Flat Parallelization

V. Aksenov, ITMO University
P. Kuznetsov, ParisTech

ITMO UNIVERSITY

July 4, 2017

Outline

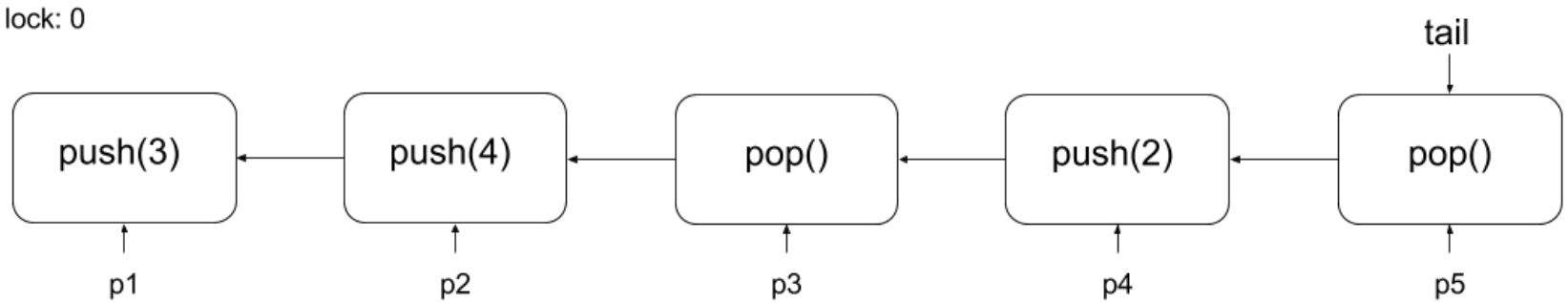
Outline

Flat-combining. Small introduction.

- Some data structures, such as queues, stacks and priority queues, have hot-spots and concurrent updates have to serialize.
- Simple lock is not a solution.
- Flat-combining (FC) proposed in [Hendler, Incze, Shavit, Tzafrir, SPAA 2010].
- We put the operation in the queue. (Similar to ReentrantLock in Java)
- One processor obtains a lock, becomes the combiner and performs all operations in queue sequentially.

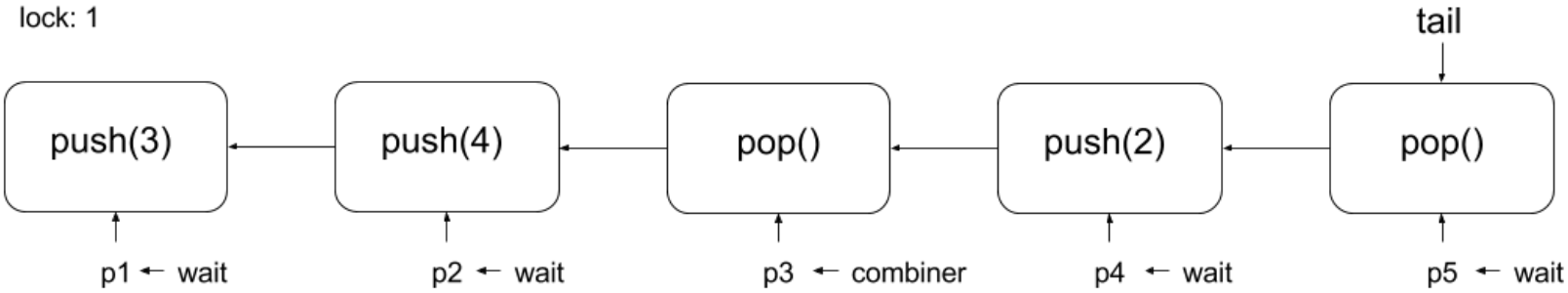
Flat-combining. Stack.

- The processor puts its operation (`push(x)` or `pop()`) in the queue and then tries to take a lock.



Flat-combining. Stack.

- If the processor succeeds in obtaining the lock it becomes the combiner, others wait.



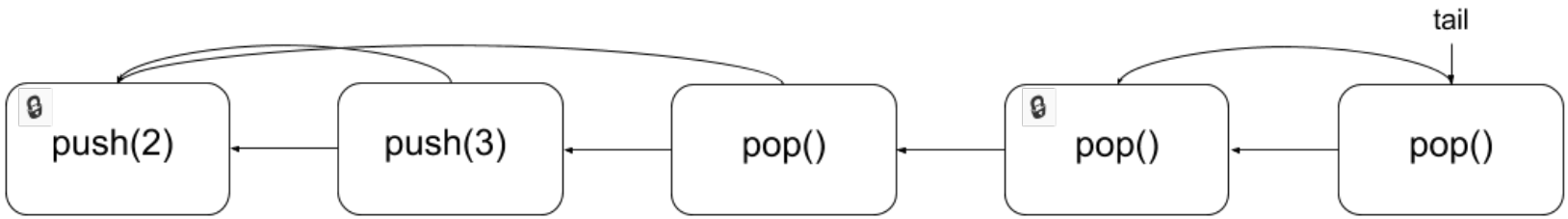
- Then the combiner reads all the requests: `push(3)`, `push(4)`, `pop()`, `push(2)`, `pop()`.
- Make a correspondence between `push(x)` and `pop()` operations. For example, `push(4)` and `push(2)` with these two `pop()`.
- And, finally, simply `push(3)` on the stack.

Parallel flat-combining

- When the number of threads is high, we lost huge computational power of waiting threads.
- Parallel FC. Introduced in [Hendler, Incze, Shavit, Tzafrir, DISC 2010].
- Applied to unfair queue, a way to solve producer-consumer problem.

Parallel flat-combining

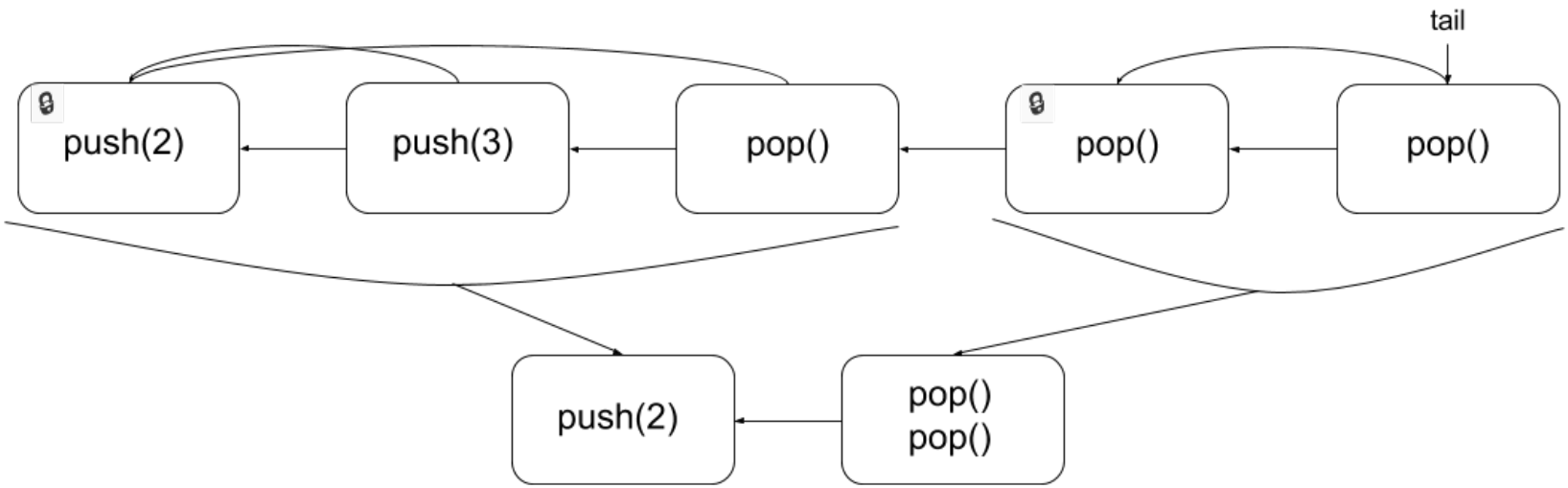
- The queue with requests is split into parts of bounded length.



- Each thread tries to take a corresponding lock to become a local combiner.

Parallel flat-combining

- They locally combine requests
- Batch of remaining requests is put into the second FC level.



Parallel flat-combining. Summary.

- This gives us additional utilization of computational power.
- Unortunately, Parallel FC does not seem to be useful for fair queues, stacks, heap-based priority queues, etc.

Outline

PRAM and Work-Time model

PRAM:

- p processors.
- Each has its own algorithm.
- Each time unit the processor performs one operation: read from and write to shared register, local computation.

Work-time model:

- Presented in terms of a sequence of parallel rounds.
- Each parallel round consists of concurrent sequences of instructions.
- The complexity in work and time.

Batch-update algorithms

- You are given list of m requests to perform.
- These requests are executed in parallel in PRAM or Work-time model.
- Example, is binary search trees. You need to remove sorted list of values in parallel and you need to insert sorted list of values.

PRAM algorithms and Flat combining

- In FC we could read all the requests and prepare them.
- And then we could perform algorithm in PRAM model.
- For example, sort values of insert and delete requests for the BST.
- As a data structure with hot-spot, heap-based priority queue could have parallel algorithm.

Concurrent Data Structures for parallel programs

- Proposed in [Agrawal et al., SPAA 2014].
- For parallel programs with fork-join. There are two graphs: core and batch.
- Batcher: publish request, tries to take a lock and goes to "batch" mode.
- Work-stealing scheduler. Core-thread perform alternating-steals from core and batch queues, batch-thread steals from batch-queues.

Flat parallelization

General form of flat parallelization for Data Structures in asynchronous model:

- Put operation into the queue.
- Try to take a lock and become a combiner.
- If combiner, prepare the requests before the batch-update algorithm.
- Wake the waiting threads.
- The threads emulate batch-update PRAM algorithm in the asynchronous system.

Outline

Binary heap as a running example

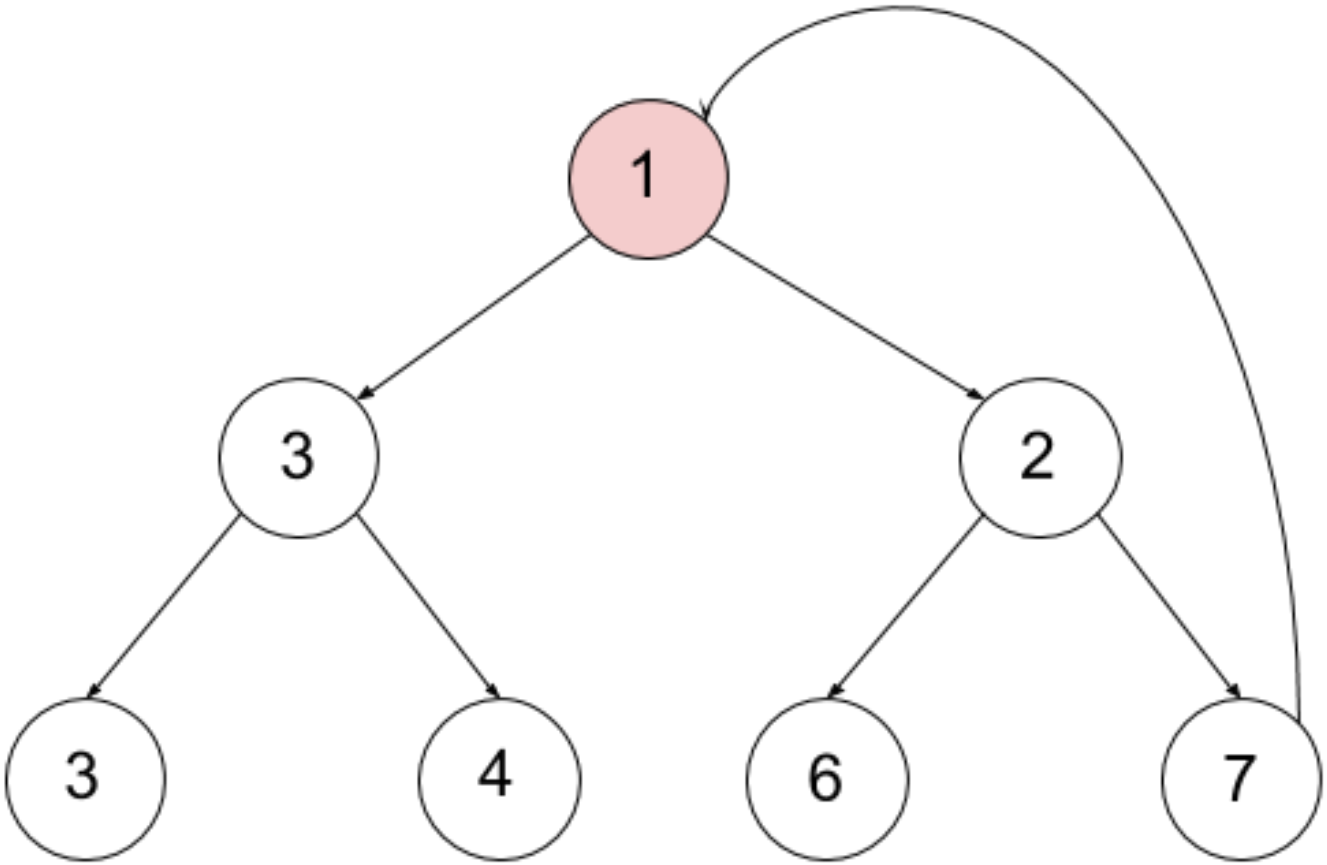
- Choose priority queue basesd on binary heap as an example.
- Two types of requests: `insert` and `extractMin`.
- We provide two algorithms for batch-insert and batch-extractMin.
- No previously known batch-update PRAM algorithm of the binary heap.

Outline

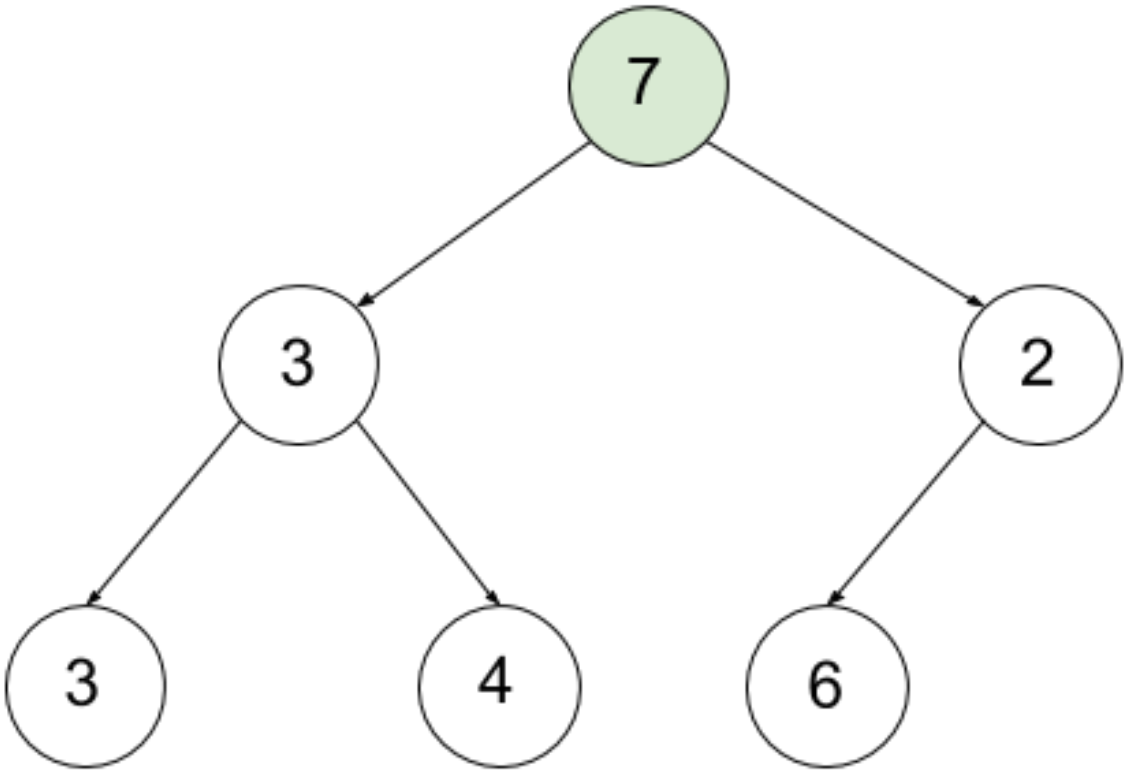
ExtractMin. Sequential algorithm.

- Swap the last element with the element in the root.
- Then sift the root down.

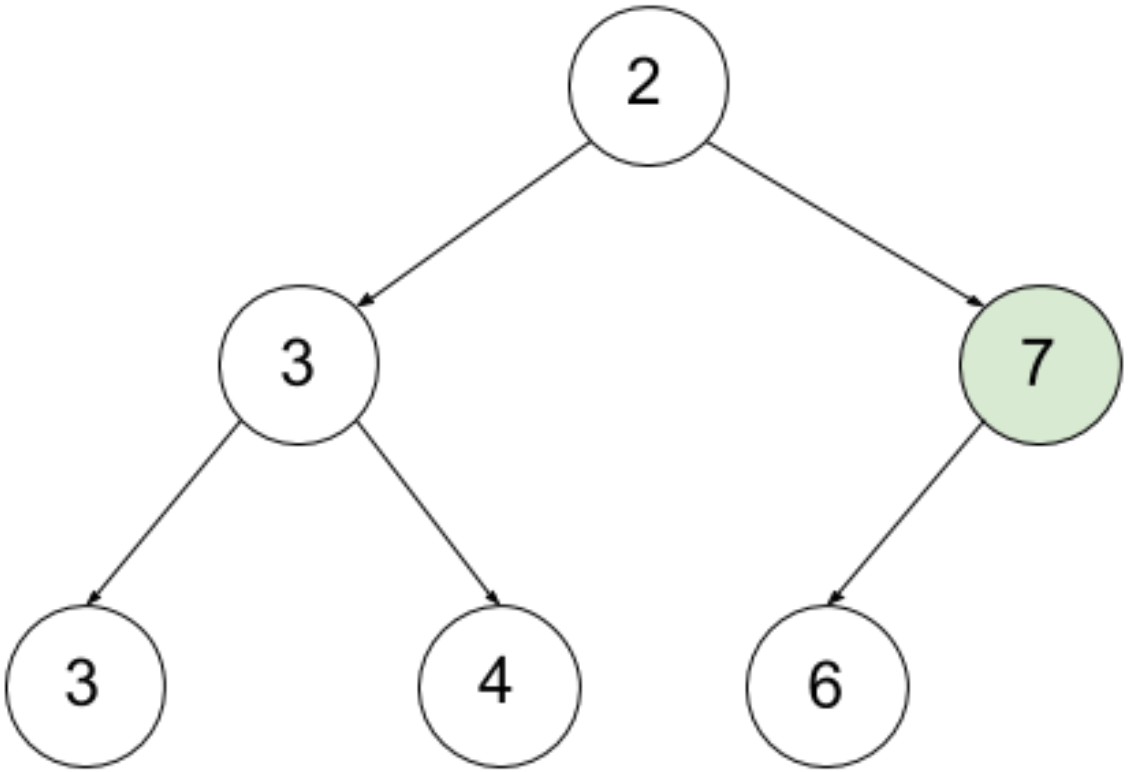
ExtractMin. Sequential algorithm.



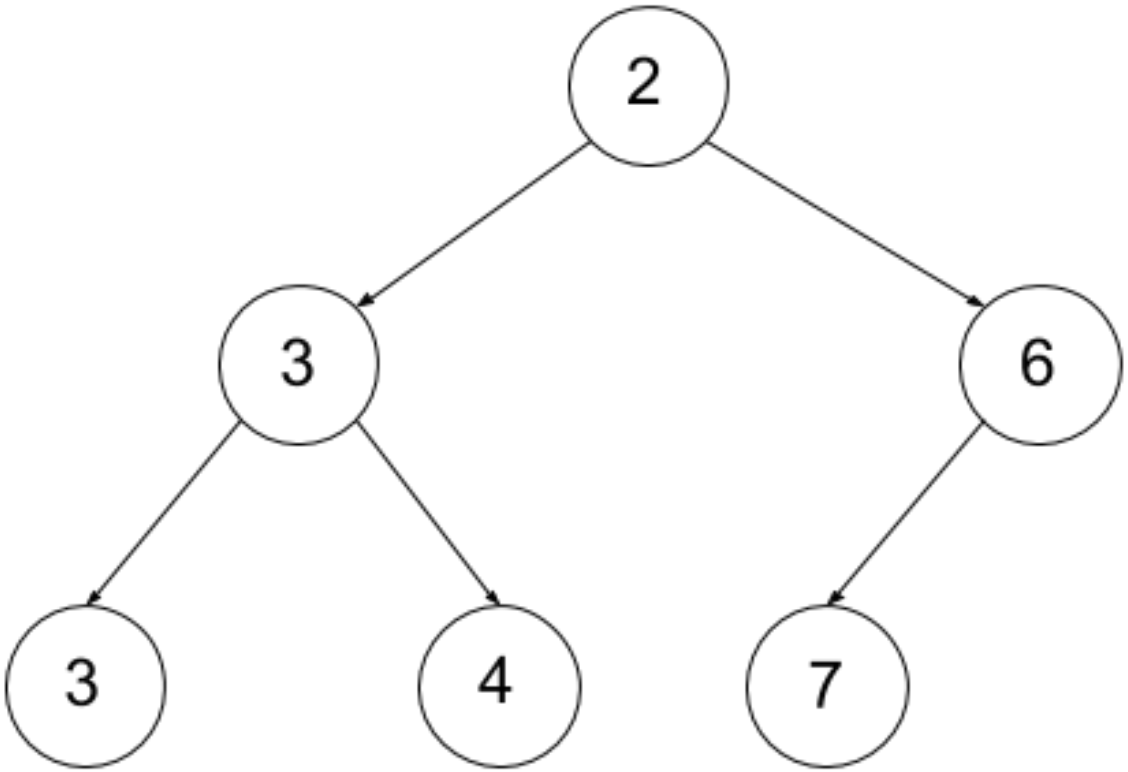
ExtractMin. Sequential algorithm.



ExtractMin. Sequential algorithm.



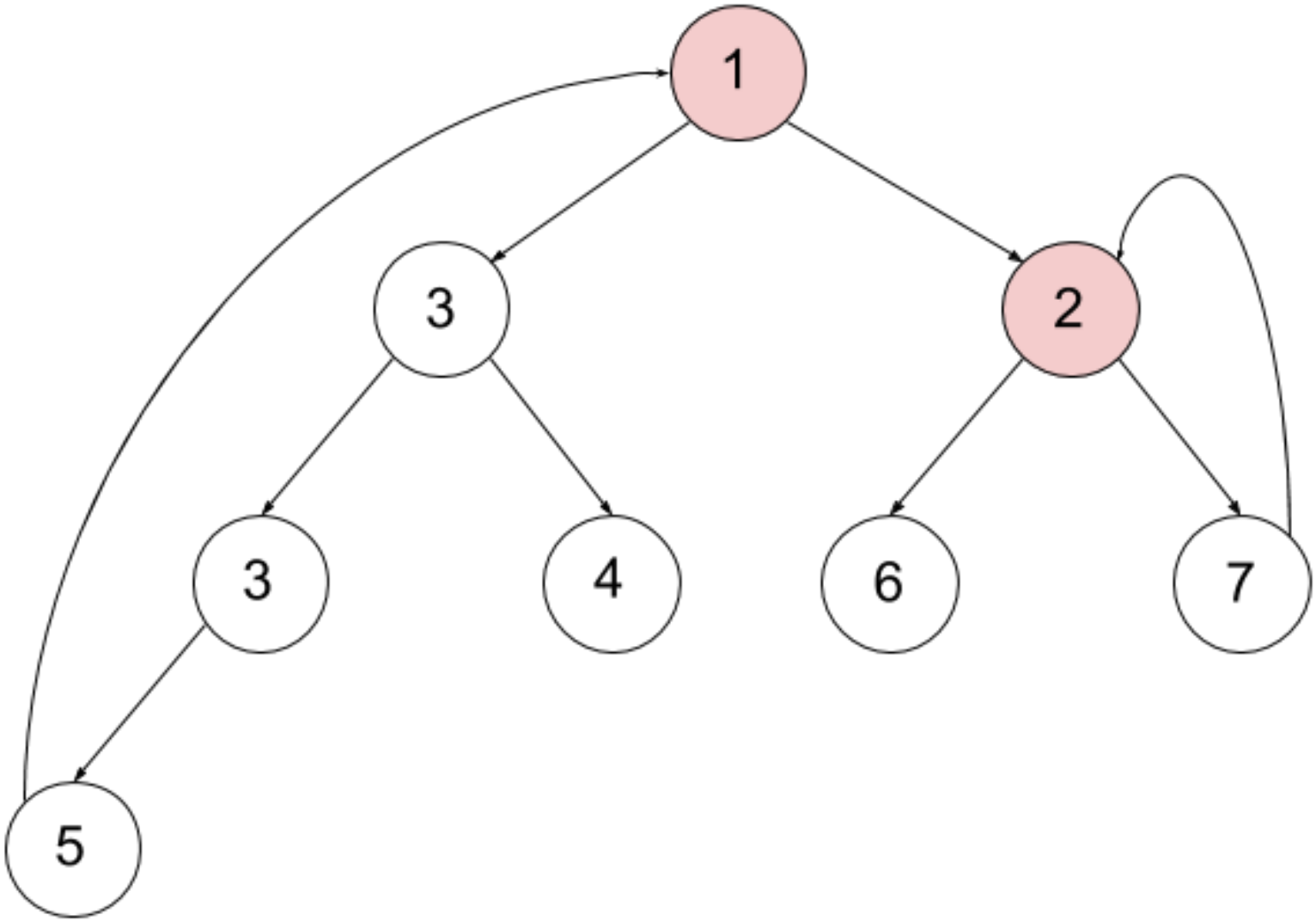
ExtractMin. Sequential algorithm.



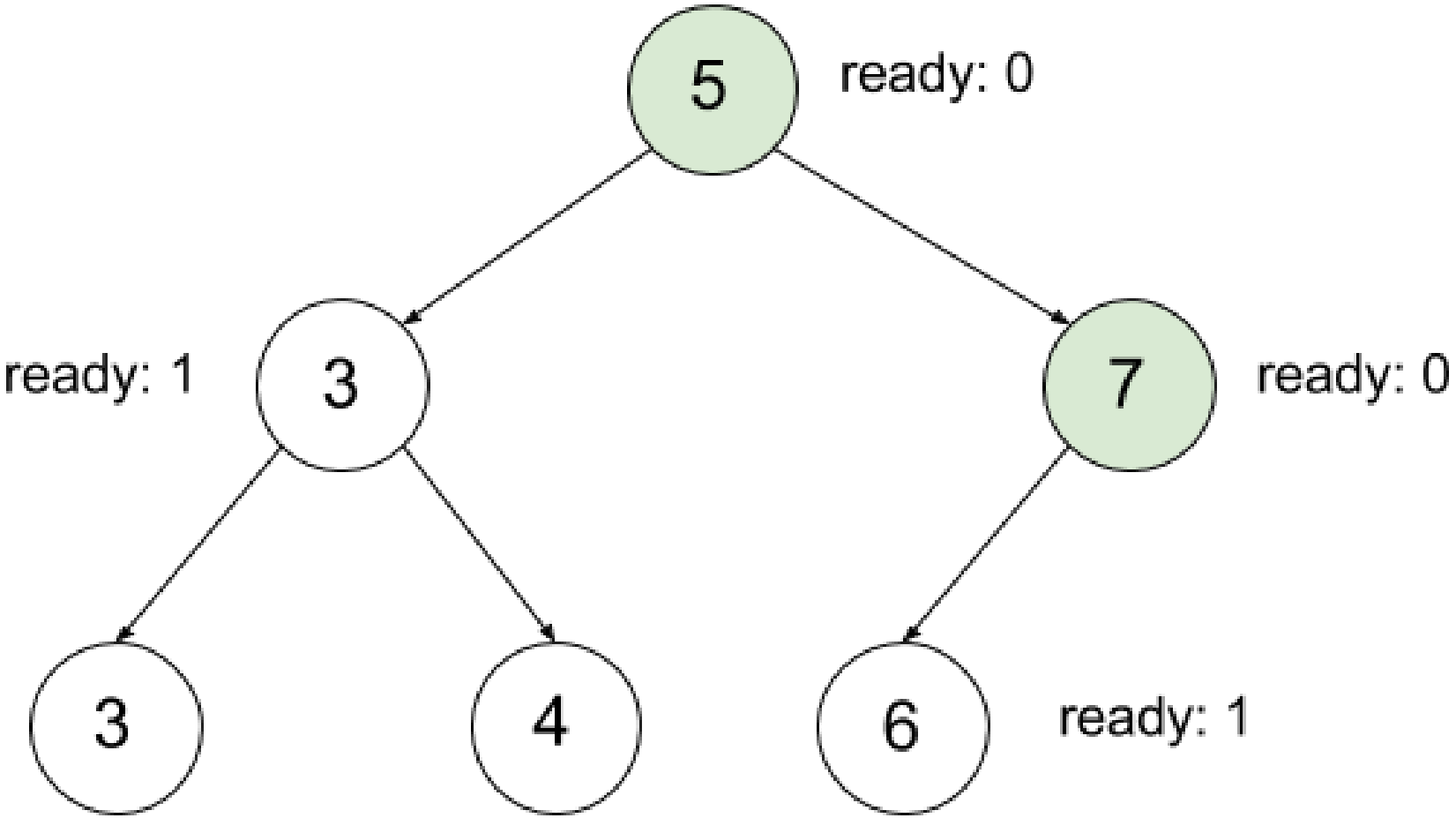
ExtractMin. Batch-update algorithm

- Swap last m elements with the smallest m elements in $O(m \log m)$ time.
- Each node has a flag `ready`. Initially, replaced elements are not ready.
- Each thread starts to work on some non-ready vertex and tries to sift down its value only if both of its child are ready.

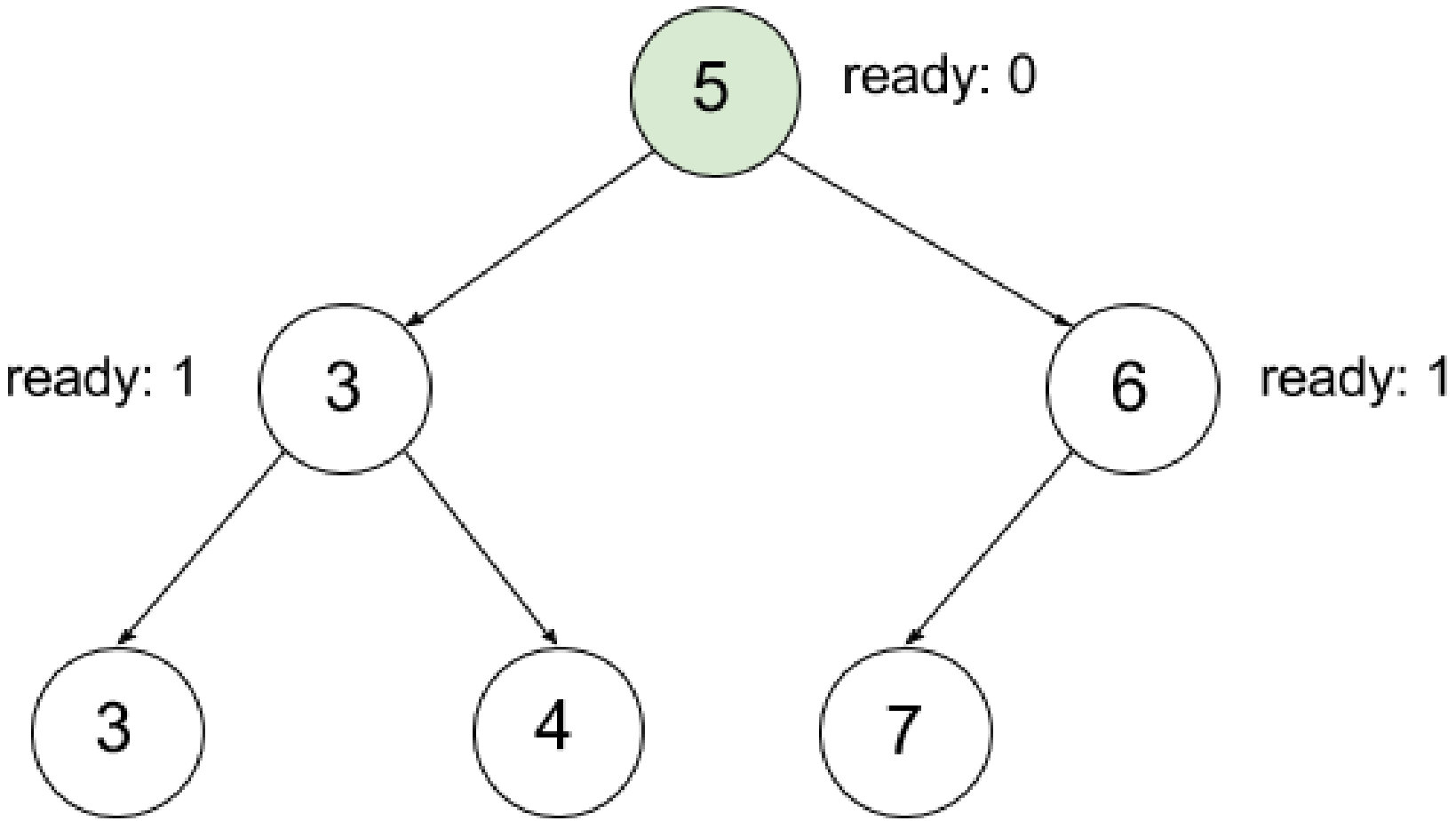
ExtractMin. Batch-update algorithm.



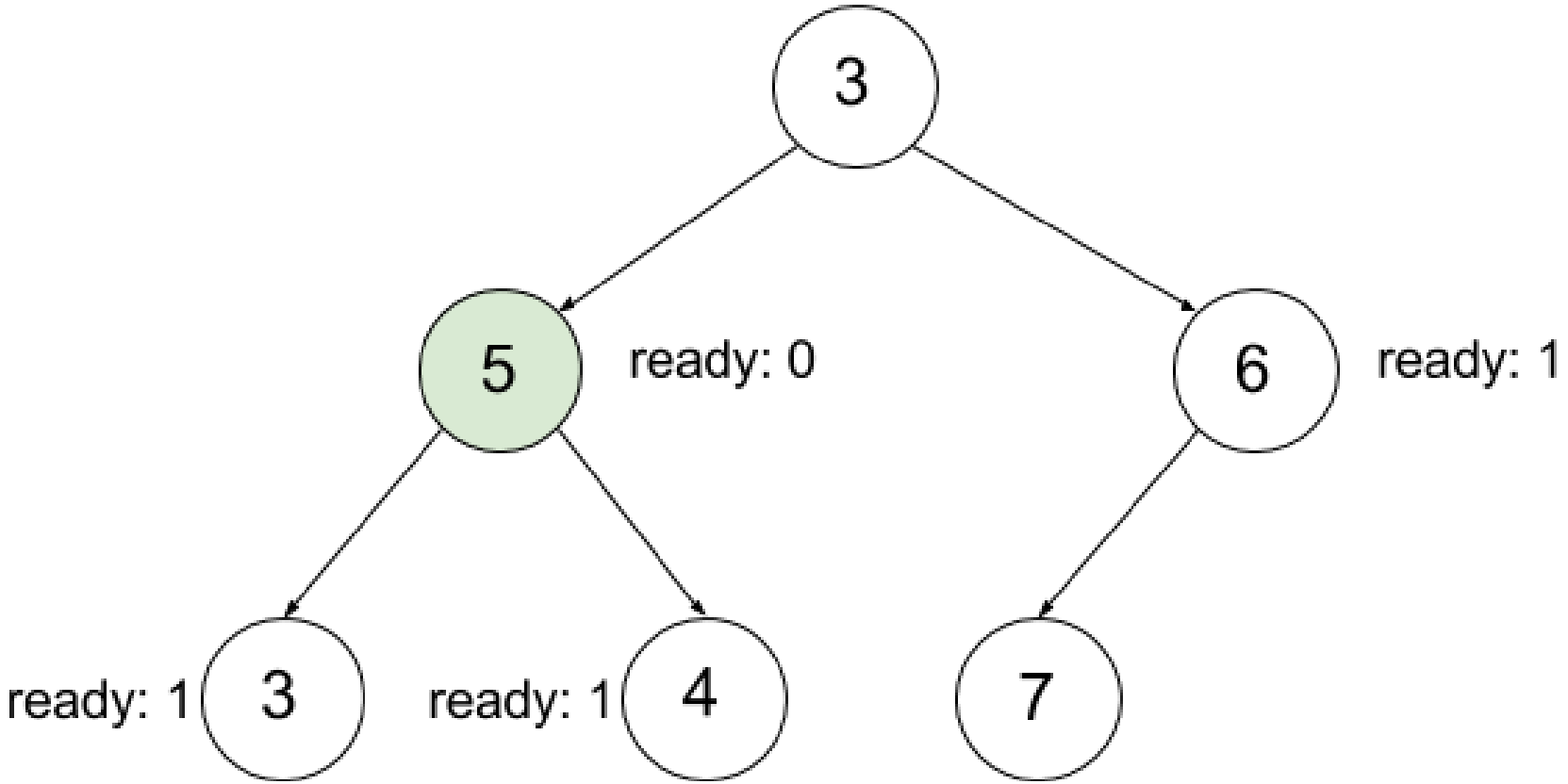
ExtractMin. Batch-update algorithm.



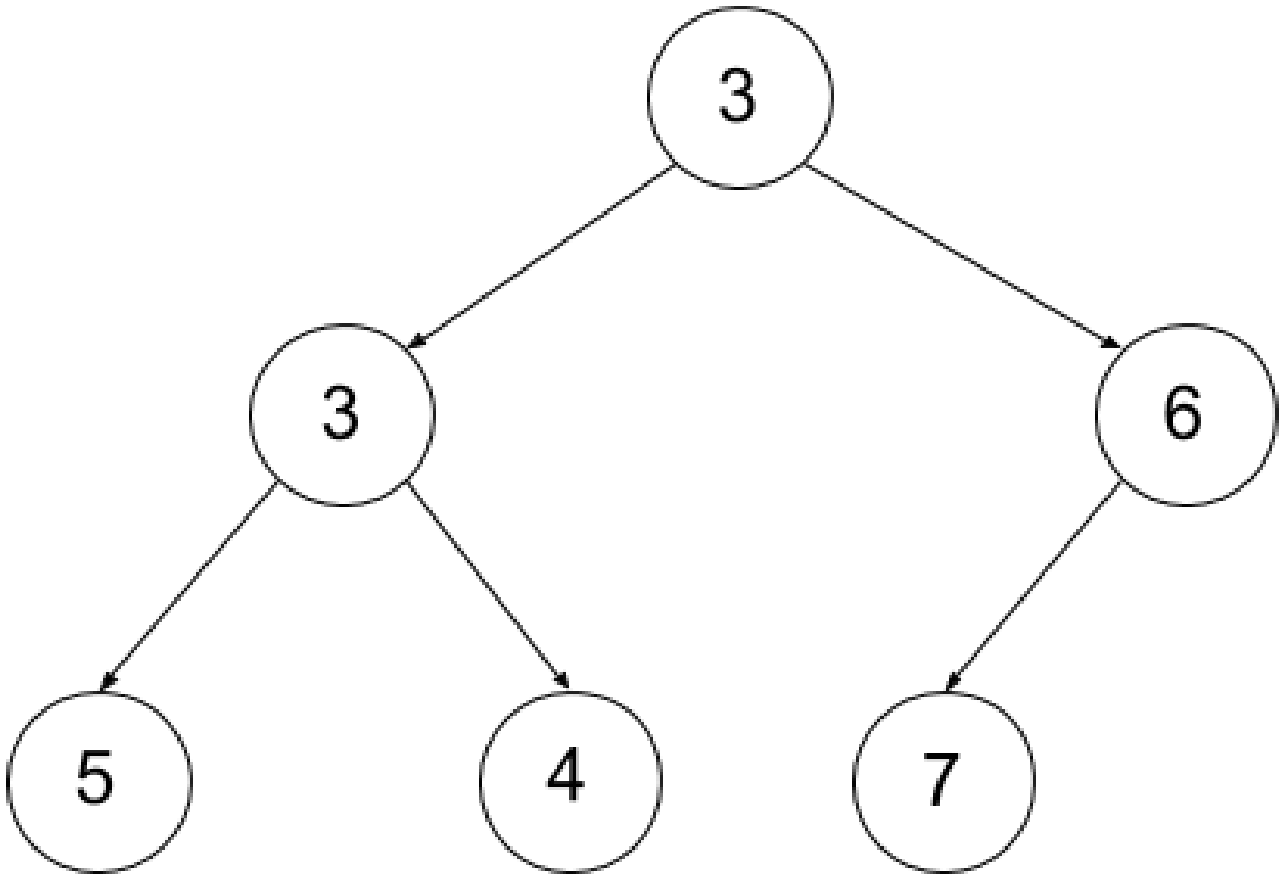
ExtractMin. Batch-update algorithm.



ExtractMin. Batch-update algorithm.



ExtractMin. Batch-update algorithm.

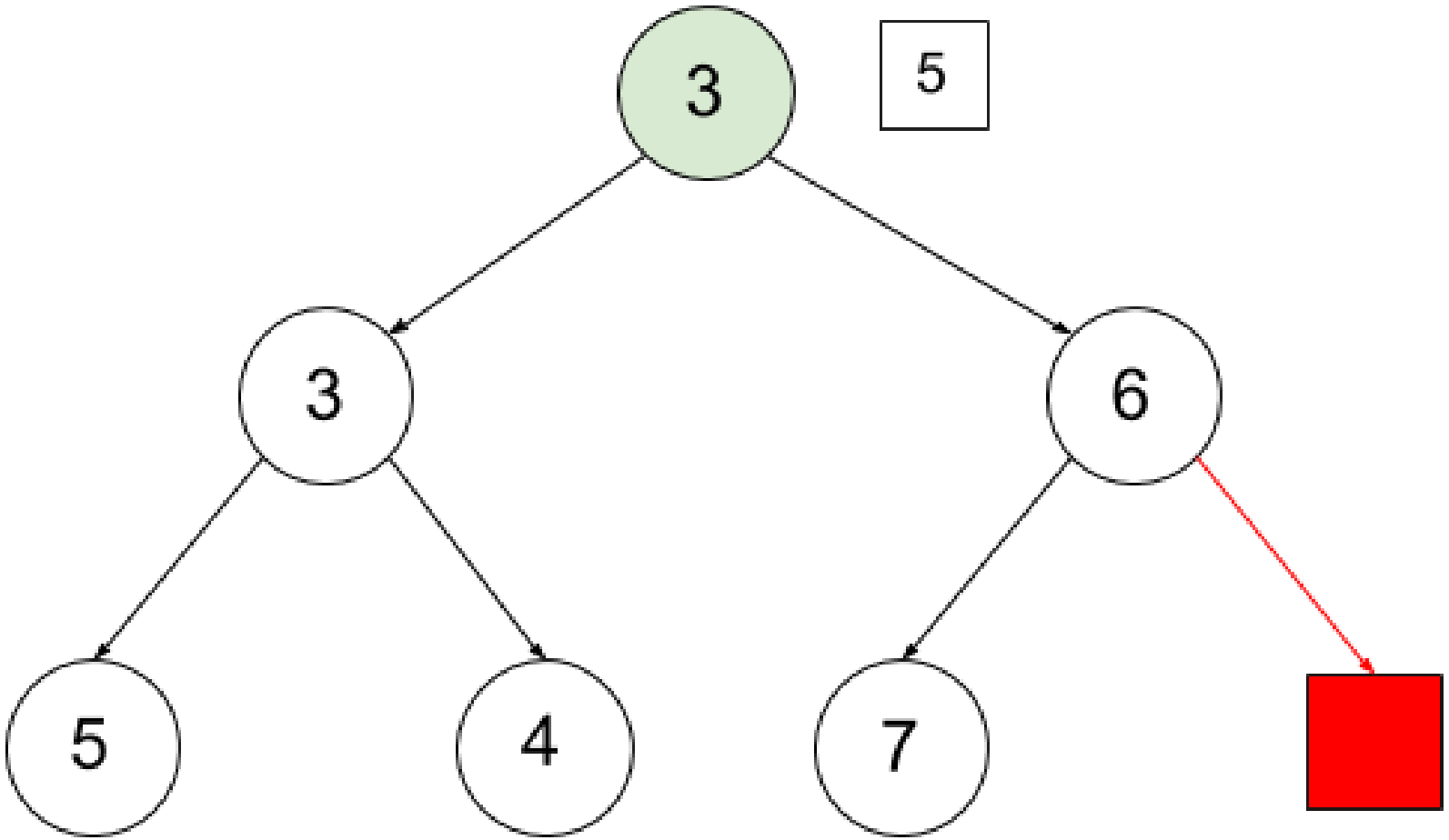


Outline

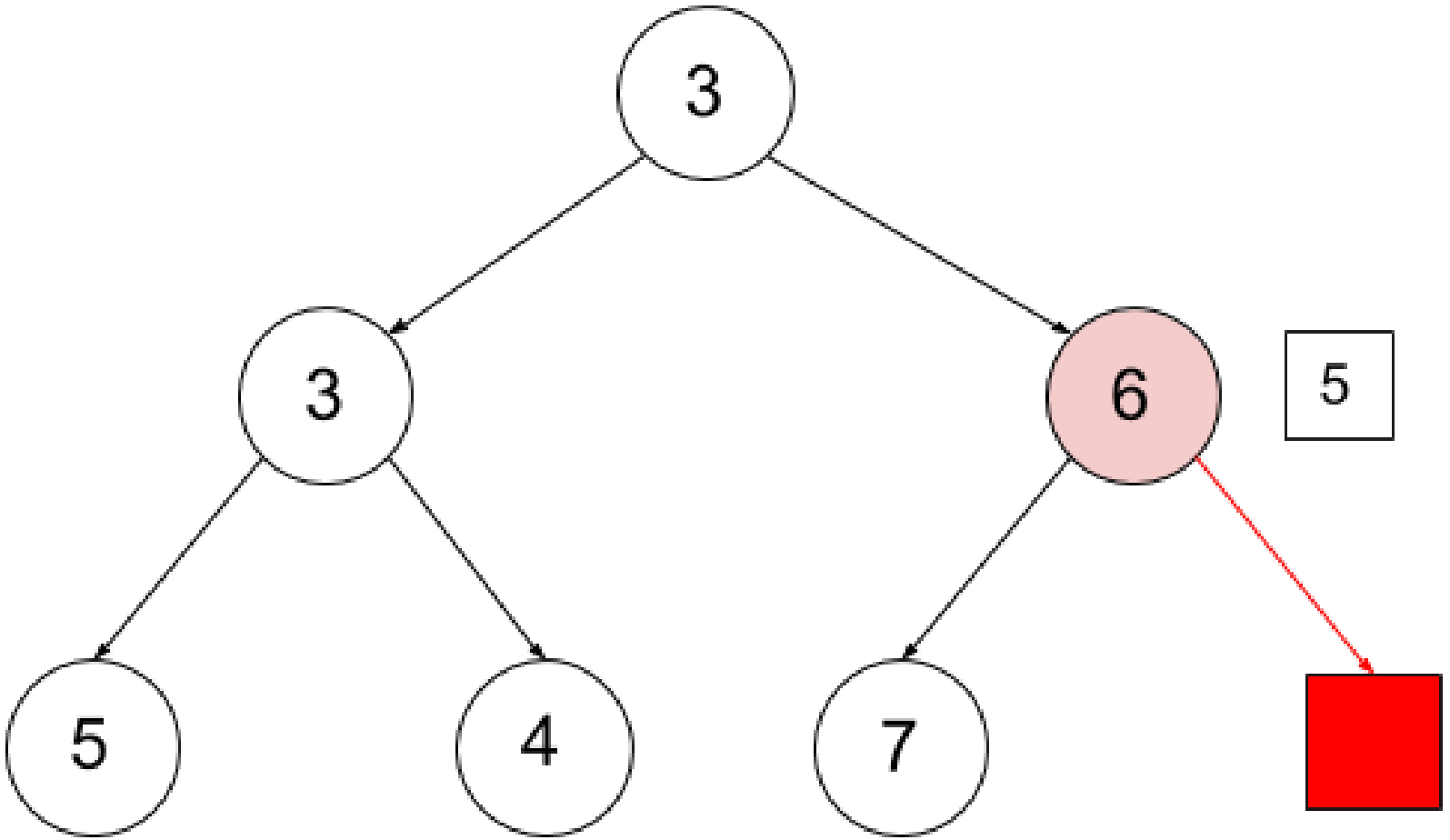
Insert. Sequential algorithm.

- We look at the path to the new position in heap.
- We go downwards by that path with inserted value x.
- Compare x with the value in the current node and if less then swap.

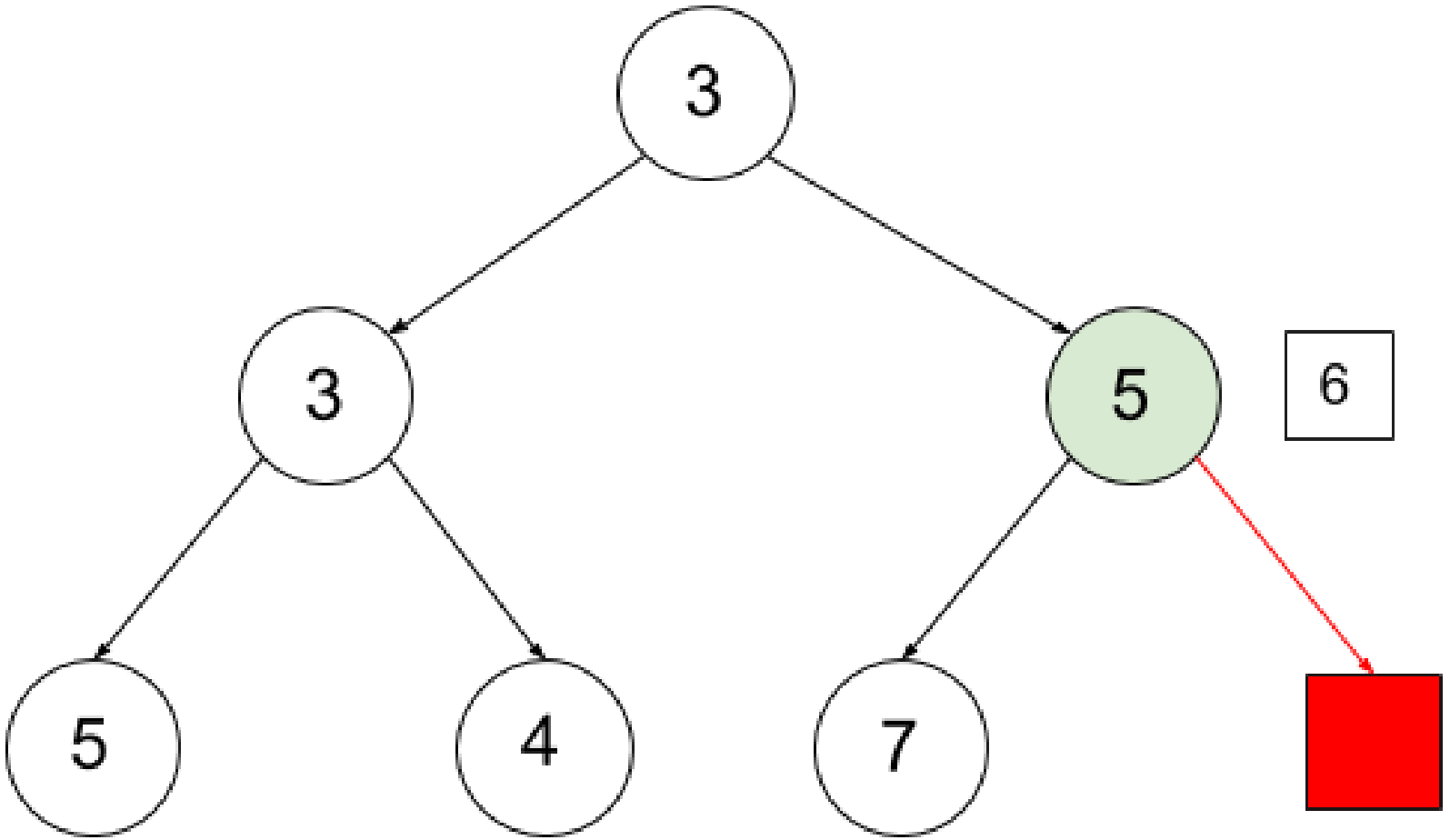
Insert. Sequential algorithm.



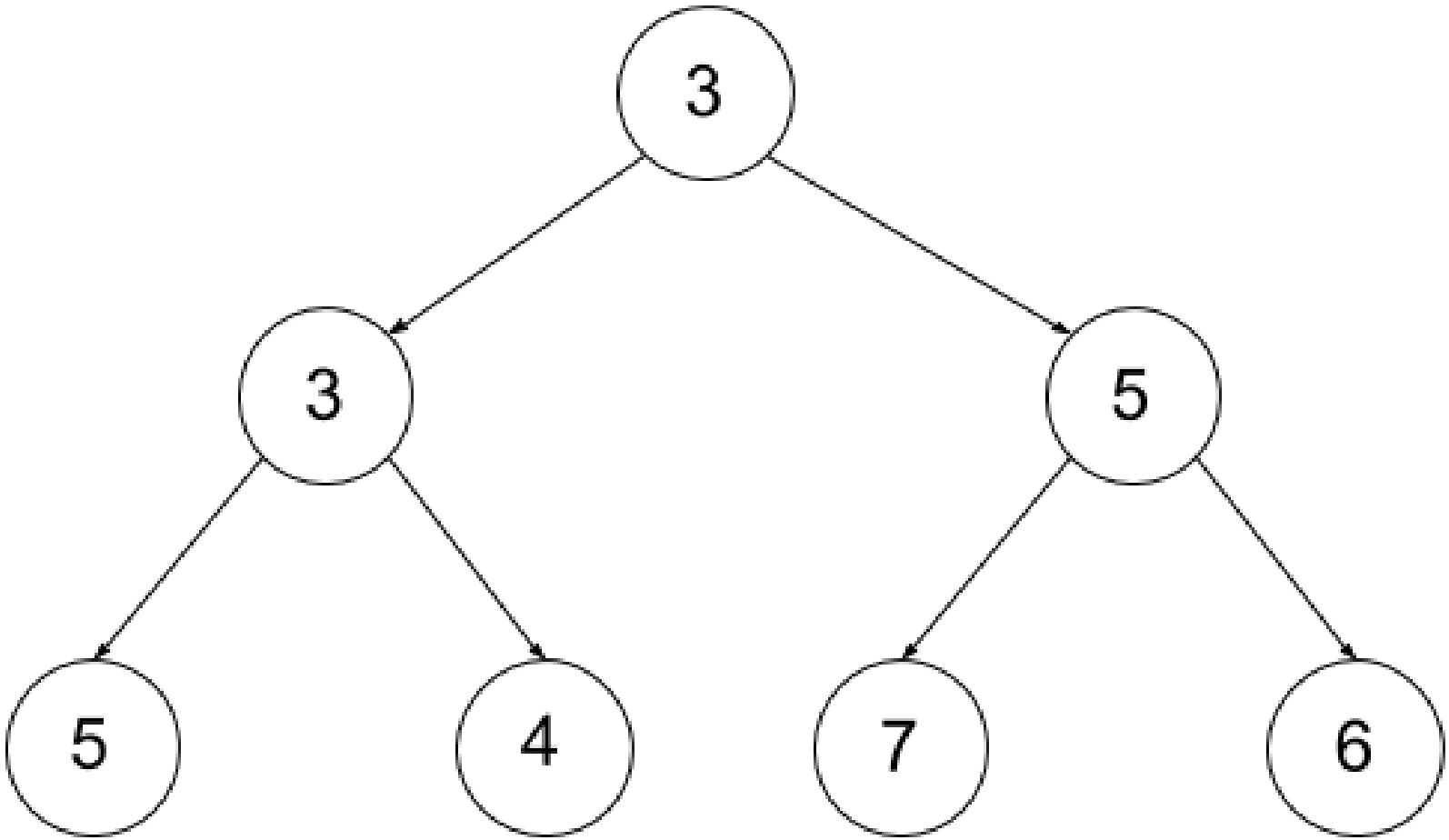
Insert. Sequential algorithm.



Insert. Sequential algorithm.



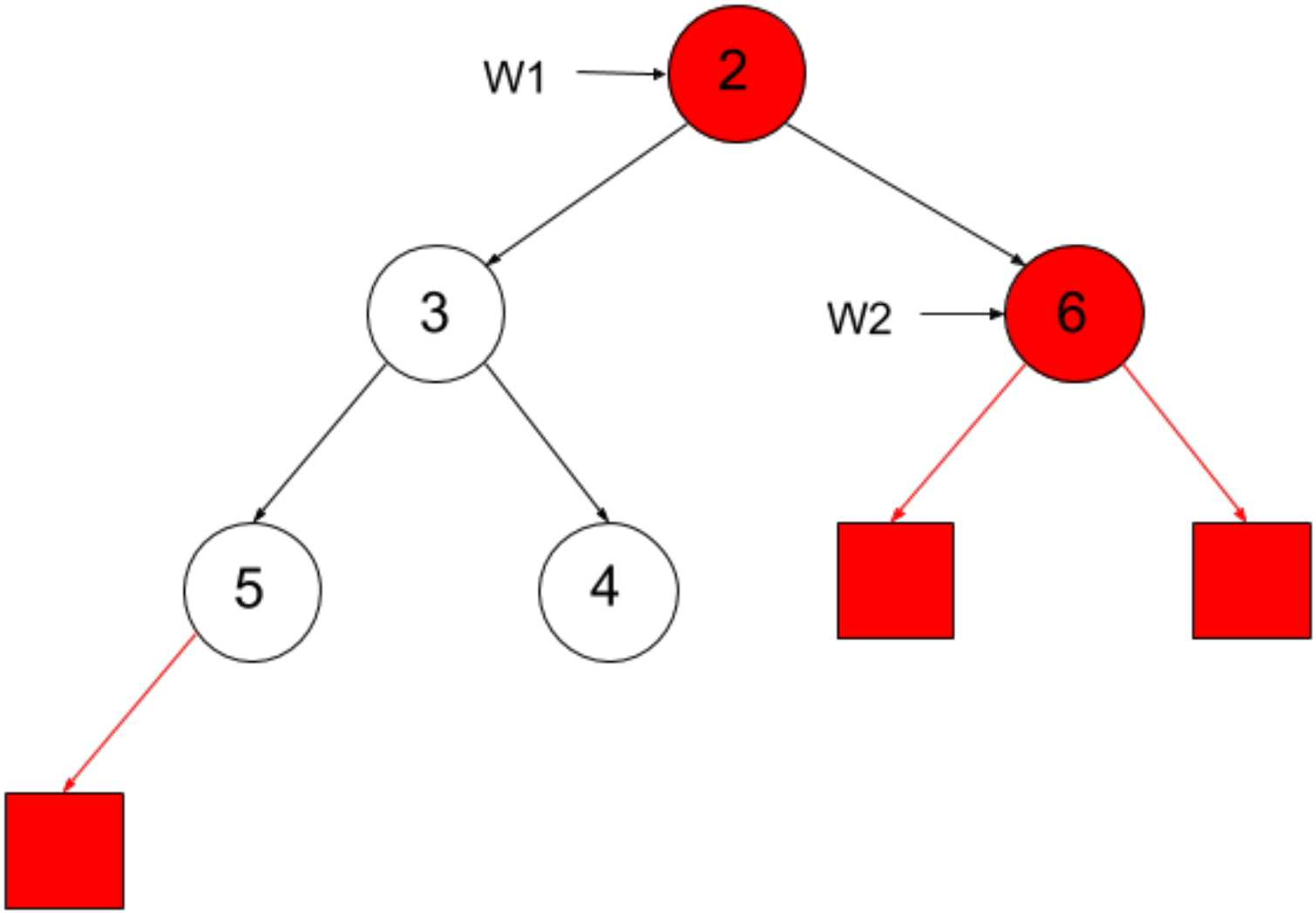
Insert. Sequential algorithm.



Insert. Batch-update algorithm.

- Consider m paths to new m positions in heap,
- Mark $m - 1$ nodes with new positions in both subtrees as split nodes.
- Exactly one thread sleeps in each split node.

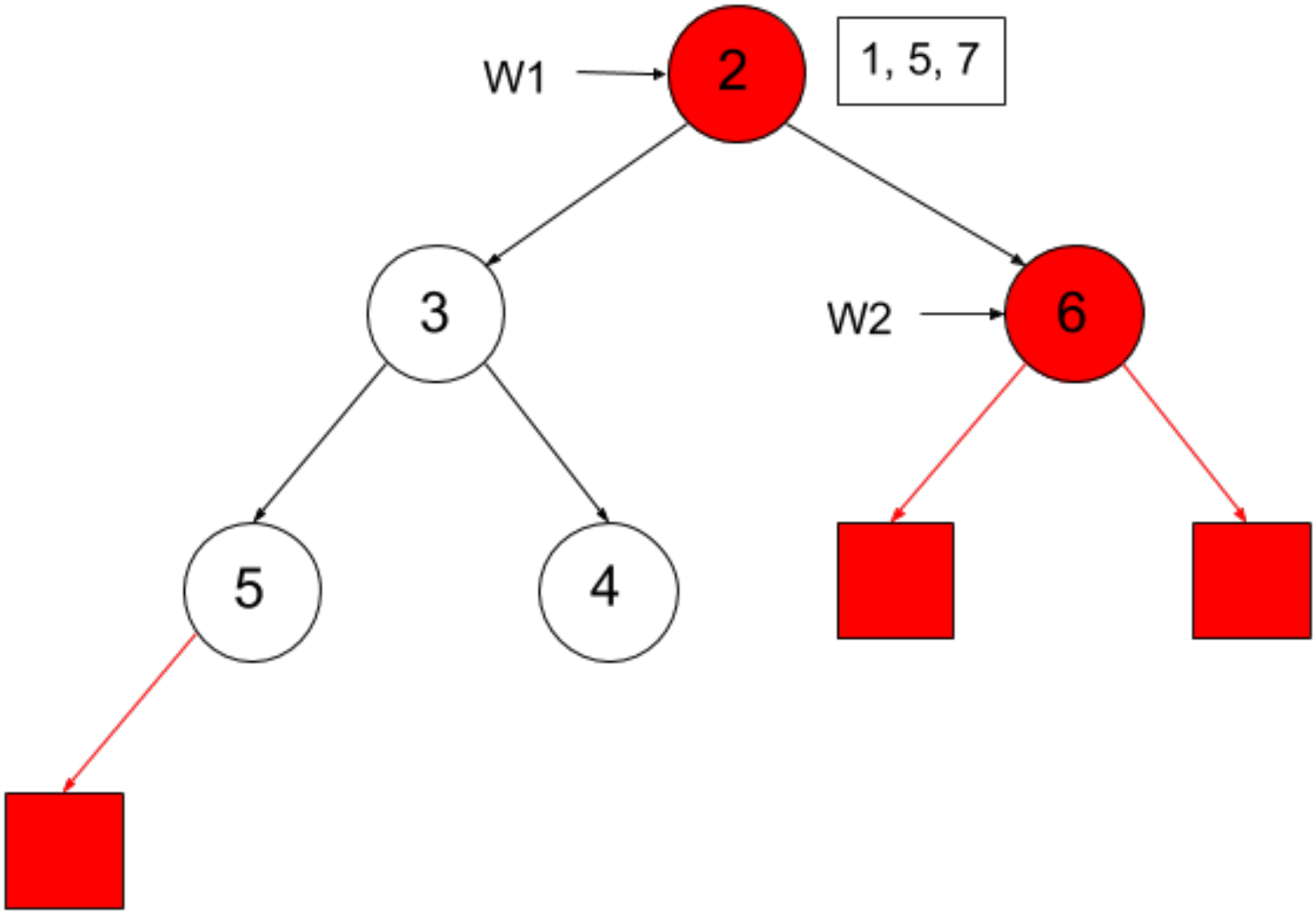
Insert. Batch-update algorithm.



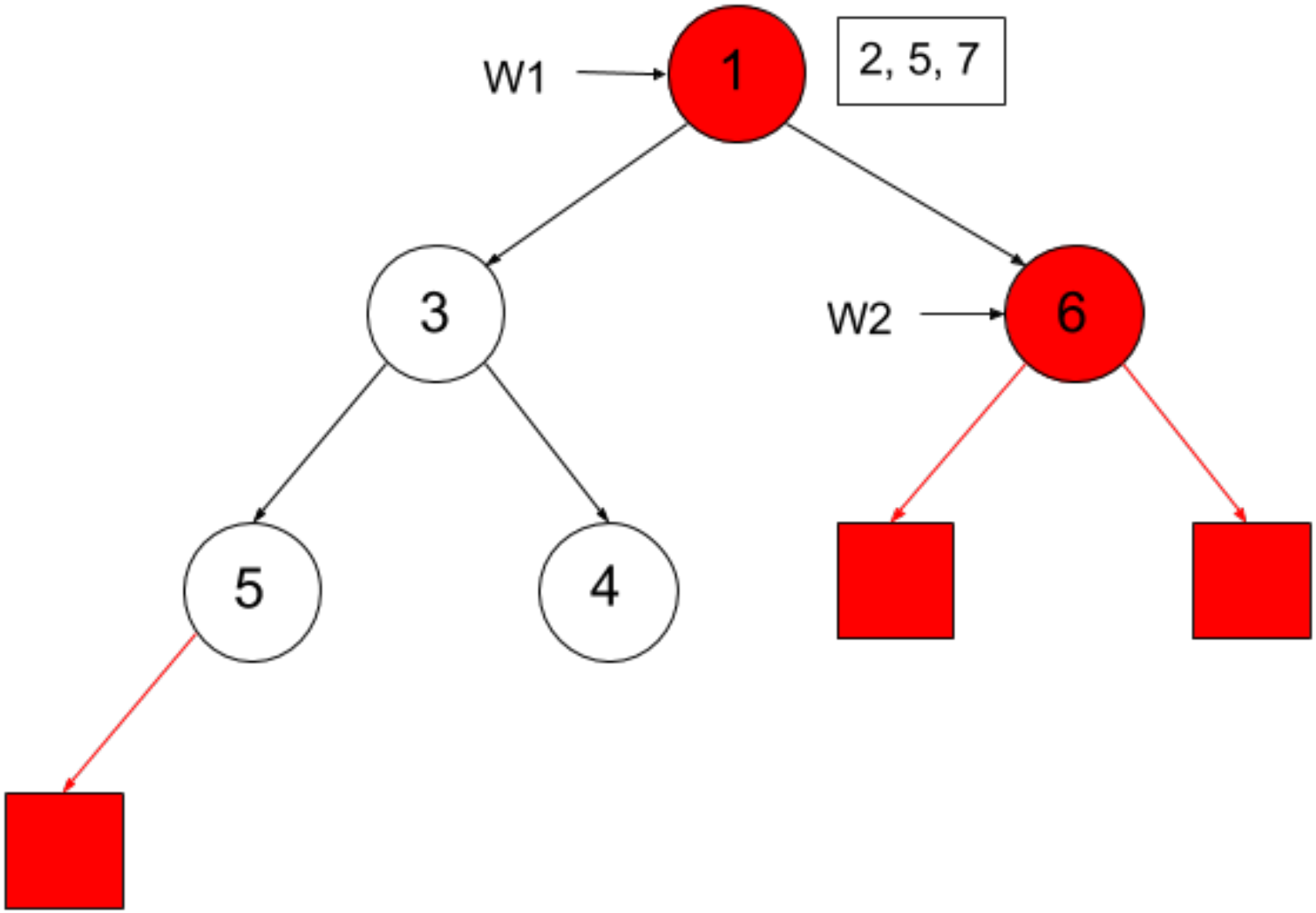
Insert. Batch-update algorithm,

- Combiner starts from the root with the sorted list of values.
- In each node, processor compares the smallest value in the list with the value in the current node.
- Probably, put the value of the node in the list and put the smallest value in the node.
- If the current node is not a split node, the processor continues with the proper child.
- Otherwise, it splits the list of values in the lists for the left and right children. Then wake up the processor in the node and give him the list for the right child. The old thread continues with the left child and the waken thread continues with the right child.

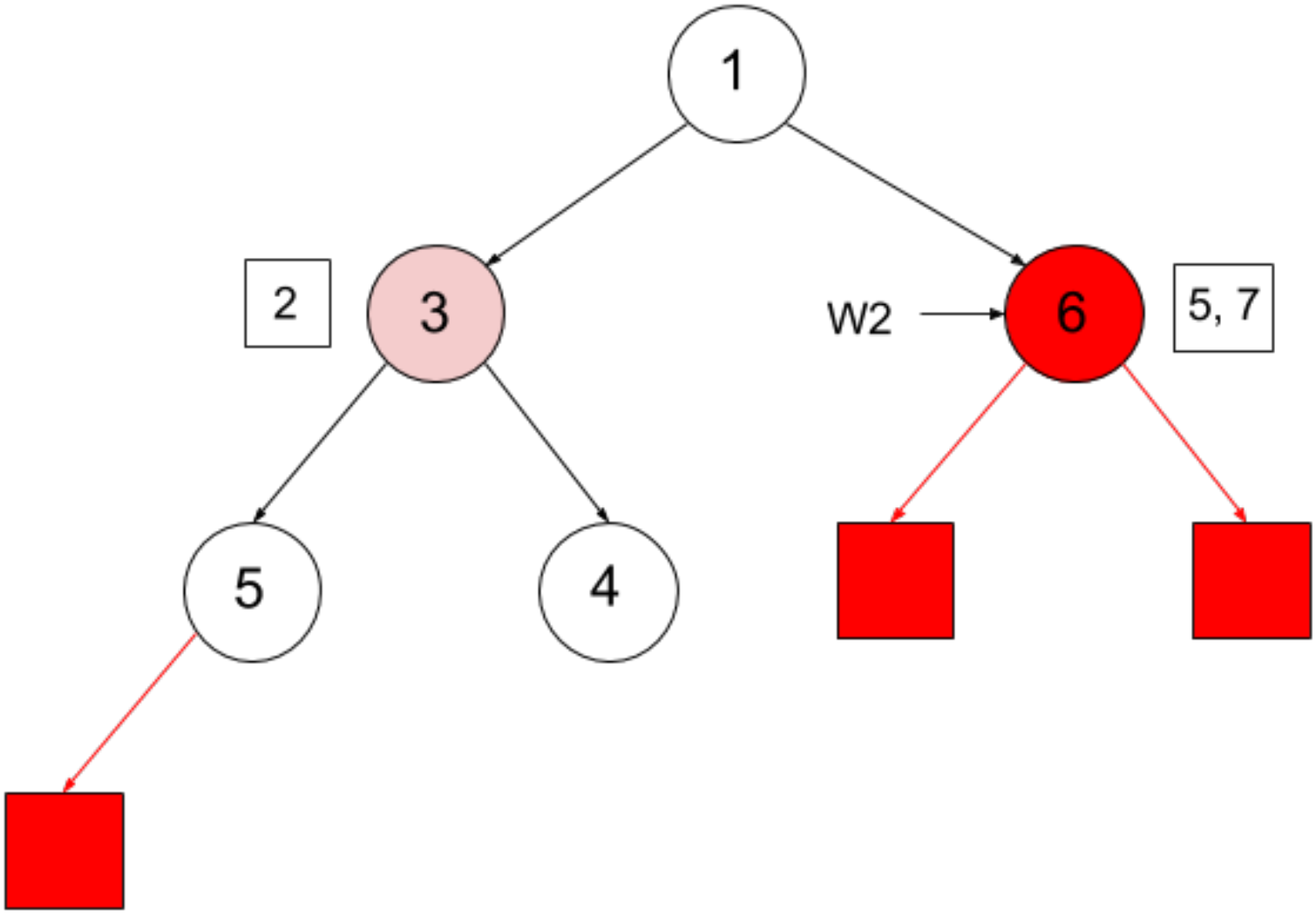
Insert. Batch-update algorithm.



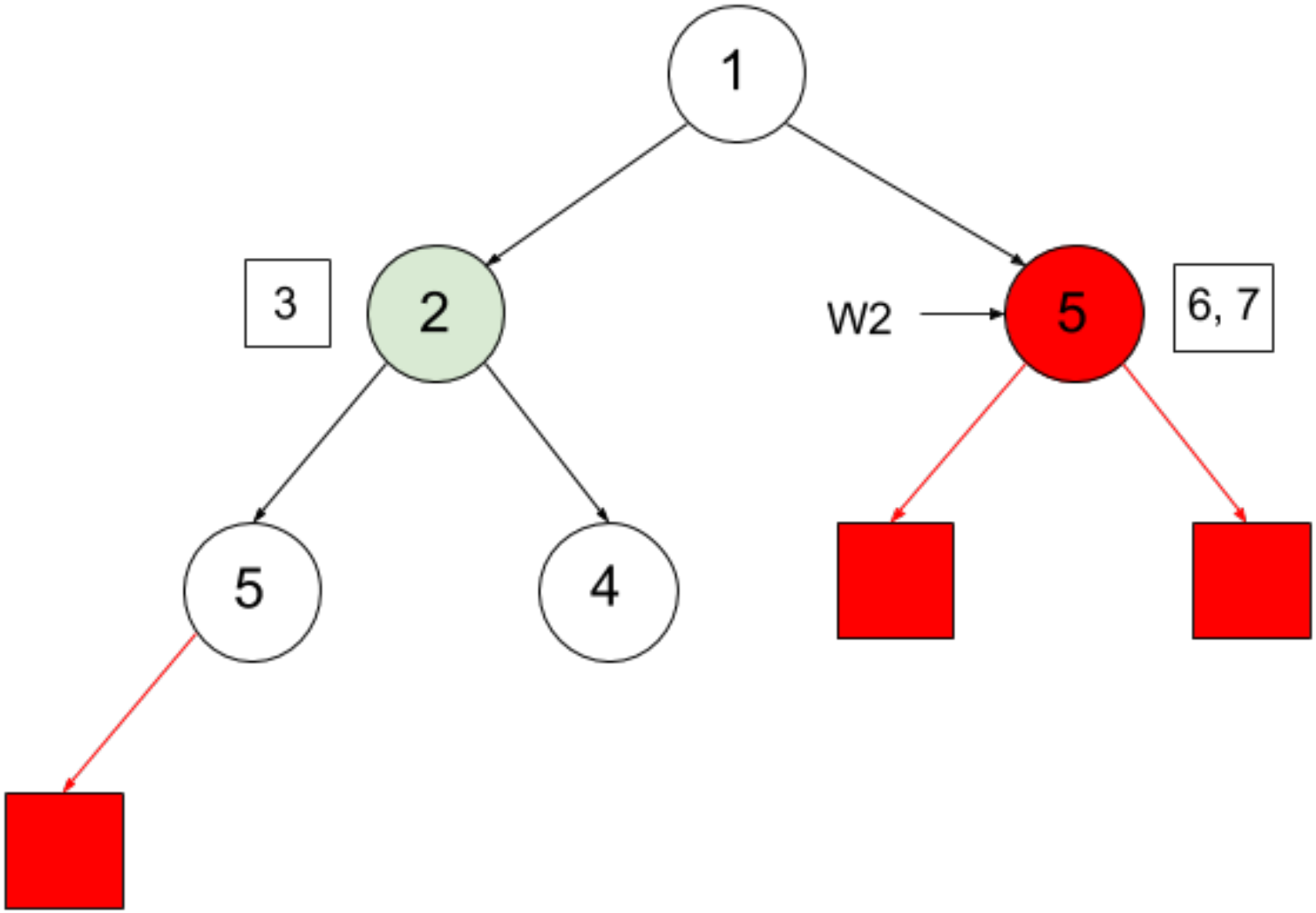
Insert. Batch-update algorithm.



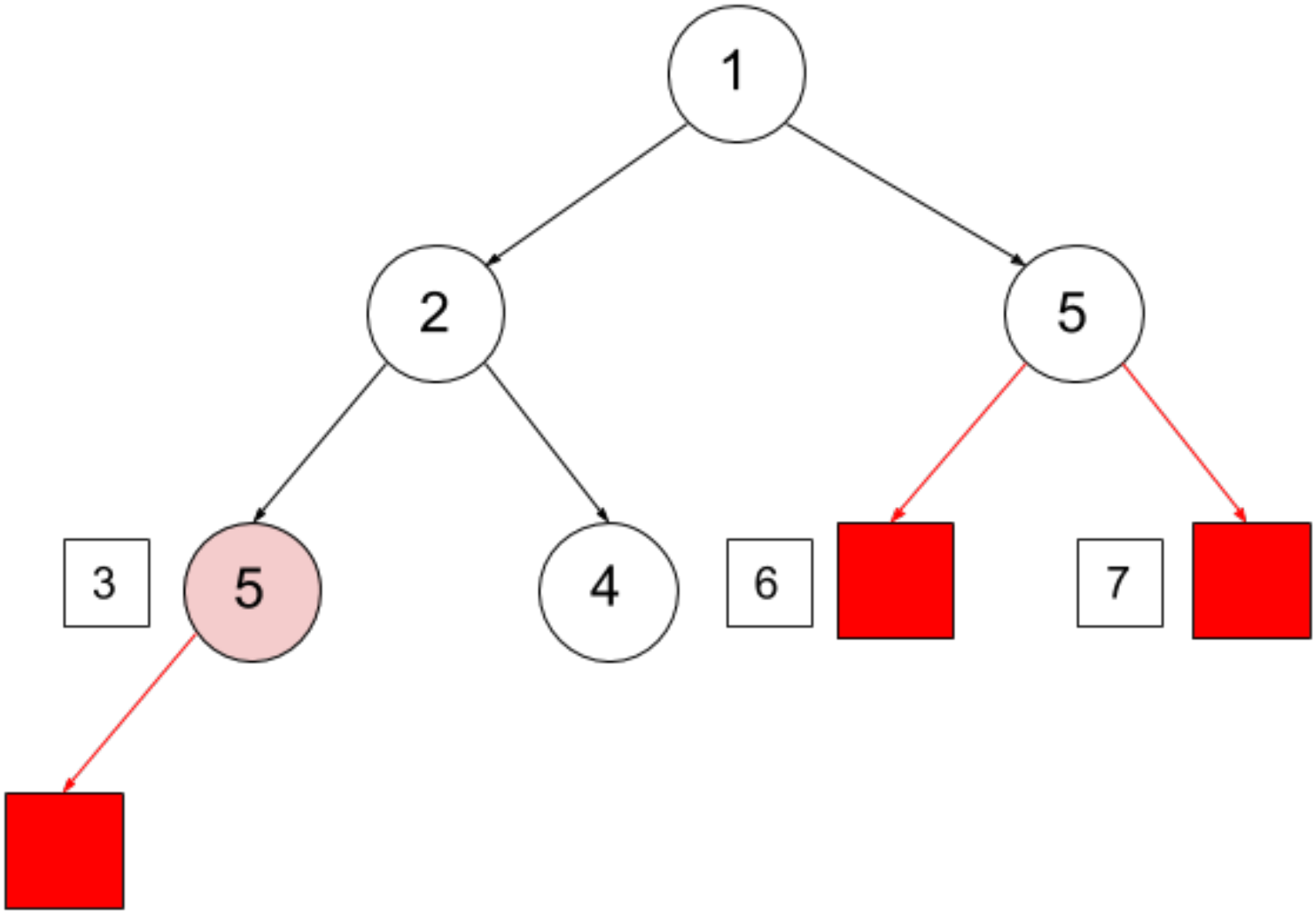
Insert. Batch-update algorithm.



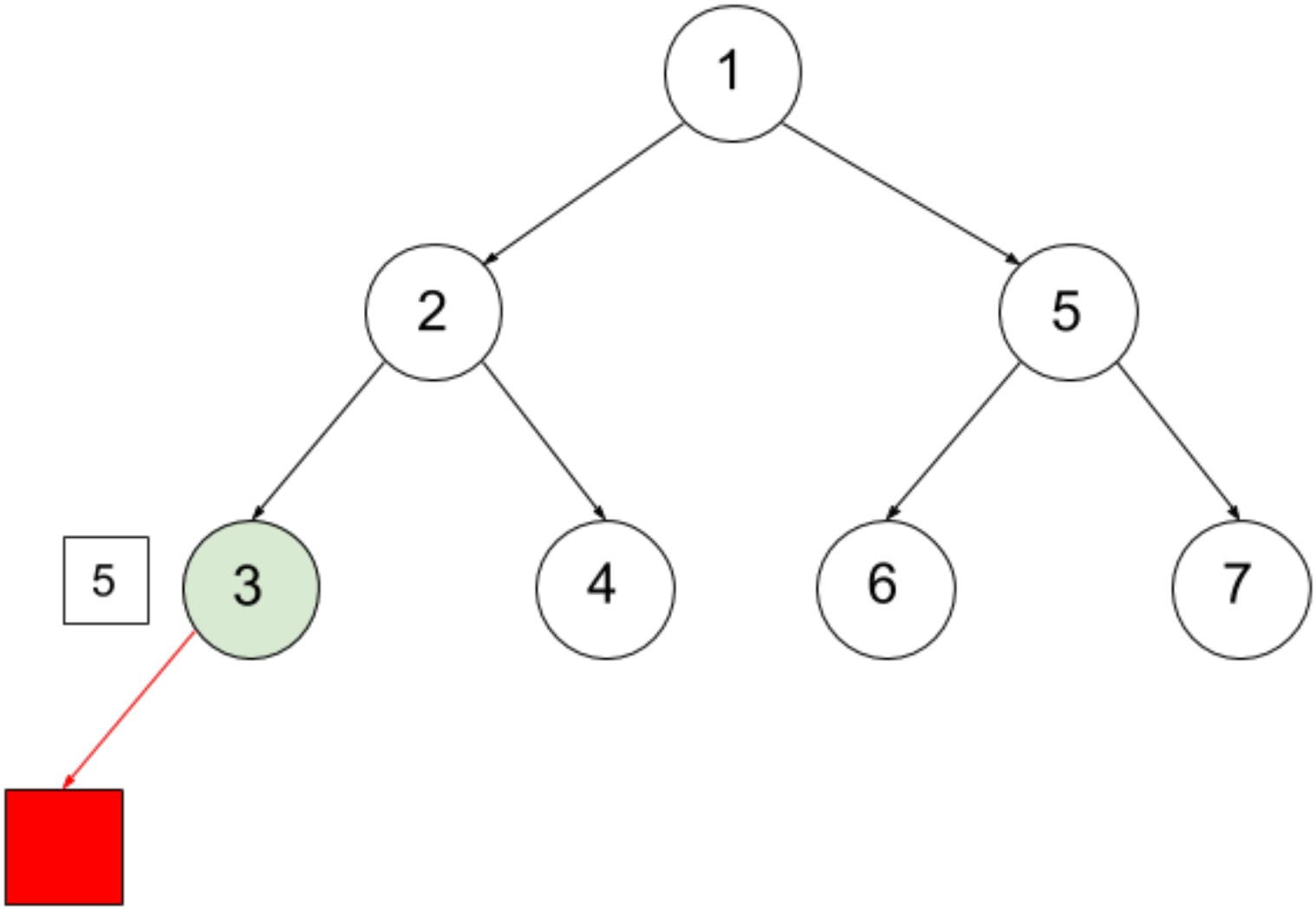
Insert. Batch-update algorithm.



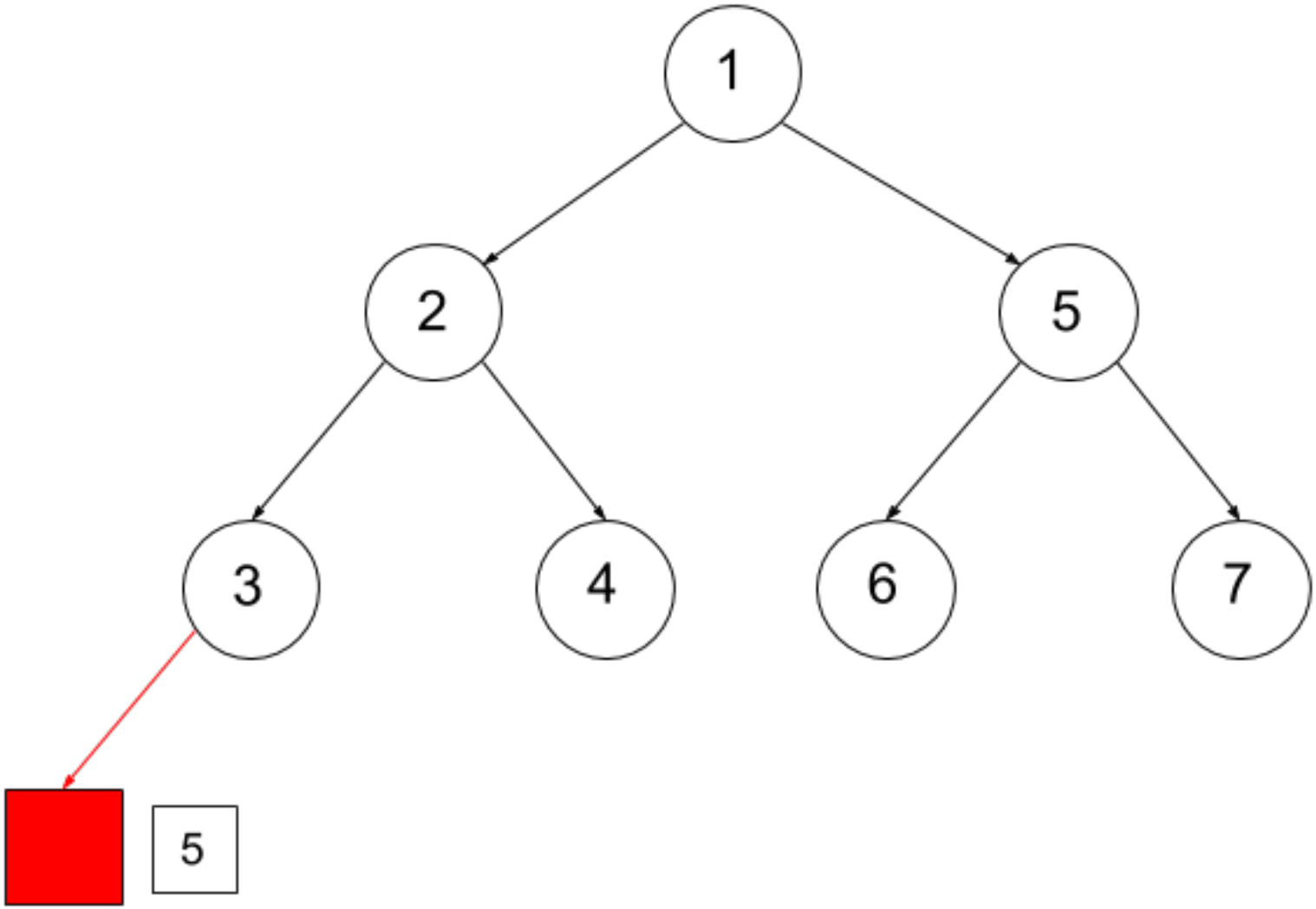
Insert. Batch-update algorithm.



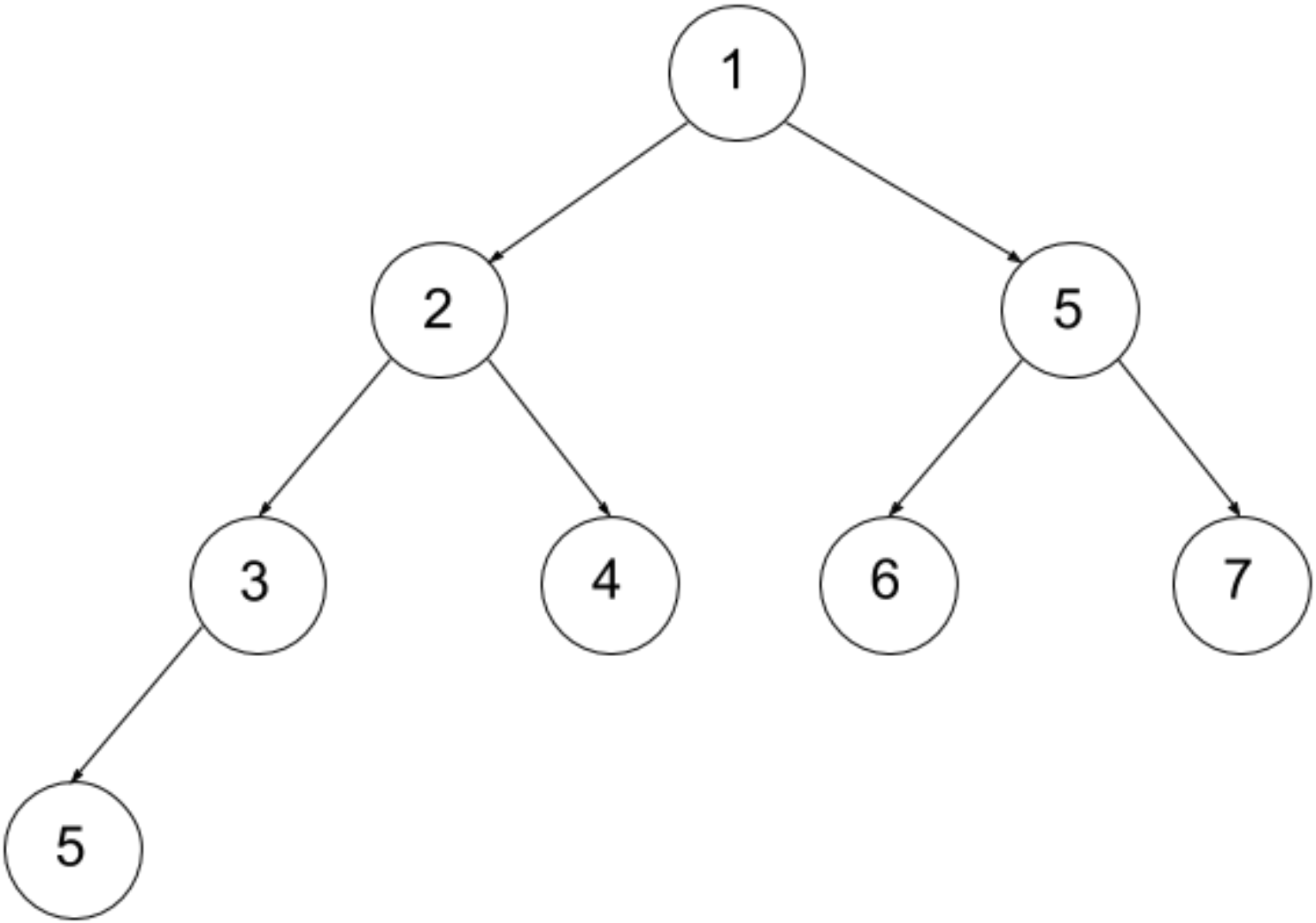
Insert. Batch-update algorithm.



Insert. Batch-update algorithm.



Insert. Batch-update algorithm.



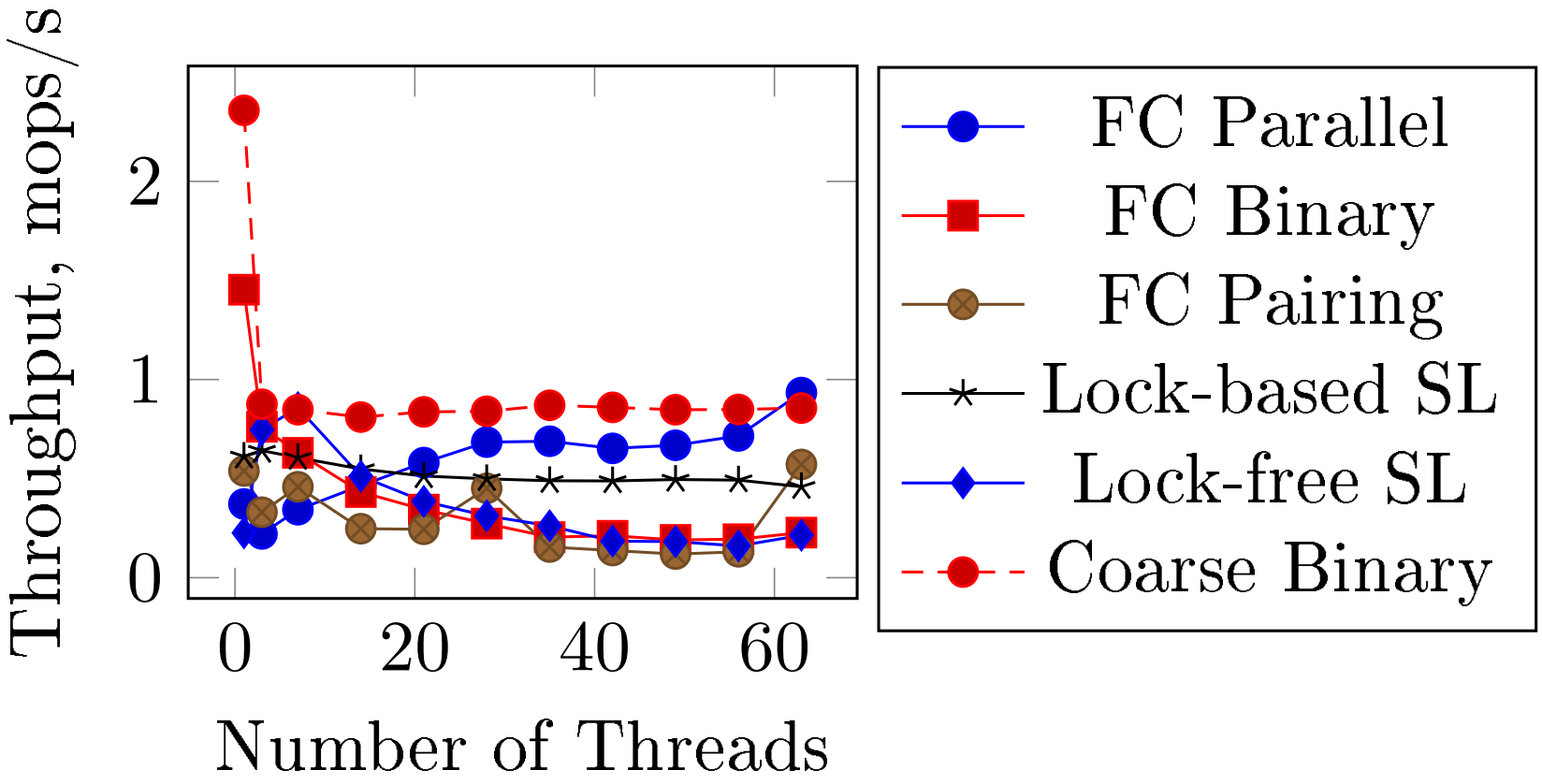
Outline

Theoretical bounds

- m batch-extractMin. $O(m \log m)$ for preparation, $O(m \log n)$ work and $O(\log n)$ span for the rest.
- m batch-insert. $O(m \log m)$ for preparation. $O(m \log n)$ work and $O(m + \log n)$ span for the rest.
- We outperform simple binary heap with flat-combining by $\frac{m \log n}{m \log m + \log n}$.
- When $m \approx \log n$, we get almost perfect speedup.

Evaluation

- Workload: 50% insert random integer and 50% extractMin.
- Initial size of the heap: $8 \cdot 10^6$.



Outline

Conclusion

Proposed Flat parallelization could be used to:

- Implement priority queue comparable to state-of-the-art approaches.
- Implement Binary Search Tree with perfect balancing. (Implemented. Works 5-10 times slower on random load.)
- Automatically provides a concurrent implementation of a data structure given PRAM counterpart. For example, dynamic tree.

Questions?

Thank you for your attention!