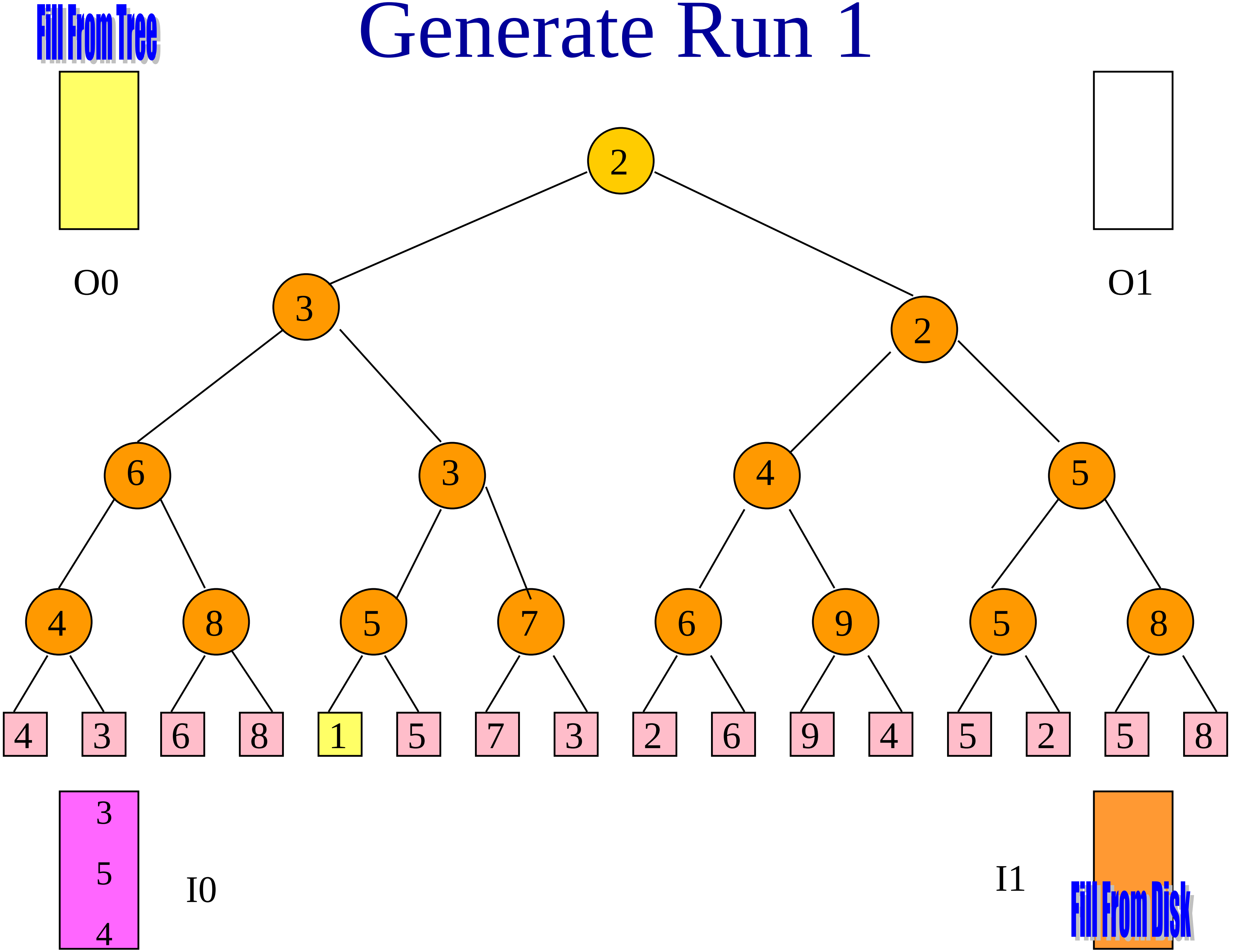
Initialize


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

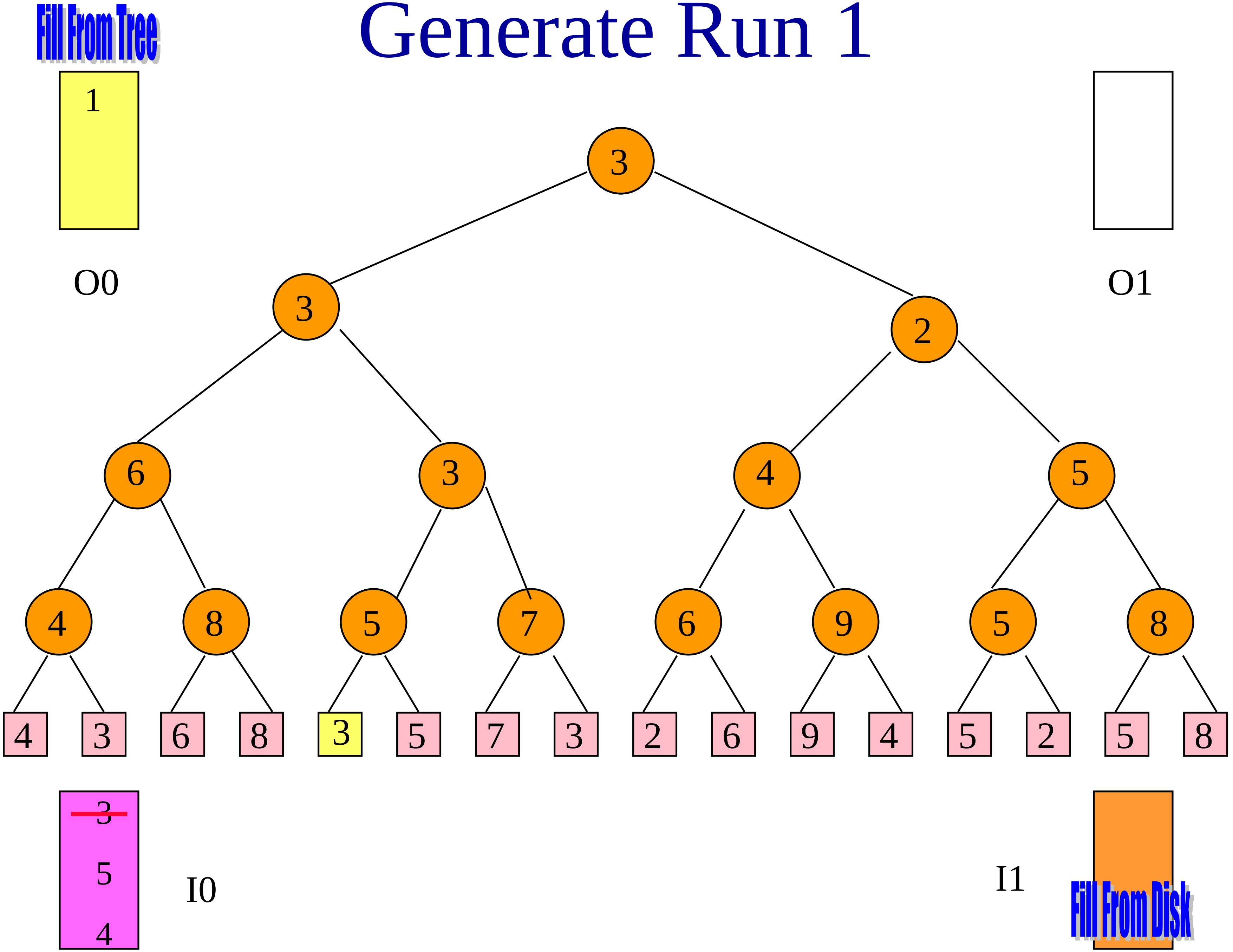
Initialize


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
Fill From Disk
I0
I1

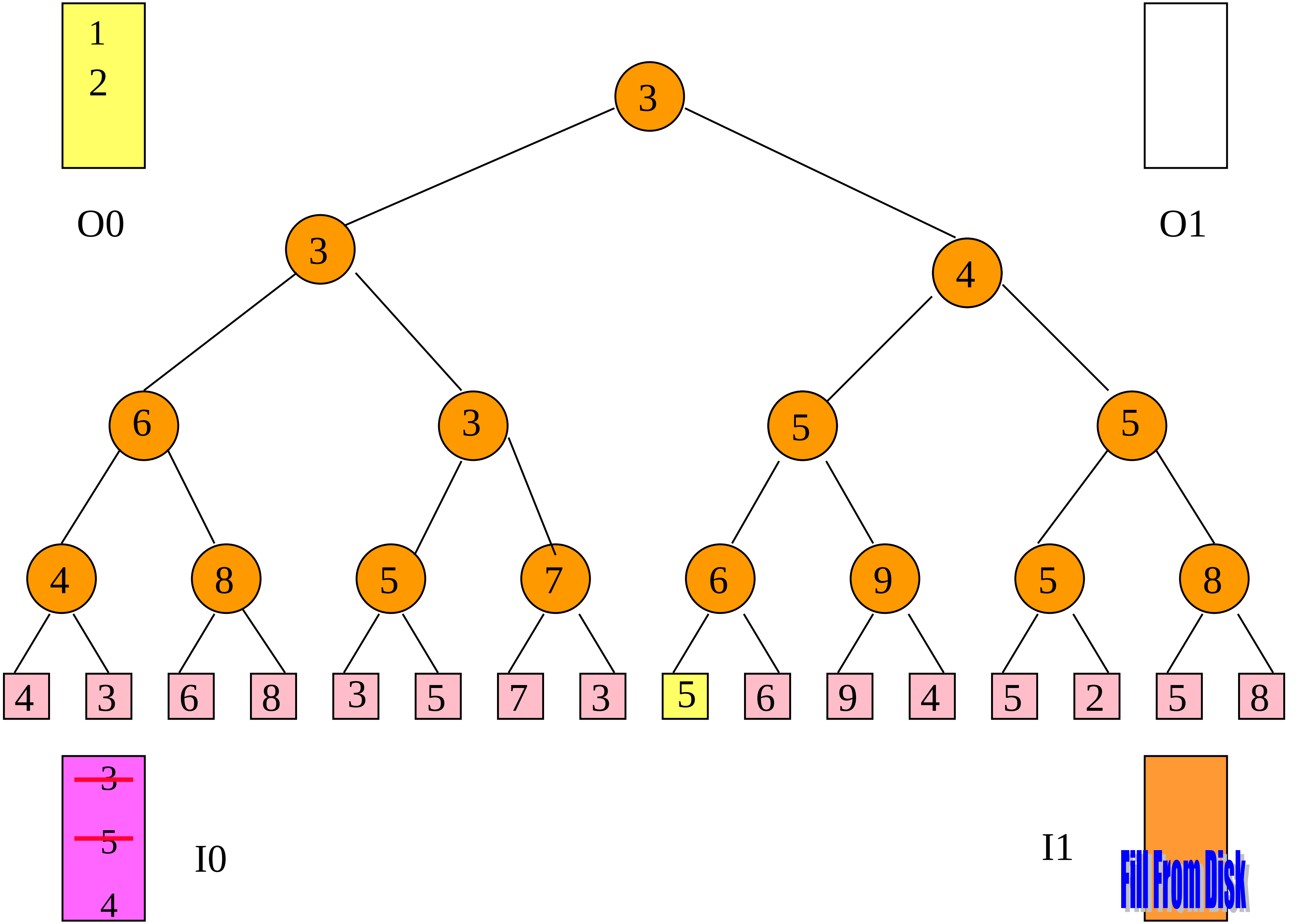
Generate Run 1



Generate Run 1

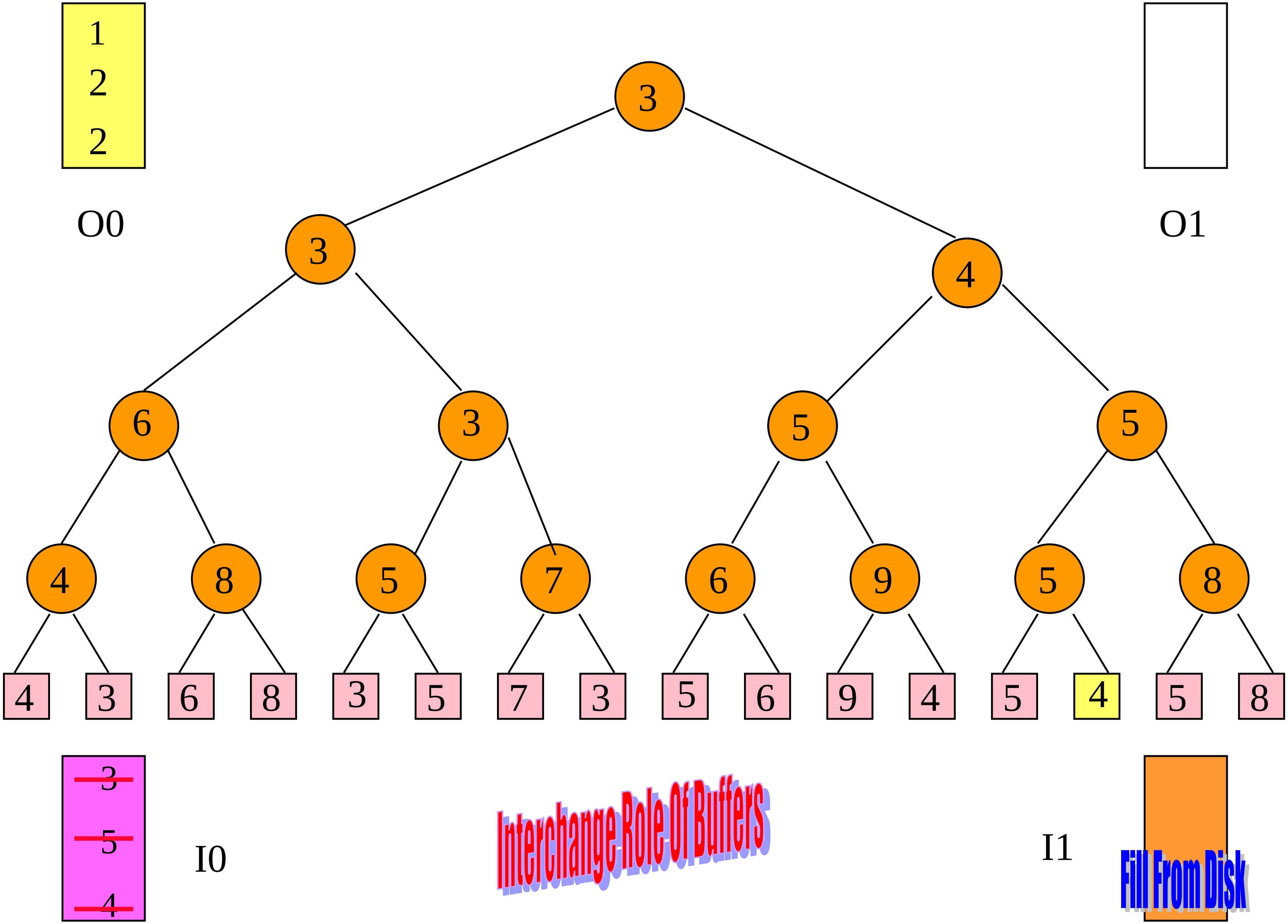


Generate Run 1



Fill From Tree

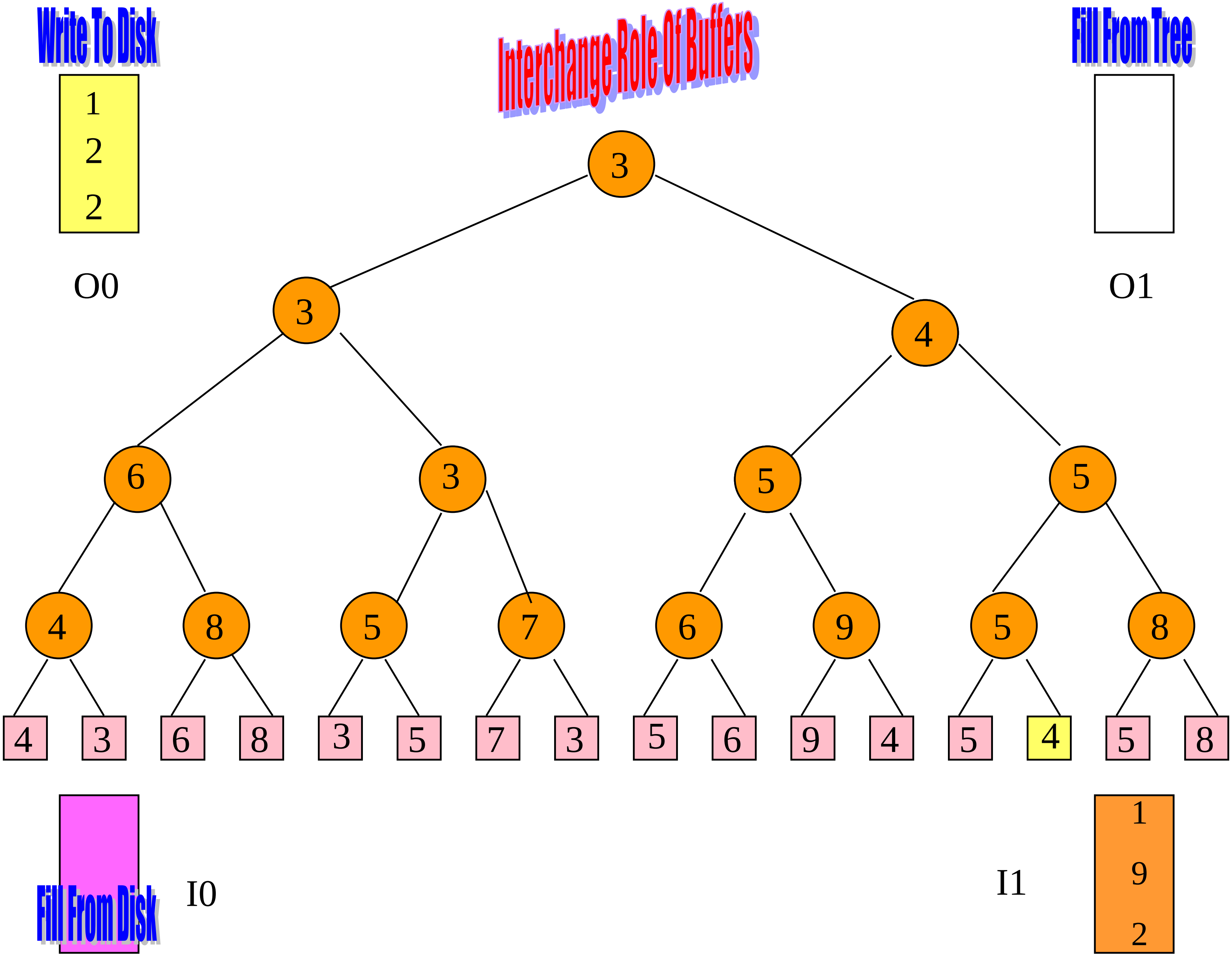
Generate Run 1



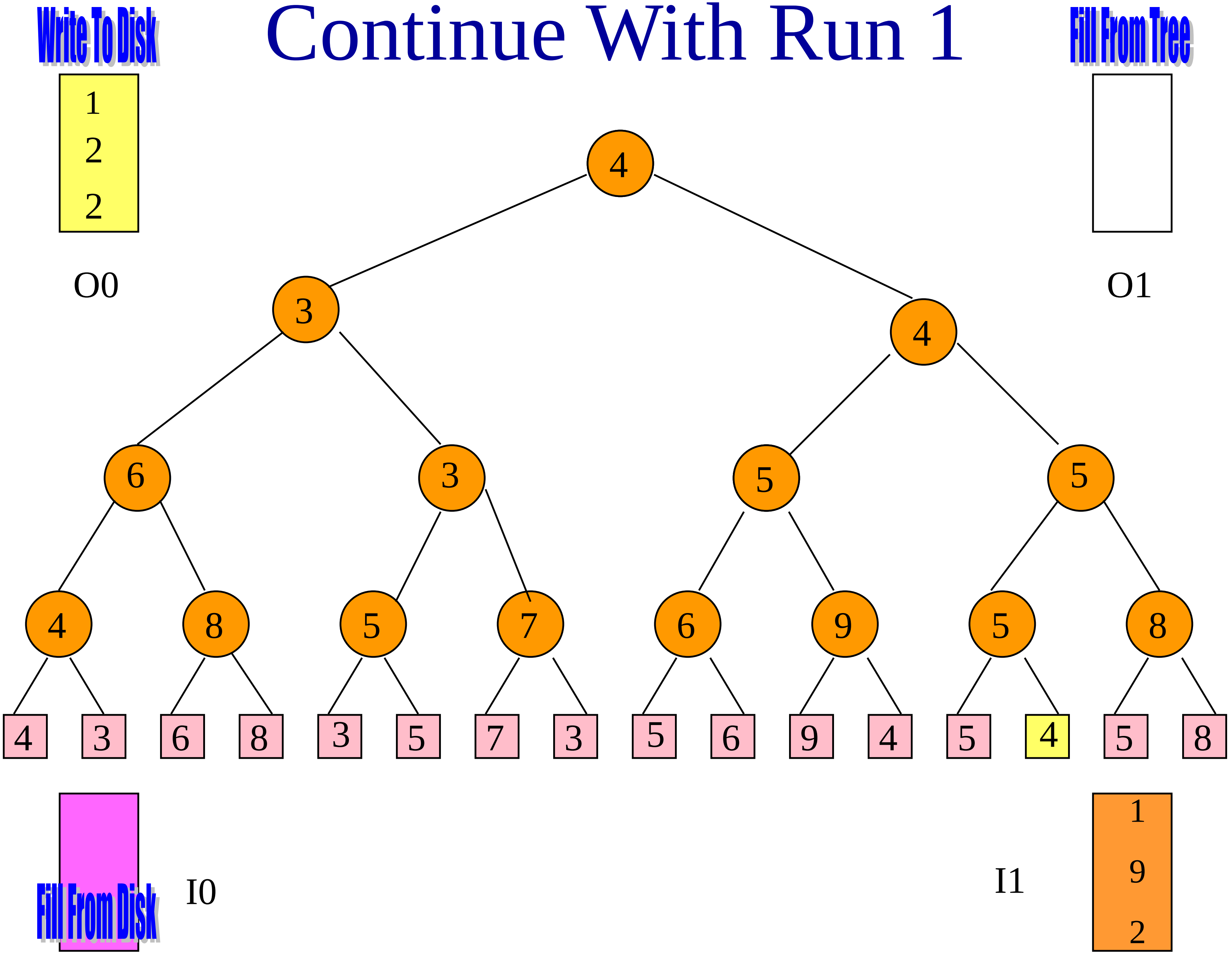
Interchange Role Of Buffers

Fill From Disk

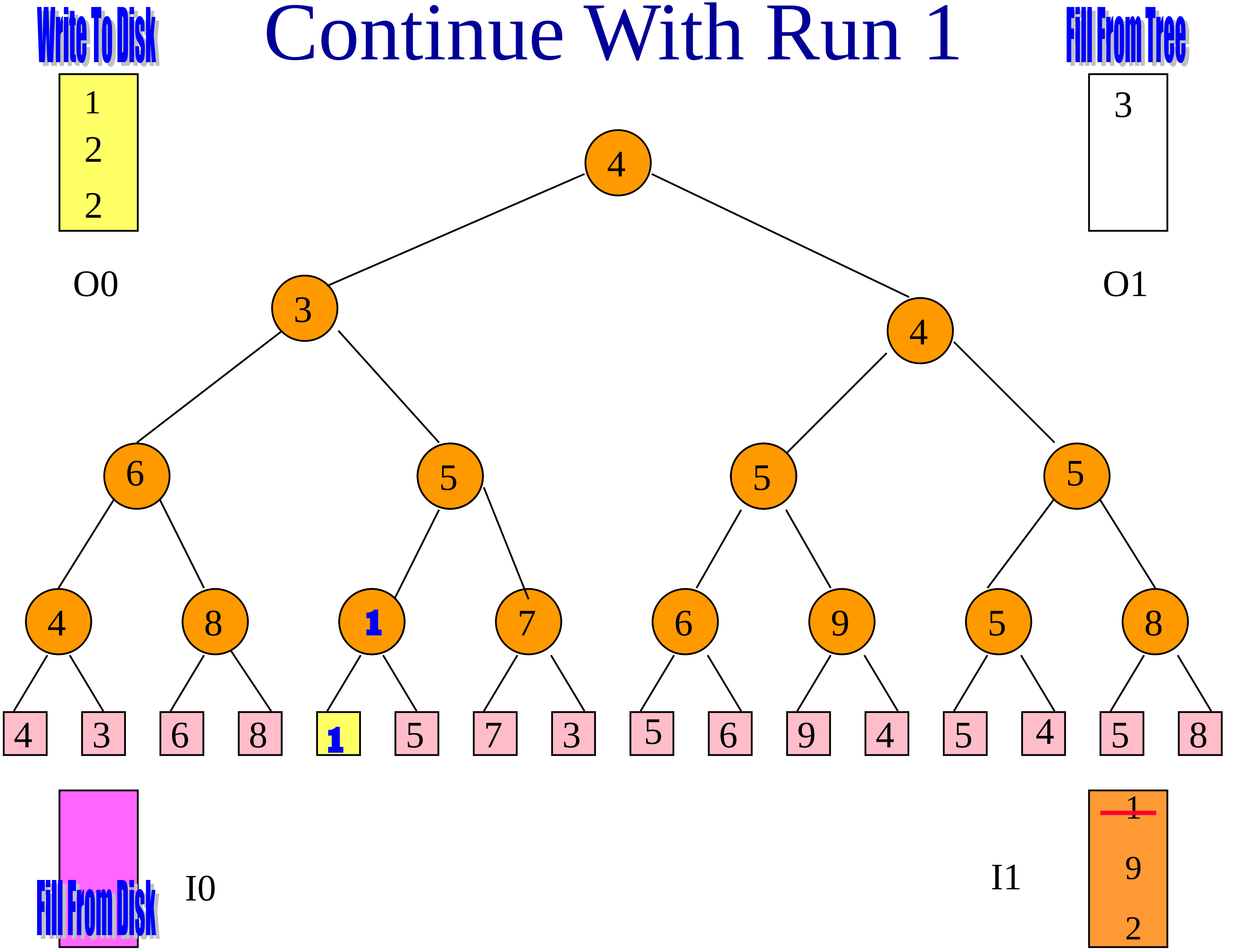
Interchange Role Of Buffers



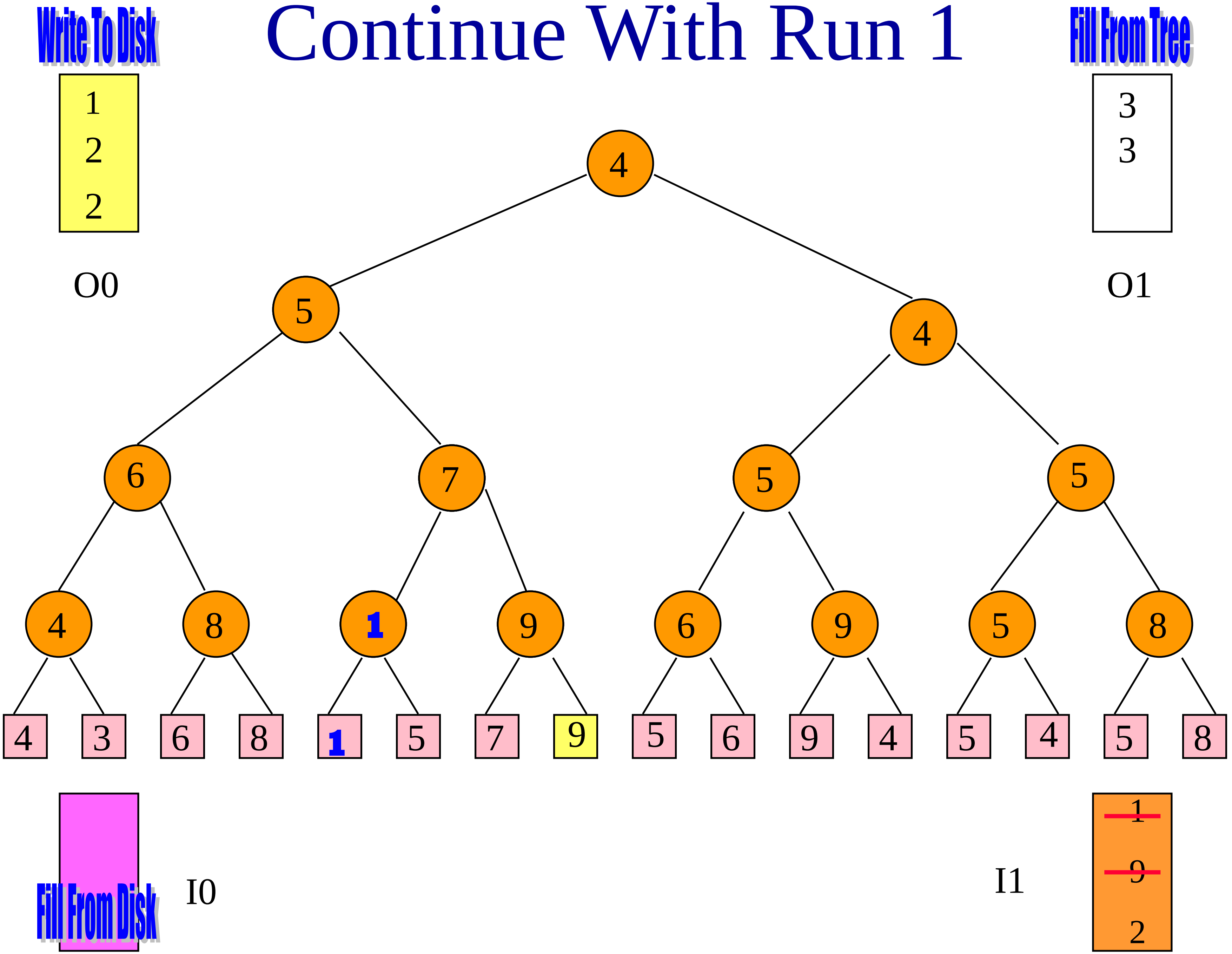
Continue With Run 1



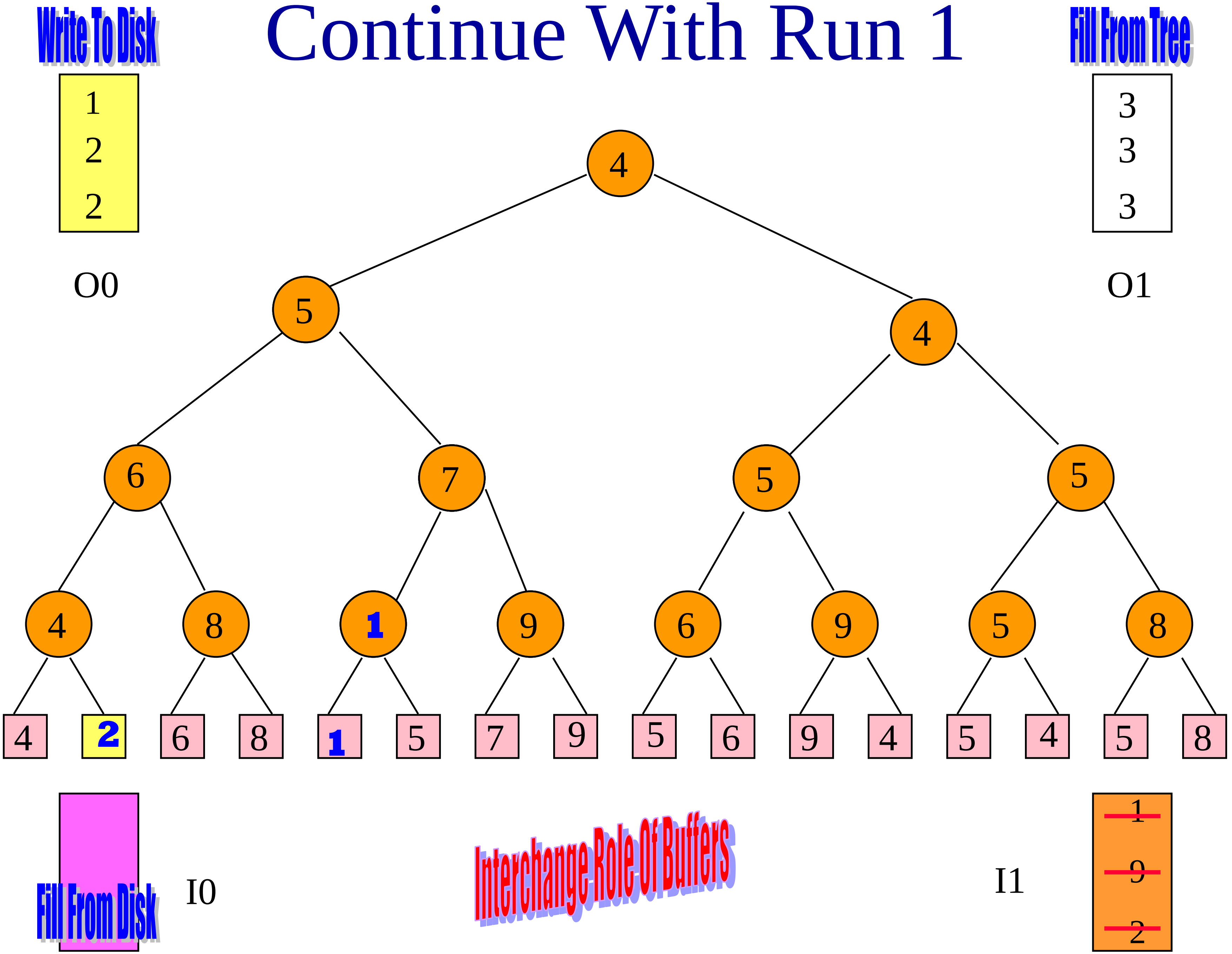
Continue With Run 1



Continue With Run 1



Continue With Run 1

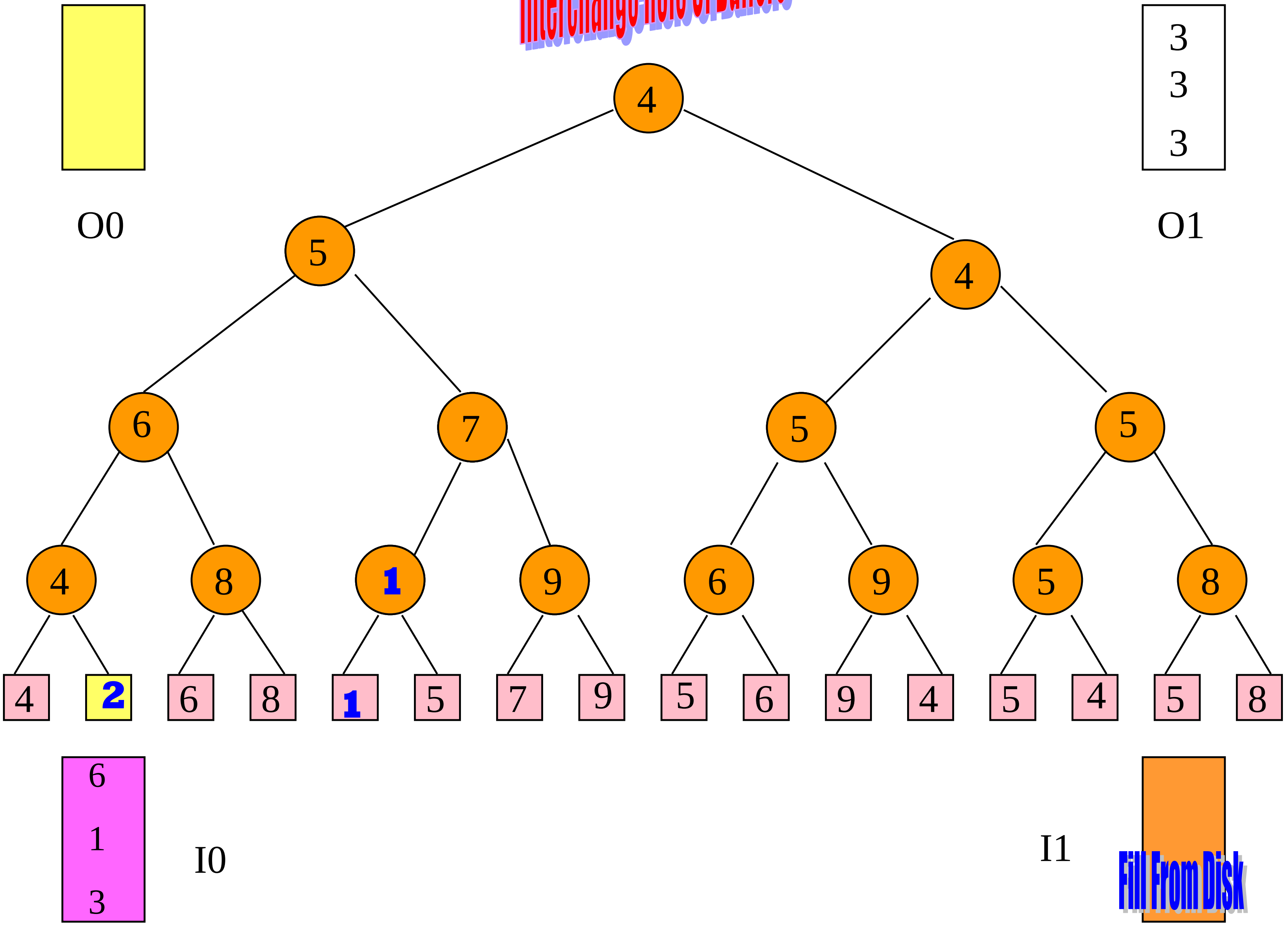


Interchange Role Of Buffers

Interchange Role Of Buffers

Fill From Tree

Write To Disk



O0

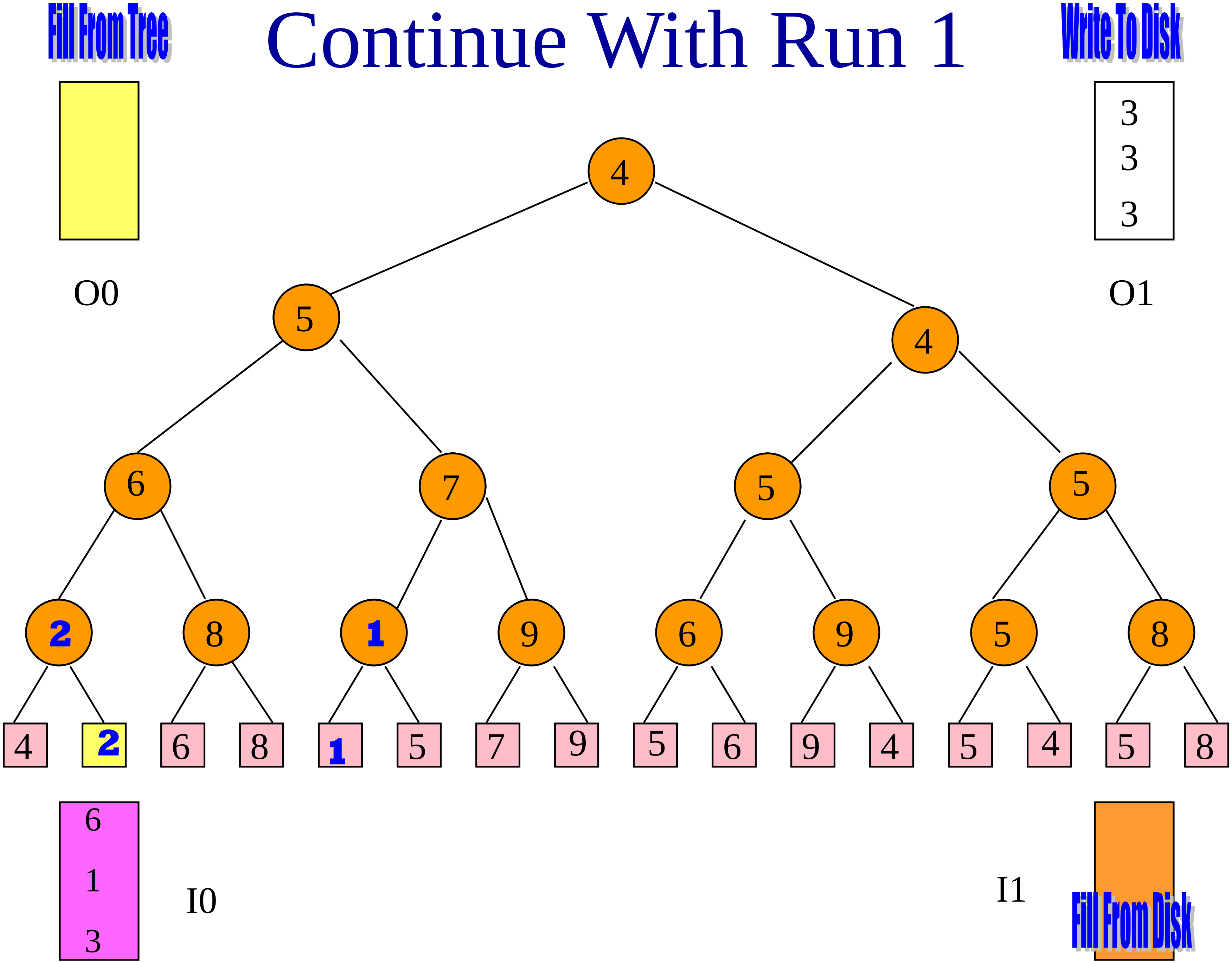
O1

I0

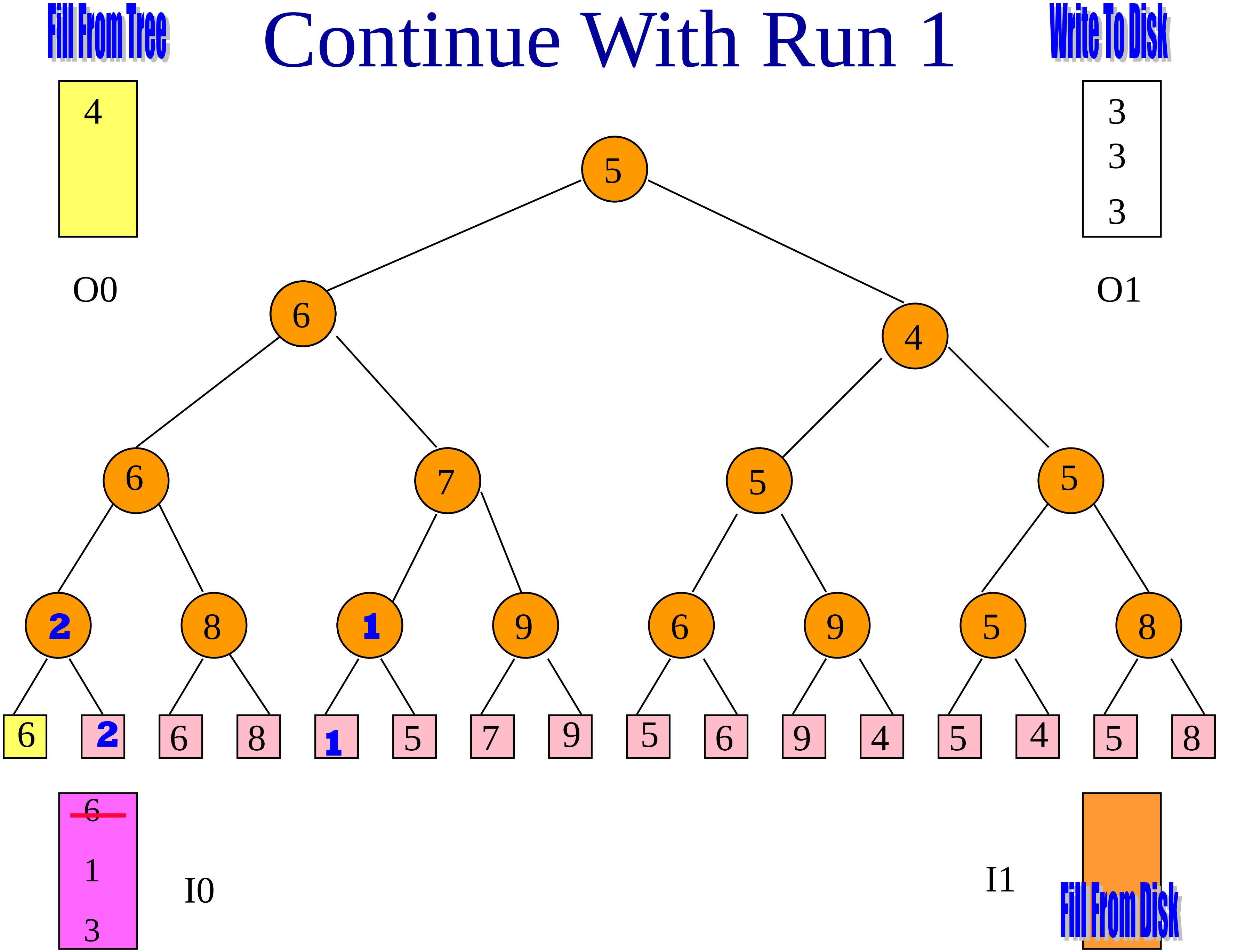
I1

Fill From Disk

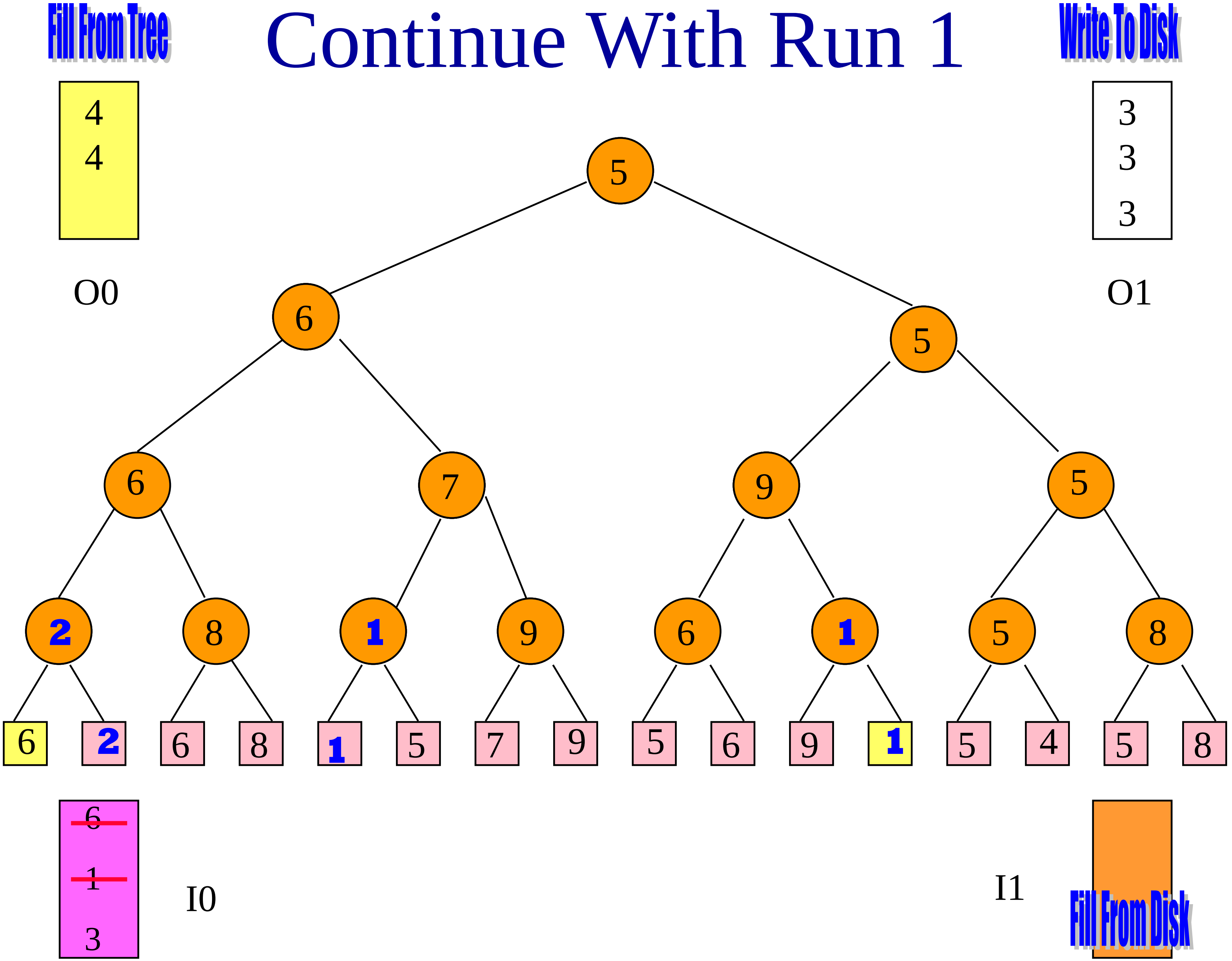
Continue With Run 1



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.

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 4 buffers (4b records).
  - Loser tree k = m – 4b.
  - Average run size = 2k = 2(m – 4b).
  - 2k >= m when m >= 8b.

Comparison

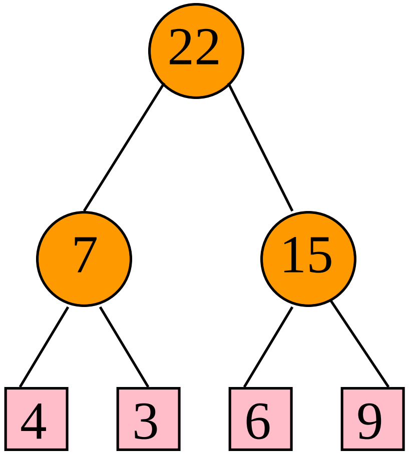
- Assume $b = 100$.

m	600	1000	5000	10000
k	200	600	4600	9600
$2k$	400	1200	9200	19200

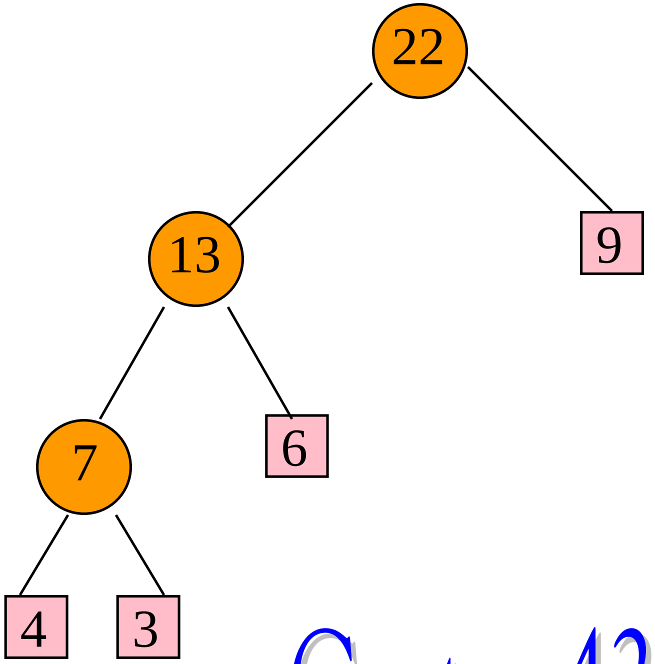
Comparison

- 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



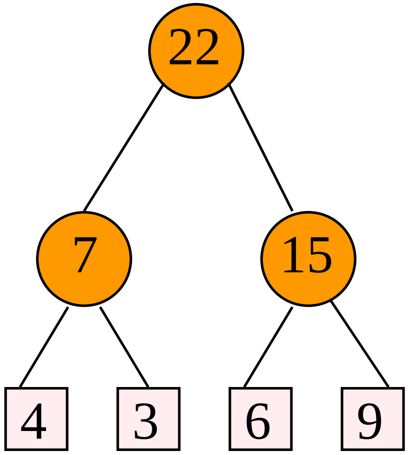
Cost = 44



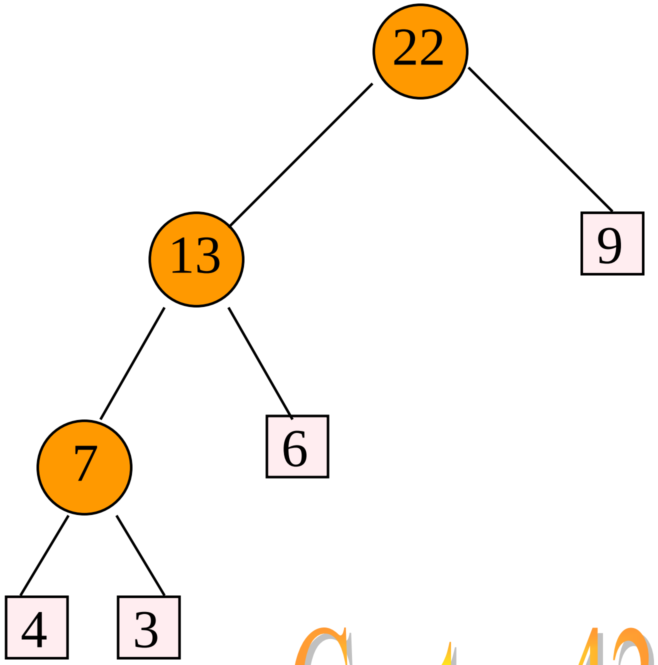
Cost = 42

Best merge order?

Optimal Merging Of Runs



Cost = 44

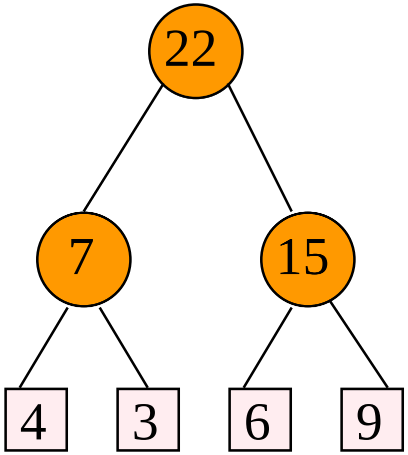


Cost = 42

Best merge order?

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

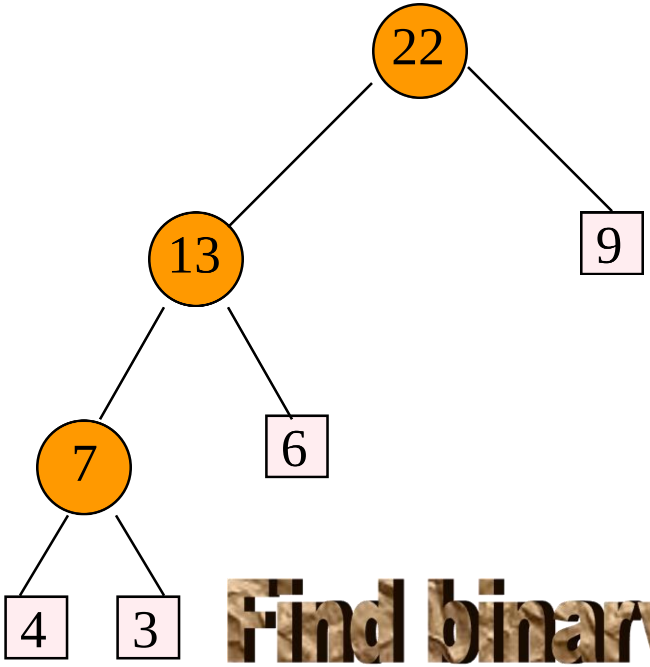
= 44

= 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

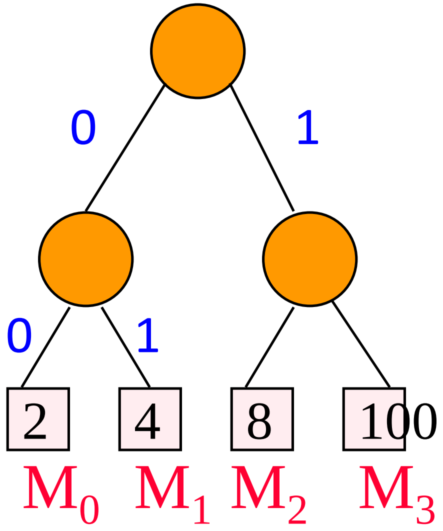
- Messages M_0, M_1, M_2, …, M_{n-1} are to be transmitted.
- The messages do not change.
- Both sender and receiver know the messages.
- So, it is adequate to transmit a code that identifies the message (e.g., message index).
- M_i is sent with frequency f_i.
- Select message codes so as to minimize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

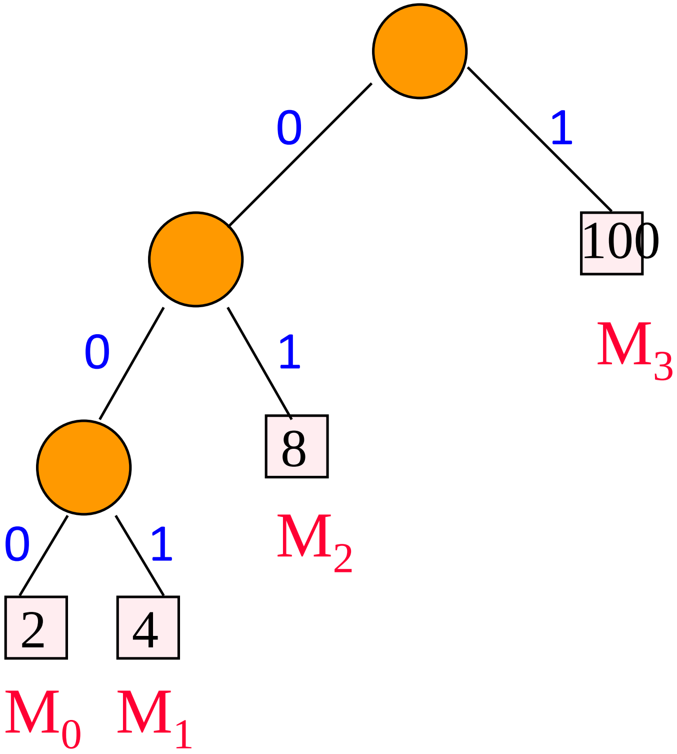
  = 228

  = transmission cost

  = WEPL

Example

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



- Decoding cost

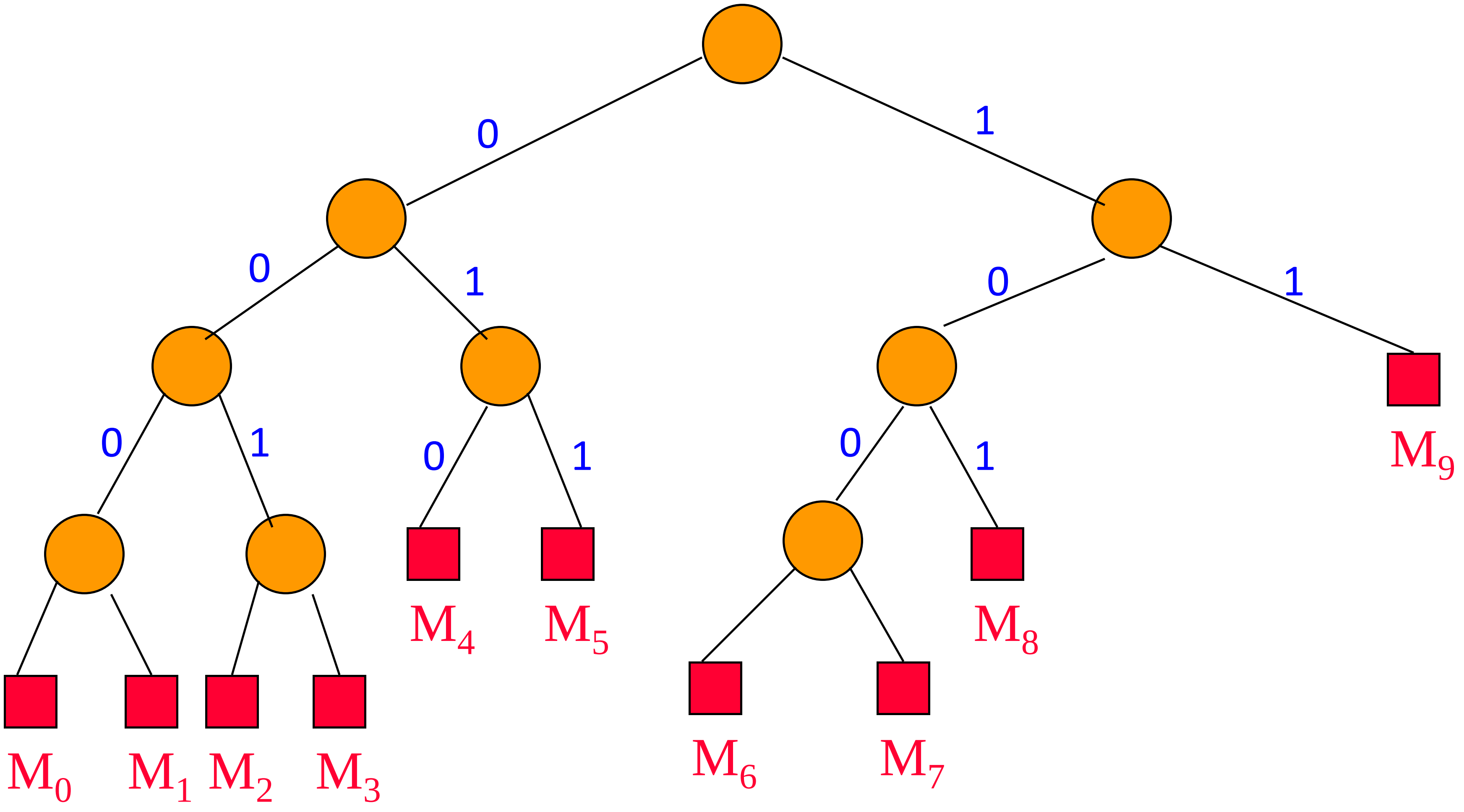
= 2*3 + 4*3 + 8*2 + 100*1

= 144

= transmission cost

= WEPL

Another Example



No code is a prefix of another!

Lossless Data Compression

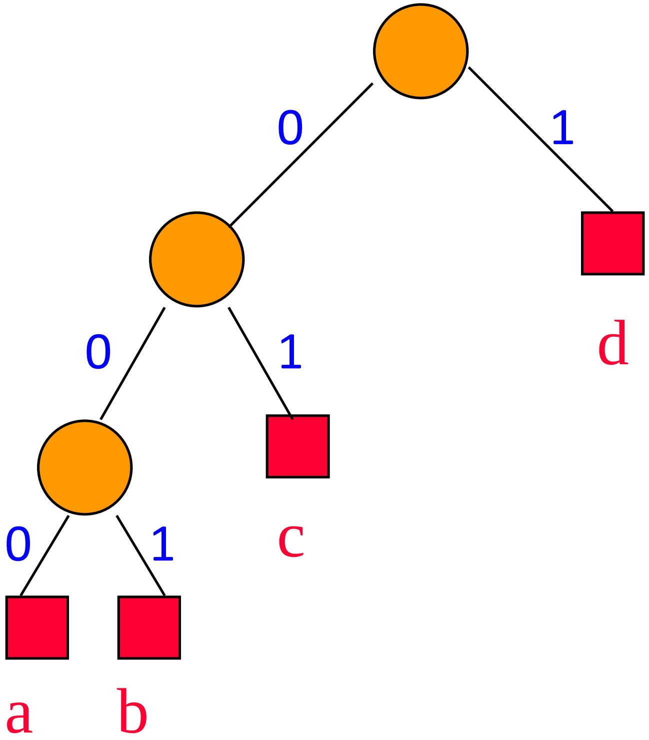
- Alphabet = {a, b, c, d}.
- String with 10 as, 5 bs, 100 cs, and 900 ds.
- Use a 2-bit code.
  - a = 00, b = 01, c = 10, d = 11.
  - Size of string = 10*2 + 5*2 + 100*2 + 900*2
    = 2030 bits.
  - Plus size of code table.

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

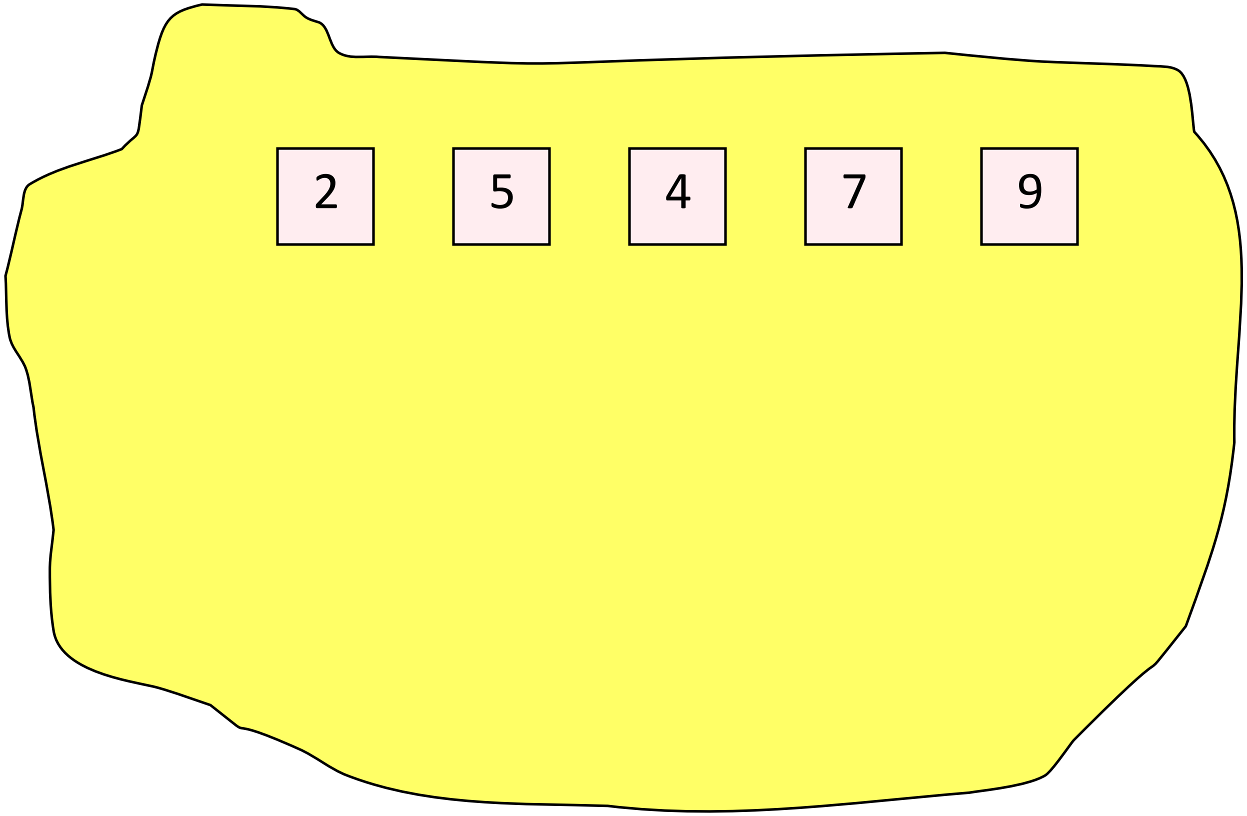
- Trees that have minimum WEPL.
- Binary trees with minimum WEPL may be constructed using a greedy algorithm.
- For higher order trees with minimum WEPL, a preprocessing step followed by the greedy algorithm may be used.
- Huffman codes: codes defined by minimum WEPL trees.

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.
  - Select 2 trees with minimum weight.
  - 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.

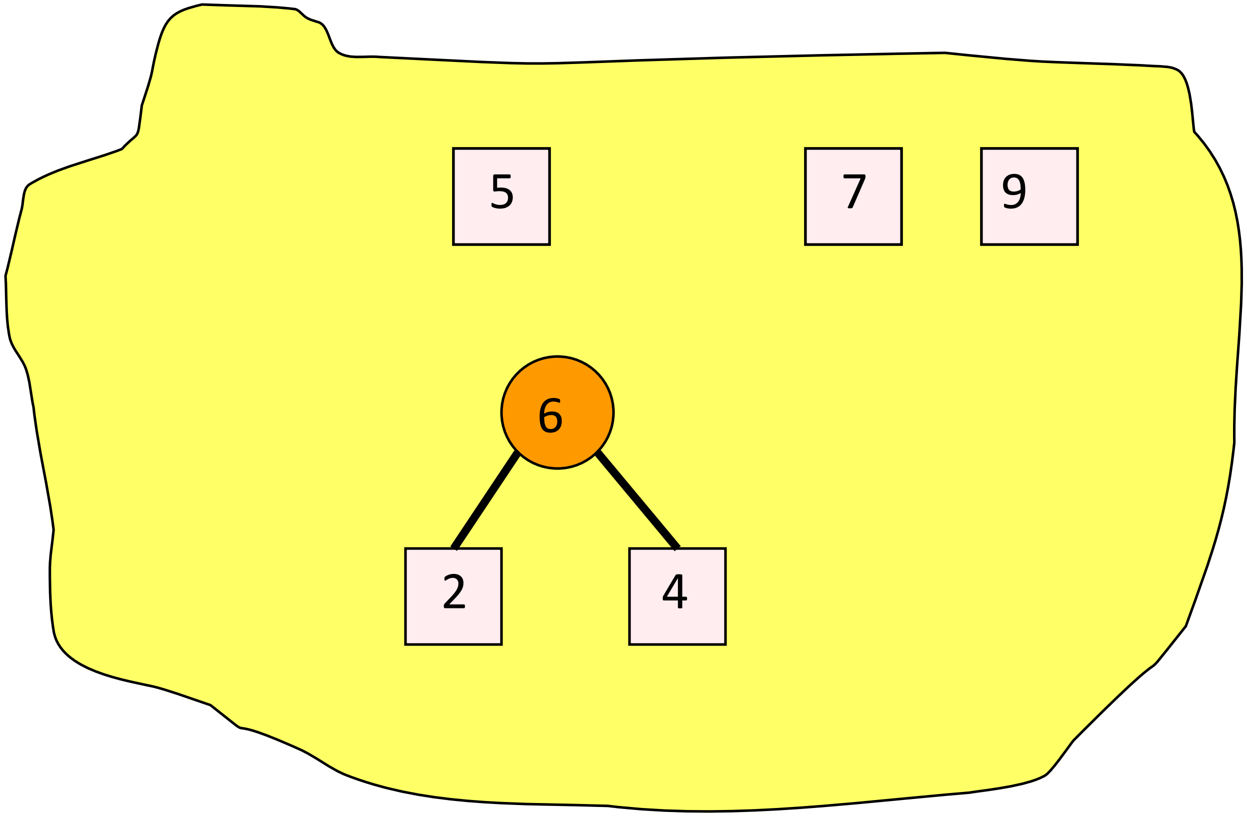
Example

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



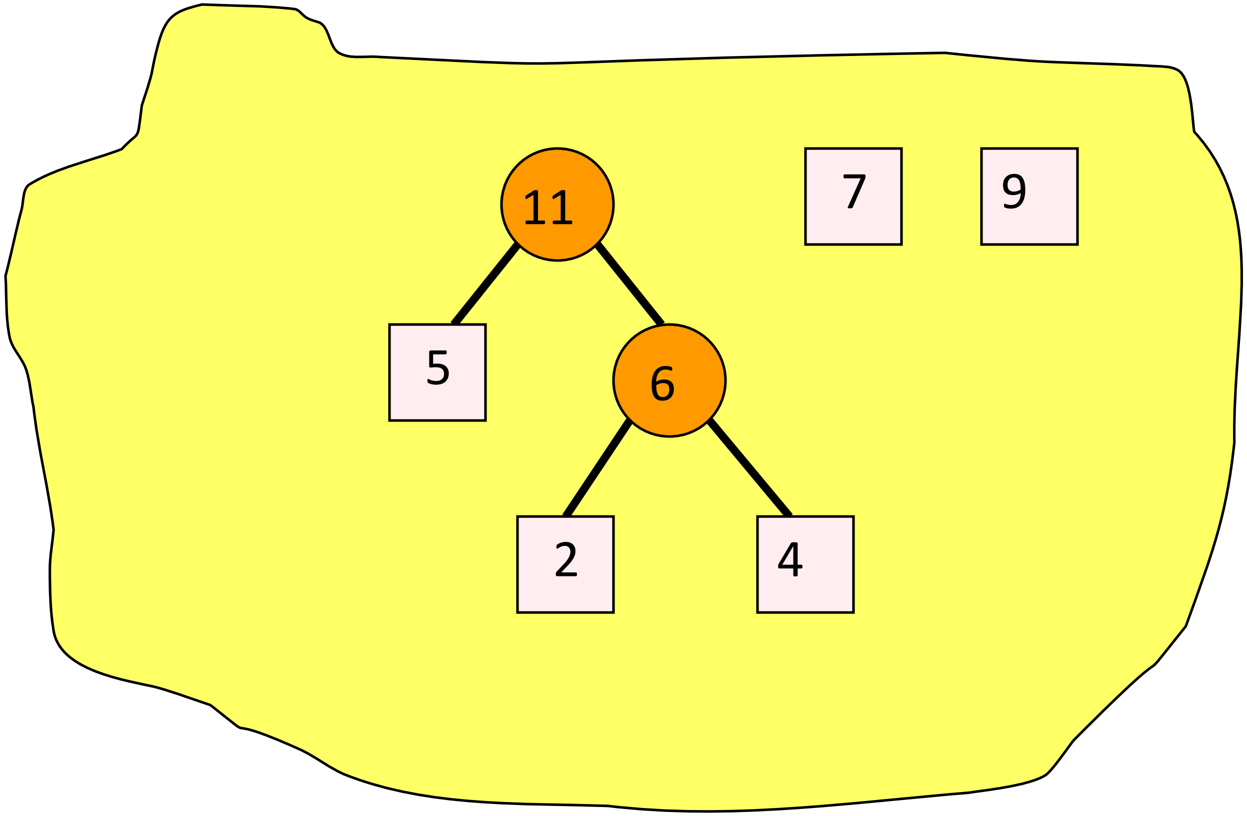
Example

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



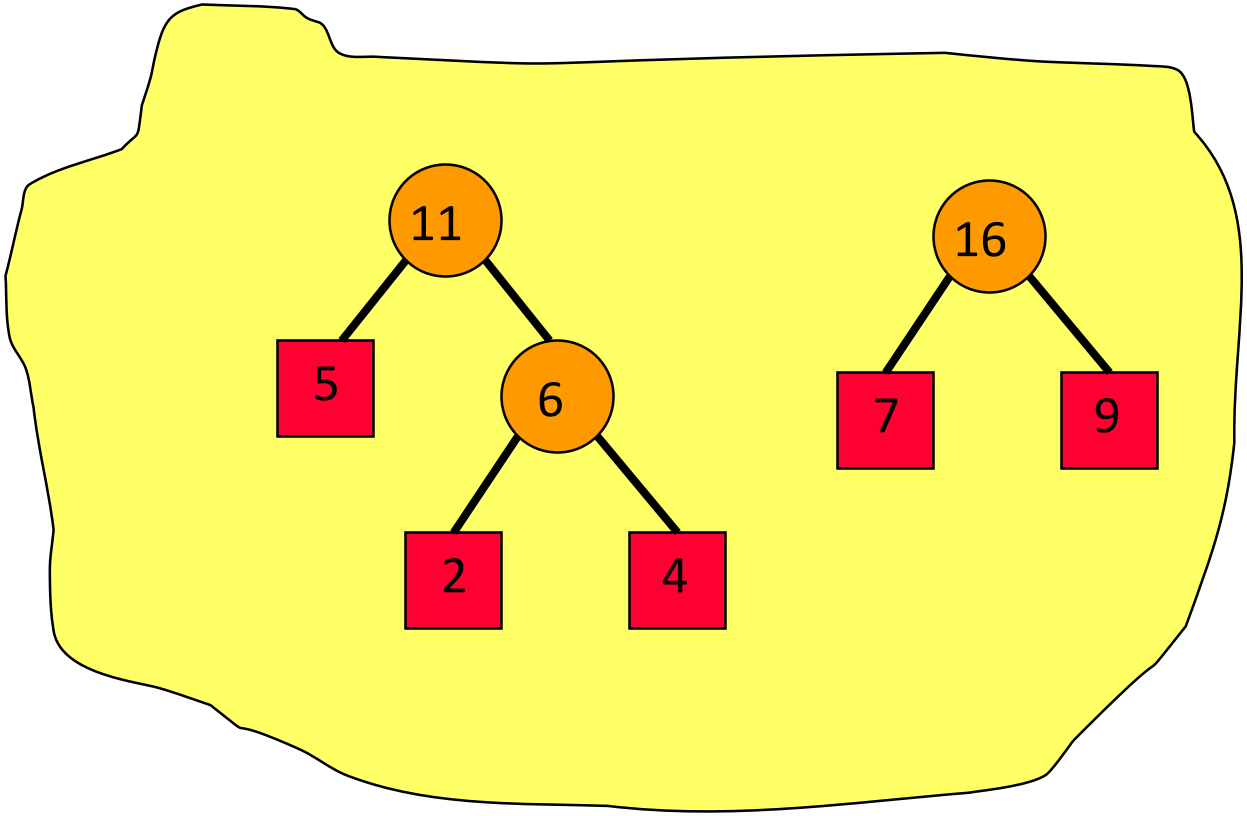
Example

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



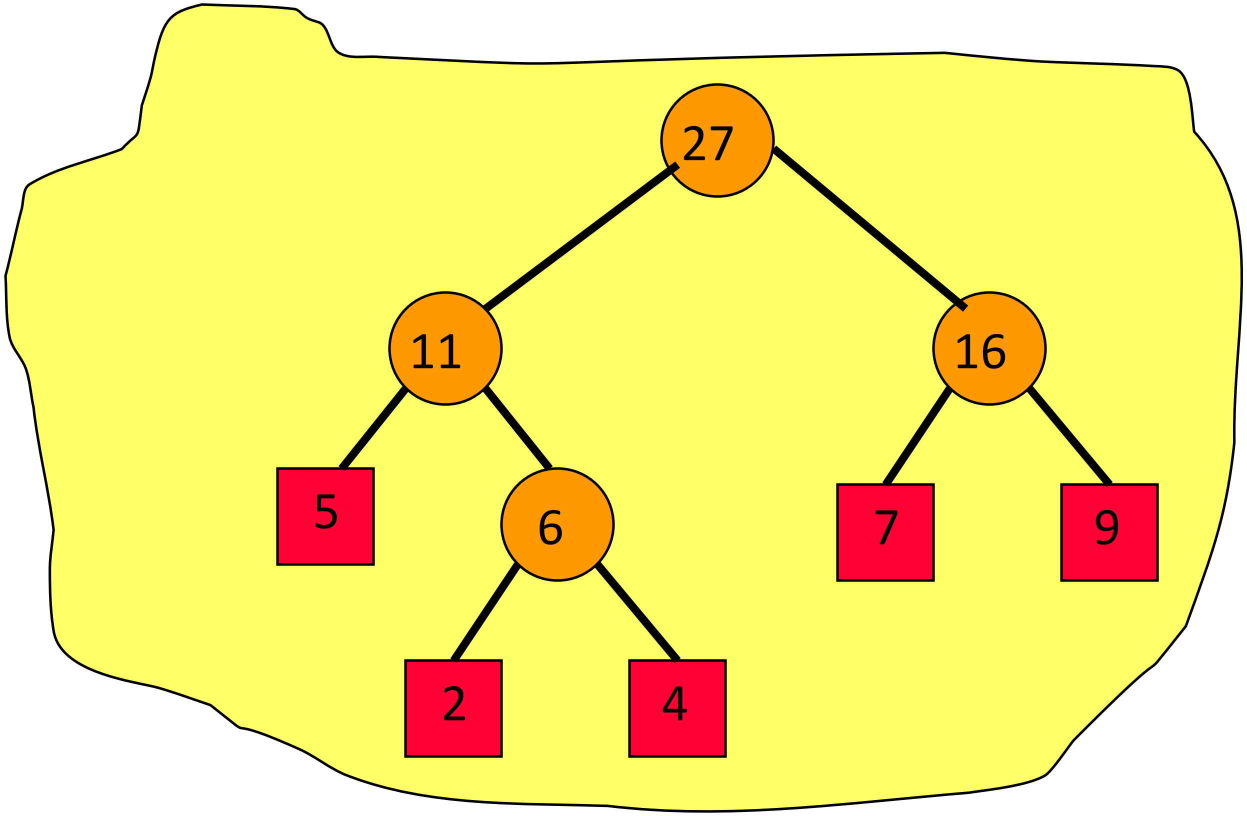
Example

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



Example

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

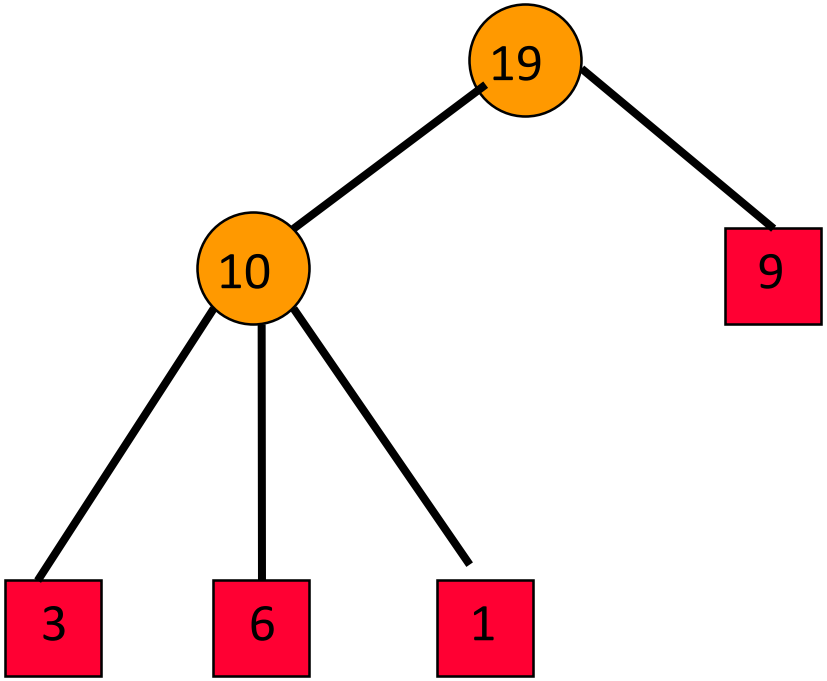


Data Structure For Tree Collection

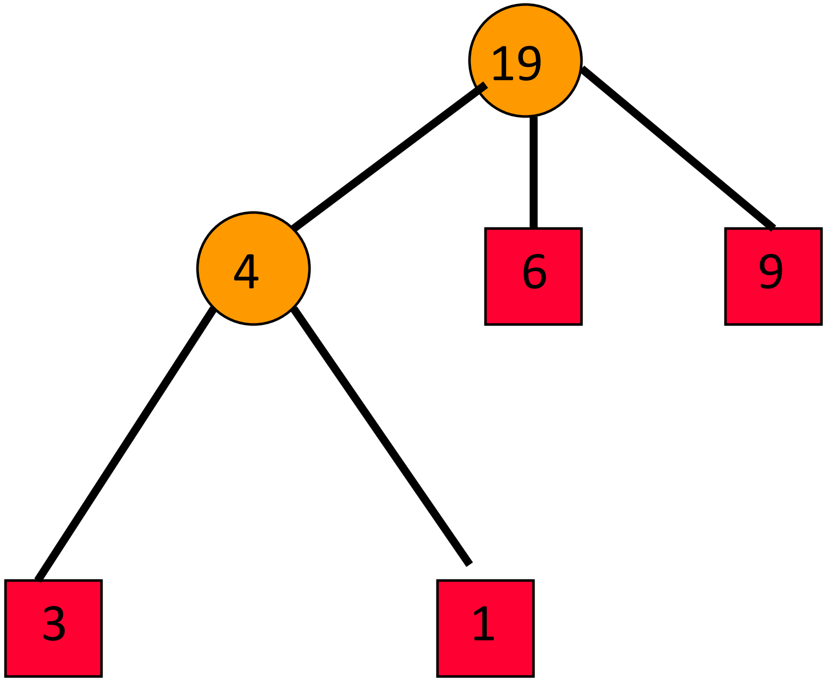
- Operations are:
  - Initialize with n trees.
  - Remove 2 trees with least weight.
  - Insert new tree.
- Use a min heap.
- Initialize … O(n).
- 2(n – 1) remove min operations … O(n log n).
- n – 1 insert operations … O(n log n).
- Total time is O(n log n).
- Or, (n – 1) remove mins and (n – 1) change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

Higher Order Trees

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

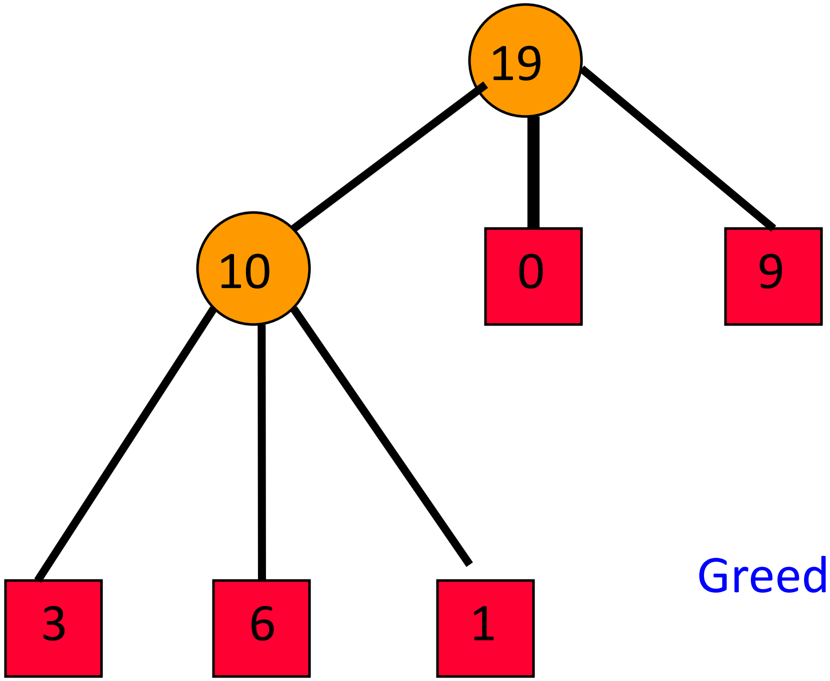


Greedy Tree Cost = 29



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) mod (k – 1) = (1 – r) mod 1 = 0.
  - So, no runs of length 0 are to be added.
- k = 4, r = 6.
  - q = (1 – r) mod (k – 1) = (1 – 6) mod 3

    = (–5)mod 3

    = (6 – 5) mod 3

    = 1.
  - So, must start with 7 runs, and then apply greedy method.