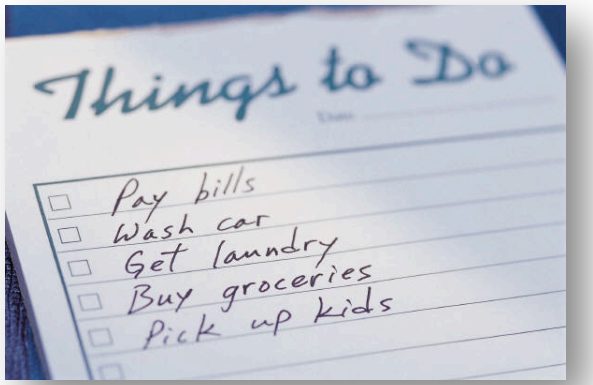
Priority Queue ADT & Heaps

Goals

- Introduce the Priority Queue ADT
- Heap Data Structure Concepts

Priority Queue ADT

- Not really a FIFO queue – misnomer!!
- Associates a "priority" with each element in the collection:
  - First element has the highest priority (typically, lowest value)
- Applications of priority queues:
  - To do list with priorities
  - Active processes in an OS




```
  UID   PID  PPID   C STIME   TTY           TIME CMD                         F PRI NI       SZ
    0     1     0   0  3Aug12 ??         5:28.80 /sbin/launchd    80004004  31  0  2453620
    0    11     1   0  3Aug12 ??         0:07.31 /usr/libexec/Use     4004  33  0  2466560
    0    12     1   0  3Aug12 ??         0:11.41 /usr/libexec/kex     4004  33  0  2454064
    0    14     1   0  3Aug12 ??         0:15.60 /usr/sbin/notify     4004  33  0  2455568
    0    15     1   0  3Aug12 ??         0:05.44 /usr/sbin/diskar     4004  33  0  2444868
```

Priority Queue ADT: Interface

- Next element returned has highest priority

```
void    add(newValue);
TYPE    getMin();
void    removeMin();
```

Priority Queue ADT: Implementation

Heap: has 2 completely different meanings

1. Classic data structure used to implement priority queues
2. Memory space used for dynamic allocation

We will study the data structure (not dynamic memory allocation)

Priority Queue ADT: Implementation

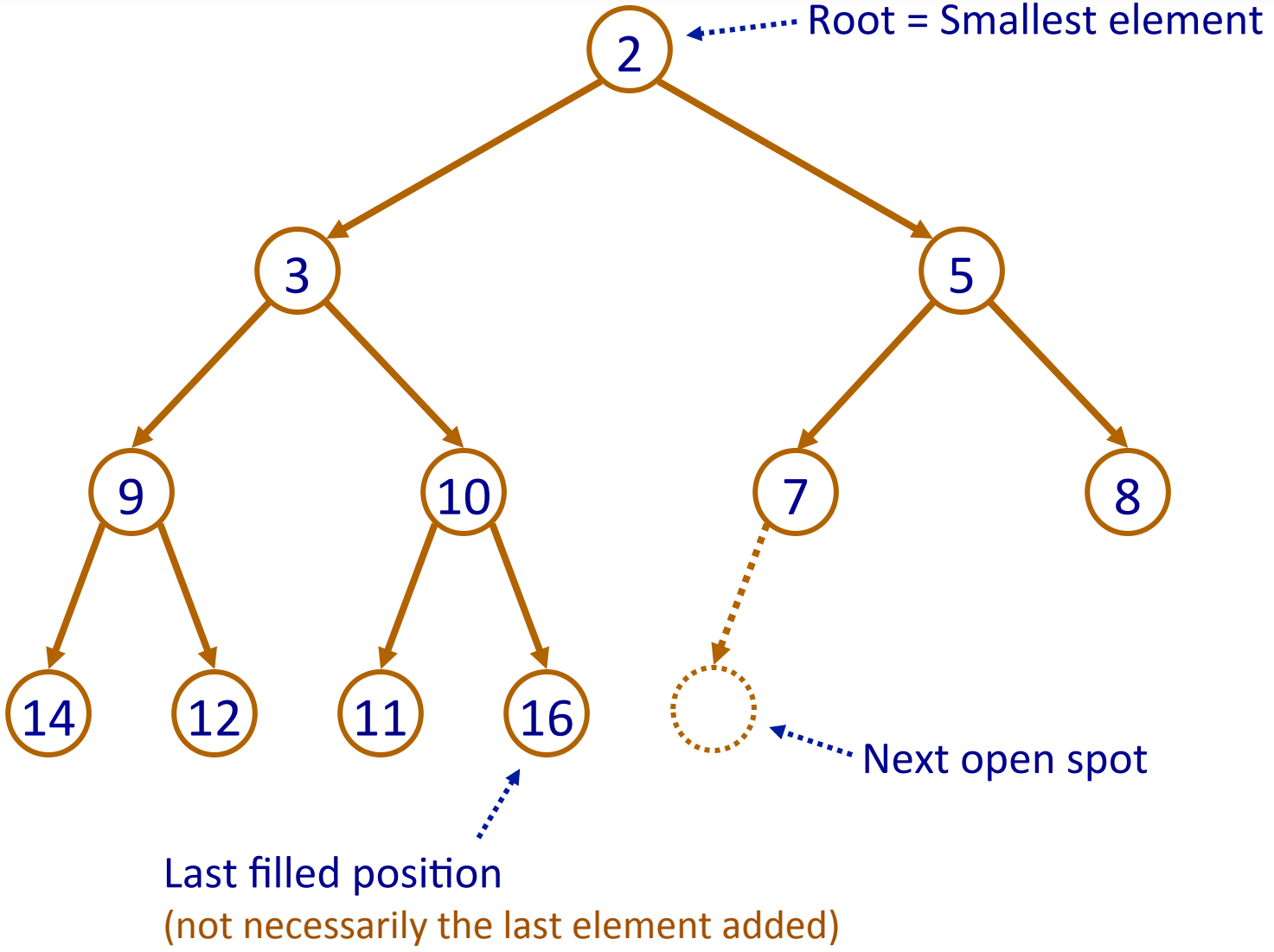
Binary Heap data structure: a *complete* binary tree in which every node's value is less than or equal to the values of its children (min heap)

Review: a complete binary tree is a tree in which

1. Every node has at most two children (binary)
2. The tree is entirely filled except for the bottom level which is filled from left to right (complete)

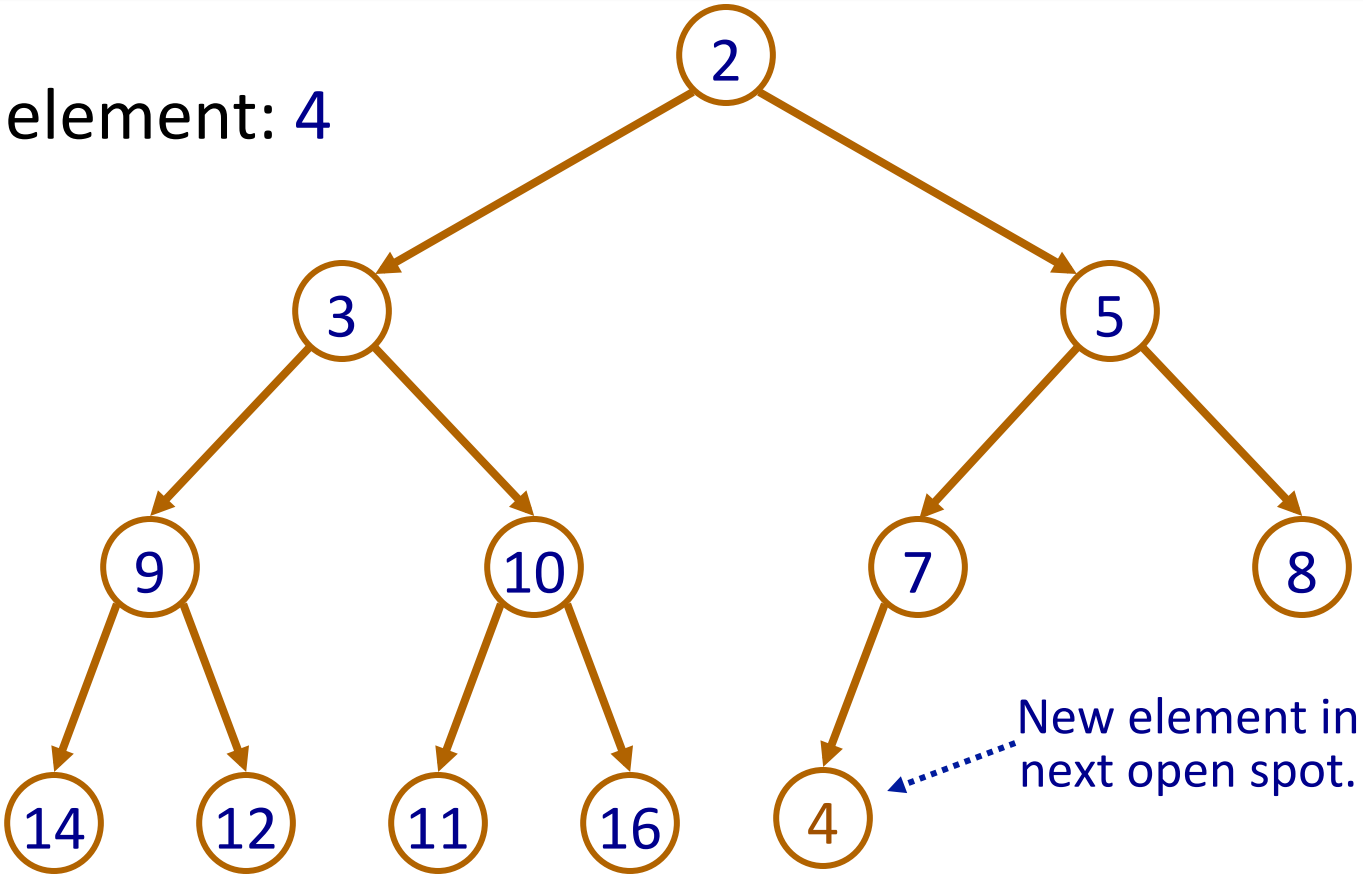
– Longest path is ceiling(log *n*) for *n* nodes

Min-Heap: Example



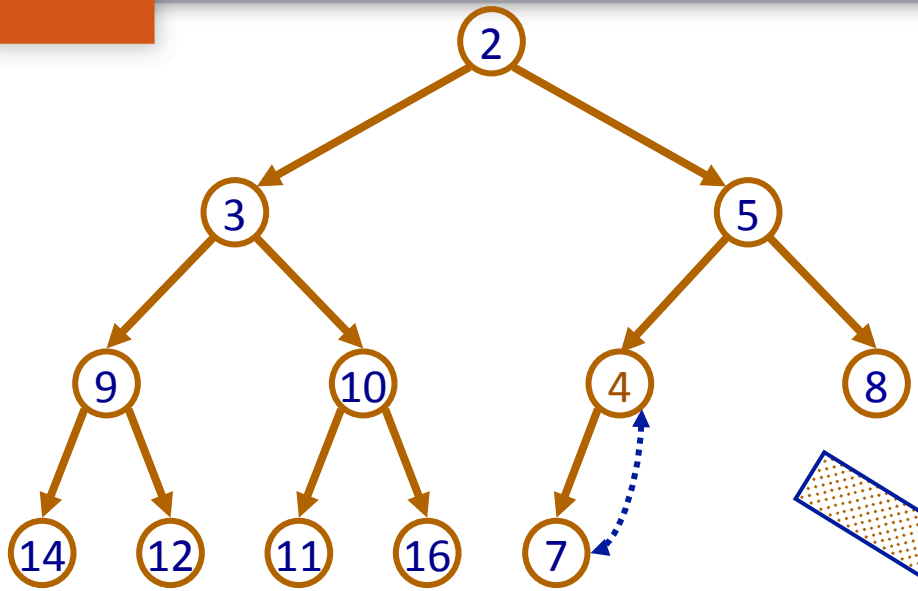
Maintaining the Heap: Addition

Add element: 4

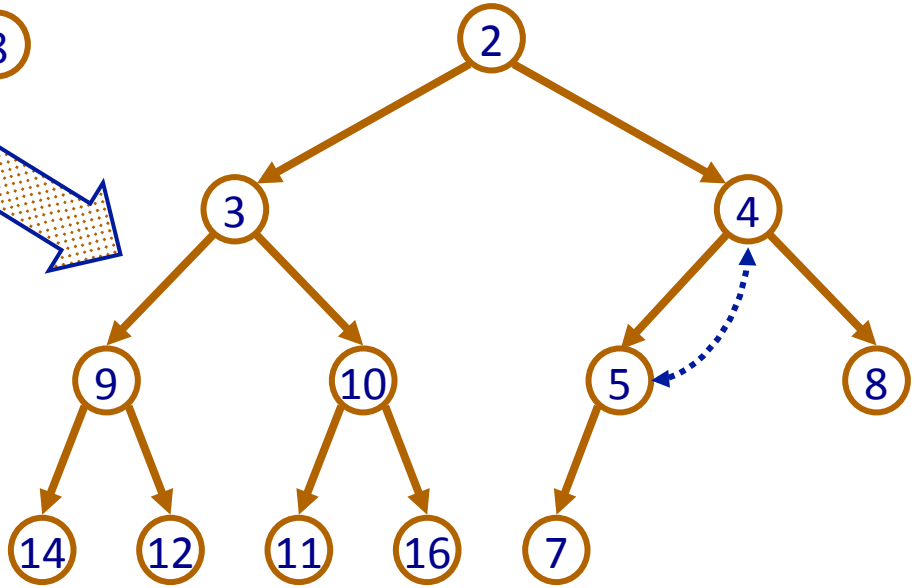


Place new element in next available position,
then fix it by "percolating up"

Maintaining the Heap: Addition (cont.)



After first iteration (swapped with 7)



After second iteration (swapped with 5)

New value not less than parent → Done

Percolating up:

while new value is less than parent,
swap value with parent

Maintaining the Heap: Removal

- Since each node's value is less than or equal to the values of its children, the root is always the smallest element

- Thus, the operations **getMin** and **removeMin** access and remove the root node, respectively

- Heap removal **(removeMin)**:

  What do we replace the root node with?

  Hint: How do we maintain the completeness of the tree?

Maintaining the Heap: Removal

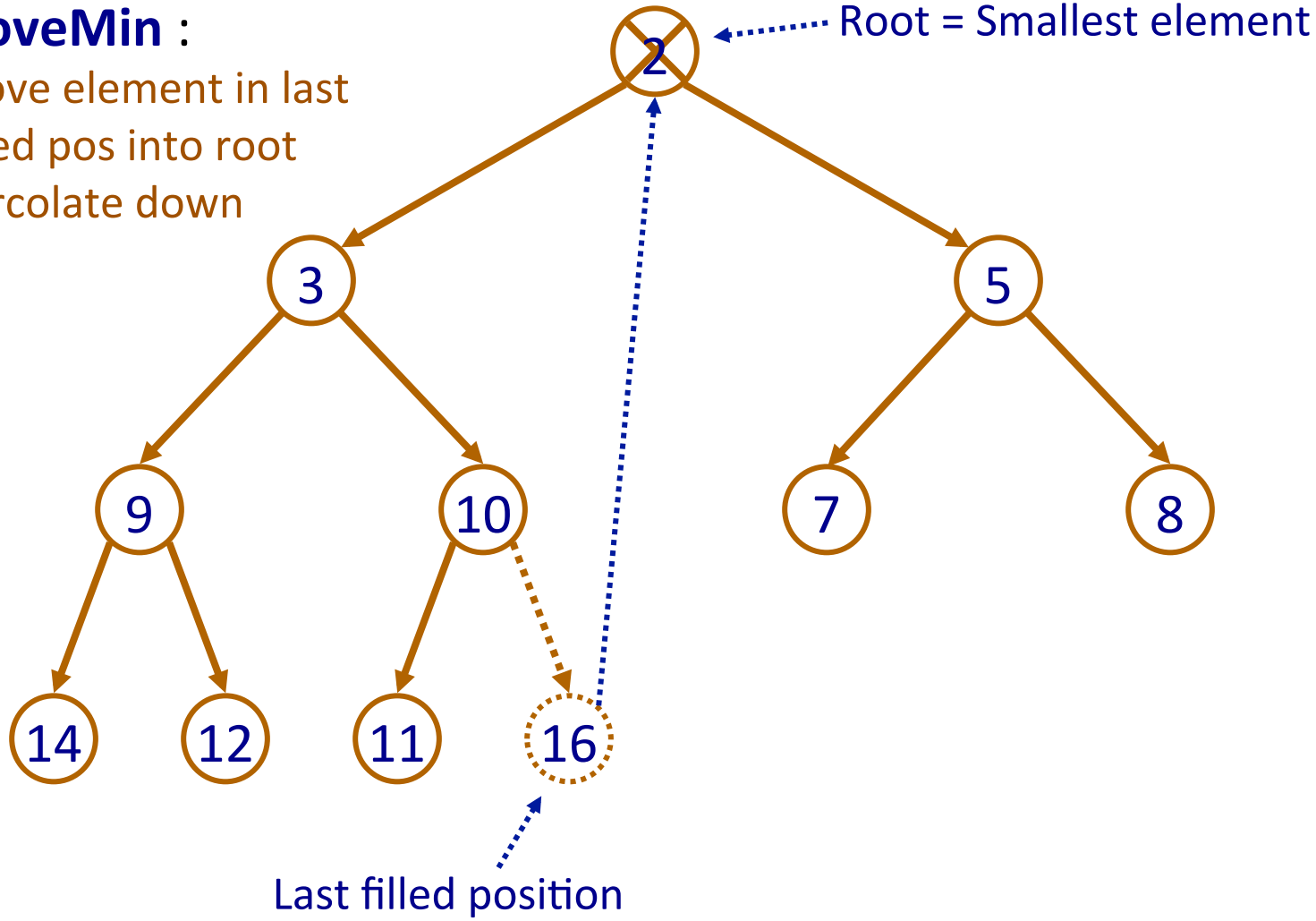
Heap removal **(removeMin)**:

1. Replace root with the element in the last filled position
2. Fix heap by "percolating down"

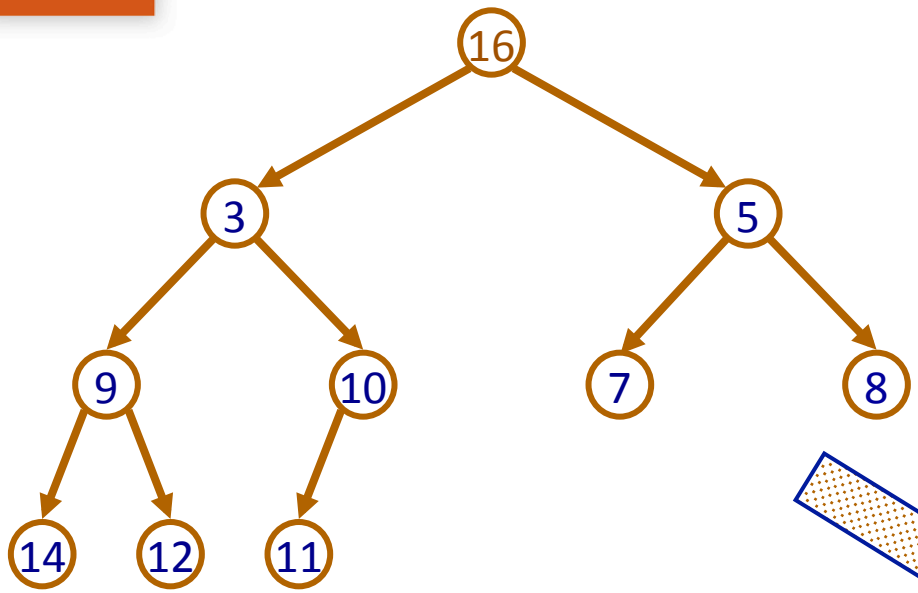
Maintaining the Heap: Removal

removeMin :

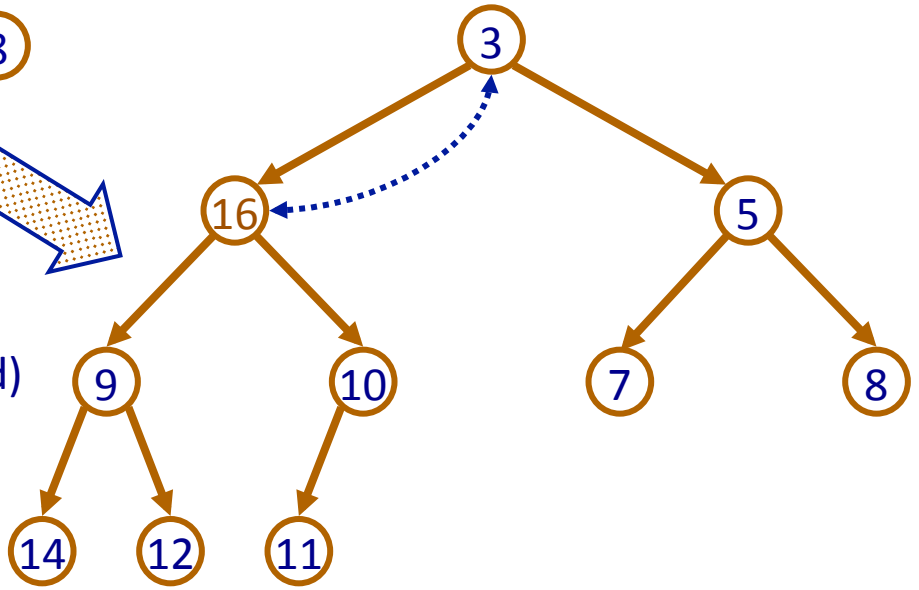
1. Move element in last filled pos into root
2. Percolate down



Maintaining the Heap: Removal (cont.)



Root value removed
(16 copied to root and last node removed)

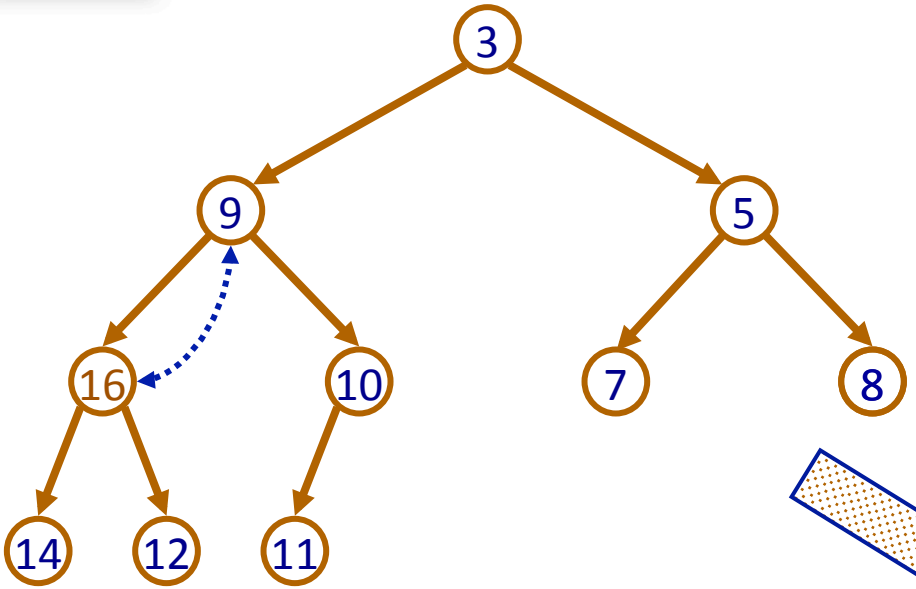


After first iteration (swapped with 3)

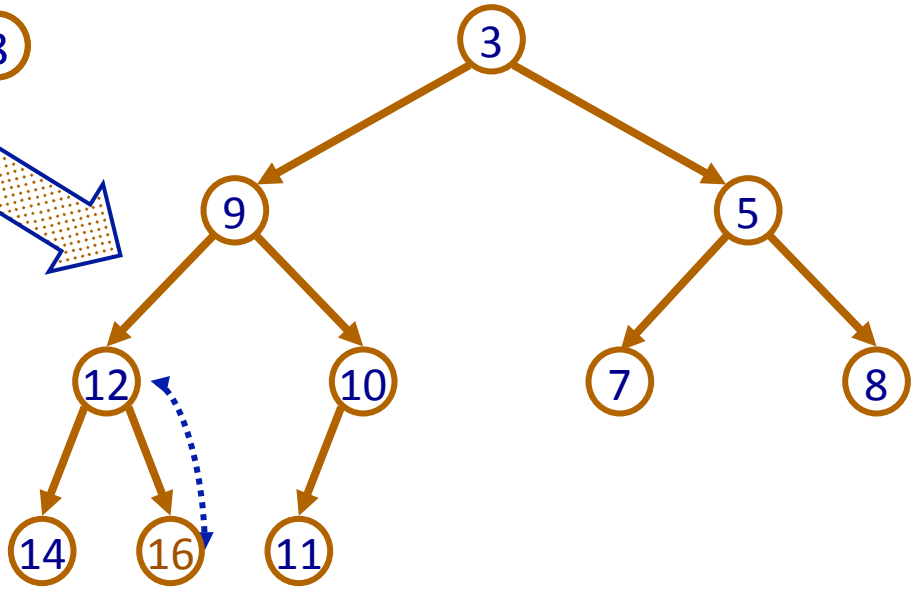
Percolating down:

while greater than smallest child
swap with smallest child

Maintaining the Heap: Removal (cont.)



After second iteration (moved 9 up)



Percolating down:

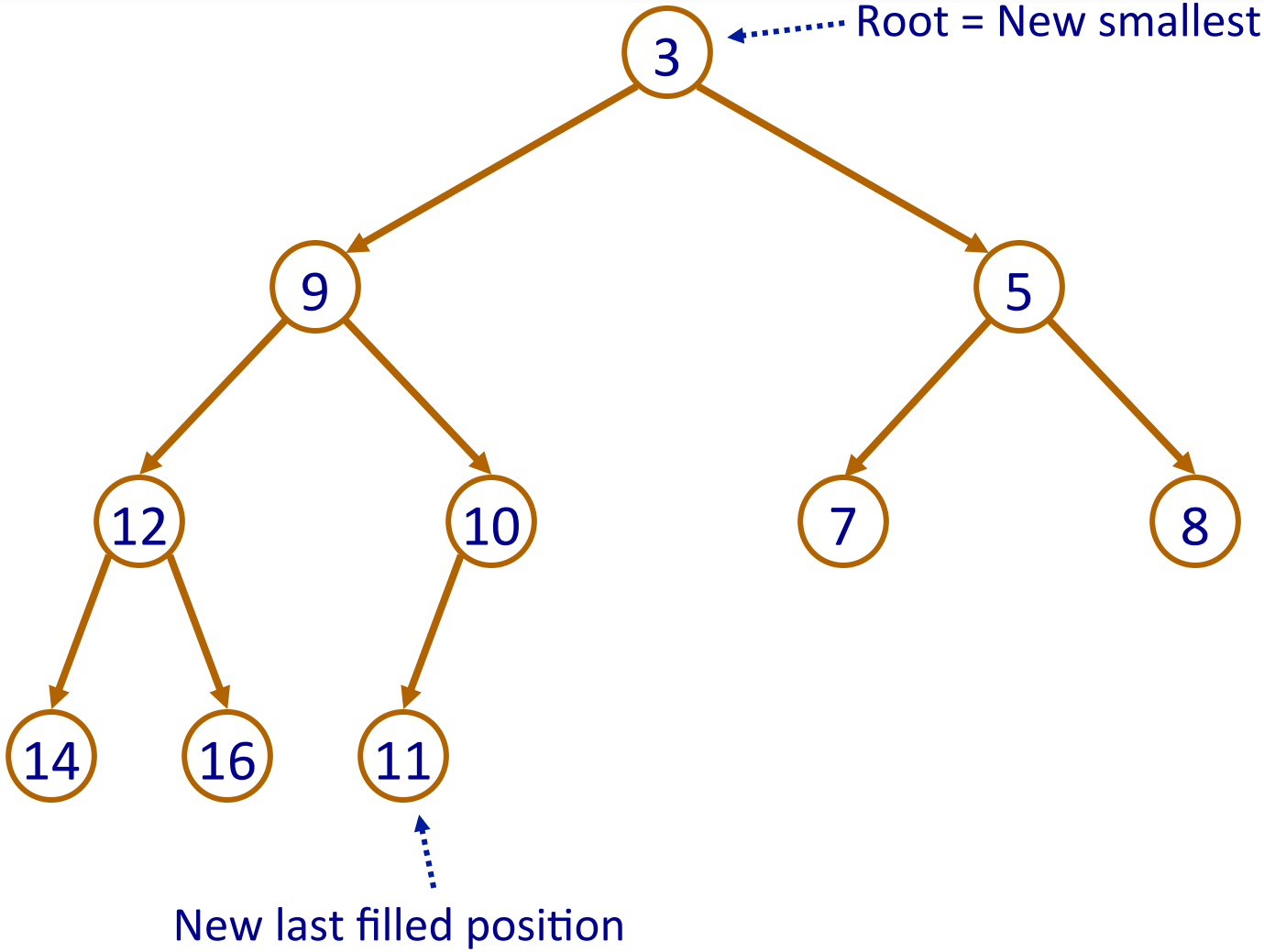
while greater than smallest child

swap with smallest child

After third iteration (moved 12 up)

Reached leaf node → Stop percolating

Maintaining the Heap: Removal (cont.)

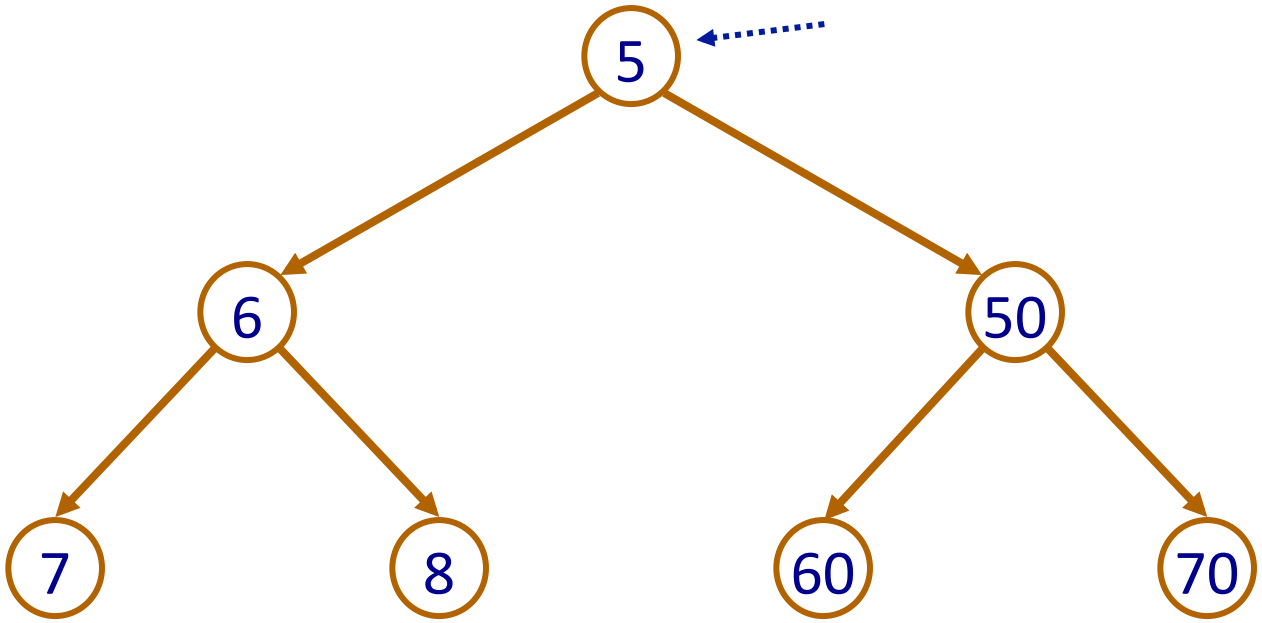


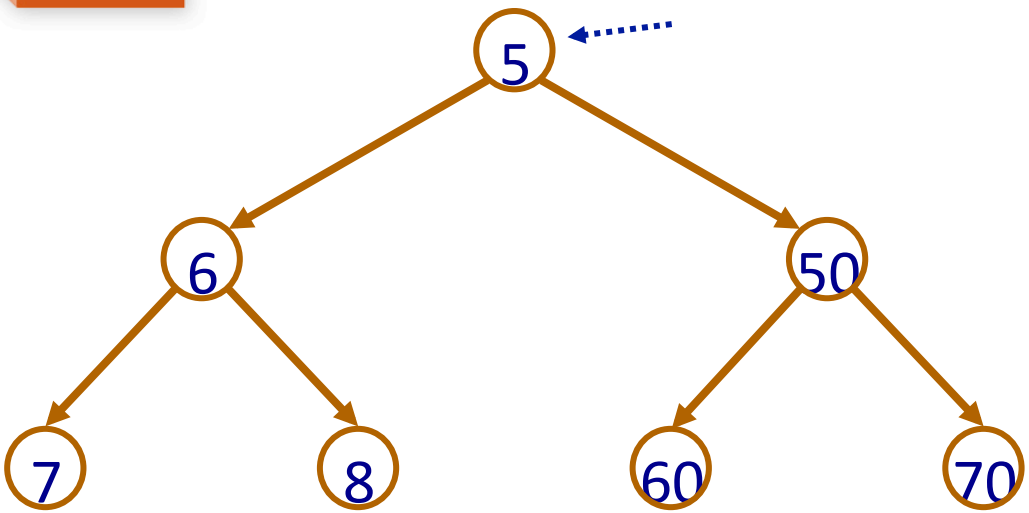
Practice

Insert the following numbers into a main-heap in the order given: 54, 13, 32, 42, 52, 12, 6, 28, 73, 36

Practice

Remove the minimum value from the min-heap




5
6
50
7
8
60
70

Your Turn

- Complete Worksheet: Heaps Practice