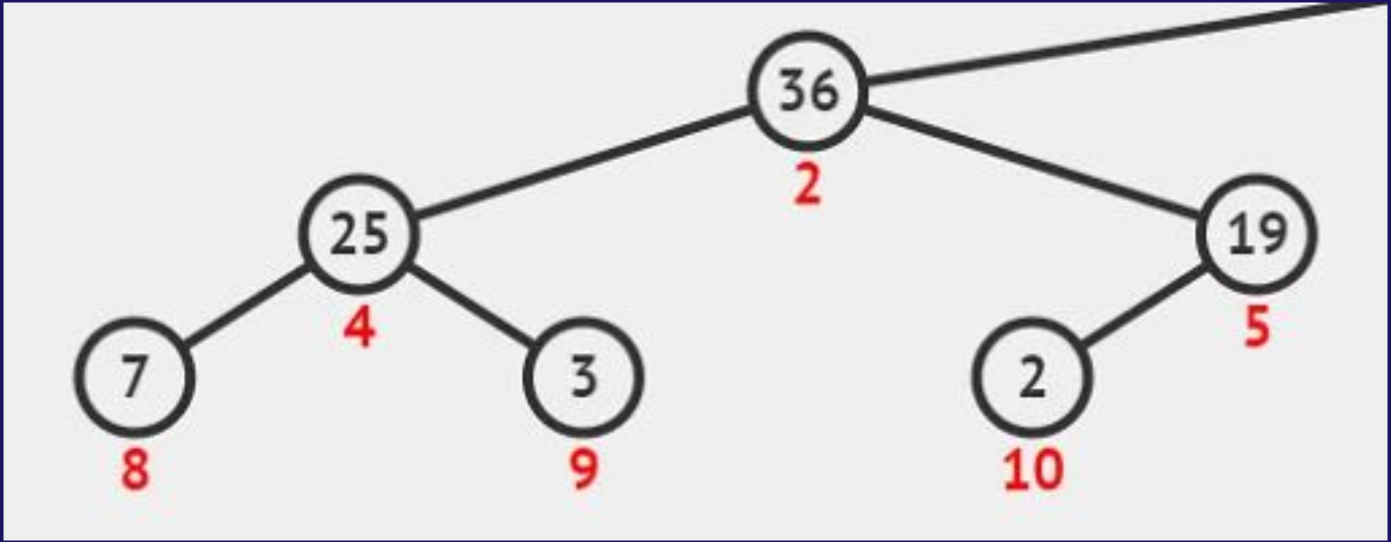
CS2040

Tutorial 5: Heaps and Priority Queues

Nicholas **Russell** Saerang (russellsaerang@u.nus.edu)

Let's start with...

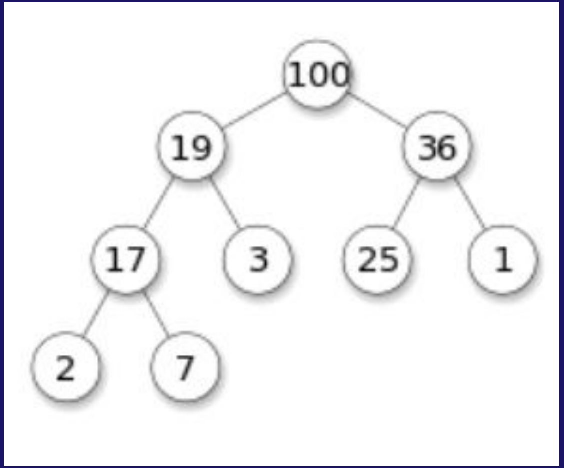


:D

BINARY HEAP

Binary Heap

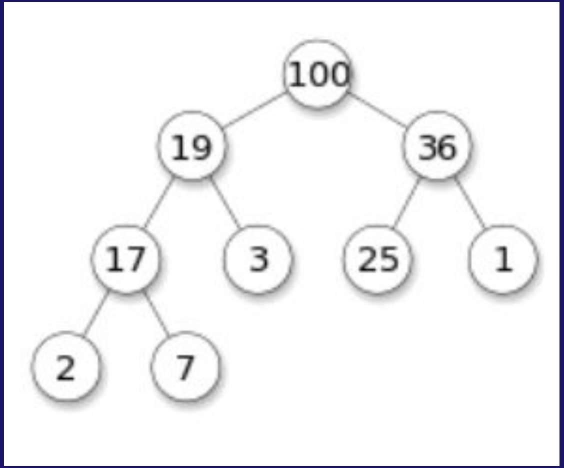
- Could be used to implement MAX/MIN priority queue
- Maintain a set of object with priority
- Items stored in a tree: biggest priority/item at root, smallest ones at leaves



Binary Heap

Property:

- priority[parent] >= priority[child]
- Complete binary tree: every level is full, except possibly the last one. All nodes are as far left as possible. Each node has at most 2 children
- Height of binary heap: O(log n)

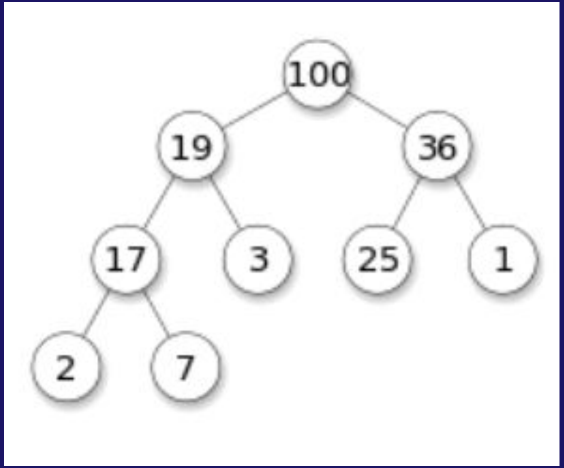


Binary Heap - Insert

Insert(v) to binary heap

- Add new leaf v
- Bubble up / shift up v until reaches correct position: keep swapping v with its parent if priority[parent] < v

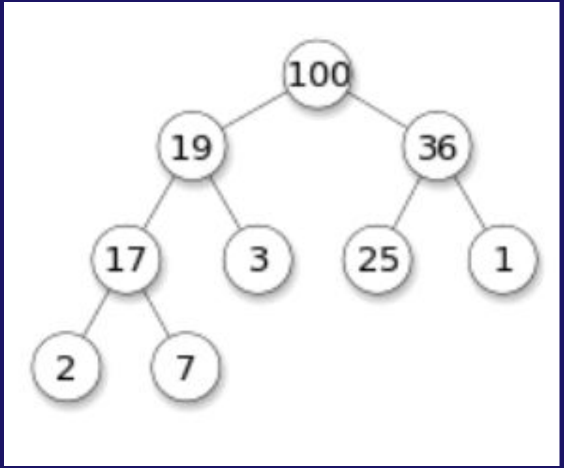
Complexity: O(log n)



Binary Heap - Delete

Delete(v) from binary heap

- Swap v with last element, let's say x
- Delete v from the tree
- After swapping v with x, x is in the wrong order. Since x was initially in the leaf, x needs to be bubbled down/shifted down. Keep swapping x with its child if priority[child] > x

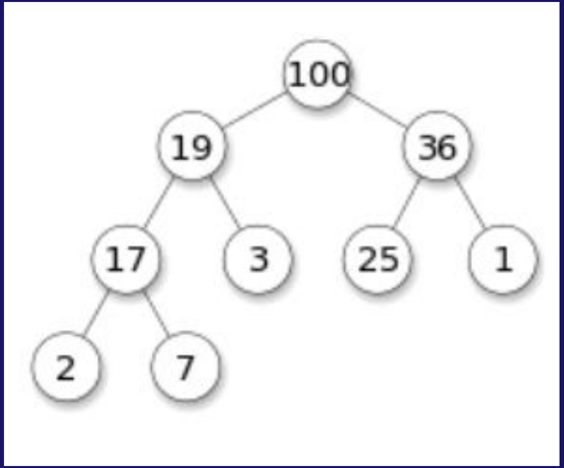


Complexity: O(log n)

Binary Heap - Extract Max/Poll

Basically delete(root).
The steps are the same as the one in the previous slide.

Complexity: O(log n)



Binary Heap in Array

Q: Why not 0-based?

As a **1-based** compact array: `A[1..size(A)]`

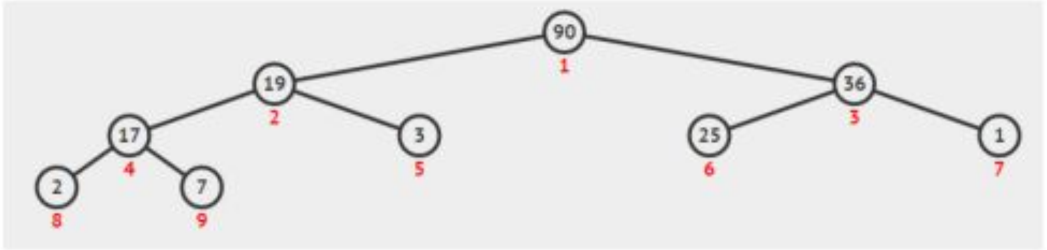
`size(A)`

0	1	2	3	4	5	6	7	8	9	10	11
NIL	90	19	36	17	3	25	1	2	7	-	-

`heapsize ≤ size(A)`

Navigation operations:

- `parent(i) = floor(i/2)`, except for `i = 1 (root)`
- `left(i) = 2*i`, No left child when: `left(i) > heapsize`
- `right(i) = 2*i+1`, No right child when: `right(i) > heapsize`

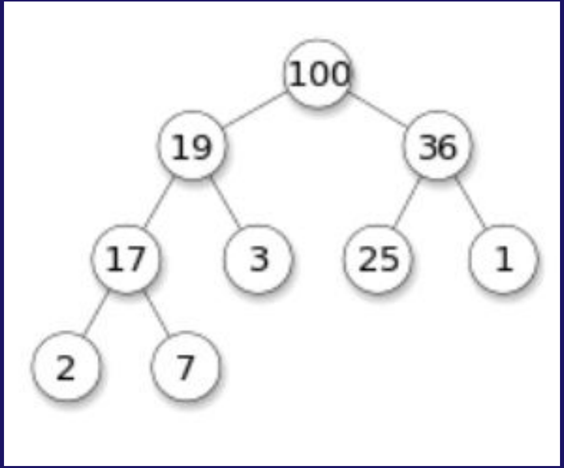


Heapify – O(N log N)

For each element, insert into binary heap: O(log n)

Insert n elements → complexity: O(n log n)

Note that heapify does not sort the array yet!

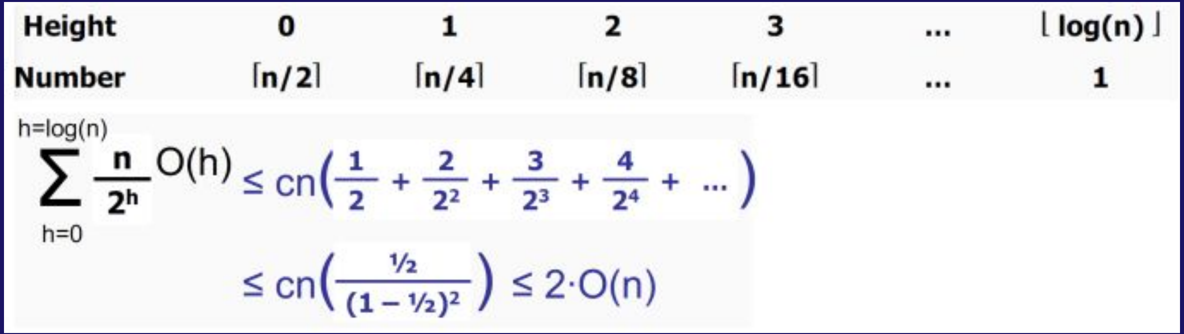


Heapify - O(N)

1. Insert all elements into 1 indexed-array
2. Loop from back (leaf) to front (root), shift down that node

Some observations:

- cost of shiftDown = height
- ceil(n/2) nodes are leaves (height 0)



Height	0	1	2	3	...	⌊log(n)⌋
Number	⌈n/2⌉	⌈n/4⌉	⌈n/8⌉	⌈n/16⌉	...	1

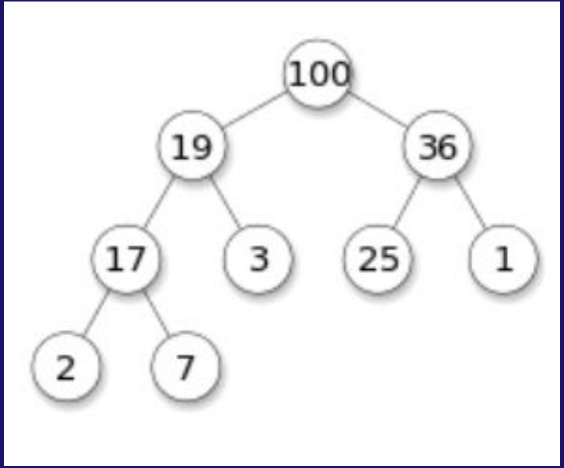
$$\sum_{h=0}^{h=\log(n)} \frac{n}{2^h} O(h) \le cn\left(\frac{1}{2}+\frac{2}{2^2}+\frac{3}{2^3}+\frac{4}{2^4}+\dots\right)$$

$$\le cn\left(\frac{1/2}{(1-1/2)^2}\right) \le 2\cdot O(n)$$

Heap Sort - O(N log N)

- Heapify an array in O(n) time
- Extract element one by one, each time O(log n)

Complexity: O(n log n)



01

TRUE OR FALSE?

Question 1a

The smallest element in a min heap is always the root.

Answer:
True. Applying min heap property, all descendants of the root must be greater than the root.

Question 1b

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.

Answer:
True. Suppose second largest element is not the child of the root. Since it cannot be the root, it must be a descendant of the children of the root. However, this violates the max heap property. (Proof by contradiction)

Question 1c

When a heap is stored in an array, finding the parent of a node takes at least O(log n) time.

Answer:

False. Only requires O(1) time by using the index. For a node at index i, parent is at index $\left\lfloor \frac{i}{2} \right\rfloor$ (using 1-indexed array).

Question 1d

Every node in the heap except the leaves has exactly 2 children.

Answer:
False. A simple example is a heap with 4 elements. The left child of the root is not a leaf, but has only one child.

Question 1e

We can obtain a sorted sequence of the heap elements in O(n) time.

Answer:
False. Heap sort takes O(n log n). (Also, note a common misconception that heapify sorts the array. This is **NOT TRUE**.)

02

GREATER THAN X

Greater Than X

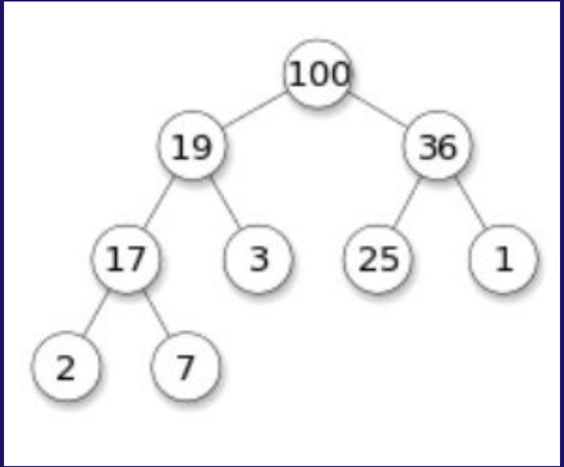
Give an algorithm to find all vertices bigger than some value x in a max heap that runs in O(k) time where k is the number of vertices in the output.

(This is different from most algorithms you have encountered which are dependent on the size of the input, instead of the size of the output. We sometimes call this output sensitive algorithms)

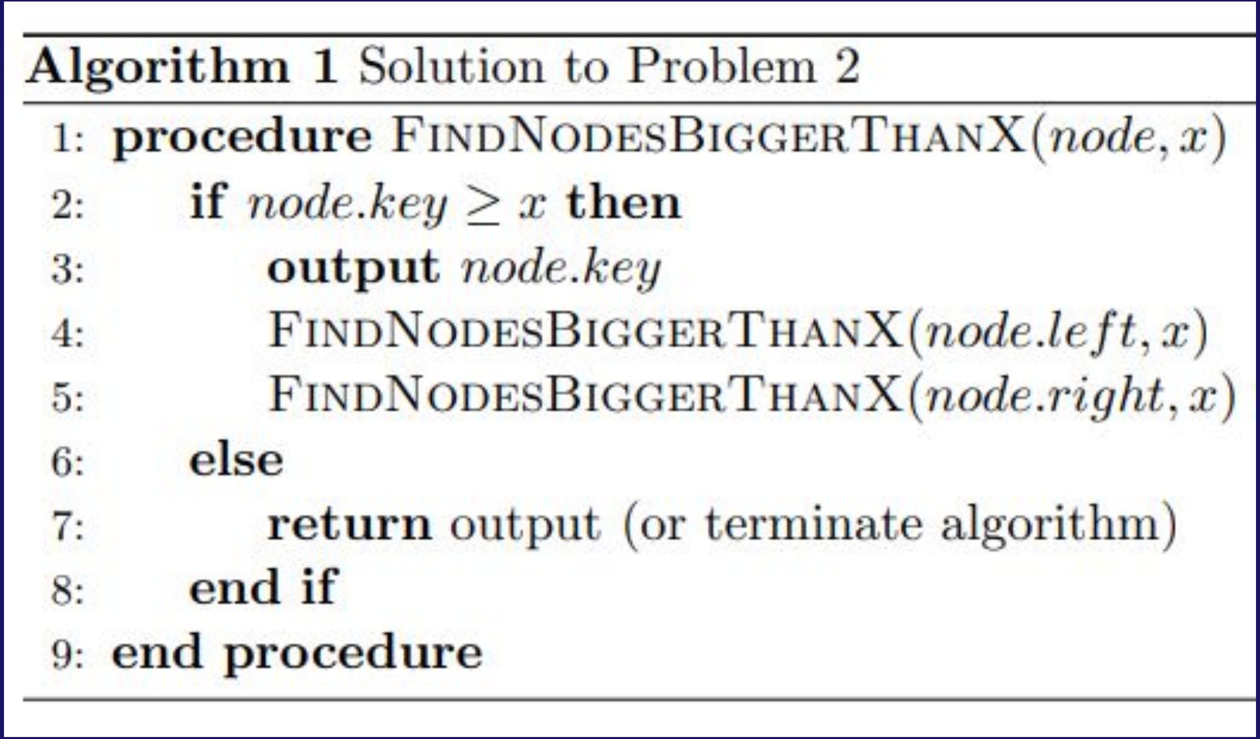
Greater Than X

Answer:

1. Perform a pre-order traversal of the max heap starting from the root.
2. At each node, check if node key is > x. If yes, output node and continue traversal. Otherwise, terminate traversal on subtree rooted at current node, return to parent and continue traversal.



Greater Than X



Algorithm 1 Solution to Problem 2

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

Greater Than X

Answer:
Since the traversal terminates when it encounters that a node's key $\leq$ x. In the worst case, it encounters 2k number of such nodes. That is, each of the left and right child of a valid node (node with key > x) are invalid. It will not process any invalid node other than those 2k nodes.

It will process all k valid nodes, since there cannot be any valid nodes in the subtrees not traversed (due to the heap property). Thus, the traversal encounters $O(k + 2k) = O(k)$ number of nodes in order to output the k valid nodes.

03

UPDATING A HEAP

Updating a Heap

Give an algorithm for the `update(int oldKey, int newKey)` operation, which updates the value of oldKey in a binary heap (max or min) with newKey in the binary heap in O(log n) time, which does not change the time complexity of the other operations. You are required to modify the other operations in the binary heap if needed, including any additional data structures used. You may assume all values in the heap will be unique. (Additional: What if the keys are not unique?)

Updating a Heap

Answer:

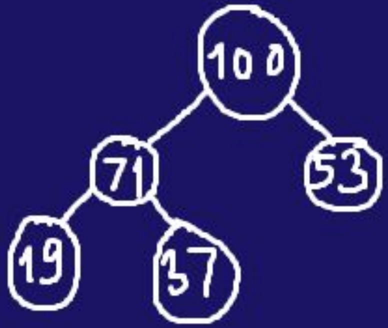
Finding where the oldKey is could take O(n) if we search through the entire heap. To support O(log n) update we can use hash table h that saves the key and value pair (k, i) where k is the key in the heap and i is the index of this key in the array (use array implementation)

Updating a Heap

Answer:

1. We start by filling h with the corresponding key-value pairs by going through the entire input array.
2. Now, whenever we call swap that is in shiftUp or shiftDown, we need to update both the values in the array, and the key-value pairs in h. Since heap creation, insert and remove are all dependent on shiftUp and shiftDown, there is no further modication required.
3. For update, because we have the hash table h, we can search for the index of oldKey in O(1) time, which is it's location in the heap. We update this key-value pair: (oldKey, i) → (newKey, i). Now we perform both shiftUp and shiftDown on newKey.
4. Since the shift operations are still in O(log n) time, all the operations, including update, are in O(log n) time.

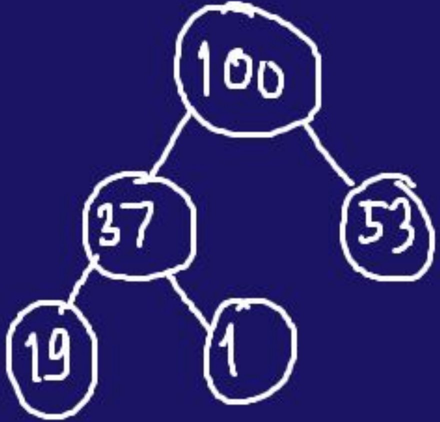
Updating a Heap



h

1	100
2	71
3	53
4	19
5	37

update(71, 1)



h

1 ↔ 100	
~~2 ↔ 71~~	5 ↔ 1
3 ↔ 53	
4 ↔ 19	
2 ~~5~~ ↔ 37	

04

SORTED? ALMOST

Sorted? Almost

An array A_k of n unique floating-point values of no fixed precision is partially sorted when each value differs from its correct position in the sorted array by no more than k positions, where k is a positive integer. For example, an array A_2 could be [1; 4; 3; 2; 6; 5; 7], where the sorted output should be [1; 2; 3; 4; 5; 6; 7].

Sorted? Almost

Give an algorithm to sort a partially sorted array in O(n) time, where k = 1.

Answer:

- Iterate from the beginning of the array A_1 to the end.
- At each step i, check whether the current element at A[i] > A[i + 1]. If so, swap A[i] and A[i + 1], else increment i.

Essentially, this is insertion sort or ‘single-pass’ bubble sort.

Sorted? Almost

Give an algorithm to sort a partially sorted array in O(n log k) time, where k can be any positive integer smaller than n.

Answer:

The key observation here is that since every element is at most k away from its actual sorted position. What this means is that the element that should be placed in A[i] (the i th index of A[1…n]), is the smallest element in A[i…(i + k)].

Sorted? Almost

Answer:

1. We use a min-heap, of size k+1 that keeps all elements in the sub-array A[i…(i+k+1)] (initialized with the first k + 1 elements).
2. Initially, i starts off at index 1, and we increase the index i by 1 as we iterate. Every time we do, we first extract the minimum value from the min-heap, since we know that this must be the value that should be at the i-th index.
3. Then, we add the item at index i+k+2. Intuitively, you can view this as sliding a heap over a window of size k+1.
4. An edge case is at the last k+1 elements. Since there will be no more elements to add after that, we simply repeatedly extract the minimum from the heap without adding anymore values into the min-heap.

05

WHICH NUMBER TO PICK?

Which Number To Pick?

Abridged problem description:

Operation: From any of the top k integers in the stack, you may remove one of them and add it to the total value (initially 0).

Given a stack of n integers, you can do the above operation repeatedly. Find an algorithm to calculate the maximum total value. State its complexity!

For example, for k = 2, where integers in the stack are [2, -10, 2, -6, 5], the output of your algorithm should be 4. Another example for k = 5, where integers in the stack are [-1, -1, -1, -1, -1, 10], the output of your algorithm should be 9. (top of stack is the first element of the array)

Which Number To Pick?

Answer:

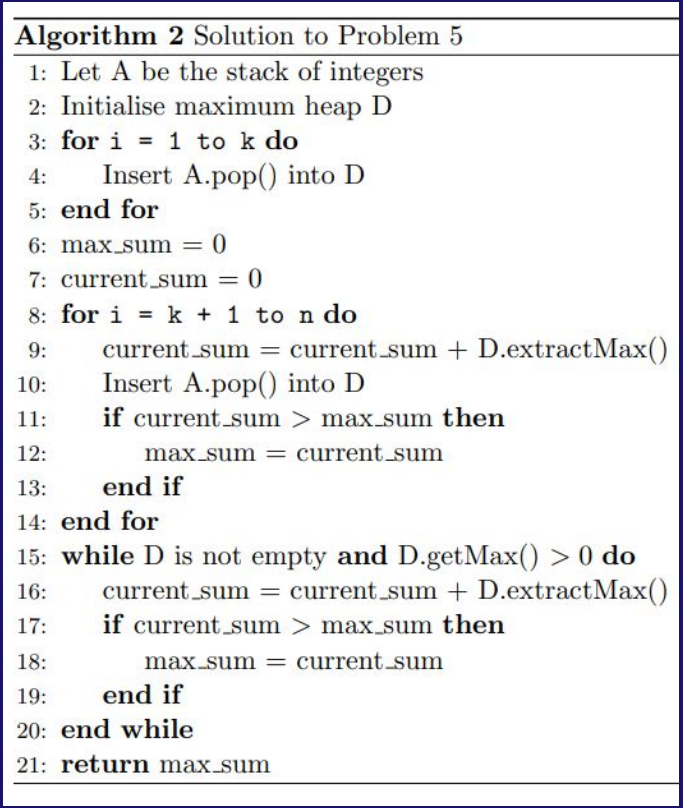
Since we can only look at the top k elements in the stack, and we want to maximize the total value of elements chosen, we make use of a maximum heap of size k. We need to maintain two values, the current/running sum, and the historical highest sum achieved.

Which Number To Pick?

Answer:

1. We will remove the largest element in the heap, followed by inserting the next element in the stack. This simulates the top k elements in the stack at any one point in time.
2. When removing the largest element, we will also add that value to the current sum and update the historical highest sum when necessary. Note that since it is possible that all the values in the heap of size k are negative, the current sum can be reduced. This is why we need to keep track of the historical highest sum.
3. We repeat this step for n - k times until the last element is added to the heap.
4. Note that since there are k elements remaining that could have positive values, we also need to check the remaining elements in the heap and update the sums as necessary.

Which Number To Pick?

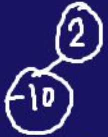


Algorithm 2 Solution to Problem 5

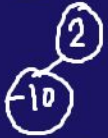
```
Let A be the stack of integers
Initialise maximum heap D
for i = 1 to k do
    Insert A.pop() into D
end for
max_sum = 0
current_sum = 0
for i = k + 1 to n do
    current_sum = current_sum + D.extractMax()
    Insert A.pop() into D
    if current_sum > max_sum then
        max_sum = current_sum
    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
```

Which Number To Pick?

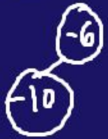
$[2 \ \ -10 \ \ \underline{2} \ \ -6 \ \ 5]$ $k=2$



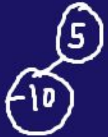
take 2, insert $\underline{2}$, current_sum = 2, max_sum = 2



take $\underline{2}$, insert -6, current_sum = 4, max_sum = 4



take -6, insert 5, current_sum = -2, max_sum = 4



take 5, current_sum = 3, max_sum = 4

take -10, current_sum = -7, $\boxed{\text{max\_sum} = 4}$

Which Number To Pick?

Answer:

Each of the n values in the stack will at most be inserted once and removed once from the heap of size k. Since each operation takes O(log k), the total time complexity of O(n log k).

The algorithm requires O(k) auxiliary space for the maximum heap of size k.

THE END!