

# Priority queues and binary heaps

Data Structures and Algorithms for Computational Linguistics III  
(ISCL-BA-07)

Çağrı Çöltekin

`ccoltekin@sfs.uni-tuebingen.de`

University of Tübingen  
Seminar für Sprachwissenschaft

Winter Semester 2021/22

# Priority queue ADT

- A *priority queue* is a collection, an abstract data type, that stores items
- The items in a priority queue are *key-value* pairs
- The key determines the priority of the item, while the value is the actual data of interest
- The interface of a priority queue is similar to a standard queue
- Instead of the first item entered into the queue, the item with the highest priority (minimum or maximum key value) is removed from the priority queue
- Priority queues have many applications ranging from data compression to discrete optimization
- We will see their application to sorting (this lecture) and searching on graphs (later)

# Priority queues

## Key operations

- `insert(k, v)` Similar to `enqueue(v)`, inserts the value `v` with priority `k` into the queue
- `remove()` Similar to `dequeue()`, removes and returns the item with highest priority
- This operation is often called `remove_min()` or `remove_max()` depending on minimum or maximum key value is considered having the highest priority

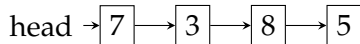
# Priority queues

## Example operations

Operation	Return value	Priority queue
insert(5, a)		{(5,a)}
insert(9, c)		{(5,a), (9,c)}
insert(3, b)		{(5,a), (9,c), (3,b)}
insert(7, d)		{(5,a), (9,c), (3,b), (7,d)}
remove()	c	{(5,a), (3,b), (7,d)}
remove()	d	{(5,a), (3,b)}
remove()	a	{(3,b)}
remove()	b	{}

# Priority queue implementation

unsorted list



# Priority queue implementation

unsorted list

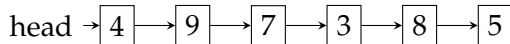
`insert(9,v)`



# Priority queue implementation

unsorted list

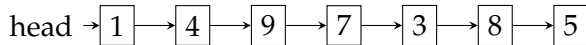
`insert(4,v)`



# Priority queue implementation

unsorted list

`insert(1,v)`

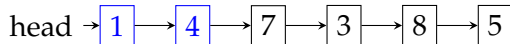




# Priority queue implementation

unsorted list

$9 \leftarrow \text{remove\_max}()$



# Priority queue implementation

unsorted list

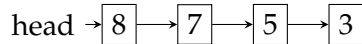
$8 \leftarrow \text{remove\_max}()$



- Insert:  $O(1)$
- Remove:  $O(n)$

# Priority queue implementation

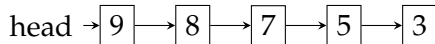
sorted list



# Priority queue implementation

sorted list

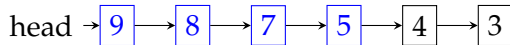
insert(9,v)



# Priority queue implementation

sorted list

`insert(4,v)`



# Priority queue implementation

sorted list

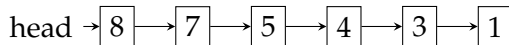
`insert(1,v)`



# Priority queue implementation

sorted list

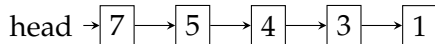
$9 \leftarrow \text{remove\_max}()$



# Priority queue implementation

sorted list

$8 \leftarrow \text{remove\_max}()$



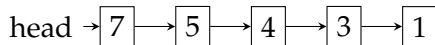
- Insert:  $O(n)$
- Remove:  $O(1)$



# Priority queue implementation

sorted list

$8 \leftarrow \text{remove\_max}()$

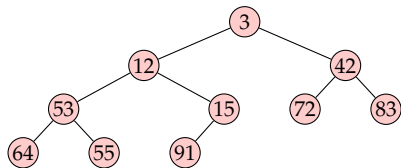
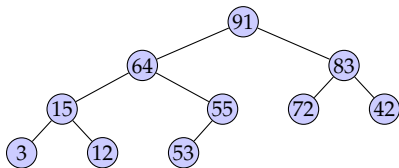


- Insert:  $O(n)$
- Remove:  $O(1)$

We can do better on average (coming soon).

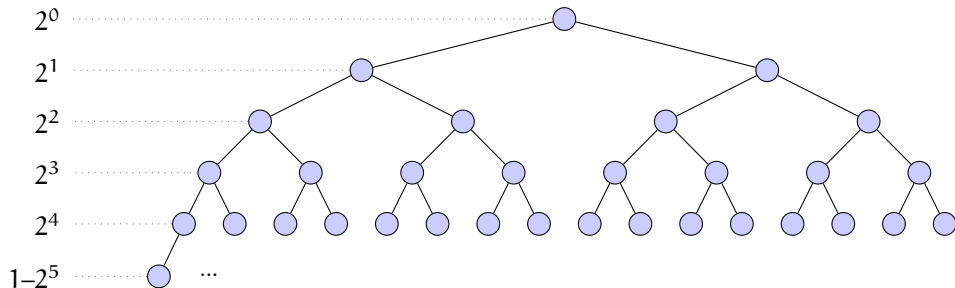
# Binary heaps

- A binary heap is a binary tree where the nodes store items with an ordering relation. A binary heap has two properties:
  1. *Shape*: a binary heap is a *complete* binary tree
    - all levels of the tree, except possibly the last one, are full
    - all empty slots (if any) are to the right of the filled nodes at the lowest level
  2. *Heap order*:
    - **max-heap** Parents' keys are larger than children's keys
    - **min-heap** Parents' keys are smaller than children's keys



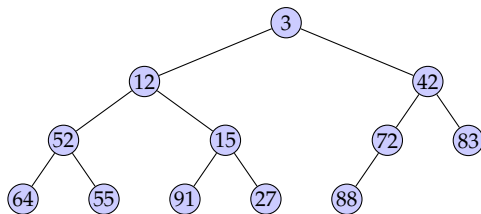
# Height of a binary heap

- Height of a binary heap is  $\lfloor \log n \rfloor$



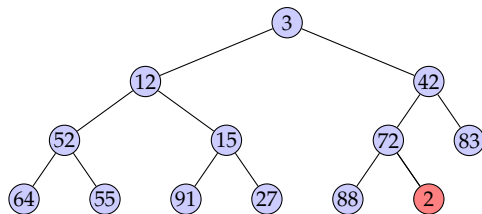
- At least  $2^h$  nodes  $\Rightarrow h \leq \log n$
- At most  $2^{h+1} - 1$  nodes  $\Rightarrow h \geq \log(n + 1) - 1$

# Adding an new item to a binary heap



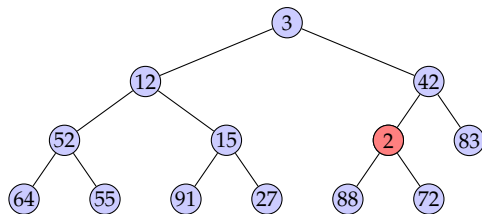
- Add the new element to the first available slot
- "Bubble up" until the heap property is satisfied
- At most  $h = \log n$  comparisons/swaps

# Adding an new item to a binary heap



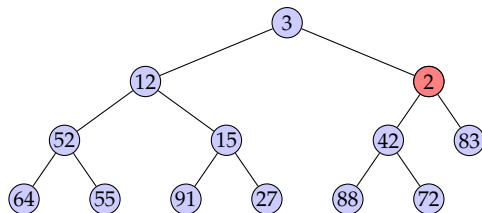
- Add the new element to the first available slot
- "Bubble up" until the heap property is satisfied
- At most  $h = \log n$  comparisons/swaps

# Adding an new item to a binary heap



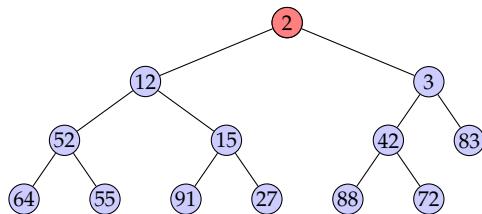
- Add the new element to the first available slot
- "Bubble up" until the heap property is satisfied
- At most  $h = \log n$  comparisons/swaps

# Adding an new item to a binary heap



- Add the new element to the first available slot
- "Bubble up" until the heap property is satisfied
- At most  $h = \log n$  comparisons/swaps

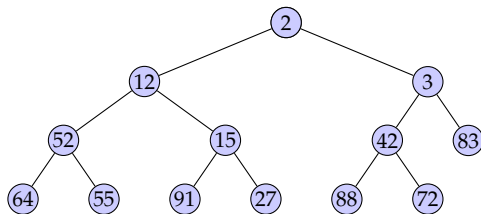
# Adding an new item to a binary heap



- Add the new element to the first available slot
- "Bubble up" until the heap property is satisfied
- At most  $h = \log n$  comparisons/swaps

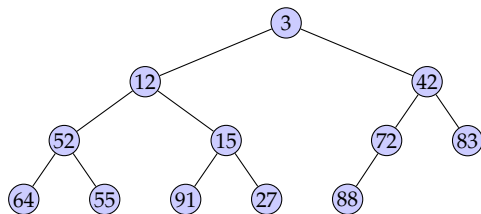


# Adding an new item to a binary heap



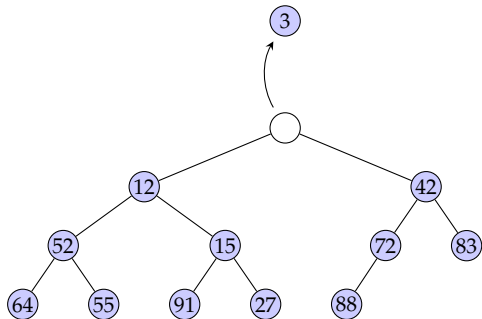
- Add the new element to the first available slot
- "Bubble up" until the heap property is satisfied
- At most  $h = \log n$  comparisons/swaps

# Removing the min/max from a binary heap



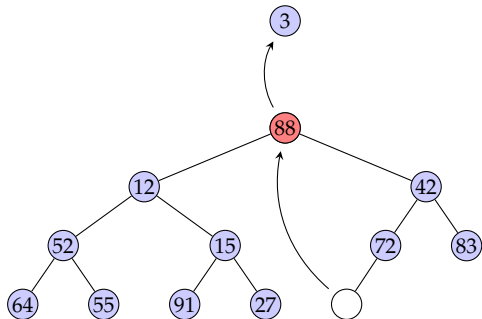
- The item to be removed is at the root
- We replace root with the element at the last slot
- "Bubble down" until the heap property is satisfied

## Removing the min/max from a binary heap



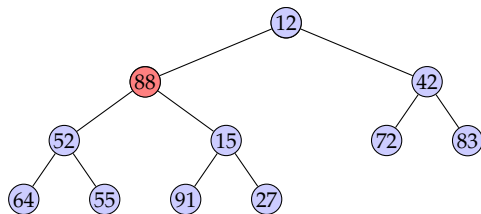
- The item to be removed is at the root
- We replace root with the element at the last slot
- "Bubble down" until the heap property is satisfied

## Removing the min/max from a binary heap



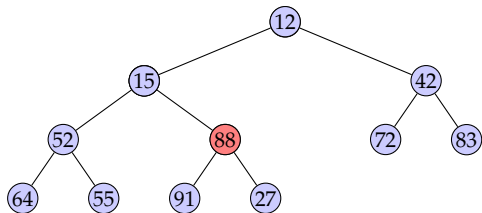
- The item to be removed is at the root
- We replace root with the element at the last slot
- "Bubble down" until the heap property is satisfied

# Removing the min/max from a binary heap



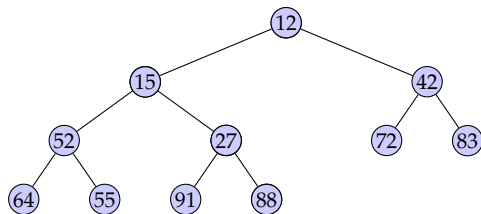
- The item to be removed is at the root
- We replace root with the element at the last slot
- "Bubble down" until the heap property is satisfied

## Removing the min/max from a binary heap



- The item to be removed is at the root
- We replace root with the element at the last slot
- "Bubble down" until the heap property is satisfied

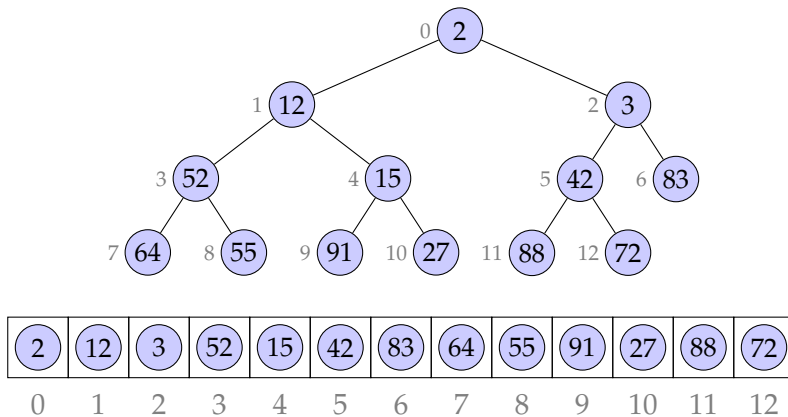
# Removing the min/max from a binary heap



- The item to be removed is at the root
- We replace root with the element at the last slot
- "Bubble down" until the heap property is satisfied

# Array based implementation of heaps

- As any complete binary tree, heaps can be stored efficiently using an array data structure



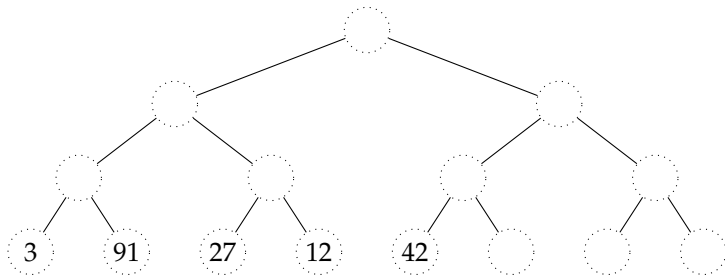


## Bottom-up heap construction

- For  $n$  items, we can construct a heap by inserting each key to the heap in  $O(n \log n)$  time
- If we have the complete list, there is a bottom-up procedure that runs in  $O(n)$  time
  1. First fill the leaf nodes, single-node trees satisfy the heap property
    - $h = \lfloor \log n \rfloor$ ,
    - we have  $2^h - 1$  internal nodes
    - $n - 2^h + 1$  leaf nodes
  2. Fill the next level, “bubble down” if necessary
  3. Repeat 2 until all elements are inserted, and heap property is satisfied

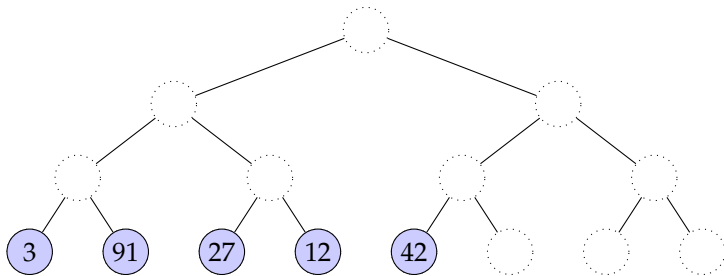
# Bottom-up heap construction

demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)



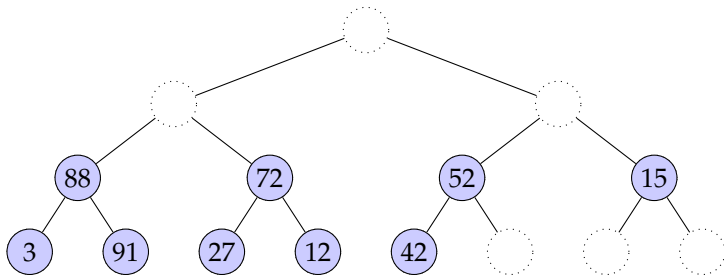
# Bottom-up heap construction

demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)



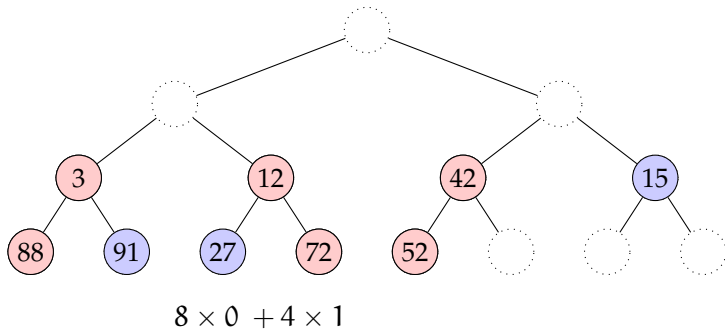
# Bottom-up heap construction

demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)



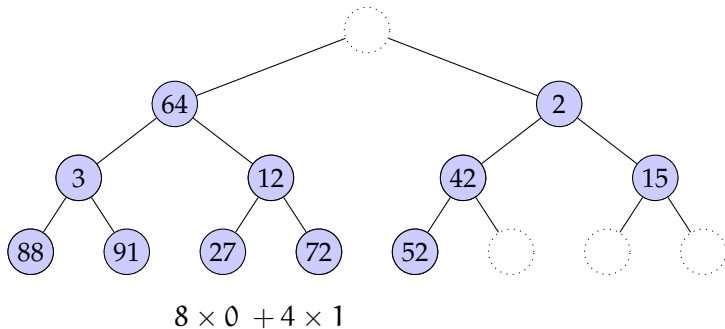
# Bottom-up heap construction

demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)



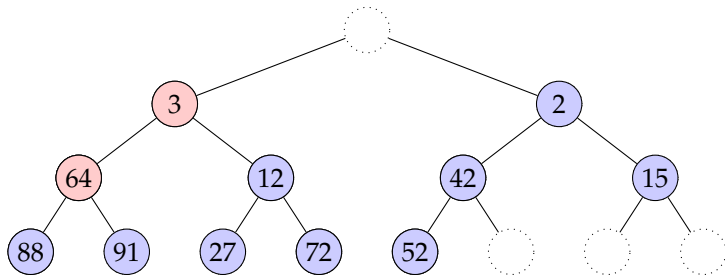
# Bottom-up heap construction

demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)



# Bottom-up heap construction

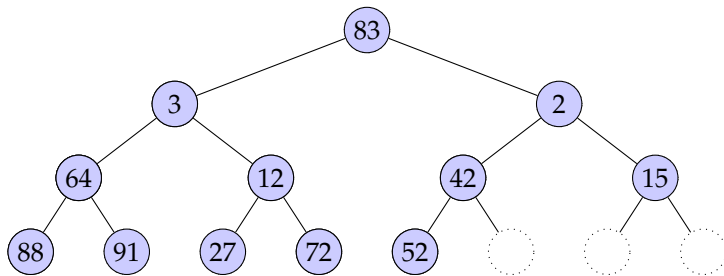
demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)



$$8 \times 0 + 4 \times 1 + 2 \times 2$$

# Bottom-up heap construction

demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)

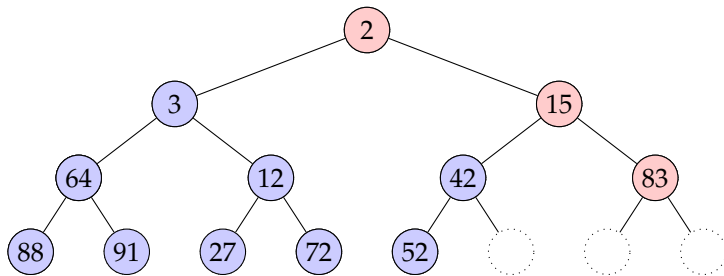


$$8 \times 0 + 4 \times 1 + 2 \times 2$$



# Bottom-up heap construction

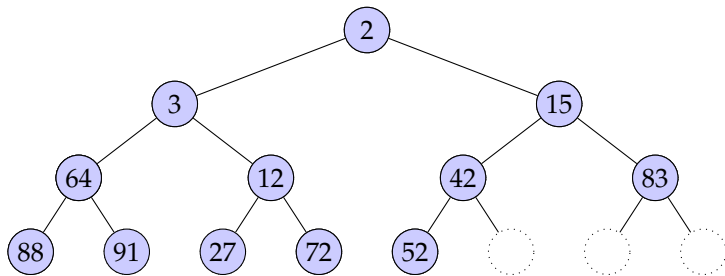
demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)



$$8 \times 0 + 4 \times 1 + 2 \times 2 + 1 \times 3$$

# Bottom-up heap construction

demonstration with: 3, 91, 27, 12, 42, 88, 72, 52, 15, 64, 2, 83 (12 items)



$$\begin{aligned}
 & 8 \times 0 + 4 \times 1 + 2 \times 2 + 1 \times 3 \\
 T(n) = & \sum_{i=0}^h i \times 2^{h-i} = \sum_{i=0}^h i \times \frac{2^h}{2^i} = n \underbrace{\sum_{i=0}^h \frac{i}{2^i}}_{\text{constant}} = O(n)
 \end{aligned}$$

# Implementing priority queues with binary heaps

- Binary heaps provide a straightforward implementation of priority queues

Implementation	insert()	remove()
Unsorted list		

# Implementing priority queues with binary heaps

- Binary heaps provide a straightforward implementation of priority queues

Implementation	insert()	remove()
Unsorted list	$O(1)$	$O(n)$
Sorted list		

# Implementing priority queues with binary heaps

- Binary heaps provide a straightforward implementation of priority queues

Implementation	insert()	remove()
Unsorted list	$O(1)$	$O(n)$
Sorted list	$O(n)$	$O(1)$
Binary heap		

# Implementing priority queues with binary heaps

- Binary heaps provide a straightforward implementation of priority queues

Implementation	insert()	remove()
Unsorted list	$O(1)$	$O(n)$
Sorted list	$O(n)$	$O(1)$
Binary heap	$O(\log n)$	$O(\log n)$

- Some improvements are possible, such as
  - d-ary heaps:  $O(\log_d n)$  insert,  $O(d \log_d n)$  remove
  - Fibonacci heaps:  $O(1)$  insert,  $O(\log n)$  remove

# Python standard heap implementation

- Python standard `heapq` module allows maintaining a list (array) based heap
  - The `heappush(h, e)` insert `e` into heap `h`
  - The `heappop(h)` return the minimum value from heap `h`
  - The `heapify(h)` construct a heap from given list `heappos(h)`

```
>>> h = []
>>> heappush(h, (3, 'this is important'))
>>> heappush(h, (9, 'this, not so much'))
>>> heappush(h, (5, 'this is quite important too'))
>>> heappush(h, (1, 'highest priority'))
>>> heappush(h, (4, 'fairly important'))
>>> h
[(1, 'highest priority'), (3, 'this is important'), (5, 'this is quite important too'),
 ↪ (9, 'this, not so much'), (4, 'fairly important')]
>>> [heappop(h) for _ in range(len(h))]
[(1, 'highest priority'), (3, 'this is important'), (4, 'fairly important'), (5, 'this is
 ↪ quite important too'), (9, 'this, not so much')]
```

# Sorting with priority queues

- Inserting the items in a priority queue and removing them effectively sorts the given array
- There is an interesting connection with this approach and some sorting algorithms
  - If we use a sorted list, the algorithm is equivalent to the insertion sort  $O(n^2)$
  - If we use a unsorted list, the algorithm is equivalent to the selection sort  $O(n^2)$
  - If use a binary heap, we get an  $O(n \log n)$  algorithm (heap sort)



# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`insert(7)`



# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`insert(2)`

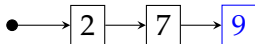


# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`insert(9)`



# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`insert(4)`

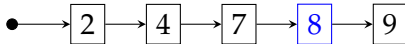


# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

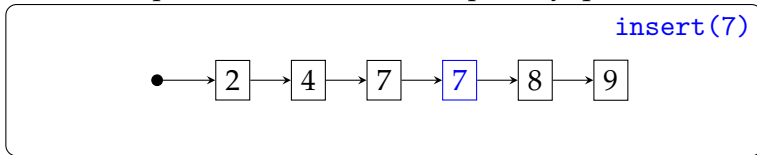
insert(8)



# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue



# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

insert(7)



Step 2: simply remove each item from the priority queue



# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue





# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue



# Insertion sort with priority queues

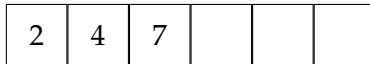
priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue

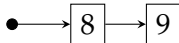


# Insertion sort with priority queues

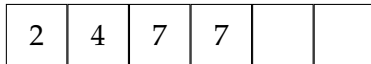
priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue



# Insertion sort with priority queues

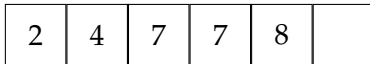
priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue



# Insertion sort with priority queues

priority queues implemented with sorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue

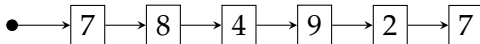
2	4	7	7	8	9
---	---	---	---	---	---

# Selection sort with priority queues

priority queues implemented with unsorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`insert(7)...`

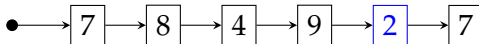


# Selection sort with priority queues

priority queues implemented with unsorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue



# Selection sort with priority queues

priority queues implemented with unsorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue



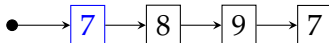


# Selection sort with priority queues

priority queues implemented with unsorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue

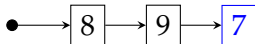


# Selection sort with priority queues

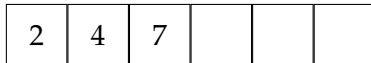
priority queues implemented with unsorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue

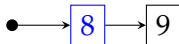


# Selection sort with priority queues

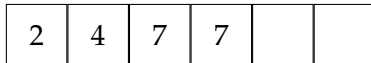
priority queues implemented with unsorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue



# Selection sort with priority queues

priority queues implemented with unsorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue

2	4	7	7	8	
---	---	---	---	---	--

# Selection sort with priority queues

priority queues implemented with unsorted lists – sorting: 7, 2, 9, 4, 8, 7

Step 1: insert the items to a priority queue

`remove_min()`



Step 2: simply remove each item from the priority queue

2	4	7	7	8	9
---	---	---	---	---	---

# Sorting with heaps

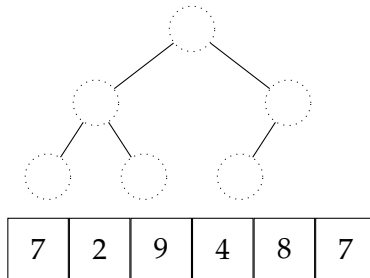
a first attempt

- The idea is simple: as before, insert all items to the heap
- Remove them in order
- Complexity of  $O(n \log n)$
- However,
  - not stable
  - not in-place: needs  $O(n)$  extra space (we can fix this)

```
def heap_sort(seq):  
    heap = []  
    for item in seq:  
        heappush(item)  
    for i in range(len(seq)):  
        seq[i] = heappop(heap)
```

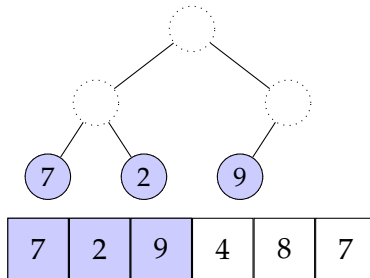
# In-place heap sort

step 1: bottom-up heap construction– sorting: 7, 2, 9, 4, 8, 7



# In-place heap sort

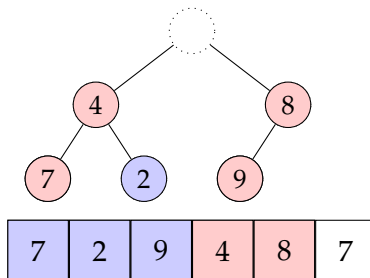
step 1: bottom-up heap construction– sorting: 7, 2, 9, 4, 8, 7





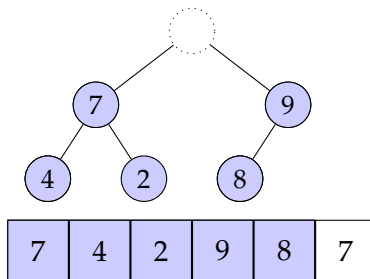
# In-place heap sort

step 1: bottom-up heap construction– sorting: 7, 2, 9, 4, 8, 7



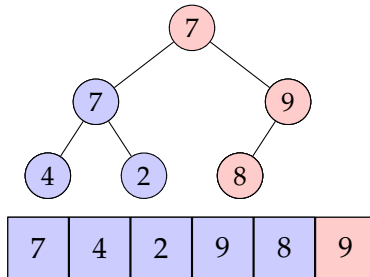
# In-place heap sort

step 1: bottom-up heap construction– sorting: 7, 2, 9, 4, 8, 7



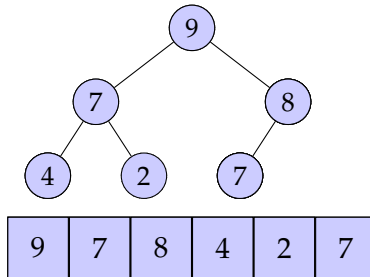
# In-place heap sort

step 1: bottom-up heap construction– sorting: 7, 2, 9, 4, 8, 7



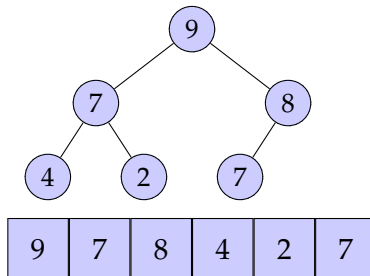
# In-place heap sort

step 1: bottom-up heap construction– sorting: 7, 2, 9, 4, 8, 7



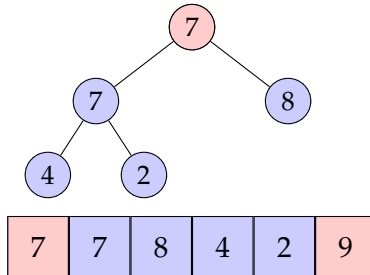
# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end



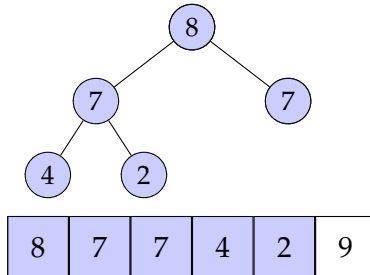
# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end



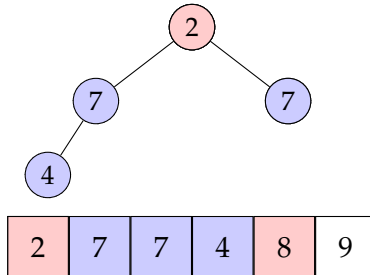
# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end



# In-place heap sort

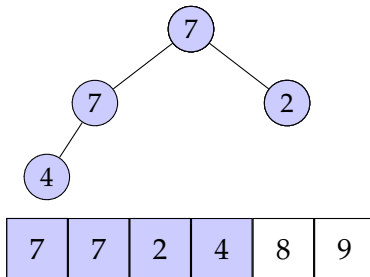
step 2: iteratively remove the maximum element, place it at the end





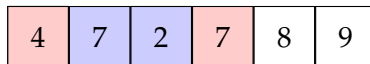
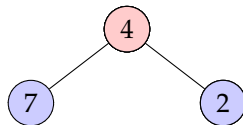
# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end



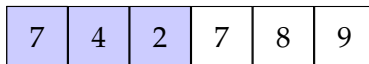
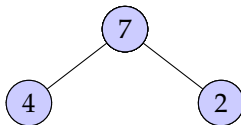
# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end



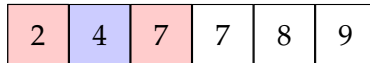
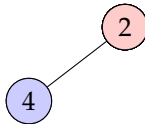
# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end



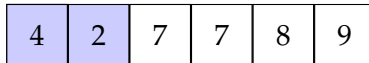
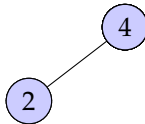
# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end



# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end



# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end

2

2	4	7	7	8	9
---	---	---	---	---	---

# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end

2

2	4	7	7	8	9
---	---	---	---	---	---

Heap construction:  $O(n) + n \times \text{remove\_min}(): O(n \log n) = O(n \log n)$

# In-place heap sort

step 2: iteratively remove the maximum element, place it at the end

2	4	7	7	8	9
---	---	---	---	---	---

Heap construction:  $O(n) + n \times \text{remove\_min}(): O(n \log n) = O(n \log n)$



## A summary of sorting algorithms so far

Algorithm	worst	average	best	memory	in-place	stable
Bubble sort	$n^2$	$n^2$	$n$	1	yes	yes
Selection sort	$n^2$	$n^2$	$n^2$	1	yes	no
Insertion sort	$n^2$	$n^2$	$n$	1	yes	yes
Merge sort	$n \log n$	$n \log n$	$n \log n$	$n$	no	yes
Quicksort	$n^2$	$n \log n$	$n \log n$	$\log n$	yes	no
Bucket sort	$n^2$	$n^2/k$	$n$	$kn$	no	yes
Heap sort	$n \log n$	$n \log n$	$n$	1	yes	no
Timsort	$n \log n$	$n \log n$	$n$	$n$	no	yes

## A summary of sorting algorithms so far

Algorithm	worst	average	best	memory	in-place	stable
Bubble sort	$n^2$	$n^2$	$n$	1	yes	yes
Selection sort	$n^2$	$n^2$	$n^2$	1	yes	no
Insertion sort	$n^2$	$n^2$	$n$	1	yes	yes
Merge sort	$n \log n$	$n \log n$	$n \log n$	$n$	no	yes
Quicksort	$n^2$	$n \log n$	$n \log n$	$\log n$	yes	no
Bucket sort	$n^2$	$n^2/k$	$n$	$kn$	no	yes
Heap sort	$n \log n$	$n \log n$	$n$	1	yes	no
Timsort	$n \log n$	$n \log n$	$n$	$n$	no	yes
?	$n \log n$	$n \log n$	$n$	1	yes	yes

# Summary

- A priority queue is a useful ADT for many purposes
- Binary heaps implement priority queues efficiently
- Heap sort is an efficient algorithm based on priority queue implementation with heaps (Goodrich, Tamassia, and Goldwasser 2013, ch. 9)

Next:

- Graphs
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 14)

# Acknowledgments, credits, references



Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013).  
*Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. ISBN:  
9781118476734.







