PRIORITY QUEUES

PRIORITIZATION PROBLEMS

- Printer Jobs : Print jobs print in a queue, what if we want the faculty jobs to print before the student jobs, irrespective of which order they are sent to the printer.

- ER Scheduling : Treating a gunshot patient before a guy with a sore neck irrespective of the time of their arrival.

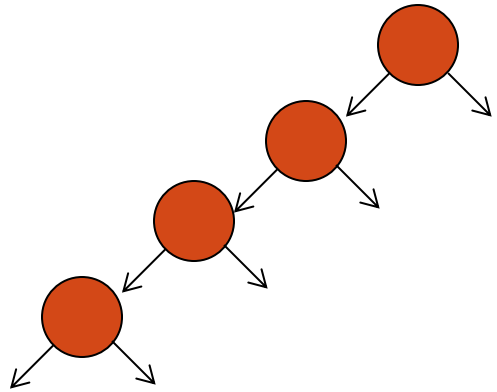
- *Why can't we solve these problems efficiently with the data structures we have (list, sorted list, queue, stack etc.)?*

PROBLEMS WITH EXISTING DATA STRUCTURES

- *list* : store jobs in a list; remove min/max by searching (O(*N*))
  - problem: expensive to search

- *sorted list* : store in sorted list; binary search it in O(log *N*) time
  - problem: expensive to add/remove (O(*N*))

- *binary search tree* : store in BST, go right for max in O(log *N*)
  - problem: tree becomes unbalanced



PRIORITY QUEUE

- **Priority Queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
  - usually implemented using a tree structure called a *heap*
- Each element of the queue has some priority, and all elements are processed based on their priorities
- An element with the high priority is processed before element with low priority.
- FIFO rule applies if two elements have same priority
- Implement it as a sorted linked list, sorted on priority
- Element with highest priority is always inserted first

PRIORITY QUEUE ADT

A priority queue P supports the following methods:

-size(): Return the number of elements in P

-isEmpty(): Test whether P is empty

-insertItem(k,e): Insert a new element e with key k into P

-minElement(): Return (but don't remove) an element of P with smallest key; an error occurs if P is empty.

-minKey(): Return the smallest key in P; an error occurs if P is empty

-removeMin(): Remove from P and return an element with the smallest key; an error conditions occurs if P is empty.

LIST BASED PRIORITY QUEUE

- Unsorted list implementation
  - Store the items of the priority queue in a list-based sequence, in arbitrary order

- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin, minKey and minElement take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Sorted list implementation
  - Store the items of the priority queue in a sequence, sorted by key

- Performance:
  - insertItem takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin, minKey and minElement take $O(1)$ time since the smallest key is at the beginning of the sequence

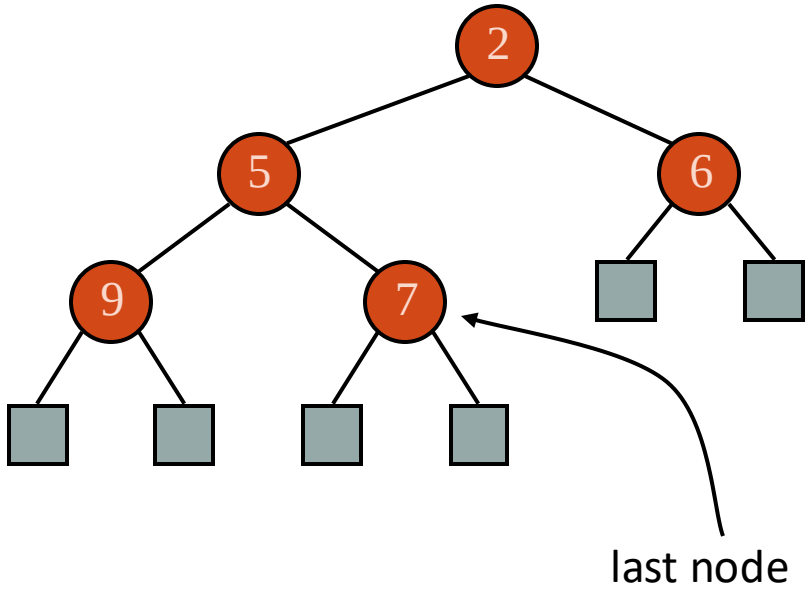
7

IMPLEMENTATION OF A PRIORITY QUEUE

HEAPS

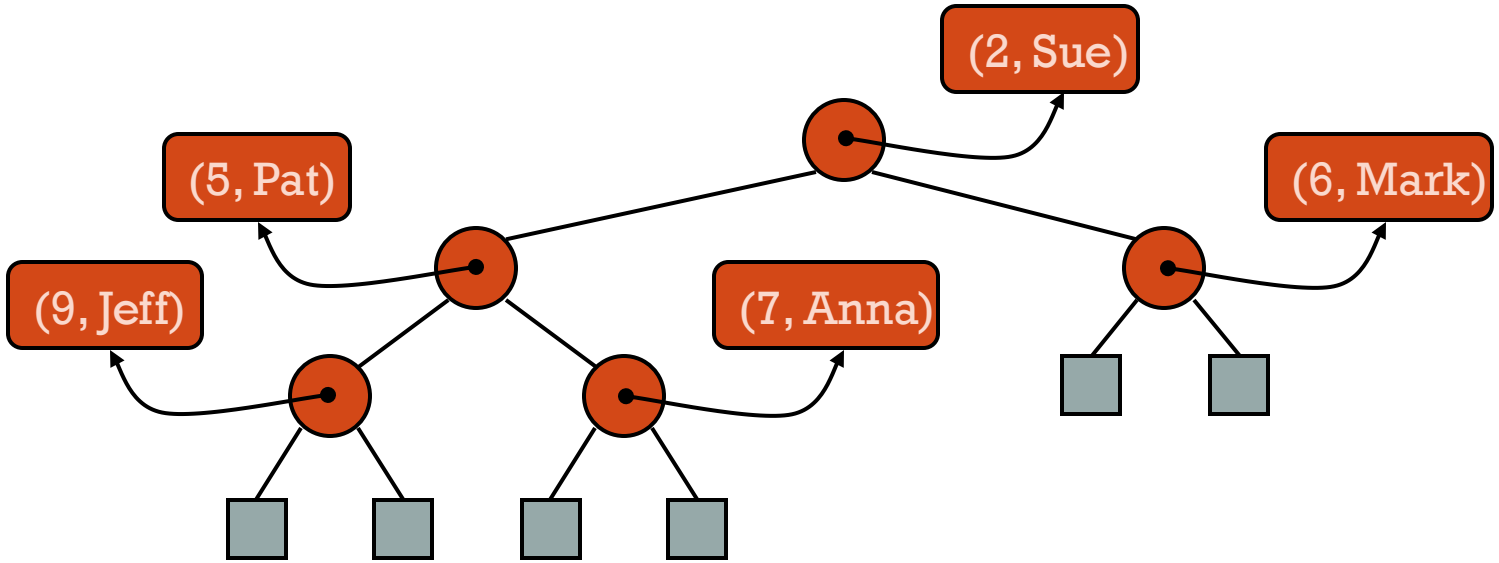
- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
    - at depth $h - 1$, the internal nodes are to the left of the leaf nodes

- The last node of a heap is the rightmost internal node of depth $h - 1$



HEAPS AND PRIORITY QUEUES

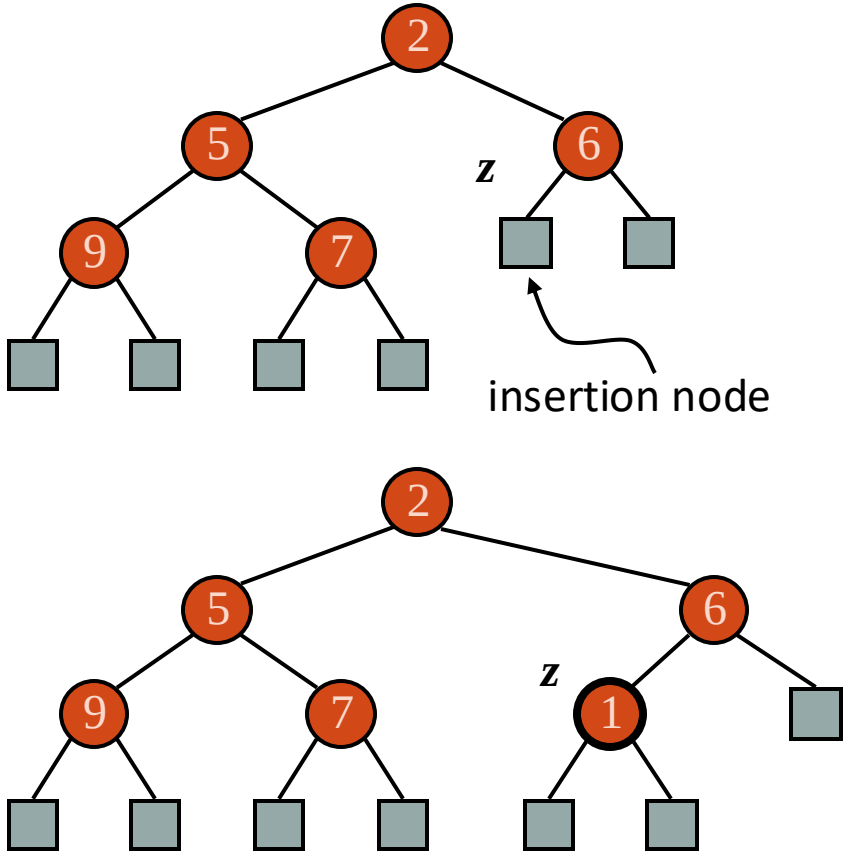
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures



INSERTION INTO A HEAP

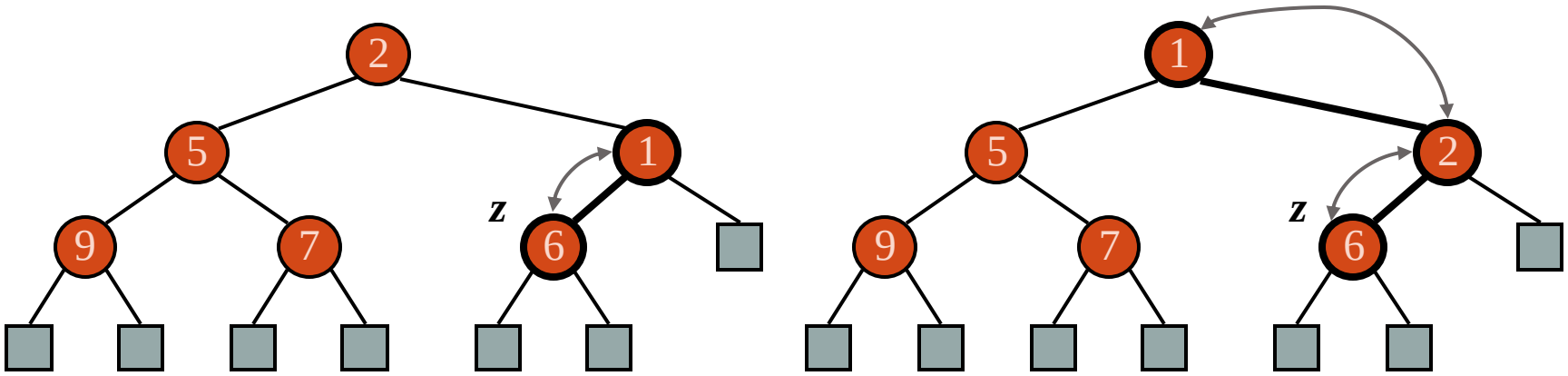
- Method insertItem of the priority queue ADT corresponds to the insertion of a key *k* to the heap

- The insertion algorithm consists of three steps
  - Find the insertion node *z* (the new last node)
  - Store *k* at *z* and expand *z* into an internal node
  - Restore the heap-order property (discussed next)



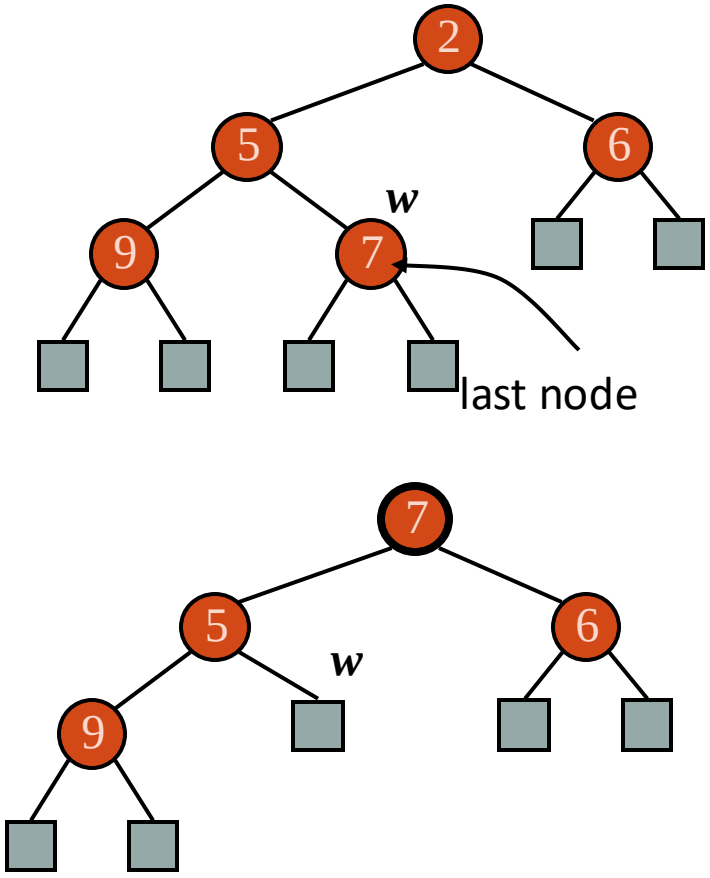
UPHEAP

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



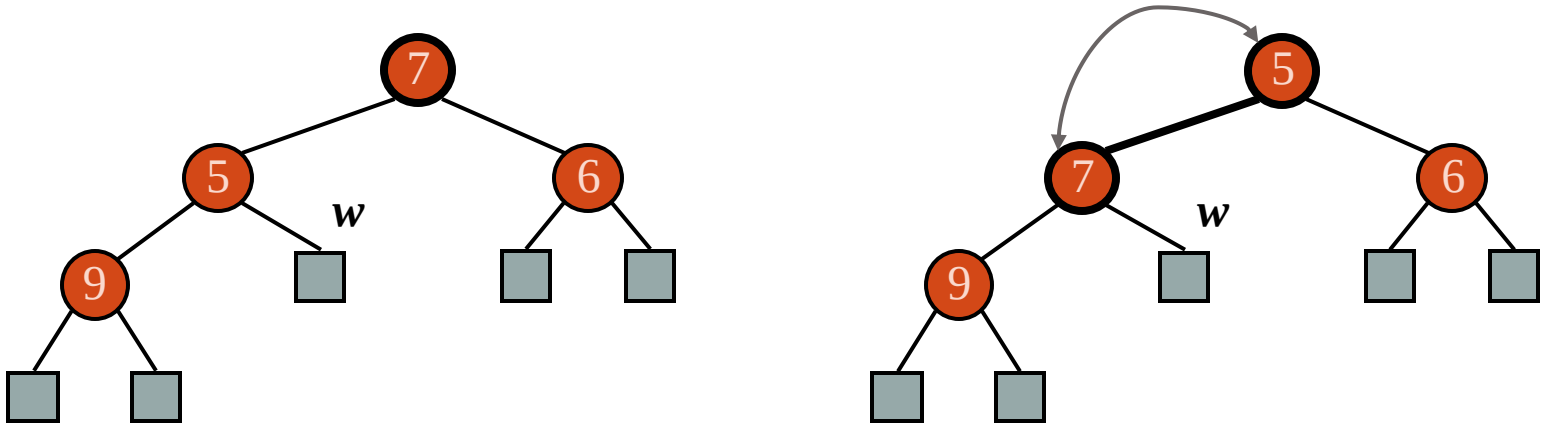
REMOVAL FROM A HEAP

- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node *w*
  - Compress *w* and its children into a leaf
  - Restore the heap-order property (discussed next)



DOWNHEAP

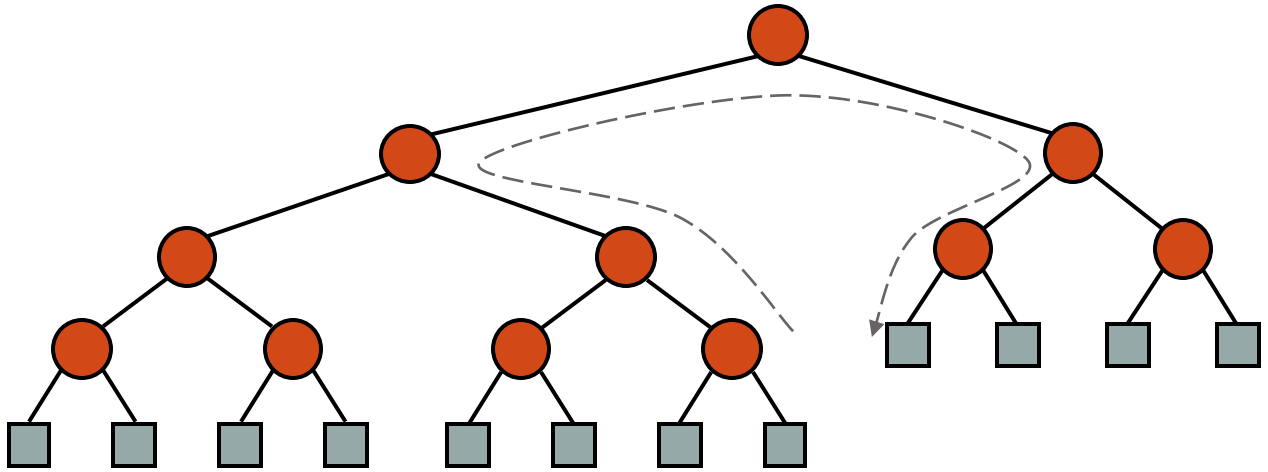
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap property by swapping key k with the child with the smallest key along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



- Start with the parent
- Compare it with both its left child and right child
  - If both children are greater than k – *Nothing to be done*
  - If one child is smaller than k – *the Smaller child value is moved up*
  - If both children are smaller than k – *the Smaller of the two children is moved* up.
- Stop when both children are greater than k or we reach a leaf node.

UPDATING THE LAST NODE

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



HEAP SORT – A PEEK

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insertItem and removeMin take $O(\log n)$ time
  - methods size, isEmpty, minKey, and minElement take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

REPRESENTING HEAPS - SEQUENTIALLY

- We can represent a heap with *n* keys by means of an array of length *n*
- For the node at rank *i*
  - the left child is at rank 2*i* ***(if 2i >n no left child)***
  - the right child is at rank 2*i* +1 ***(if 2i+1 > n no right child)***
  - parent is i/2
- Links between nodes are not explicitly stored
- The leaves are not represented