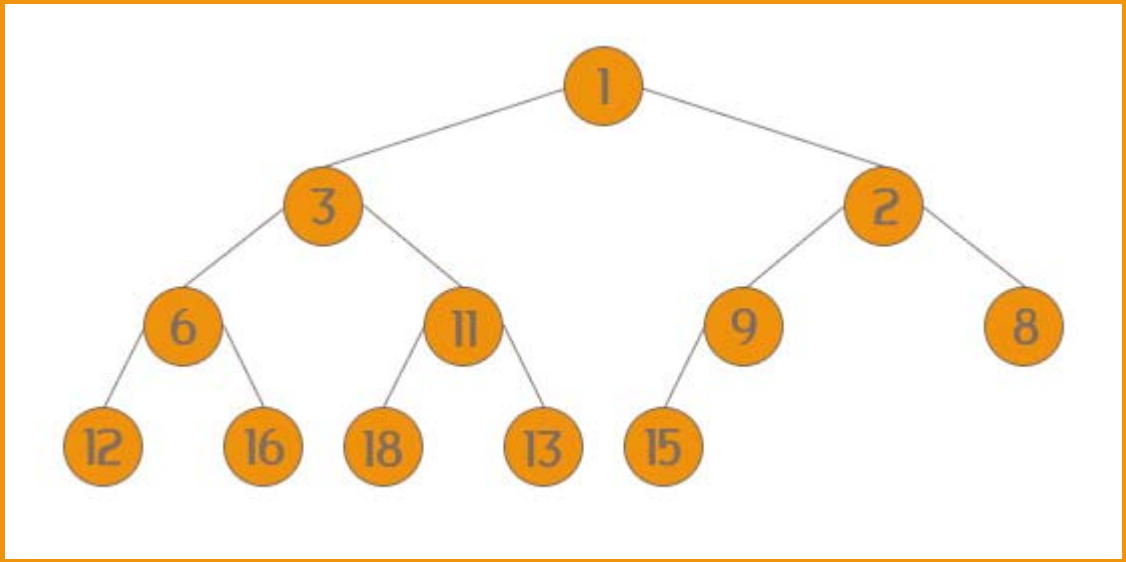
Concurrent Access of Priority Queues

R.V. Nageshwara and V. Kumar

Binary Heap

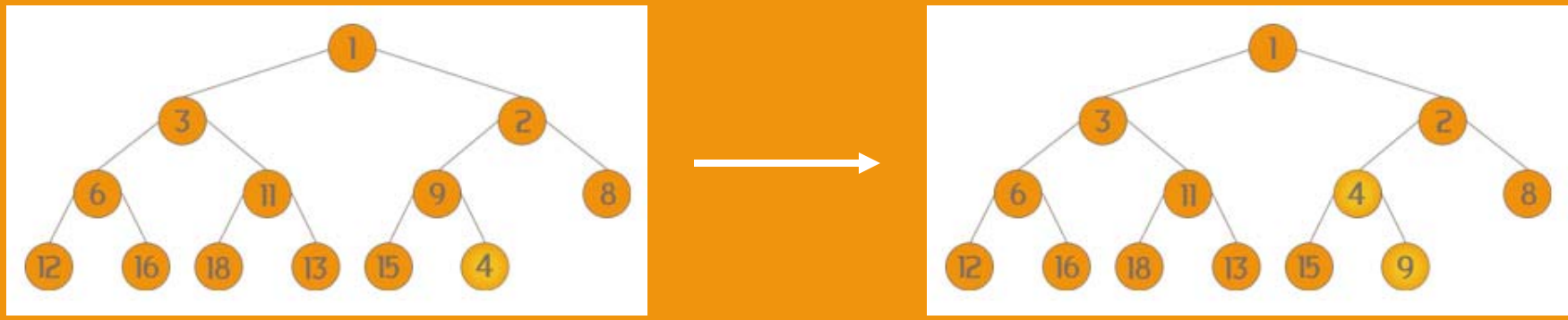
A **Binary Heap** is a data structure that is an array object that can be viewed as a nearly complete binary tree. The tree is completely filled on all levels except the lowest, which may only be partially filled. [CLRS]

Heap Order Property: Key stored at the parent is smaller or equal to the key stored at either child.



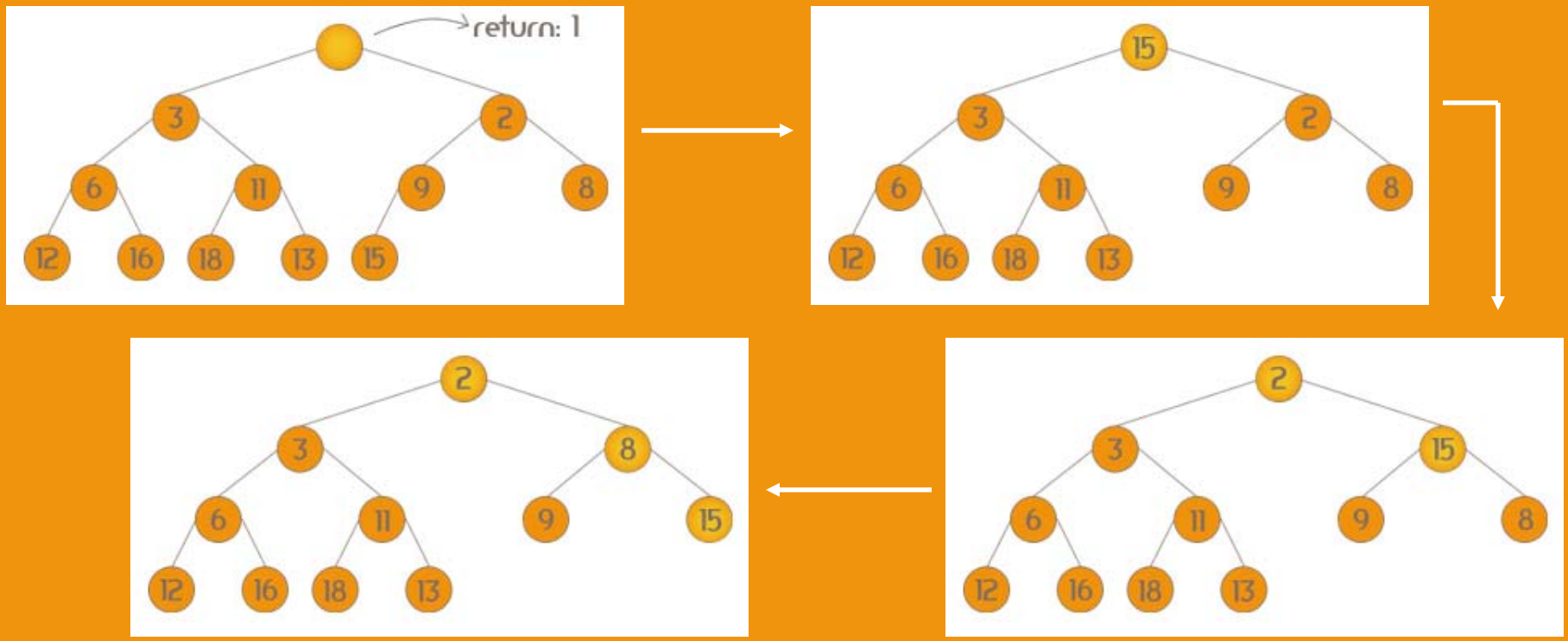
Binary Heap

Inserts into a Binary Heap are done in a bottom up manner. The new element is inserted into the next available spot and bubbled up the tree to satisfy the heap order property.



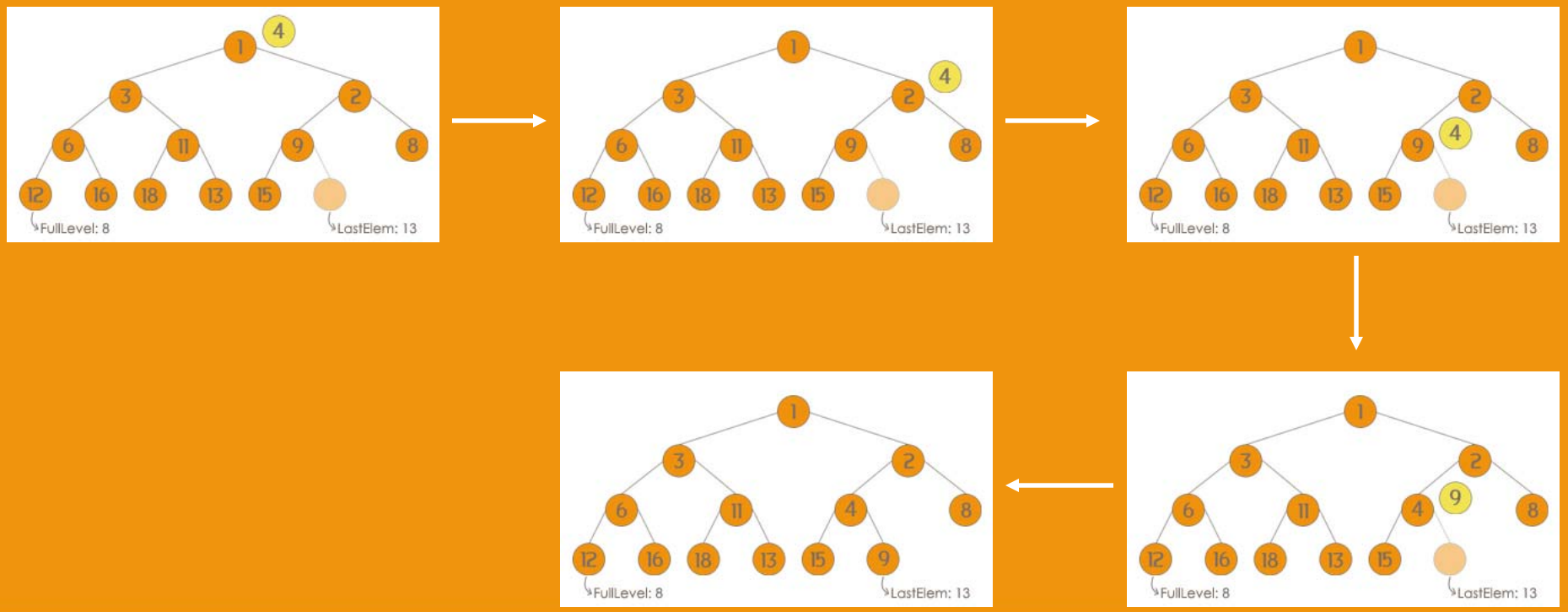
Binary Heap

Deletes from a Binary Heap are done in a top down manner. The min element is removed and replaced with the most recently inserted element. Then the element is bubbled down the tree to maintain the heap order property.



Modified Binary Heap

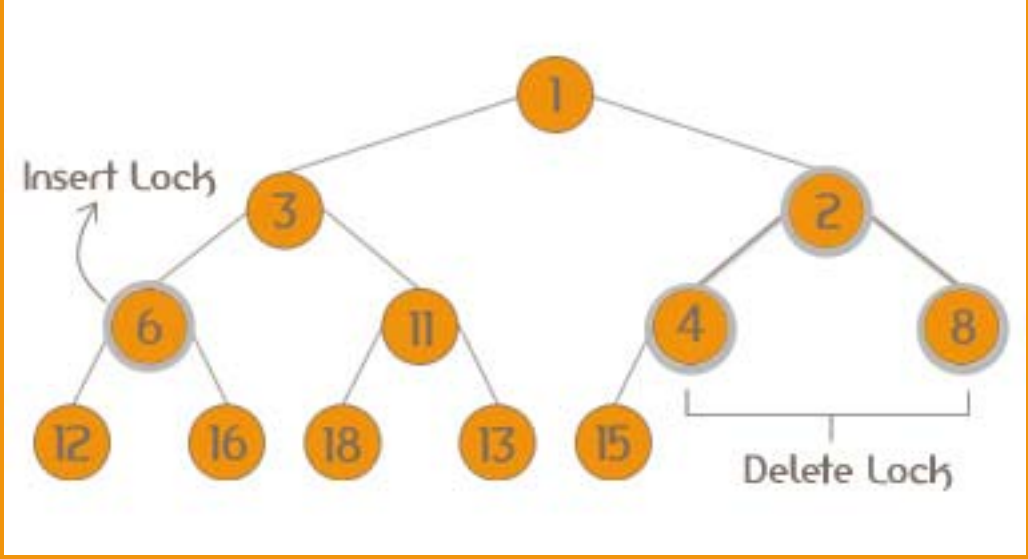
Difference = LastElem - FullLevel = 13 – 8 = 5
Height of Heap = 3
Binary Representation with 3 digits = 101
Path from Root: right → left → right



Concurrent
Binary Heap

With the top down insert procedure it now becomes possible to open the heap to multiple processes and lock it on a node by node basis.

- **Insert Processes** would lock only one node
- **Delete Processes** would lock 3 nodes: The parent and the 2 children



Implementation Details

Implementation of the concurrent heap data structure followed directly from the pseudo code provided in the original paper.

Considerations for the implementation:

- Integer based implementation v.s generic types
- Array based structure v.s linked nodes
- Lock granularity and node based locking
- Message passing mechanism for state changes

Implementation Details

Integers v.s Generic Types

- Generic types do not provides the ability to make an array neatly
- The implementation discussed in the paper makes the assumption that the tree stores data of type integer
- Defining a MAX value for some data types (Strings, user defined Objects) is difficult and thus making comparisons is not always possible

Implementation Details

Array v.s Linked Implementation

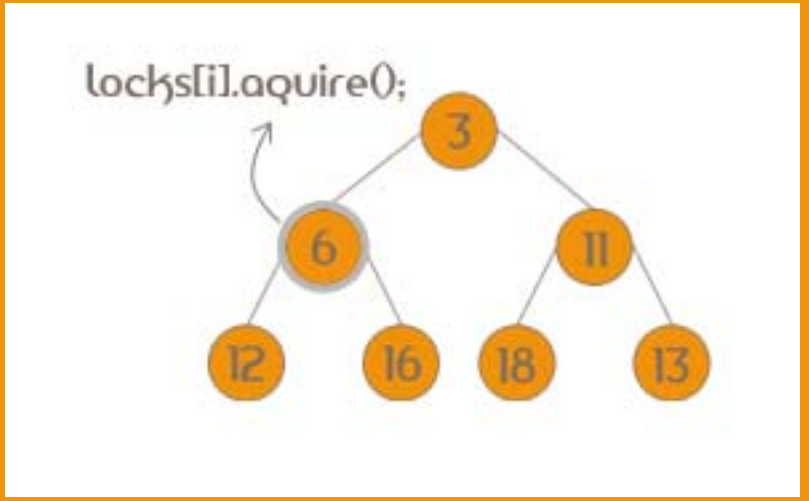
- Using an array based implementation puts a limit on the number of elements that can be inserted. The size of the heap becomes fixed
- Array implementation is necessary because the calculation of the path to the destination node for the top down insert is heavily dependant on the use of indices
- The array implementation described in the paper uses indices that start at 1 instead of the traditional 0. This in this implementation all arrays are of size HEAP_SIZE + 1 and the first element (at index 0) is always empty

Implementation Details

Node Based Locking

- Since the implementation of the heap was array based the series of locks that corresponded to each node were also in an array
- Semaphores were used to create the locks for each node

```
Semaphore[] locks
    = new Semaphore[HEAP_SIZE+1]
```



Implementation Details

Message Passing for Node State Changes

- One of the tricky parts of this algorithm is checking and changing the state of each node
- When a delete process requires a node that has not yet been inserted it sets the state of the destination node as wanted and then waits for it to be filled

Status	Meaning
PRESENT	A key exists at the node
PENDING	An insert is in progress which will ultimately insert a value to the node
WANTED	A delete process is waiting for the key
ABSENT	No key is present

Implementation Details

Message Passing for Node State Changes

- In the implementation the delete process sets the status of the required node to *WANTED*, deletes the value of the root node, sets the root node's status to *ABSENT* and then waits on the root node. The insert process then will then check at each iteration if the status of the destination node has changed and if so it will change the status to *ABSENT* and then put what ever node it can into the root, change the root's status to *PRESENT* and notify on the root node

```
delete()
{
        …
        status[1] = ABSENT;
        status[j] = WANTED;
        …
        while (status[1] == ABSENT)
        Thread.sleep(1);
        …
}
```

Implementation Details

Other Considerations

- Every time a heap is created the MAX_INT value needs to be specified. This is because the re-heapification loop that was designed by the authors requires that all nodes have their values set to MAX_INT instead of empty.
- There are no guards in the original algorithm that check for array index being within the bounds of the actual heap array during the heapification loop, they assume that all will be well. This was found to be not the case for some sample runs.

Implementation Details

Testing

- Correctness of the heap implementation was validated by using a single thread and running a random series of inserts and deletes and monitoring the progress of the data structure with a print method.

- The algorithm was tested on a quad-core machine with a heap of size 10000. There were 10000 randomly generated inserts and deletes, the inserts would insert a random integer in the range of [0,10000]. There was a total of 500 times that each experiment was run to get the average running time of the set (for higher thread count only 10 samples were run).

Thread Count	Running Time in ms
1 (10000 op ea)	54
2 (5000 op ea)	~2500
4 (2500 op ea)	~3500
10 (1000 op ea)	~3600

Implementation Details

Verification Considerations

- Test for deadlocks in the delete and insert processes
- Create boundary test cases that would cause incorrect heap behavior in an ordinary heap and check to see if those cases are treated properly
- Check if the status of each node is actually updated safely, personal inclination is that the authors did not create a safe access policy for reading/writing the status of each node, or went on the assumption that reading and writing the status would be an atomic process

Sources:

[NK]

R.V. Nageshwara, V. Kumar. *Concurrent Access of Priority Queues*. IEEE Transactions on Computers, 37(12): 1657-1665, December 1988.

[CLRS]

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001.1