

Priority queues and binary heaps

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

Çağrı Çöltekin
ccoltekin@ifa.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2022/23

version: 09/2020 002223-11-14

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)

Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 1 / 22

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

Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 2 / 22

Priority queues

Example operations

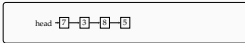
Operation	Return value	Priority queue
insert(5, a)		{(5,a)}
insert(9, c)		{(5,a), (9,c)}
insert(3, b)		{(5,a), (9,c), (3,b)}
insert(7, d)		{(5,a), (9,c), (3,b), (7,d)}
remove()	c	{(5,a), (3,b), (7,d)}
remove()	d	{(5,a), (3,b)}
remove()	a	{(3,b)}
remove()	b	{}

Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 3 / 22

Priority queue implementation

unsorted list

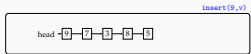


Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 4 / 22

Priority queue implementation

unsorted list



Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 5 / 22

Priority queue implementation

unsorted list

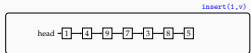


Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 6 / 22

Priority queue implementation

unsorted list



Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 7 / 22

Priority queue implementation

unsorted list



Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 8 / 22

Priority queue implementation

unsorted list



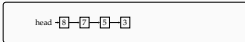
- Insert: $O(1)$
- Remove: $O(n)$

Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 9 / 22

Priority queue implementation

sorted list

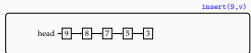


Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 10 / 22

Priority queue implementation

sorted list



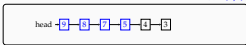
Ç. Çöltekin, INF | University of Tübingen

Winter Semester 2022/23 11 / 22

Priority queue implementation

sorted list

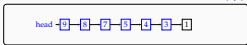
insert(4,v)



Priority queue implementation

sorted list

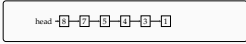
insert(1,v)



Priority queue implementation

sorted list

9 ← remove_max()



Priority queue implementation

sorted list

8 ← remove_max()

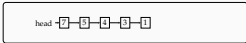


- Insert: $O(n)$
- Remove: $O(1)$

Priority queue implementation

sorted list

8 ← 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



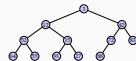
Height of a binary heap

- 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 a new item to a binary heap



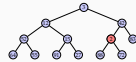
- Add the new element to the first available slot
- "Bubble up" until the heap property is satisfied
- At most $h = \log n$ comparisons/swaps

Adding a 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 a 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 a 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 a 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 a new item to a binary heap



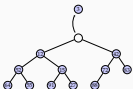
- Add the new element to the first available slot
- "Bubble up" until the heap property is satisfied
- At most $h = \log n$ comparisons/swaps

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



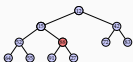
- The item to be removed is at the root
- We replace root with the element at the last slot
- "Bubble down" until the heap property is satisfied

Removing the min/max from a binary heap



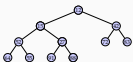
- The item to be removed is at the root
- We replace root with the element at the last slot
- "Bubble down" until the heap property is satisfied

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

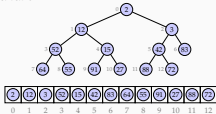
Removing the min/max from a binary heap



- 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

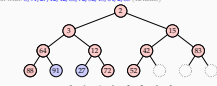


Bottom-up heap construction

- 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)



$$T(n) = \sum_{i=0}^h i \times 2^{h-i} = \sum_{i=0}^h i \times \frac{1}{2^i} \times 2^h = 2^h \sum_{i=0}^h \frac{i}{2^i} = \frac{n+1}{2} \sum_{i=0}^h \frac{1}{2^i} = O(n)$$

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 `heappify(h)` construct a heap from given list `heappop(h)`

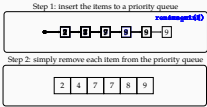
```
>>> h = []
>>> heappush(h, ('this is important'))
>>> heappush(h, ('this, not so much'))
>>> heappush(h, ('this is quite important too'))
>>> heappush(h, ('highest priority'))
>>> heappush(h, ('fairly important'))
>>> h
[('highest priority'), ('this is important'), ('this is quite important too'),
 ('this, not so much'), ('fairly important')]
>>> heappop(h)
('this, not so much')
>>> heappop(h)
('highest priority'), ('this is important'), ('this is quite important too'),
 ('fairly important'), ('this, not so much')
>>> h
[('highest priority'), ('this is important'), ('this is quite important too'),
 ('fairly important'), ('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 an unsorted list, the algorithm is equivalent to the selection sort $O(n^2)$
 - If we use a binary heap, we get an $O(n \log n)$ algorithm (heap sort)

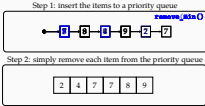
Insertion sort with priority queues

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



Selection sort with priority queues

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



Sorting with heaps

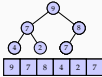
a first attempt

- 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(heap, item)
    for i in range(len(seq)):
        seq[i] = heappop(heap)
```

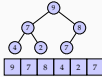
In-place heap sort

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



In-place heap sort

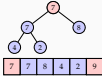
step 2: iteratively remove the maximum element, place it at the end



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

In-place heap sort

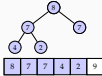
step 2: iteratively remove the maximum element, place it at the end



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

In-place heap sort

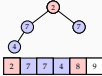
step 2: iteratively remove the maximum element, place it at the end



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

In-place heap sort

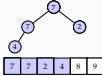
step 2: iteratively remove the maximum element, place it at the end



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

In-place heap sort

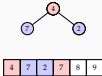
step 2: iteratively remove the maximum element, place it at the end



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

In-place heap sort

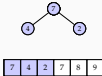
step 2: iteratively remove the maximum element, place it at the end



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

In-place heap sort

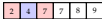
step 2: iteratively remove the maximum element, place it at the end



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

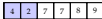
In-place heap sort

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

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

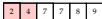
In-place heap sort

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

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

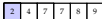
In-place heap sort

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

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

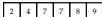
In-place heap sort

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

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

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
?	$n \log n$	$n \log n$	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 ([goodrich2013](#))

Next:

- Graphs
- Reading: [goodrich2013](#)

Acknowledgments, credits, references