Heaps and Priority Queues

COMP2611: Data Structures
2019/2020

Outline

- Priority Queue ADT
  - Motivations
  - Operations
  - Naive Implementations
- Heaps
  - Binary Heaps as a special binary tree
  - Min Heaps vs Max Heaps
  - Insertion and removal in Heaps
  - Heaps as an efficient way to implement a Priority Queue

Priority Queue

- Queues operate in FIFO manner
  - “Clients” are served in order of arrival
  - Useful way to order data for sequential processing
  - Not always useful for many cases :-(
- Sometimes we want to serve clients in (ascending or descending) order of some priority value:
  - Patients in a hospital ER room
  - Bandwidth and connection management with some networking applications
  - Process on multi-tasking processor
  - Edges in a graph for processing during Dijkstra and Prim’s algorithm

Priority Queue

- Assume that our data (“clients”) have among their other fields, a priority or importance score
  - Sometimes an id field that uniquely identifies data (think primary key)
- Want to enqueue data…
- Then remove most important item…
- Sometimes we might need to update priority of an item using item id

Priority Queue ADT

- **enqueue** - inserts data into the priority queue
- **dequeue** - removes most important data from the priority queue
- (optionally) **update** - updates priority of data by id

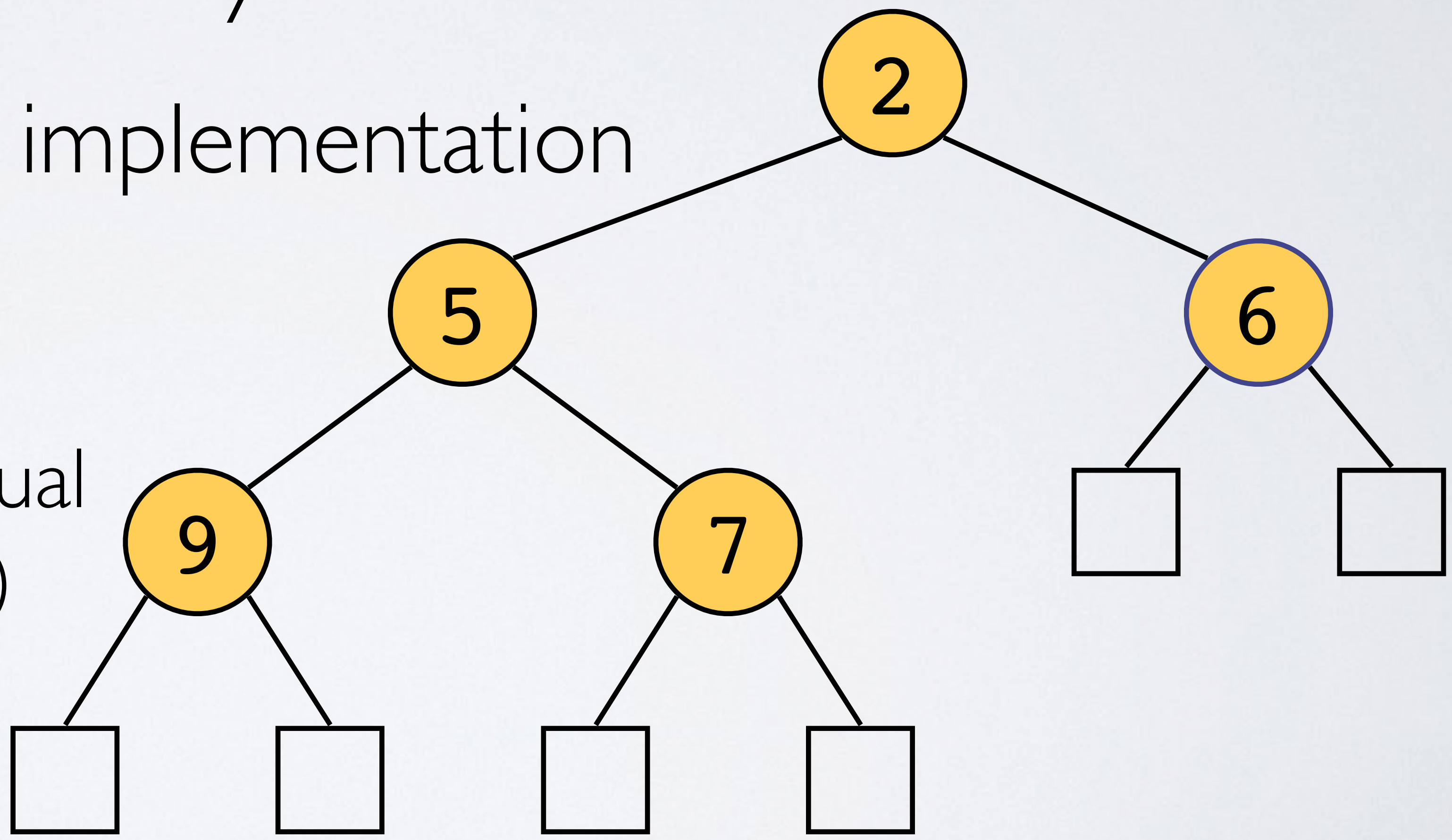
Priority Queue Efficiency

Implementation	insert	pop	update
Unsorted Dynamic Array	`O(1)`	`O(n)`	`O(n)`
Sorted Dynamic Array	`O(n)`	`O(1)`	`O(n)`
Dictionary (Hashtable)	`O(1)`	`O(n)`	`O(1)`
Binary Heap	`O(logn)`	`O(logn)`	`O(n)*`

`*` - Binary Heap + Dictionary is `O(logn)`

What is a Binary Heap?

- Data structure often used to implement a priority queue
- Binary Heaps are Binary Trees:
  - Typically use Array implementation
  - Parent is either
    - Greater than or equal to child (max heap)
    - Less than or equal to child (min heap)



Heap Properties

- Binary tree
  - each node has at most **2** children
- Each node has a priority (usually we make this the key of the entry in the heap)
- Heap has an order
  - min-heap: n.parent.key $\leq$ n.key
  - max-heap: n.parent.key $\geq$ n.key
- Left-complete
- Height of **`O(log n)`**

Min-heaps

- Will look at min heaps as examples
- But trivial to modify for max heaps
- Typically, we just need to fix comparisons
- We will use trees to visualise operations, but remember you will use the array representation of the binary tree for heaps!

Utility functions

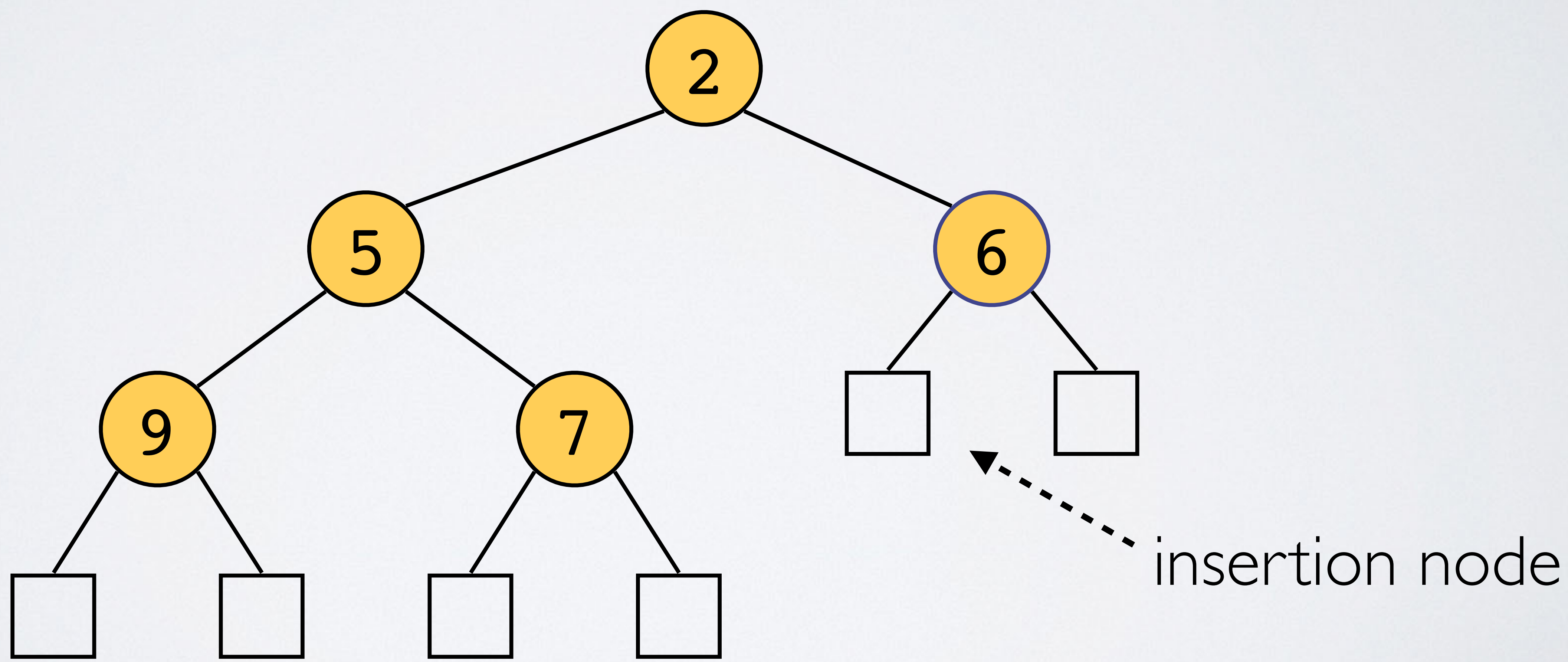
```
function parent(index):
  return floor(index / 2)
```

```
function left_child(index):
  return index * 2
```

```
function right_child(index):
  return index * 2 + 1
```

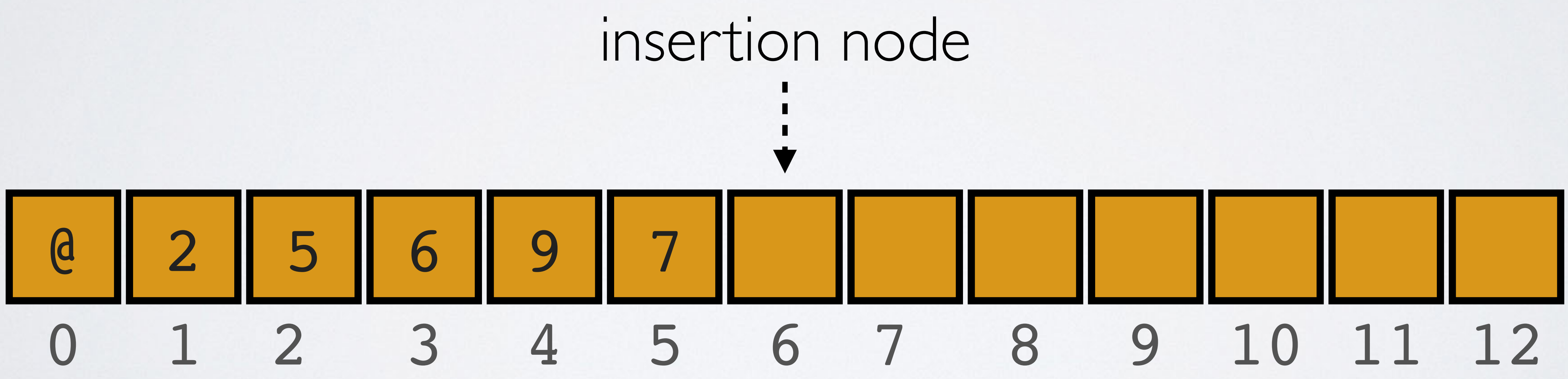
Heap — insert( )

- Need to keep track of "insertion node"
  - leaf where we will insert new node…
  - …so we can keep heap left-complete



Heap — insert( )

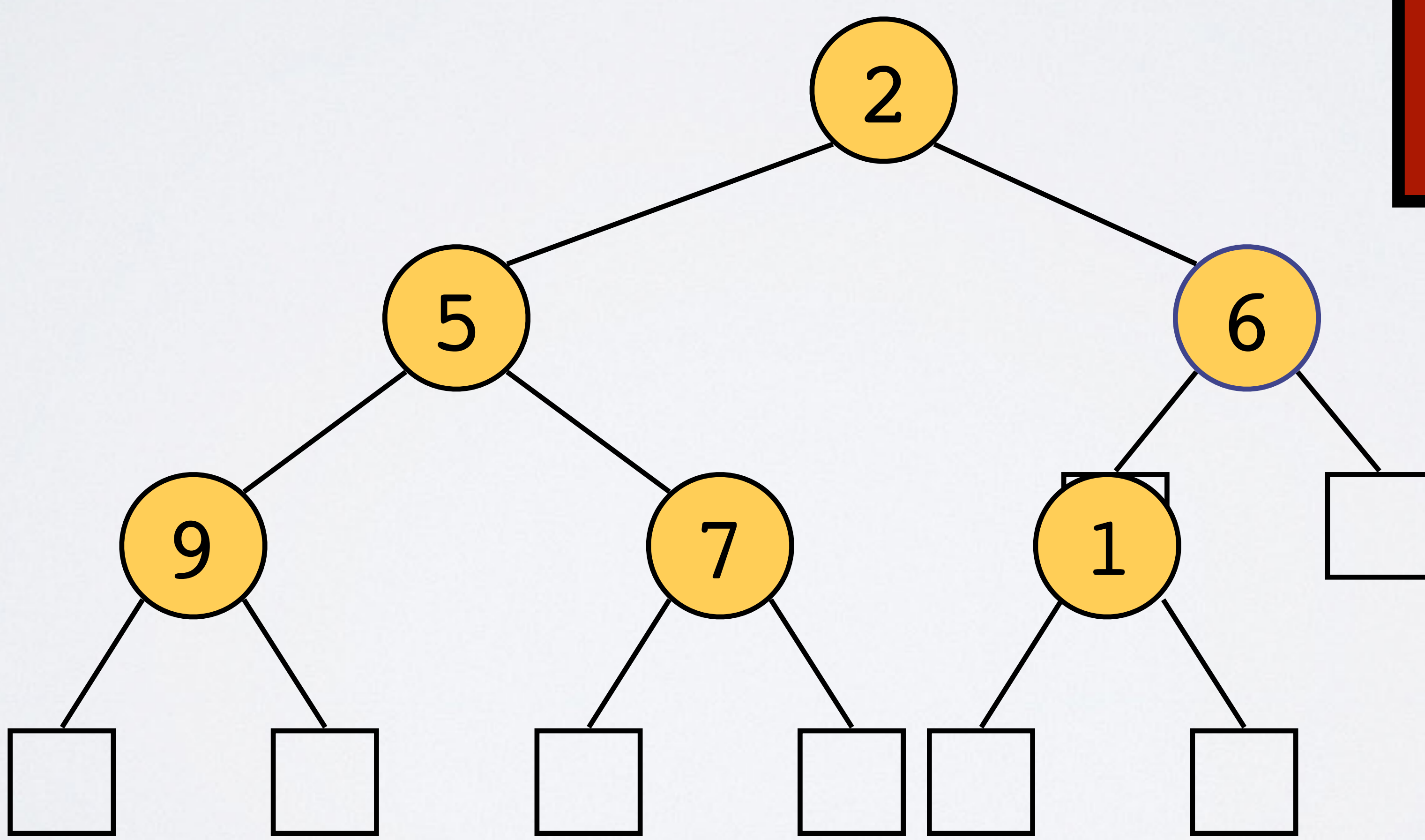
- Need to keep track of "insertion node"
  - leaf where we will insert new node…
  - …so we can keep heap left-complete



Heap — insert( )

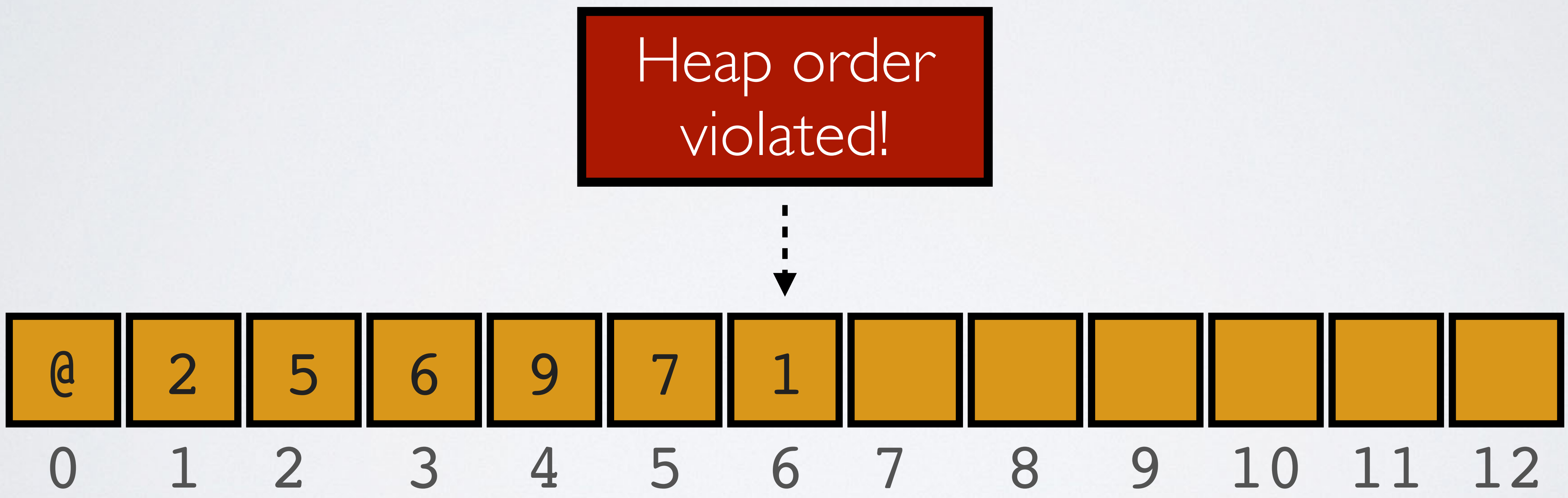
- Ex: insert(1)
  - replace insertion node w/ new node

Heap order violated!



Heap — insert( )

- Need to keep track of "insertion node"
  - leaf where we will insert new node…
  - …so we can keep heap left-complete

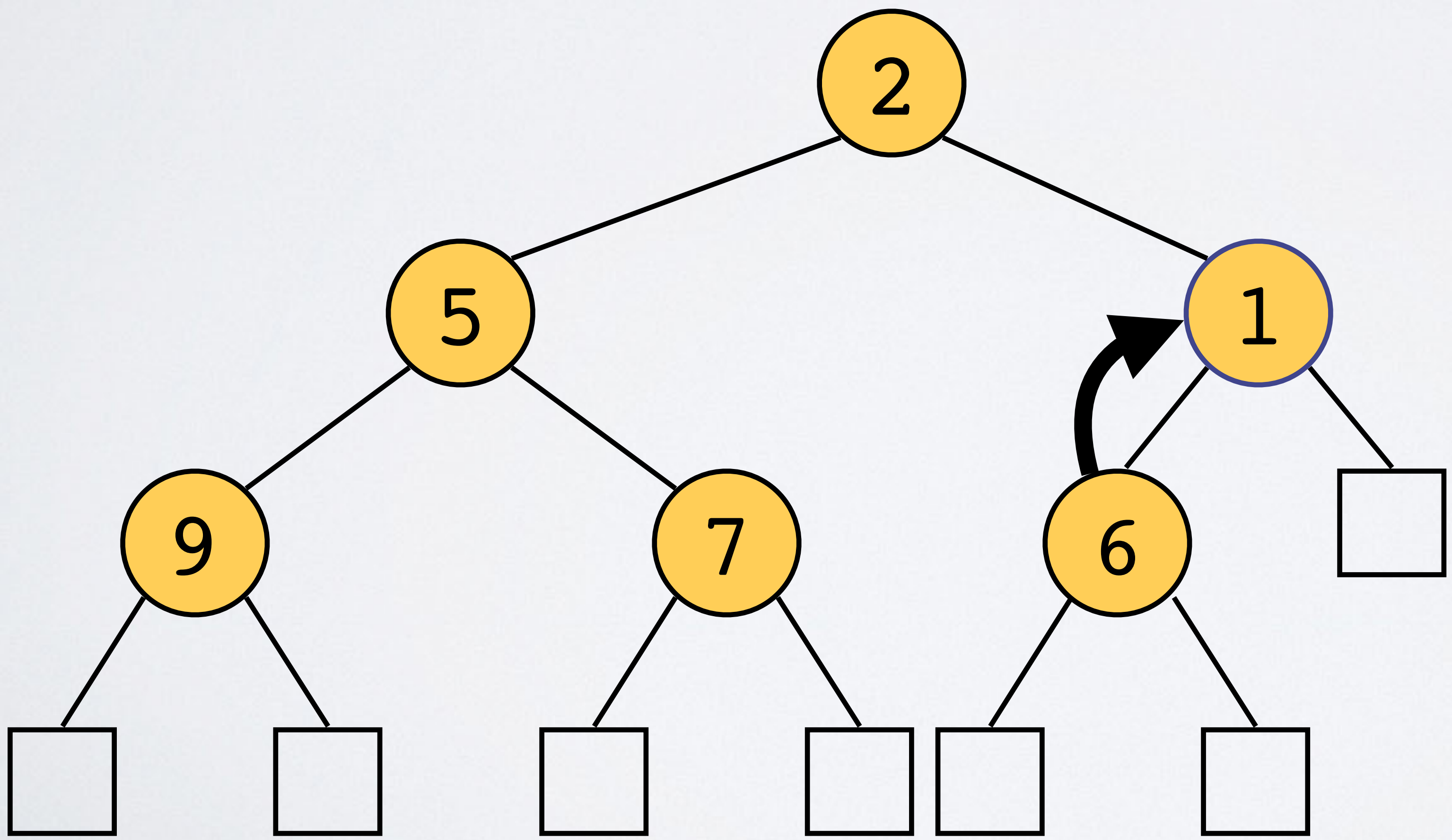


Heap - insert()

- If we insert data at the end of the heap, we would destroy Heap properties :-(
- Solution:
  - insert at end
  - repair heap
    - move entry up through tree if needed

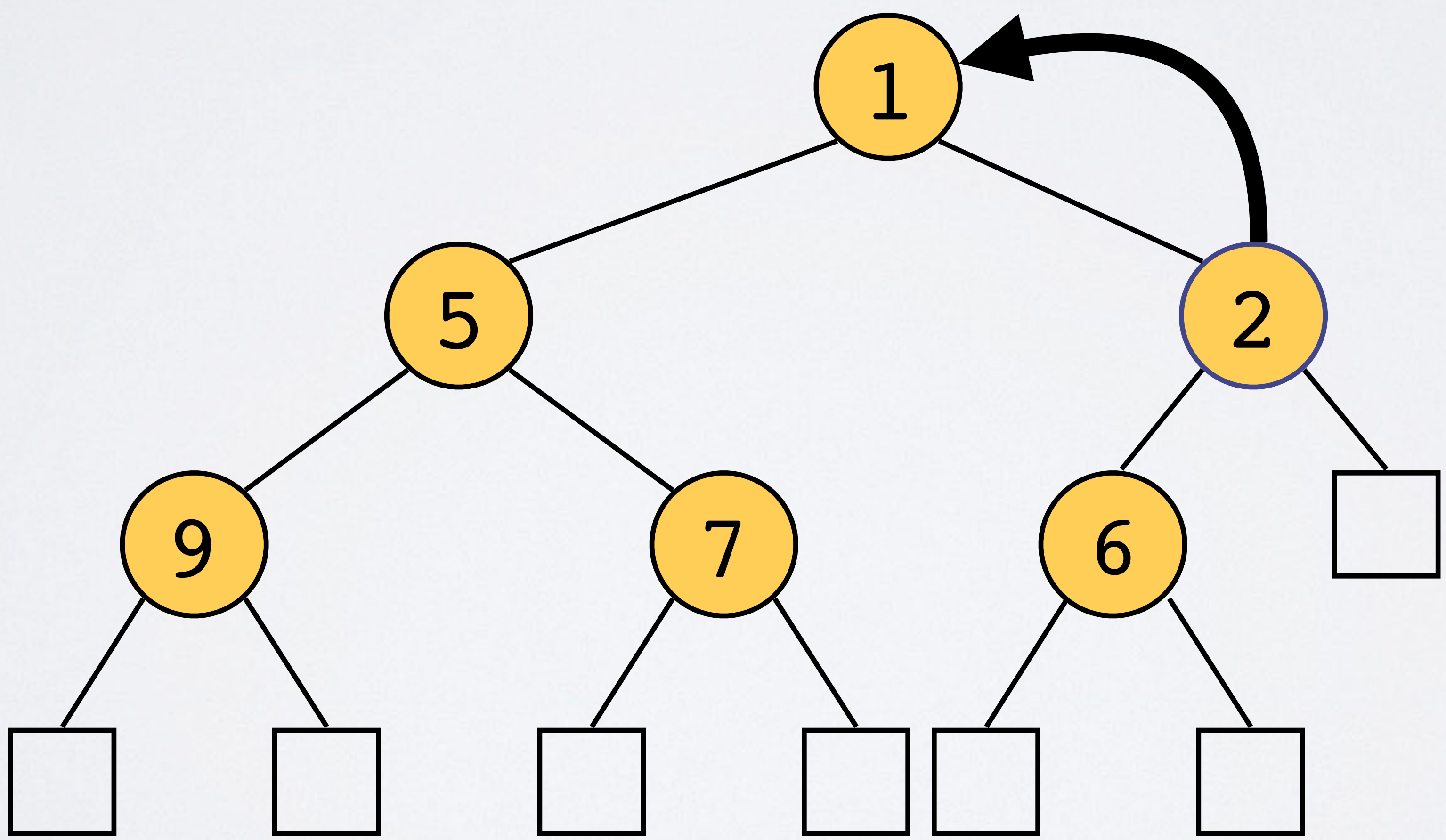
Heap — sift_up

- Repair heap: swap new element up tree until keys are sorted
- First swap fixes everything below new location
  - since every node below **6**'s old location has to be at least **6**…
  - …they must be at least **1**



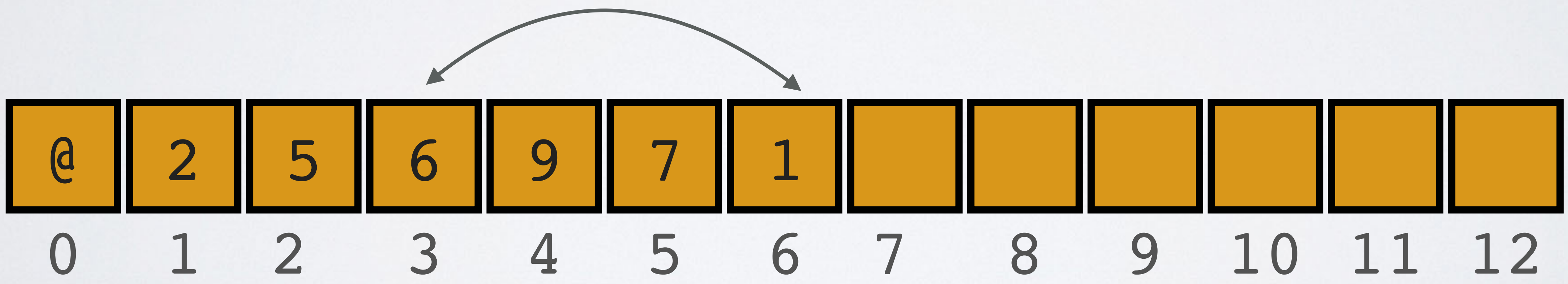
Heap — sift_up

- One more swap since `1≤2`
- Now left-completeness and order are satisfied



Heap — insert( )

- Need to keep track of "insertion node"
  - leaf where we will insert new node…
  - …so we can keep heap left-complete



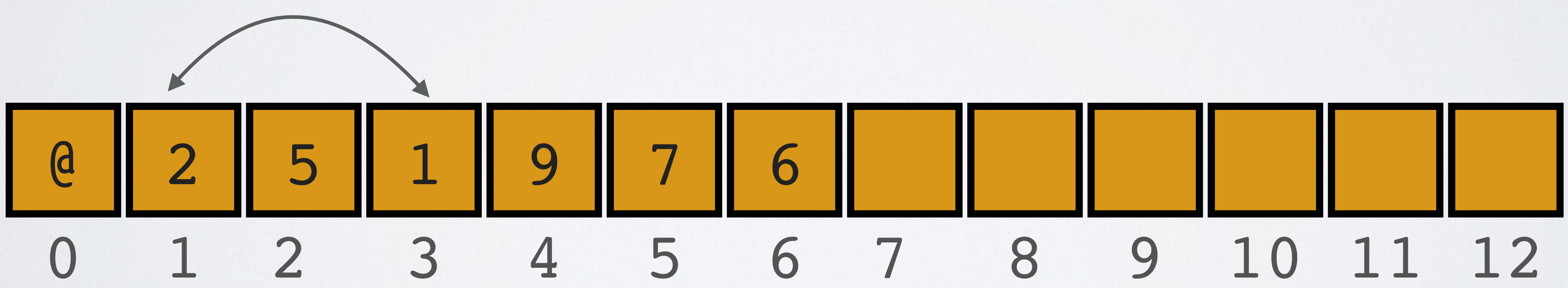
Heap — insert( )

- Need to keep track of "insertion node"
  - leaf where we will insert new node…
  - …so we can keep heap left-complete

@	2	5	1	9	7	6						
0	1	2	3	4	5	6	7	8	9	10	11	12

Heap — insert( )

- Need to keep track of "insertion node"
  - leaf where we will insert new node…
  - …so we can keep heap left-complete



Heap — insert( )

- Need to keep track of "insertion node"
  - leaf where we will insert new node…
  - …so we can keep heap left-complete

@	1	5	2	9	7	6						
0	1	2	3	4	5	6	7	8	9	10	11	12

should_move

We will abstract away our decision on whether we need to swap two entries

```
function should_move(x, y, min_heap):
  if x > y and min_heap == True:
    return True
  if x < y and min_heap == False:
    return True
  return False
```

sift_up

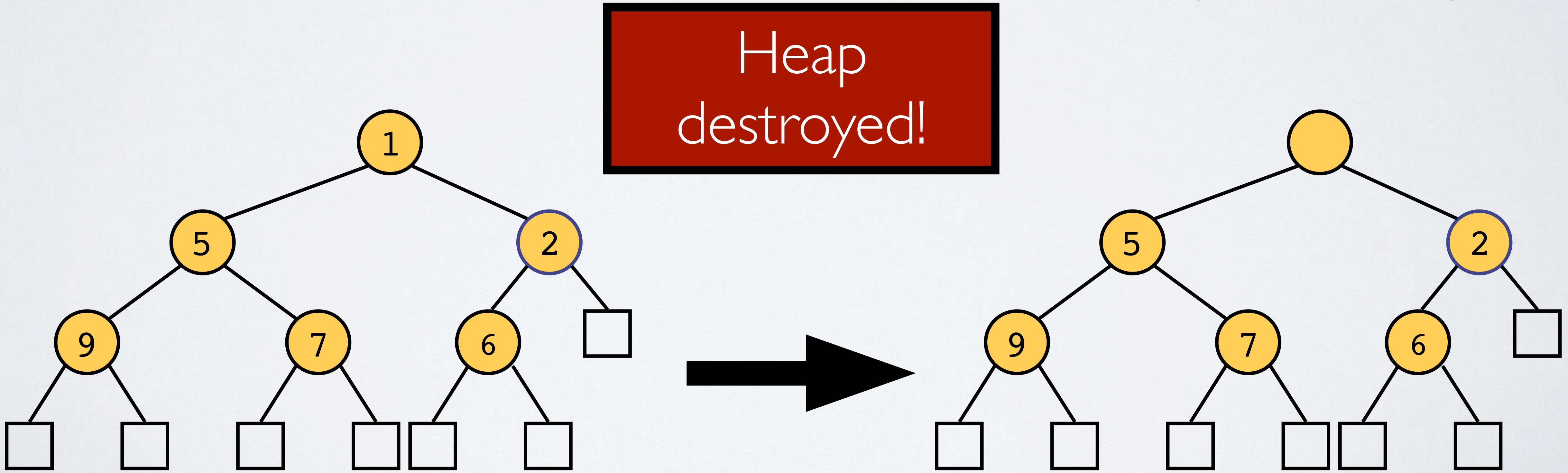
```
function sift_up(arr, index, min_heap):
  while should_move(arr[index], arr[parent(index)], min_heap)
   and index > 1:
    swap(arr, index, parent(index))
    index = parent(index)
```

insert

```
function insert(arr, entry, min_heap):
  arr.append(entry)
  index = length(arr) - 1
  sift_up(arr, index, min_heap)
```

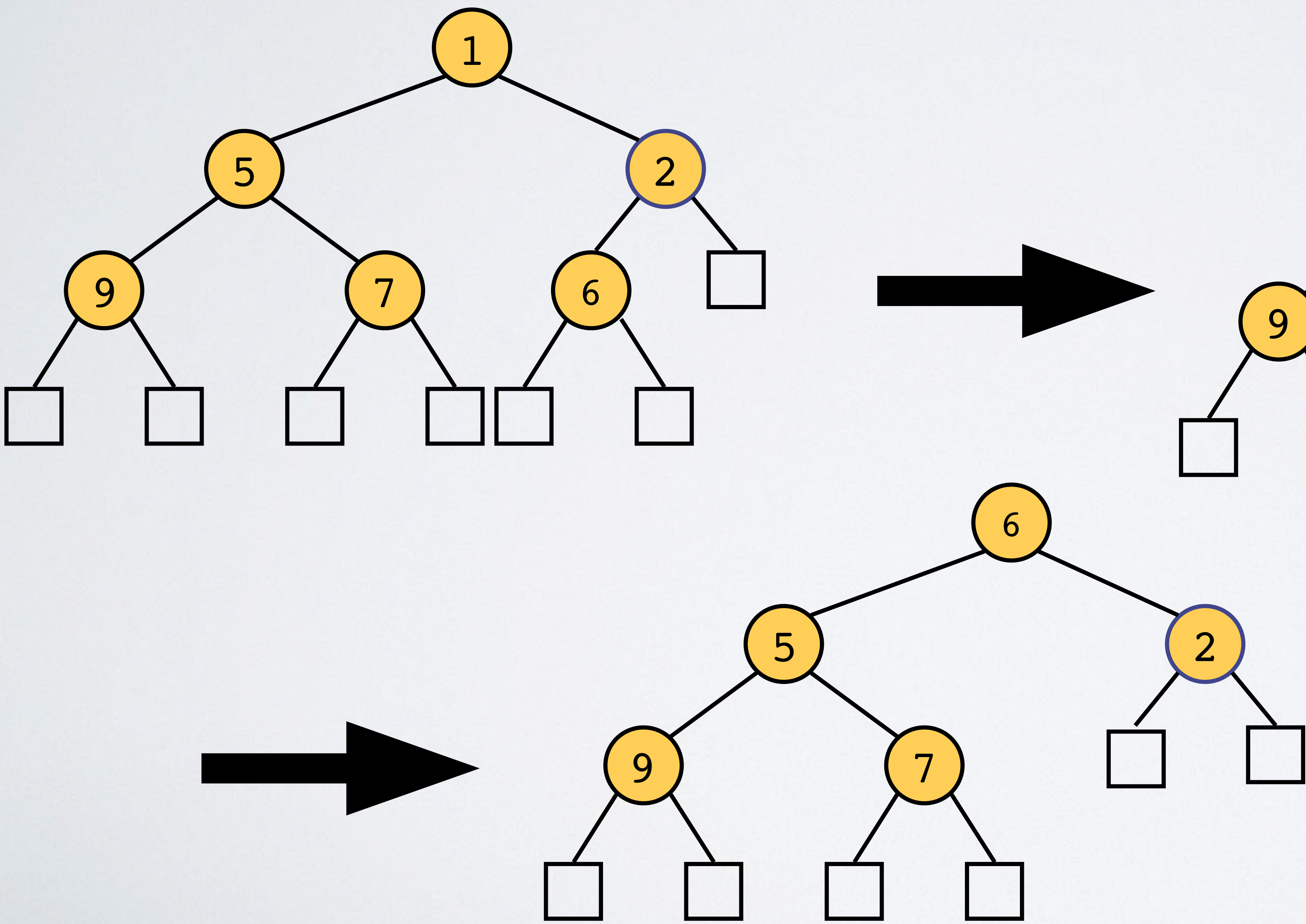
Heap — dequeue( )

- Remove root
  - because it is always the smallest (min-heap) or largest (max-heap) element
- How can we remove root w/o destroying heap?



Heap — dequeue( )

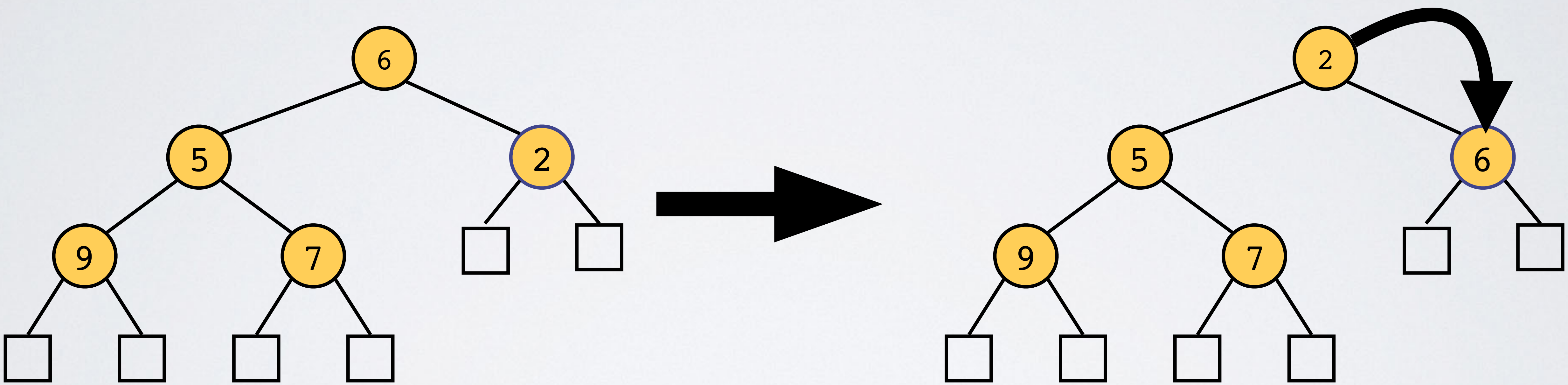
- Instead swap root with last element & remove it
  - removing last element is easy



Order destroyed!

Heap — dequeue( )

- Now swap root down as necessary



Heap is in order!

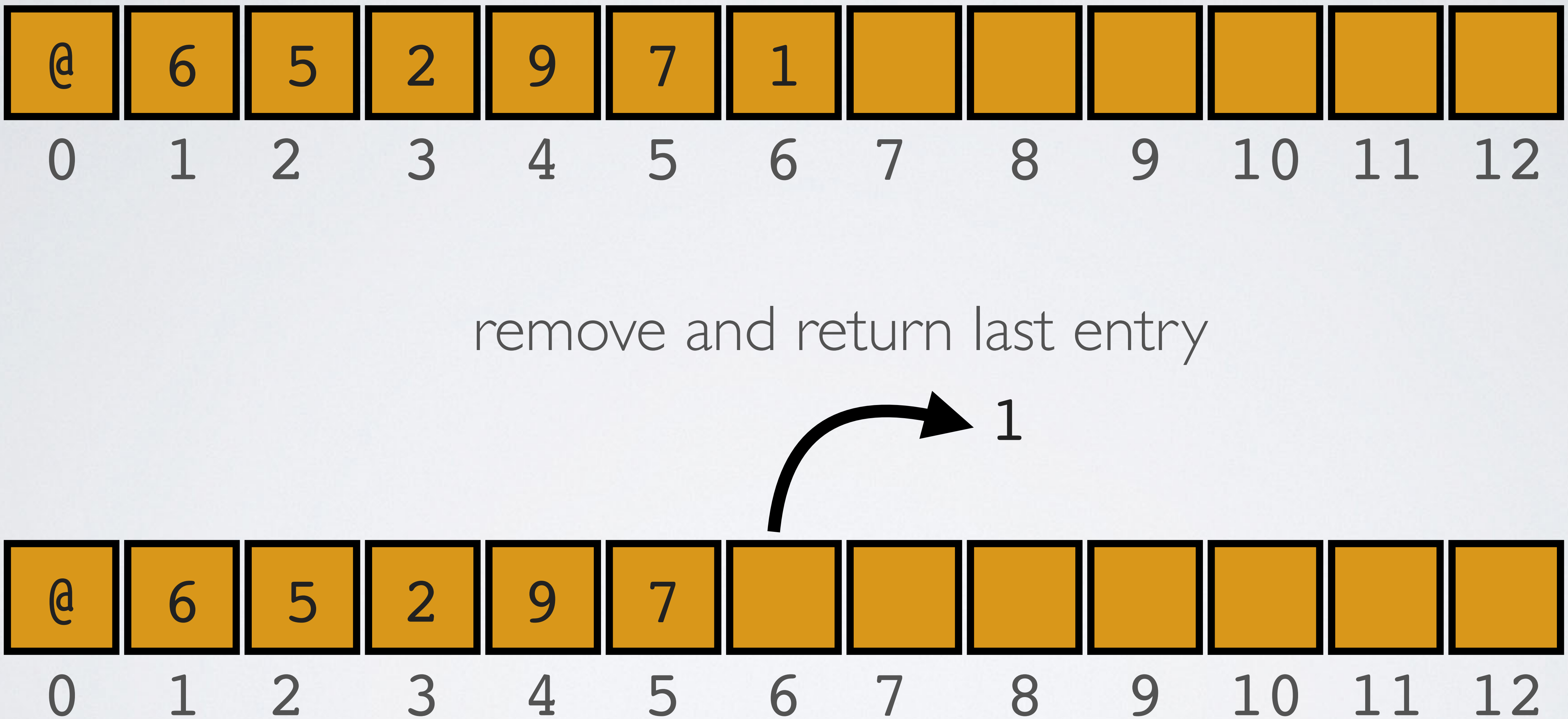
Heap — dequeue( )

0	1	2	3	4	5	6	7	8	9	10	11	12
@	1	5	2	9	7	6						

Swap first and last items

0	1	2	3	4	5	6	7	8	9	10	11	12
@	6	5	2	9	7	1						

Heap — dequeue( )



Heap — dequeue( )

@	6	5	2	9	7							
0	1	2	3	4	5	6	7	8	9	10	11	12

Swap down until order is retained

@	2	5	6	9	7							
0	1	2	3	4	5	6	7	8	9	10	11	12

sift_down

```
function sift_down(arr, index, min_heap):
   n = length(arr) - 1
   child1 = left_child(index)
   while child1 <= n:
      curr_child = child1
      child2 = right_child(index)
       if (child2 <= n) and should_move(arr[child2], arr[child1], min_heap):
         curr_child = child2
      if(not should_move(arr[index], arr[curr_child]):
         break
      swap(arr, index, curr_child)
      index = curr_child
      child1 = left_child(index)
```

deuqueue

```
function dequeue(arr, min_heap):
  index = length(arr) - 1
  swap(arr, 1, index)
  ans = arr[index]
  arr[index] = NIL
  sift_down(arr, 1, min_heap)
```

HeapSort

- Can use heaps to help sort data
- Core idea:
  - Insert data into heap
  - Remove data from heap until heap is empty

HeapSort

```
function heapsort(arr, ascending):
  n = length(arr)
  if n == 0:
    return
  heap = [None, arr[0]]
  for i in 1 to n-1:
    insert(heap, arr[i], ascending)
  for i in 0 to n-1:
    arr[i] = dequeue(heap, ascending)
```

Updating Priorities

- Some applications require us to update priorities
- Using a heap alone, we forced to search for an item linearly since heaps are organised by a key that represents priority
- Solution:
  - Use heap and dictionary (hashtable) together!
  - Assume that every entry has id field
  - Store index in heap of an item in hashtable such that we can retrieve by id!
  - Have sift_up and sift_down return index in array

References

- Slides adapted from Brown's CS16