Tutorial 5: Heaps and Priority Queues

September 26, 2022

Gu Zhenhao

** Partly adopted from tutorial slides by Wang Zhi Jian.*

Priority Queue ADT

Why do we need the Priority Queue ADT?

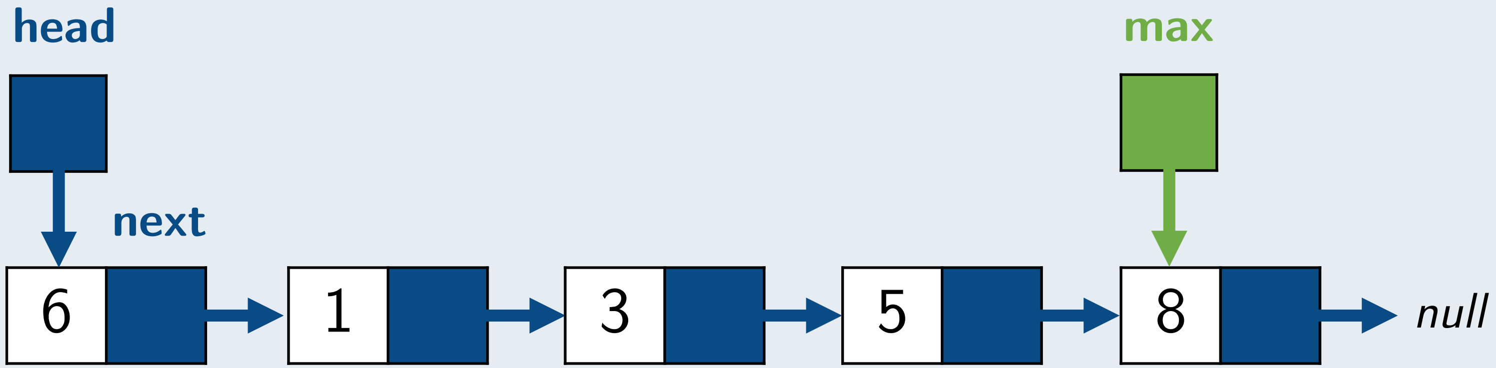
Why Priority Queue?

Operations	Array	Linked List
`getItemAtIndex`	$O(1)$	$O(n)$
`getFirst/getLast`	$O(1)$	$O(1)^*$
`addAtIndex/removeAtIndex`	$O(n)$	$O(n)$
`addFront/removeFront`	$O(n)$	$O(1)$
`addBack/removeBack`	$O(n)$ ($O(1)$ amortized)	$O(1)^*$
`findMax/extractMax`	$O(n)$	$O(n)$

- Searching for the key with highest priority (largest value) in arrays and linked lists is slow.
- We will have to look through all keys to find the max key.

Implementation of Priority Queue

- **Idea**: Maintain a pointer that points to the largest key.

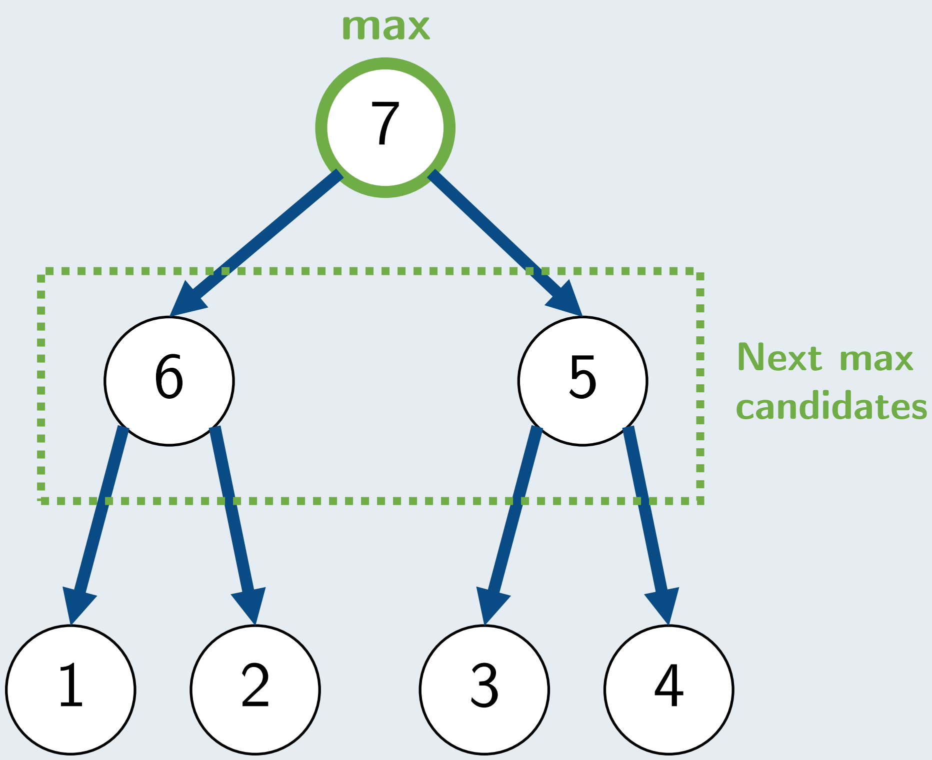


Finding the max now costs $O(1)$.

- **Problem**: If we extract the max key, need $O(n)$ to **search the whole list** to find the 2nd largest key.

Is there a way to limit our search range to $O(1)$ number of keys?

Implementation of Priority Queue



- **Idea**: Let the max key point to the next largest keys.

We use a binary tree where:

- We keep the max key at root,
- Each node has at most 2 children,
- Every child key $\leq$ parent key.

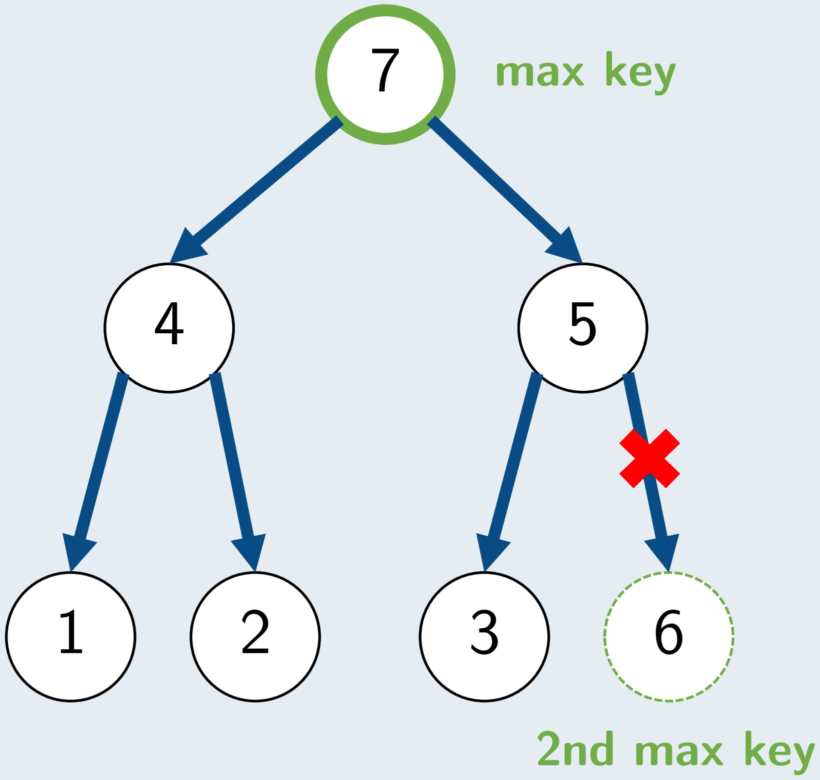
Problem 1.a

True or false: *The smallest element in a min heap is always the root.*

True. This can be directly inferred from property of min heap.

Problem 1.b

True or false: *The second largest element in a max heap with more than two elements (all elements are unique) is always one of the children of the root.*



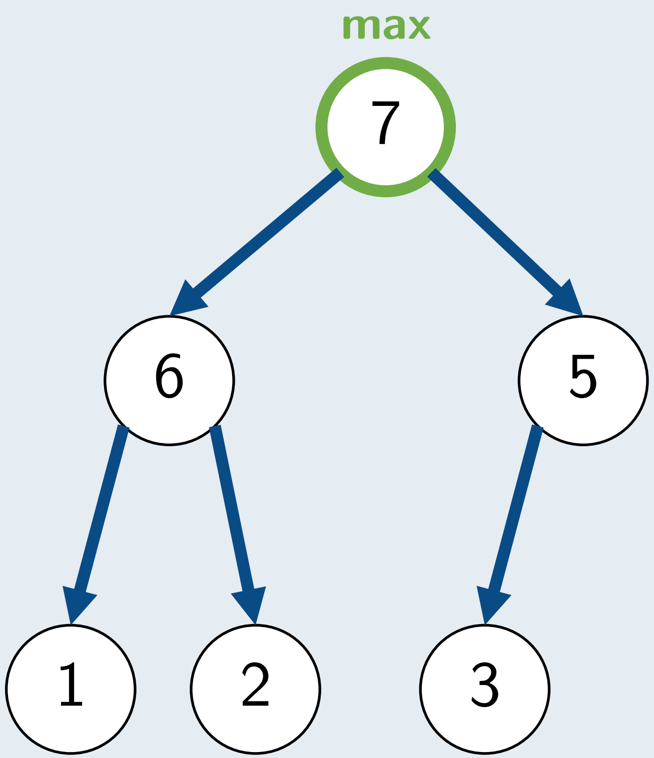
True. This can be proved by contradiction.

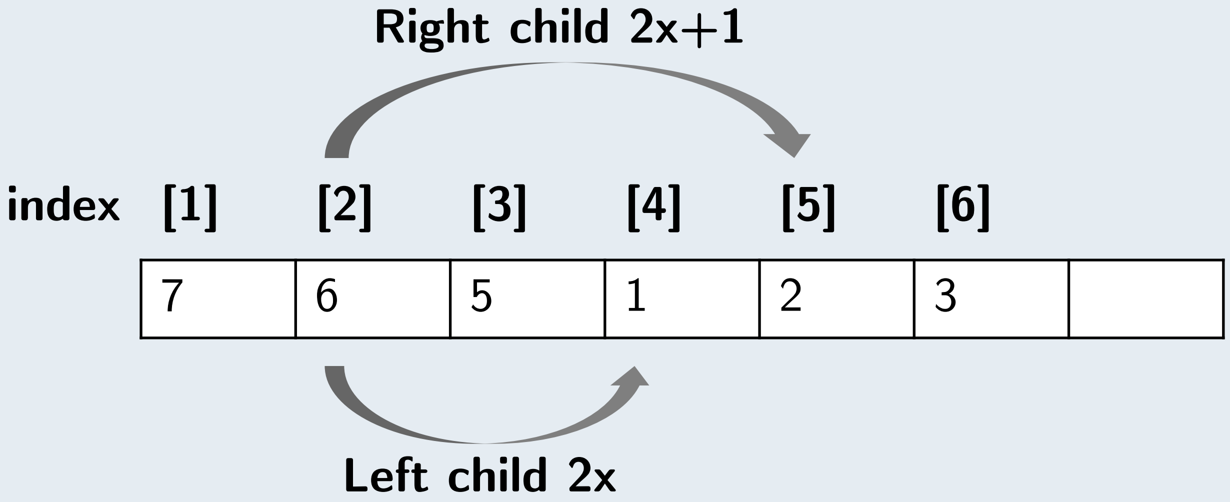
Suppose not, then the 2^{nd} largest key must be some descendent of a child of the root.

The child is smaller than the max key but larger than 2^{nd} largest key. $\lightning$

Problem 1.c

True or false: *When a heap is stored in an array, finding the parent of a node takes at least* $O(\log n)$ *time.*

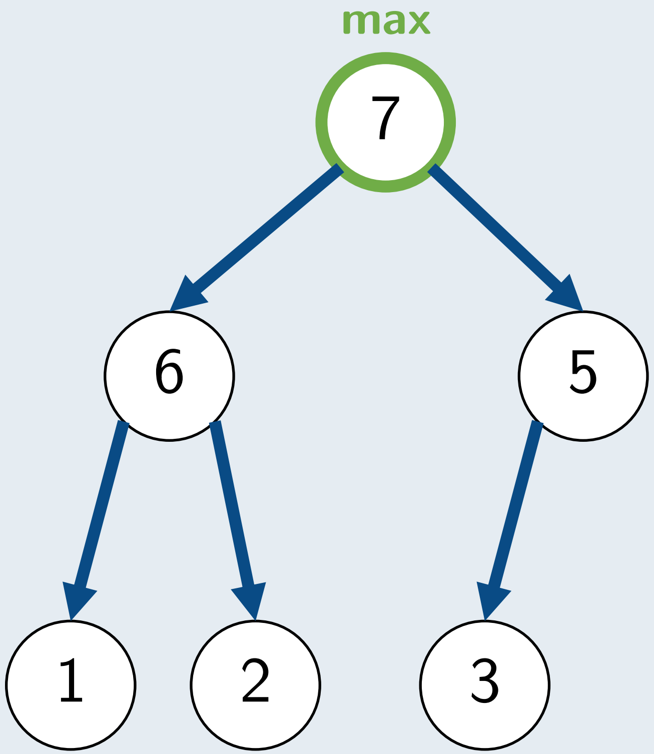




False. For a node at index x, its parent is at index $\lfloor x/2 \rfloor$. We can find it in $O(1)$ time.

Problem 1.d

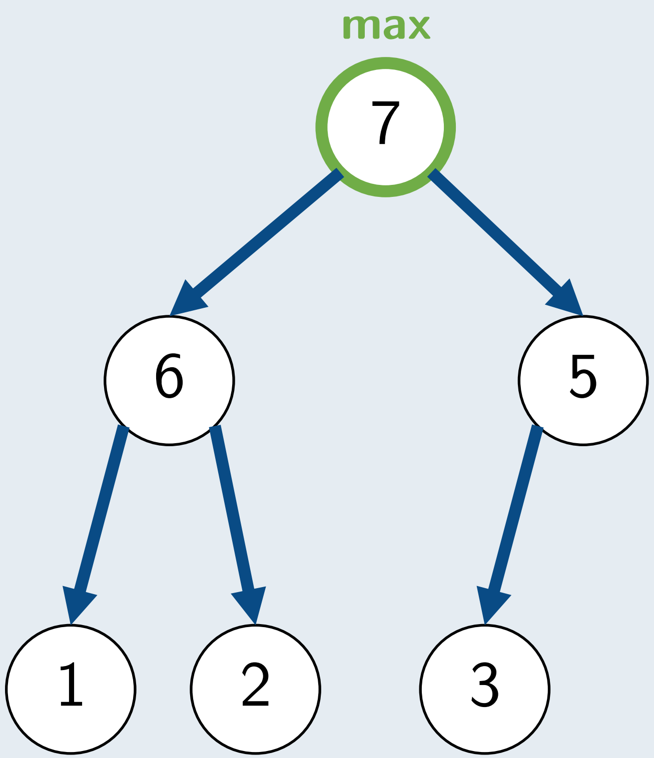
True or false: *Every node in the heap except the leaves has exactly 2 children.*

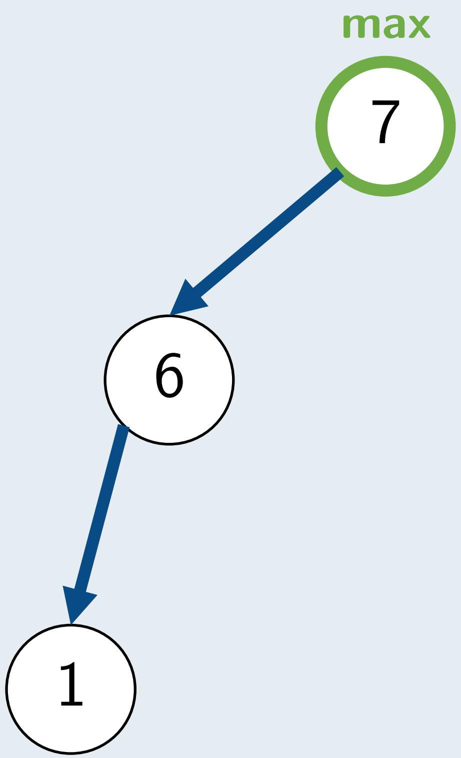


False. As an counter-example, the tree on the left has a node 5 that has only 1 child.

Pause and Ponder

Question: what is the time complexity for `extractMax` if our tree is extremely biased: each node has only one child?





** Find the answer in the Appendix!*

Problem 1.e

True or false: *We can obtain a sorted sequence of the heap elements in $O(n)$ time.*

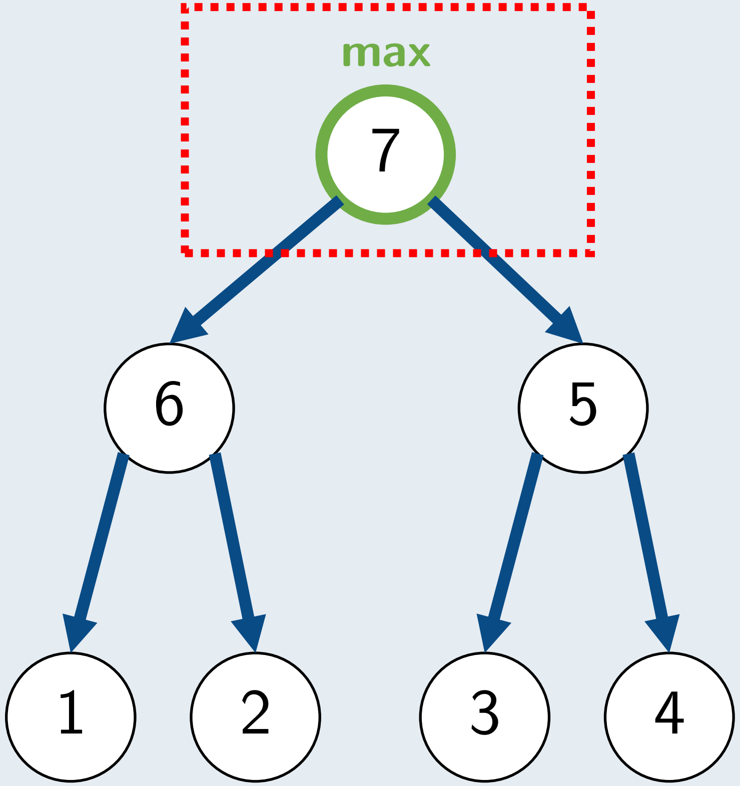
False. Using a min heap, we need to do `extractMin` repeatedly for n times until we extract everything. Each `extractMin` costs $O(\log n)$ time. In total we still need $O(n \log n)$ time.

Tree Traversal

Why do we go through all nodes in a tree?

Problem 2

Give an algorithm to find all vertices bigger than some value x in a max heap that runs in $O(k)$ time where k is the number of vertices in the output.



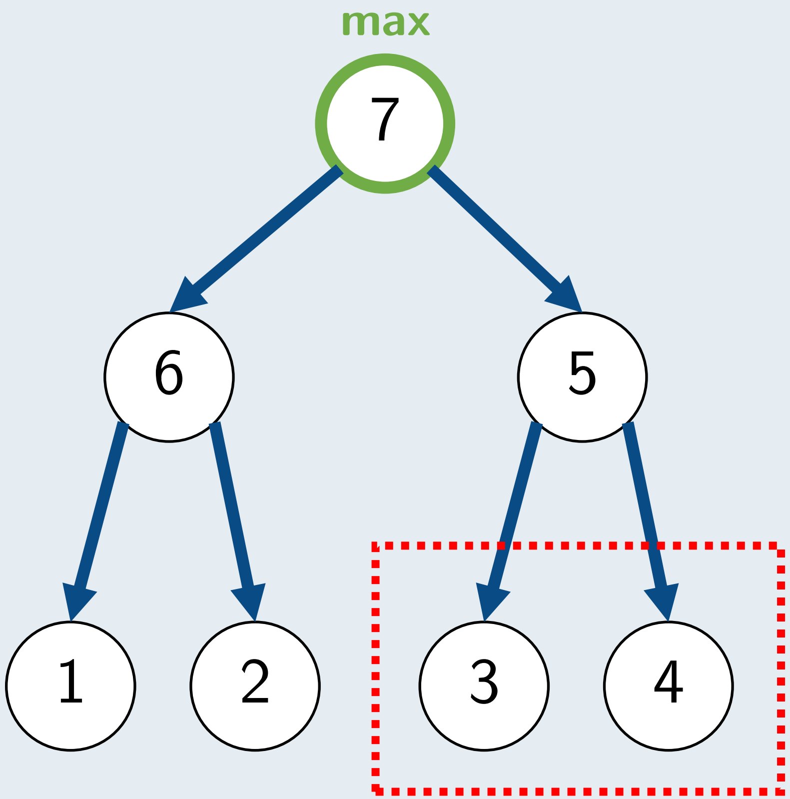
Example: Let $x = 8$.

Trivial Answer: Scan through all nodes and report those nodes with value > 8.

No need to check any of the nodes because the root is already < 8!

Problem 2

Give an algorithm to find all vertices bigger than some value x in a max heap that runs in $O(k)$ time where k is the number of vertices in the output.



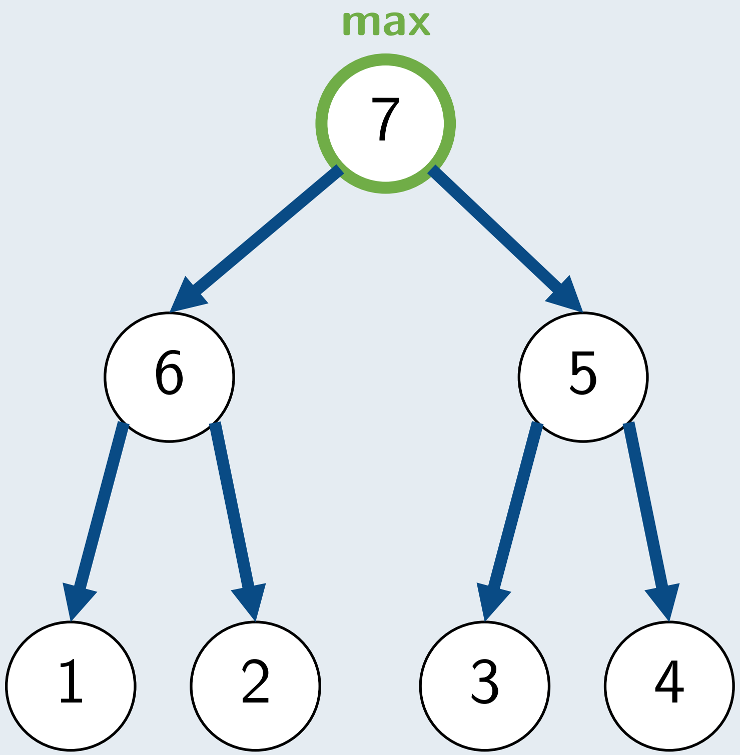
Example: Let $x = 5$.

Trivial Answer: Scan through all nodes and report those nodes with value > 5.

No need to check these nodes as their parent is already ≤ 5!

Problem 2

Give an algorithm to find all vertices bigger than some value x in a max heap that runs in $O(k)$ time where k is the number of vertices in the output.



Idea:

- Start from the root,
  - If it is $> x$, record the root and go on to check its children,
  - Otherwise, no need to check its children.

Problem 2

Algorithm 1 Solution to Problem 2

```
procedure FindNodesBiggerThanX(node, x)
    if node.key > x then
        output node.key
        FindNodesBiggerThanX(node.left, x)
        FindNodesBiggerThanX(node.right, x)
    else
        return output (or terminate algorithm)
    end if
end procedure
```

- The k nodes that are $> x$ will be visited only once.
- At most $2k$ nodes that are $\leq x$ will be visited.
- Total time $O(k + 2k) = O(k)$.

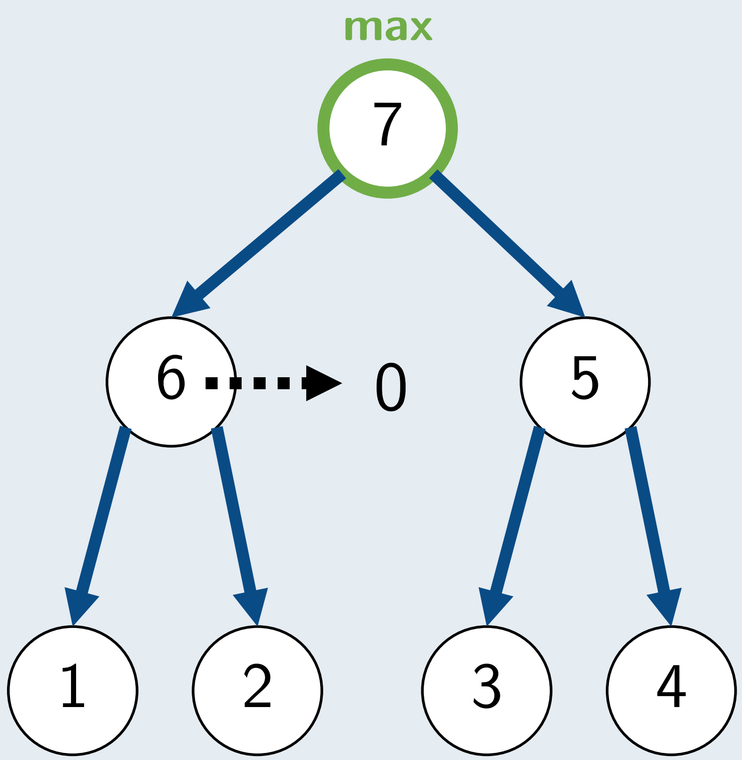
* *This is similar to **pre-order traversal**: visit the root, then visit left subtree, and finally right subtree.*

Adjusting Priority

Why to adjust the priority of a key?

Problem 3

Enable `update(int old_key, int new_key)` *operation that updates the value of* `old_key` *in a binary heap with* `new_key`*.*



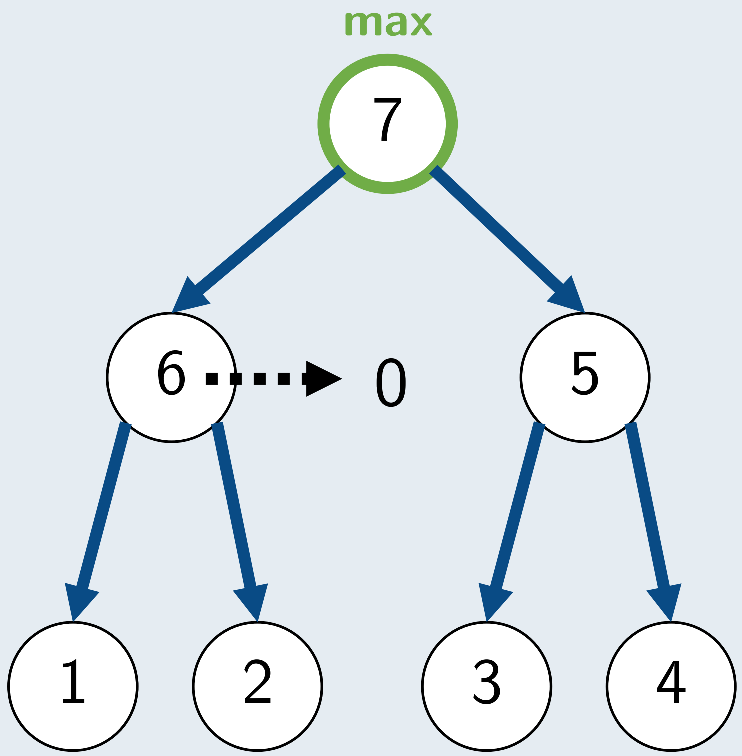
Example: `update(6, 0)`.

Steps:

1. Find the node to be updated.
2. Update the key and move the node to the correct position.

Problem 3

***Step 1**: Find the node to be updated.*

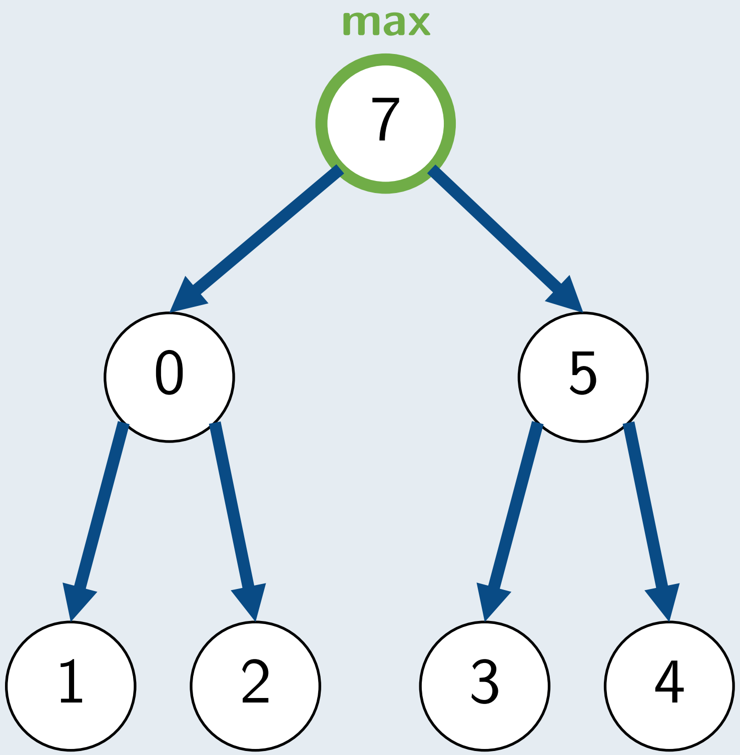


Trivial answer: traverse through all nodes and find the required key.

Can use a **HashMap⟨key, node⟩** to store all nodes beforehand to find it in $O(1)$ time.

Problem 3

Step 2: Update the key and move the node to the correct position.

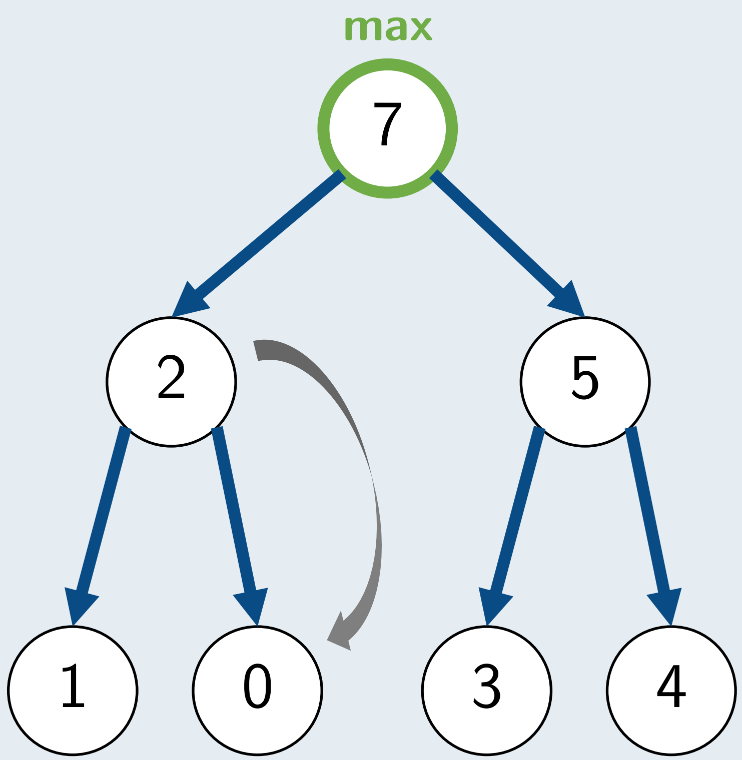


Can just use the `shiftUp` and `shiftDown` method in the lecture!

The shifting needs $O(\log n)$ time.

Problem 3

Step 2: Update the key and move the node to the correct position.



Can just use the `shiftUp` and `shiftDown` method in the lecture!

The shifting needs $O(\log n)$ time.

Implementation of Priority Queue

Operations	Linked List	Binary Heap
`insert`	$O(1)$	$O(\log n)$
`extractMax`	$O(n)$	$O(\log n)$
`findMax`	$O(n)$	$O(1)$
`update`	$O(1)$	$O(\log n)$

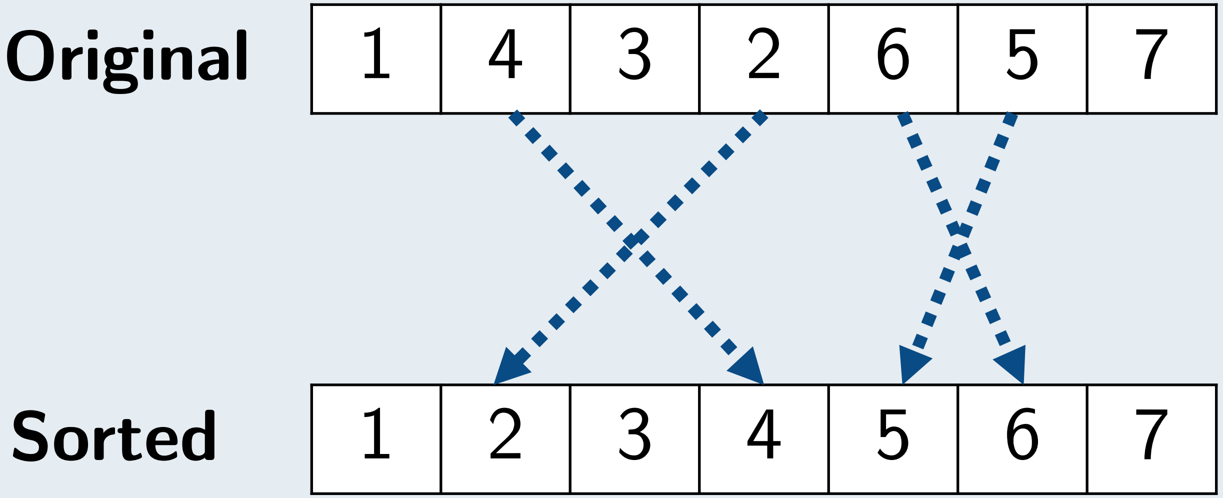
- There are several other ways of implementation, e.g. binomial heap*, Fibonacci heap* that make certain operations run faster.

Applications

How to make use of heap?

Problem 4

Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions.



Problem 4.a

Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, ***where*** $\boldsymbol{k = 1}$.

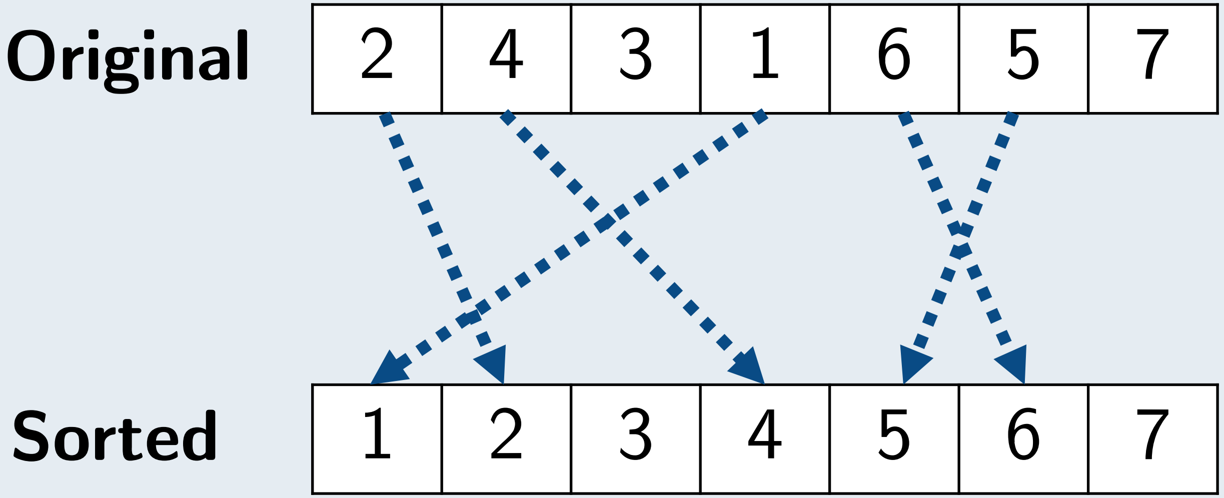
Original	1	3	2	4	6	5	7
Sorted	1	2	3	4	5	6	7

If $k = 1$, each wrongly-placed values can be corrected by a single swap.

Use insertion sort or improved bubble sort!

Problem 4.b

*Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, **where k is arbitrary**.*



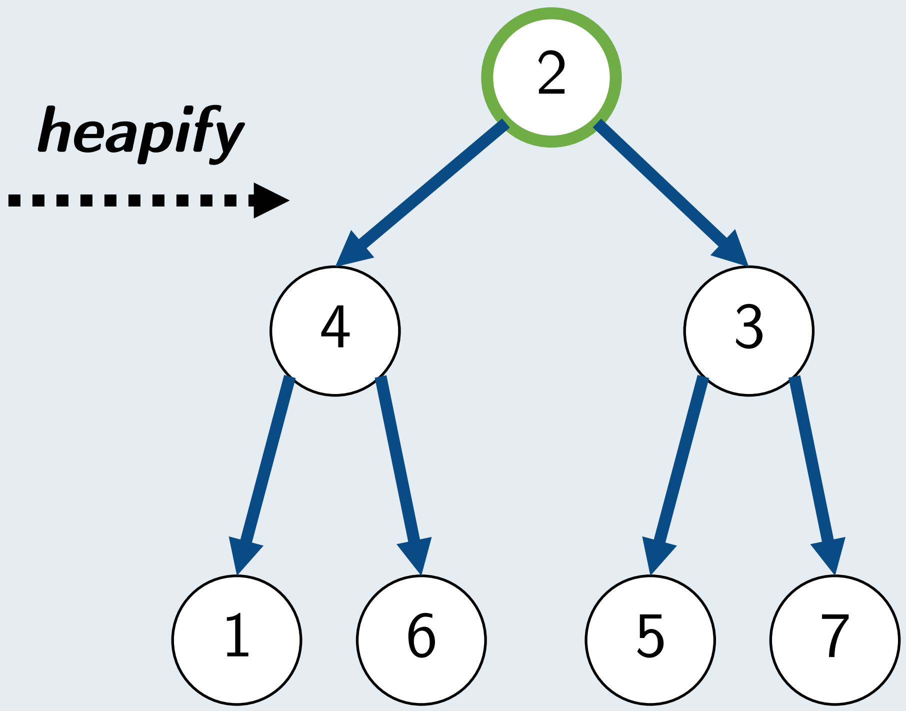
Example: $k = 3$.

Trivial answer: simply use a traditional sorting algorithm, e.g. heap sort, $O(n \log n)$.

Problem 4.b

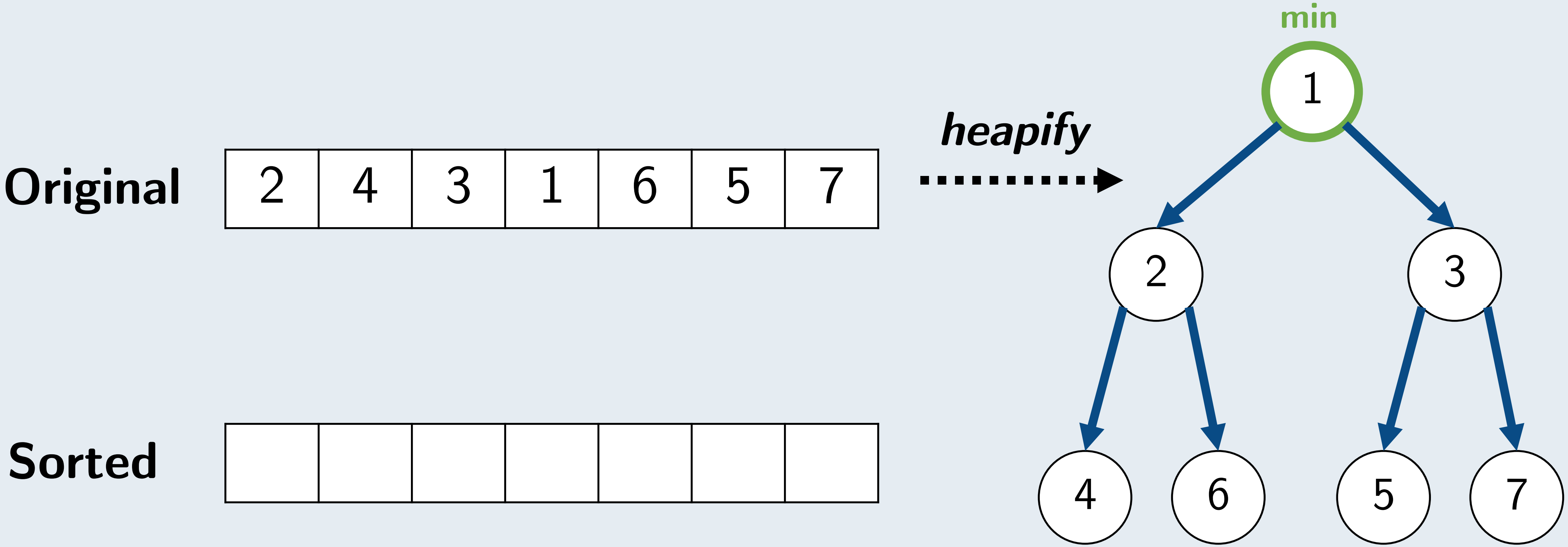
*Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, **where k is arbitrary**.*

Original	2	4	3	1	6	5	7
Sorted							



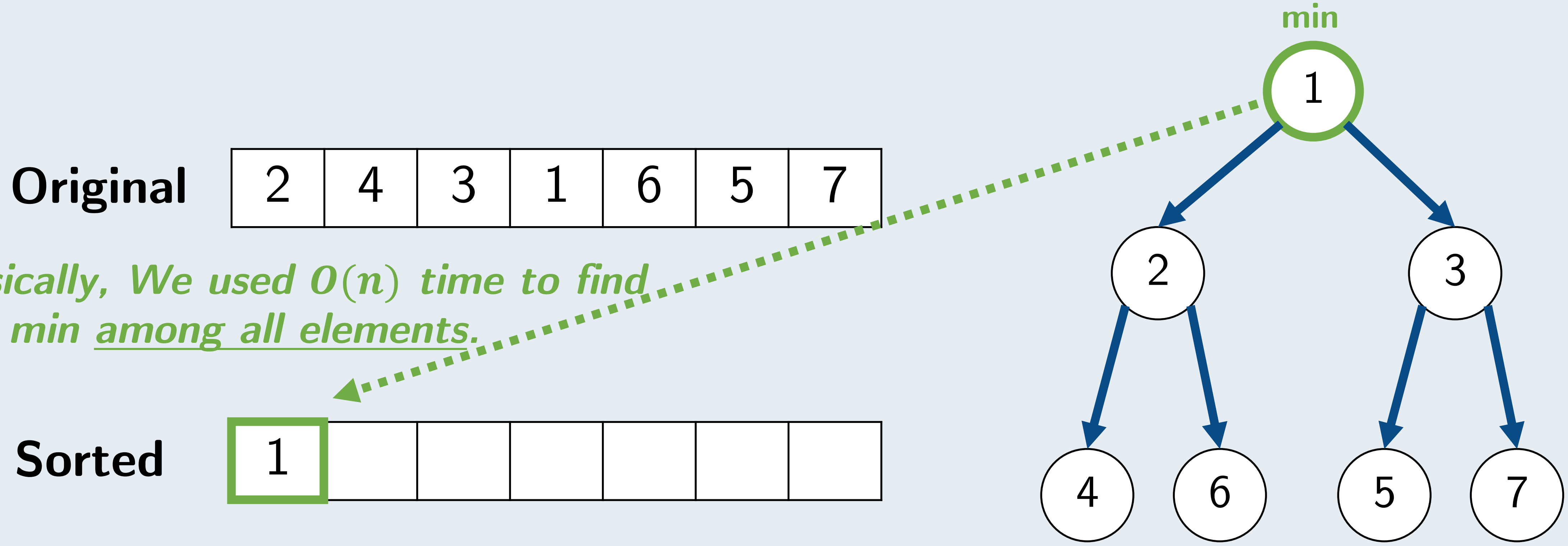
Problem 4.b

*Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, **where k is arbitrary**.*



Problem 4.b

Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, ***where k is arbitrary****.*



Basically, We used $O(n)$ time to find the min among all elements.

Problem 4.b

Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, ***where k is arbitrary***.

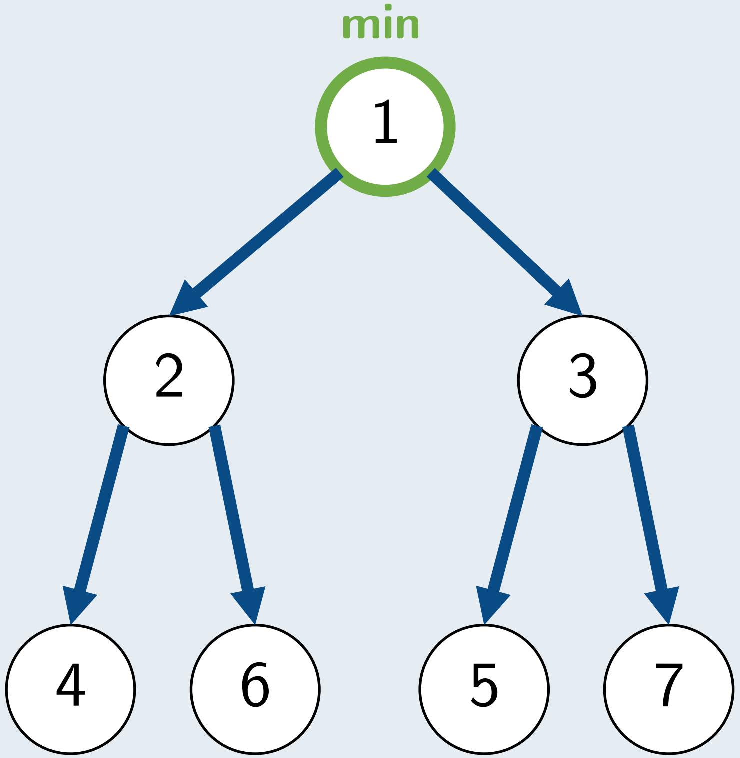
Original

2	4	3	1	6	5	7

Question: *Do we actually need to scan through all elements to find the min?*

Sorted

1						



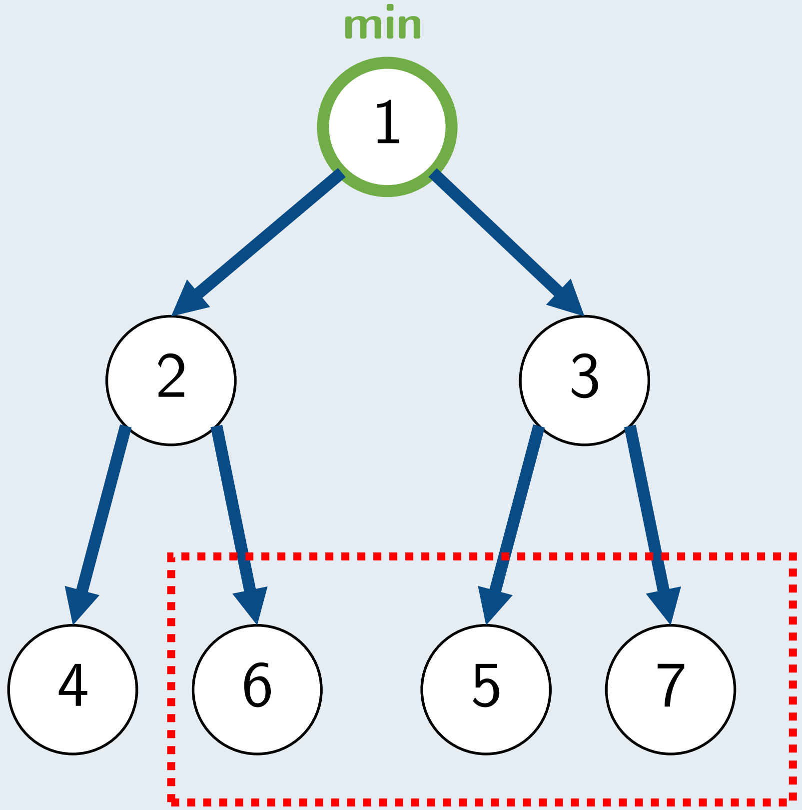
Problem 4.b

*Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, **where k is arbitrary**.*

Original	2	4	3	1	6	5	7

Claim. *We can find min among the first $k+1$ elements!*

Sorted	1						



No need to include these nodes for now!

Problem 4.b

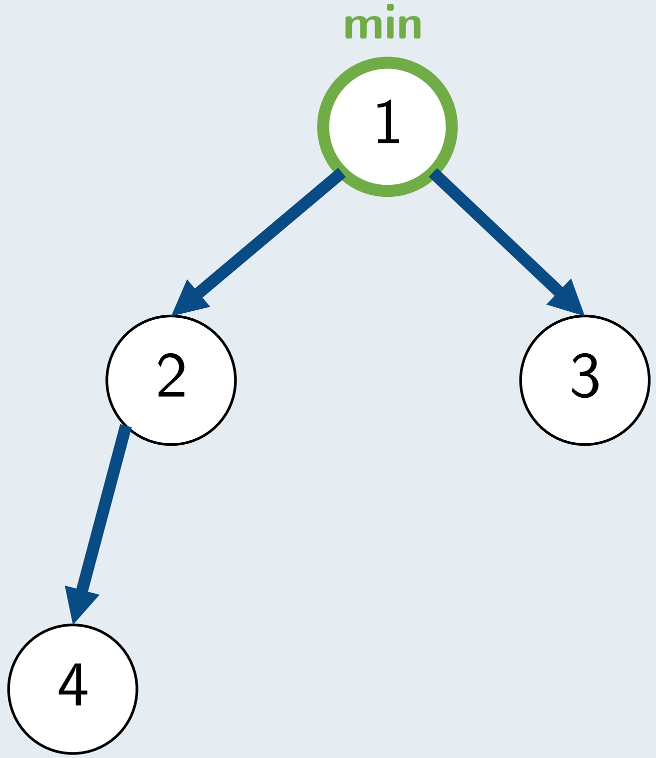
Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, ***where k is arbitrary****.*

Original

2	4	3	1	6	5	7

Claim. *We can find min among the first $k+1$ elements!*

Sorted



Problem 4.b

Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, ***where k is arbitrary****.*

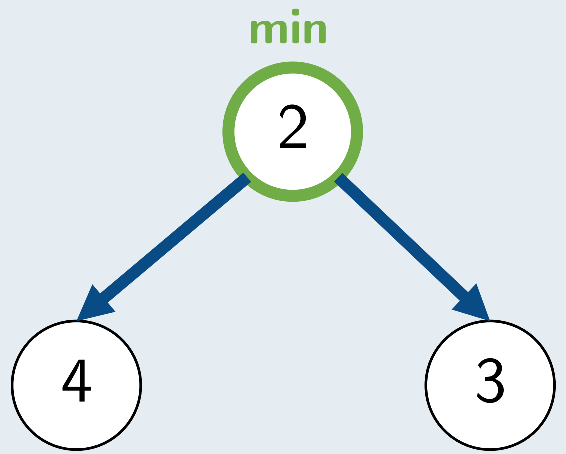
Original

	2	4	3	6	5	7

Claim. *We can find min among the first $k + 1$ elements!*

Sorted

1						



Problem 4.b

Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, ***where k is arbitrary****.*

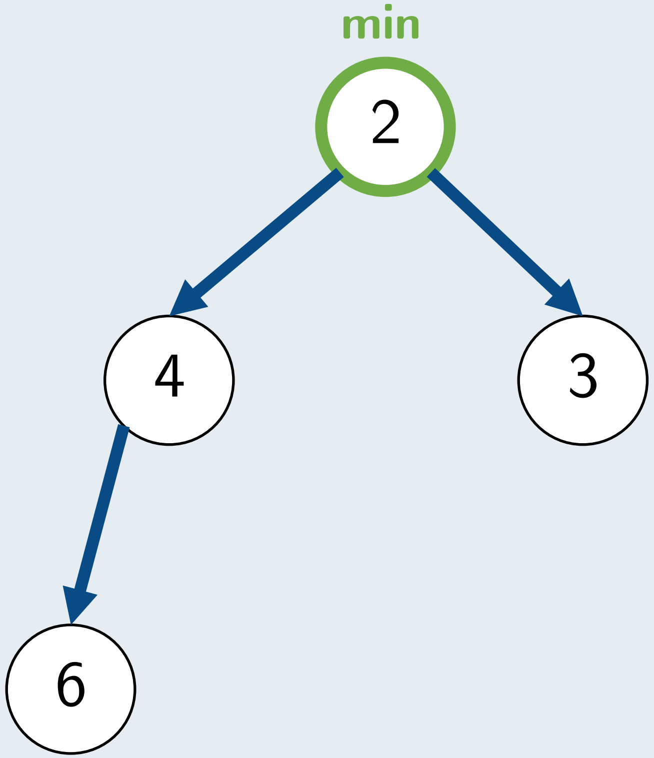
Original

	2	4	3	6	5	7

Claim. *We can find the second smallest between the 2nd and $(k+2)^{th}$ elements!*

Sorted

1	2					

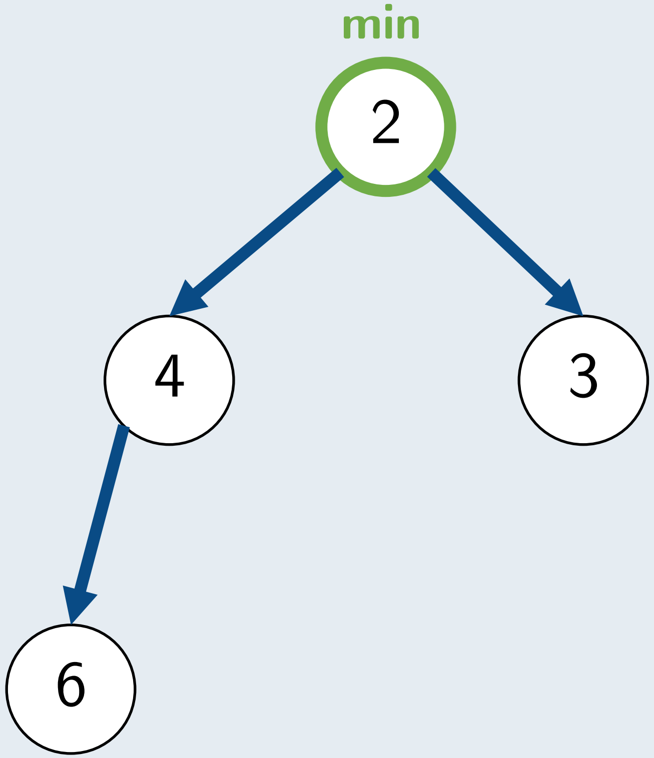


Problem 4.b

*Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, **where k is arbitrary**.*

Idea:

- Keep a min heap of size $k+1$, and initialize with first $k+1$ elements in array.
- Extract min, then insert the next element in array to the heap.
- Repeat until we extracted all elements in the heap.

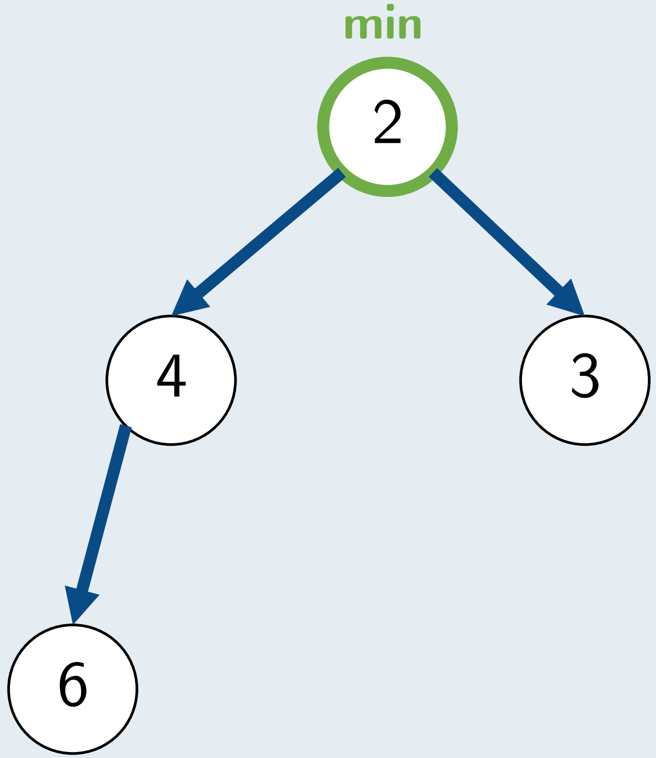


Problem 4.b

*Sort an almost sorted array, where each value differs from its correct position in the sorted array by no more than k positions, **where k is arbitrary**.*

Analysis:

- Need $O(k)$ extra space for min heap.
- Initializing heap costs $O(k)$, each time we extract min takes $O(\log k)$. In total we need $O(k + n \log k)$ time.



Problem 5

We have a stack of n integers. Each time we can pop an integer from the top k integers in the stack. We want to maximize the sum of popped integers.

$k = 2$

2
-10
2
-6
5

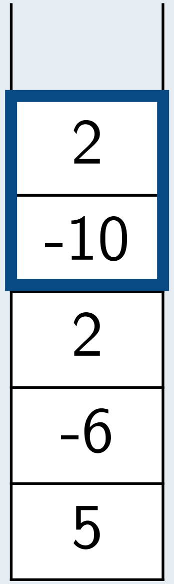
Simplified goal:

- If we choose to pop no integer, $\text{sum} = 0$.
- If we choose to pop only one integer, which one should we pop?
- If we choose to pop two integers, which two should we pop?

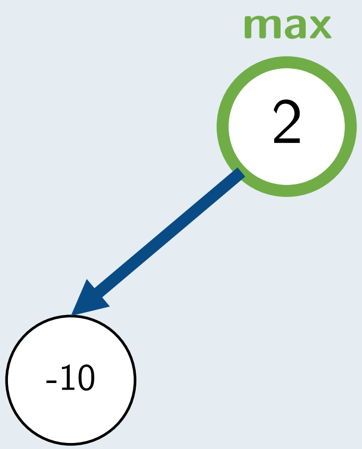
Problem 5

We have a stack of n integers. Each time we can pop an integer from the top k integers in the stack. We want to maximize the sum of popped integers.

$k = 2$



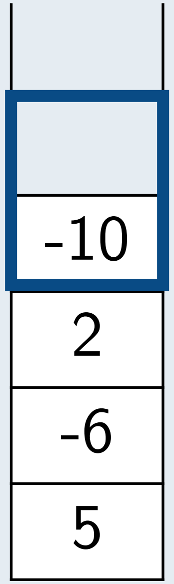
No. of pops	**0**	**1**	**2**	**3**	**4**	**5**
Max sum we can achieve	0					



Problem 5

We have a stack of n integers. Each time we can pop an integer from the top k integers in the stack. We want to maximize the sum of popped integers.

$k = 2$

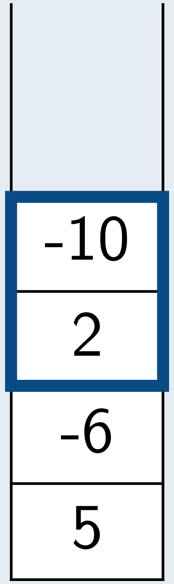


No. of pops	**0**	**1**	**2**	**3**	**4**	**5**
Max sum we can achieve	0	2				

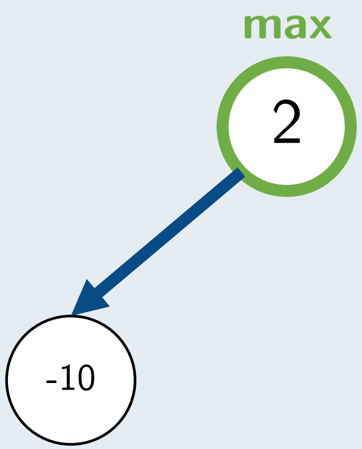
Problem 5

We have a stack of n integers. Each time we can pop an integer from the top k integers in the stack. We want to maximize the sum of popped integers.

$k = 2$



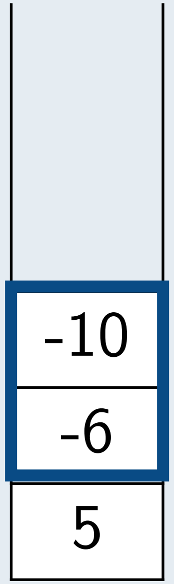
No. of pops	0	1	2	3	4	5
Max sum we can achieve	0	2				



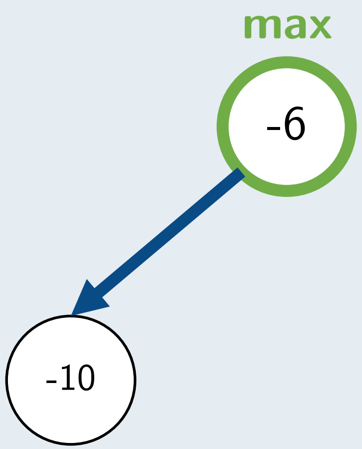
Problem 5

We have a stack of n integers. Each time we can pop an integer from the top k integers in the stack. We want to maximize the sum of popped integers.

$k = 2$



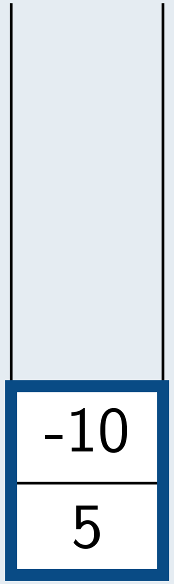
No. of pops	**0**	**1**	**2**	**3**	**4**	**5**
Max sum we can achieve	0	2	4			



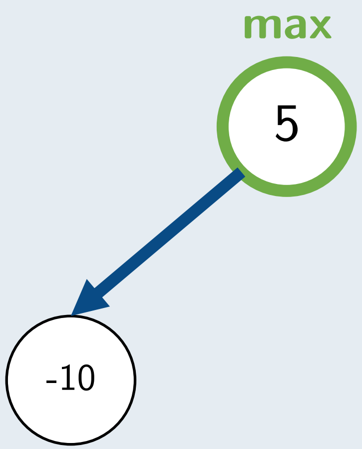
Problem 5

We have a stack of n integers. Each time we can pop an integer from the top k integers in the stack. We want to maximize the sum of popped integers.

$k = 2$



No. of pops	0	1	2	3	4	5
Max sum we can achieve	0	2	4	-2		



Problem 5

We have a stack of n integers. Each time we can pop an integer from the top k integers in the stack. We want to maximize the sum of popped integers.

$k = 2$

-10

No. of pops	**0**	**1**	**2**	**3**	**4**	**5**
Max sum we can achieve	0	2	4	-2	3	

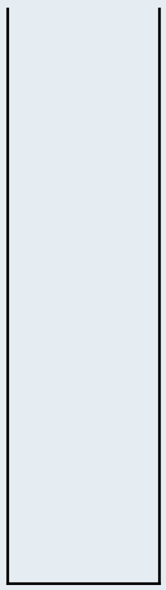
max

-10

Problem 5

We have a stack of n integers. Each time we can pop an integer from the top k integers in the stack. We want to maximize the sum of popped integers.

$k = 2$



No. of pops	**0**	**1**	**2**	**3**	**4**	**5**
Max sum we can achieve	0	2	4	-2	3	-7

The best sum
we can achieve.

Problem 5

Algorithm 2 Solution to Problem 5

```
Let A be the stack of integers
Initialise maximum heap D
for i = 1 to k do
    Insert A.pop() into D
end for
max_sum = 0
current_sum = 0
for i = k + 1 to n do
    current_sum = current_sum + D.extractMax()
    Insert A.pop() into D
    if current_sum > max_sum then
        max_sum = current_sum
    end if
end for
while D is not empty and D.getMax() > 0 do
    current_sum = current_sum + D.extractMax()
    if current_sum > max_sum then
        max_sum = current_sum
    end if
end while
return max_sum
```

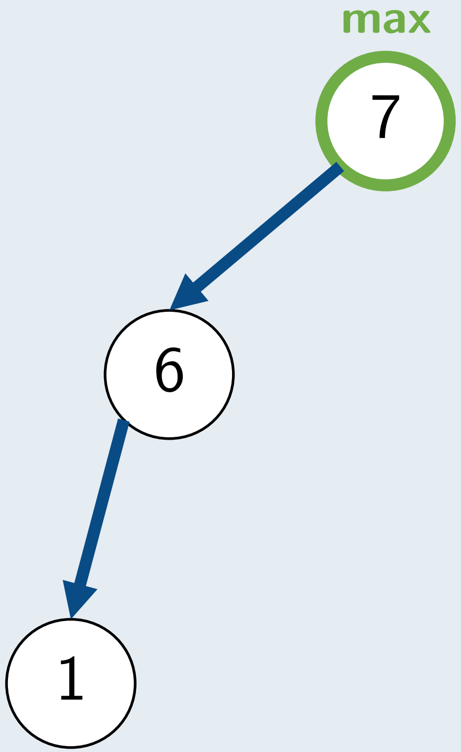
Analysis:

- We again need $O(k)$ extra space for the heap.
- Initializing the heap with top k elements cost $O(k)$. Each operation (extract max, then insert) costs $O(\log k)$ time. In total $O(k + n \log k)$ time.
- Similar to problem 4!

Appendix

Pause and Ponder

Question: what is the time complexity for `extractMax` if our tree is extremely biased: each node has only one child?



ExtractMax will cost $O(n)$!

This is similar to the case in the linked list. Therefore it is also important that the tree is balanced.

Interesting Problem: Median Heap

Design a data structure that allows

- Inserting a number in $O(\log n)$ time,
- Finding the median of all inserted numbers in $O(1)$ time.

Hint: use 2 heaps!

** Feel free to post your answer/question in the Telegram group!*

Interesting Problem: “Priority Stack”? *

Design a stack that allows

- Usual push and pop operations in $O(1)$ time,
- Retrieving the minimum element in $O(1)$ time.

** Feel free to post your answer/question in the Telegram group!*

End of File

Thank you very much for your attention :-)