Lecture 4: Priority Queues, Heaps and Heapsort

Reading: CLRS 6.1-6.4
Priority Queue (PQ)

An abstract data structure (aka data type) maintaining a set $S$ of elements, each associated with a key, supporting the following operations:

- $\text{insert}(S, x)$: insert element $x$ into set $S$
- $\text{max}(S)$: return element of $S$ with largest key
- $\text{extract}\_\text{max}(S)$: return element of $S$ with largest key and remove it from $S$
- $\text{increase}\_\text{key}(S, x, k)$: increase value of $x$'s key to new value $k$
  (assumed to be $\geq$ the current key value)

**Think:** All the operations you would need to organize triage in an emergency room $\Rightarrow$ key value = severity of patient’s condition

(Tons of applications in algorithms and across the whole CS too.)
Priority Queue (PQ)

An abstract data structure (aka data type) maintaining a set $S$ of elements, each associated with a key, supporting the following operations:

- $\text{insert}(S, x)$: insert element $x$ into set $S$
- $\text{max}(S)$: return element of $S$ with largest key
- $\text{extract}\_\text{max}(S)$: return element of $S$ with largest key and remove it from $S$
- $\text{increase}\_\text{key}(S, x, k)$: increase value of $x$'s key to new value $k$ (assumed to be $\geq$ the current key value)

This abstraction specifies desired functionality/interface, but how to implement it?

Naïve way: Use an (unsorted) array and scan all elements to find max
- each insert and increase key takes $\Theta(1)$ time
- all the other operations take $\Theta(n)$ (worst-case) time

Can we do better?
(Max) Heap

- **Data structure** implementing a priority queue
- It is an **array** that:
  - we visualize as a (nearly complete) **binary tree**
  - satisfies **Max Heap Property (MHP)**:
    - Key of a node is $\geq$ than the keys of its children
- (Min Heap defined analogously)

**Important fact:**
Height of the tree is always $O(\log n)$
Mapping Tree to a Heap

root: first element in the array (i=1)
parent(i): floor(i/2) returns index of node's parent
left(i): 2i returns index of node's left child
right(i): 2i+1 returns index of node's right child

(Note: No pointers needed!)

Important detail: We index elements starting from i=1 here
Why Heaps?

Key consequence of Max Heap Property:
Root/first element is always the max \( \Rightarrow \) can do \( \text{max}(S) \) in \( \Theta(1) \) time!
(Note: the array is not sorted though!)

**But:** How to maintain the Max Heap Property property after \( \text{insert/extract\_max/increase\_key} \)?

**In fact:**
How to build a Max Heap to begin with?
Key Primitive

max_heapify(A[i]): Corrects a **single** violation of Max Heap Property in a subtree rooted at i only

**How to implement it?**

- Assume that the trees rooted at left(i) and right(i) are Max Heaps
- If element A[i] violates the MHP, correct violation by “trickling” this element down the tree, making the subtree rooted at i a Max Heap

(Important: Always swap with the **larger** of two children. Why?)

![Max Heap Diagram](image-url)
Key Primitive

max_heapify(A[i]): Corrects a **single** violation of Max Heap Property in a subtree rooted at i only

In other words:

⇒ Assume that the trees rooted at left(i) and right(i) are Max Heaps
⇒ If element A[i] violates the MHP, correct violation by “trickling” this element down the tree, making the subtree rooted at i a Max Heap

(Important: Always swap with the larger of two children)
Key Primitive

max_heapify(A[i]): Corrects a **single** violation of Max Heap Property in a subtree rooted at i **only**

*In other words:*

⇒ Assume that the trees rooted at `left(i)` and `right(i)` are Max Heaps

⇒ If element `A[i]` violates the MHP, correct violation by “trickling” this element down the tree, making the subtree rooted at i a Max Heap

(Important: Always swap with the **larger** of two children)

![Max Heap Tree](image)

**Run time?**

⇒ $\Theta(1)$ at each level

⇒ total: $O$(subtree height) = $\Theta$ (log n) (worst case)

(Recall: tree is balanced)

Guaranteed to be a Max Heap now
Implementing Extract_Max
Implementing Extract_Max

- Swap the root with the last element of the heap
Implementing Extract_Max

- Swap the root with the last element of the heap
- Now we can remove it from the heap (decrease the heap size by one)
- To fix MHP: Max_heapify the root
Implementing Extract_Max

- Swap the root with the last element of the heap
- Now we can remove it from the heap (decrease the heap size by one)
- To fix MHP: Max_heapify the root
Implementing Extract_Max

- Swap the root with the last element of the heap
- Now we can remove it from the heap (decrease the heap size by one)
- To fix MHP: Max_heapify the root
Implementing Extract_Max

- Swap the root with the last element of the heap
- Now we can remove it from the heap (decrease the heap size by one)
- To fix MHP: Max_heapify the root
Implementing Extract_Max

- Swap the root with the last element of the heap
- Now we can remove it from the heap (decrease the heap size by one)
- To fix MHP: Max_heapify the root
- Done!

Run time?

→ $\Theta(1)$ (swapping) + $\Theta(1)$ (removal) + $O(\log n)$ (max_heapify)
→ total: $\Theta(\log n)$ (worst case)
Implementing Insert
(in a sense: “reversing” the extract_max)
Implementing Insert
(in a sense: “reversing” the extract_max)

- Add the new element as the last one
- To fix MHP:
  “Promote” the new element up the tree
  (“reversed” max_heapify)
Implementing Insert
(in a sense: “reversing” the extract_max)

- Add the new element as the last one
- To fix MHP:
  “Promote” the new element up the tree
  (“reversed” max_heapify)
Implementing Insert  
(in a sense: “reversing” the extract_max)

• Add the new element as the last one

• To fix MHP:
  “Promote” the new element up the tree
  (“reversed” max_heapify)

• Done!

Run time?
⇒ \(\Theta(1)\) (addition) + \(O(\log n)\) (promotion up the tree)
⇒ total: \(\Theta(\log n)\) (worst case)
Implementing Increase_key
(Similar to Insert)
Implementing Increase_key (Similar to Insert)

- Increase the key value
- To fix MHP:
  Again, “promote” the new element up the tree

This is still a heap
Implementing Increase_key
(Similar to Insert)

- Increase the key value
- To fix MHP:
  Again, “promote” the new element up the tree
Implementing Increase_key
(Similar to Insert)

- Increase the key value
- To fix MHP: Again, “promote” the new element up the tree

Done!

Run time?

⇒ \( \Theta(1) \) (key value increase) + \( O(\log n) \) (promotion up the tree)
⇒ total: \( \Theta(\log n) \) (worst case)
How to build a heap from a scratch?

Simple way:
- Start with an empty heap
- Insert all the $n$ elements into it
- Total time: $\Theta(n \log n)$ (worst-case)

Better way:
- Use divide & conquer! (see blackboard)
- $T(n) = 2 \cdot T(n/2)$ (conquer) + $O(\log n)$ (in-place divide & combine)
- Total time: $\Theta(n)$ (by Master Theorem)

Iterative (and in-place) way:
```
build_max_heap(A):
    for $i=n$ downto $1$
        do max_heapify(A, i)
```

Total time? At first glance: $\Theta(n \log n)$ (cost of $n$ max_heapify)
Actually: $\Theta(n)$ (see blackboard)
Cool application: Sorting

**Heapsort:** Sorting using a heap/priority queue

- Build a heap out of all elements
- Extract max all elements one-by-one in an (inversely) sorted order!
- Total time: $\Theta(n \log n)$ (worst-case)

This is a different algorithm than Merge sort!

**In particular:** Heapsort is actually an in-place algorithm (once we unravel the implementation of the heap)