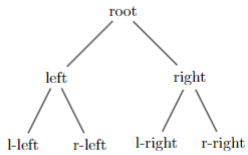
Problem Set 5

Problem 1 (a) Prove by mathematical induction that a complete binary tree with height, h, contains precisely $2^{h+1} - 1$ nodes.

We know that a binary tree is a tree where each node has at most two children, with the topmost node called the root. A complete binary tree is a tree without any "holes" - all nodes from left to right, top to bottom are present in the tree. An incomplete tree is a tree which is conversely, not filled all the way from left to right and top to bottom.

Below is a complete (perfect) binary tree :



The height of a complete binary tree is the distance from the root node to the leaf nodes. It is also equal to the number of levels for a complete binary tree.

To prove : Given a complete (perfect) binary tree with height h, it has $2^{h+1} - 1$ nodes.

Now suppose we have a complete binary tree with height h. We will assume the statement $2^{h+1} - 1$ is true. Let's check our base case for $h = 0$ and $h = 1$ first.

Case : $h = 0$ and $h = 1$

If $h = 0$, the tree will have a single root node with no children.

Hence, putting $h = 0$ into the equation:

$$2^{h+1} - 1$$

$$2^0 - 1 = 2 - 1 = 1$$

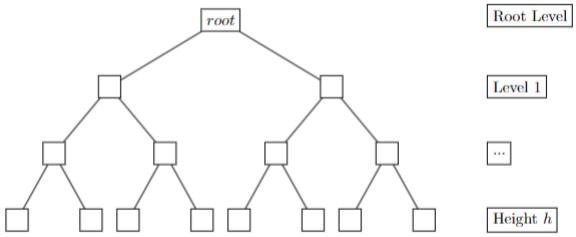
We can depict it as the following :

If $h = 1$, we gain two 2 child nodes on top of 1 root node, hence the number of nodes is 3. We can depict this mathematically

We can represent it as :

Case : height h

We assume that the statement $2^{h+1} - 1 = n$ where n is the number of nodes is true for height h.

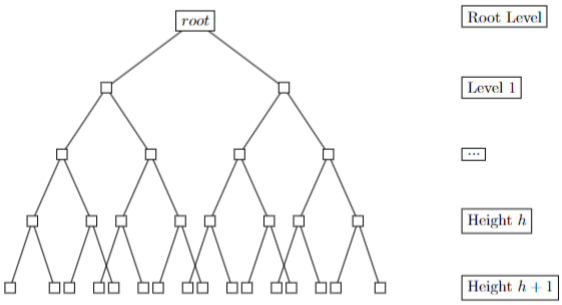


Case : height $h + 1$

Now, to prove the assumed statement by induction, we must prove it for height = $h + 1$. Hence we must prove that the total number of nodes in a perfect binary tree with $h + 1$ height is defined by :

$$2^{h+1+1} - 1 \Rightarrow 2^{h+2} - 1$$

A binary tree with $h+1$ height is a binary tree with a root that has two subtrees of height h.



Note : Apologies for the $LaTeX$ render error in the tree leaves.

Hence, we can denote the number of nodes mathematically as :

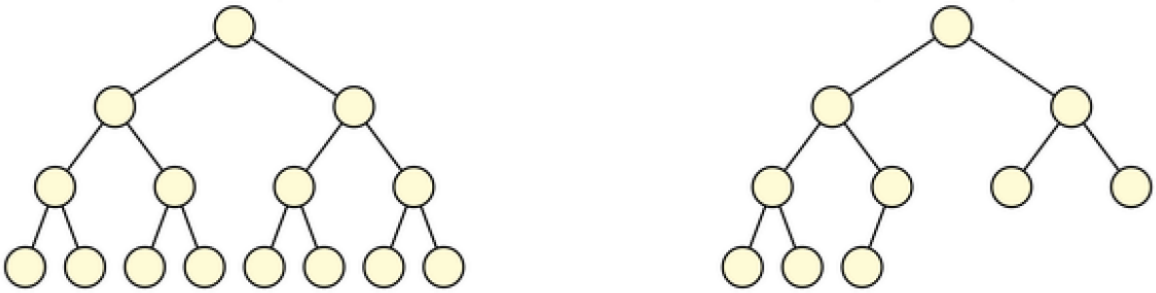
$$\begin{aligned} & 1 + 2^{(h+1)} - 1 + 2^{(h+1)} - 1 \\ & \Rightarrow 2 * (2^{(h+1)}) - 1 \\ & = 2^{(h+1)+1} - 1 \\ & \Rightarrow 2^{(h+2)} - 1 \end{aligned}$$

Hence, we have proved that the number of nodes in a tree with $h+1$ height must be $2^{(h+2)} - 1$.

Now since the equation stands for $h = 0$ and $h = 1$, and it stands for height h and $h+1$; by the process of mathematical induction we have proved that the statement must be true $\forall h \geq 0$.

1 (b) How many leaves does an almost complete binary tree of height, h, have? Give the smallest and largest possible values, and explain. Note, by definition every complete binary tree is almost complete tree.

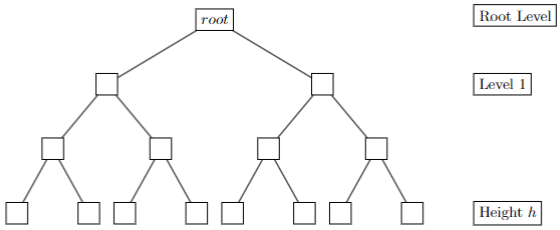
Let's explore the definition of both an almost complete and complete (perfect) binary trees first. A complete (perfect) binary tree is a tree without any "holes" - all nodes from left to right, top to bottom are present in the tree. An almost complete tree is a tree which has all the elements filled from top to bottom and left to right, except the last level which may or may not be completely filled.



[Image Source : Problem Set 5]

Hence, a perfect binary tree can be thought of as a special case of an almost complete binary tree where the last level is completely filled.

Hence, such a perfect binary tree of height h would be the maximum limit on how many leaves a tree can have.



As in, the maximum possible value of the number of leaves an almost complete tree of height h is equal to the number of leaves in a perfect binary tree of height h.

In answer **1(a)**, we proved that the number of nodes in a perfect binary tree of height h is :

$$n = 2^{h+1} - 1$$

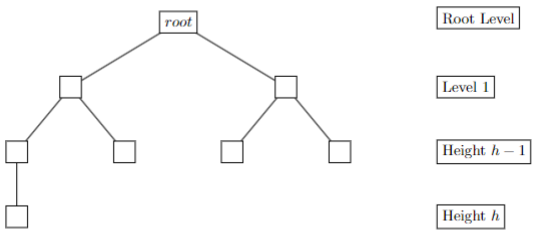
Now, the number of leaves in any perfect binary tree with height h must be equal to the number of nodes in a perfect binary tree of height h minus the number of nodes in a perfect binary tree of height $h - 1$:

$$l = 2^{h+1} - 2^{h+1-1} + 1 - 1$$
$$l = 2 * 2^h - 2^h$$
$$l = 2^h$$

Here, number of leaves = l.

Hence, the number of nodes in a perfect binary tree are 2^h, **and so the maximum number of leaves in an almost complete binary tree must also be 2^h.**

Now, we must talk about the minimum number of leaves that an almost complete binary tree must have. For any binary tree to be almost complete while also retaining all the properties of a binary tree, it must be populated by at least 1 leaf node in the last level, or at height h. We can think of this as slowly removing all the leaves from a perfect tree of height h till 1 leaf is left with height h. As we remove the leaves, the number of leaves at height h decreases, but the parents at the $h-1$ level also start becoming leaves. When a single leaf is left, all the nodes at $h-1$ become leaves except the leftmost node, which has a single child leaf at height h.



Hence, the total number of minimum possible leaves for an almost complete binary tree becomes the total number of leaves of a perfect binary tree of a height $h-1$, minus 1 node and plus 1 node in the h level:

$$2^{h-1} + 1 - 1 \Rightarrow 2^{(h-1)}$$

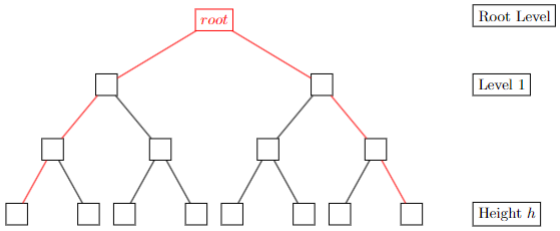
Hence, the minimum number of nodes in an almost complete binary tree of height h is 2^{h-1}.

1 (c) The diameter of a tree or a graph is the maximum distance (length of the longest path) between nodes. What's the diameter of an almost complete binary tree of height, h. Give the smallest and largest possible values and explain.

The diameter of a binary tree is the maximum distance between two nodes. To find the minimum and maximum diameter of an almost complete binary tree, we must understand when the minimum and maximum condition is achieved.

An almost complete binary tree has the most number of nodes when it is a perfect binary tree. Since it has the most number of nodes, this is when there will be the most separation between the leftmost and rightmost leaf. For a perfect binary tree, the diameter will be the distance or the number of edges between the leftmost and the rightmost leaf at the last level.

Suppose we have a perfect binary tree of height h, then the diameter will look something like this:

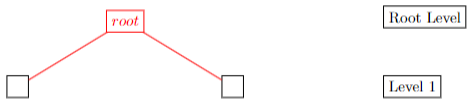


Now, as we can see in the diagram, for a perfect tree and since the tree is filled from left to right, the diameter should connect the leftmost and rightmost leaf nodes. Since the distance for both the leftmost and the rightmost leaf in a perfect binary tree is equal to the height h, we can mathematically represent diameter d as :

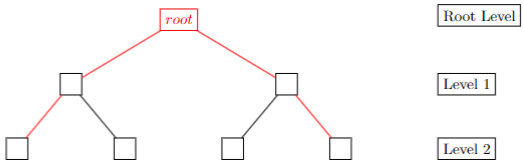
$$d = h + h \Rightarrow 2h$$

Hence the maximum diameter of an almost complete binary tree is $2h$, where h is the height.

We can see how this applies for $h = 1, d = 2 * 1 = 1$



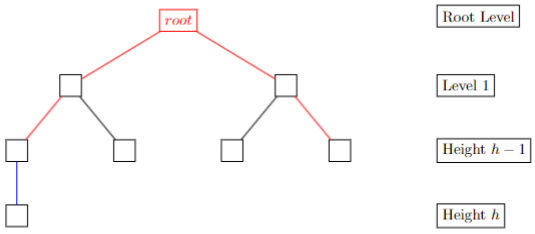
for $h = 2, d = 2 * 2 = 4$



It is important to note that for each parent node, the distance to both its children will be the same. Hence, both the rightmost and second rightmost elements at height h will satisfy the condition for diameter. Similarly both the leftmost and second leftmost element will satisfy the condition for diameter, since both of them are equidistant from their parent. The leaf nodes that fall in the same subtree will always have the same diameter.

Now, let's move on to the minimum case. In the minimum case, we must minimize the distance between the furthest nodes. This can be done by making it so there is only one node at the height of h such that it is the furthest away from any leaf nodes on the same level or any other level.

The diameter of such a node can be thought of as the diameter of a tree with height $h-1$ + one edge which connects the singular leftmost element on the h^{th} level.



Here we can see the diameter of a tree with height $h-1$ in red and the extra edge in blue.

This can be mathematically represented as :

$$2(h-1)+1 \Rightarrow 2h-2+1 = 2h-1$$

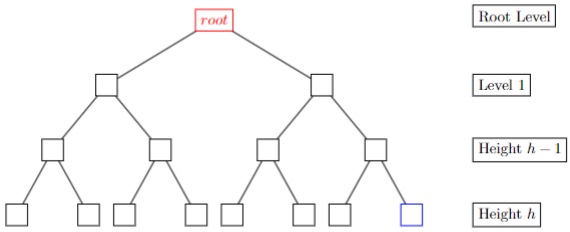
Hence the minimum distance must be $2h-1$ for an almost complete binary tree.

1 (d) Suppose that we "reroot" a complete binary tree of height, h, by designating one of the erstwhile leaves as the root. What is the height of the rerooted tree?

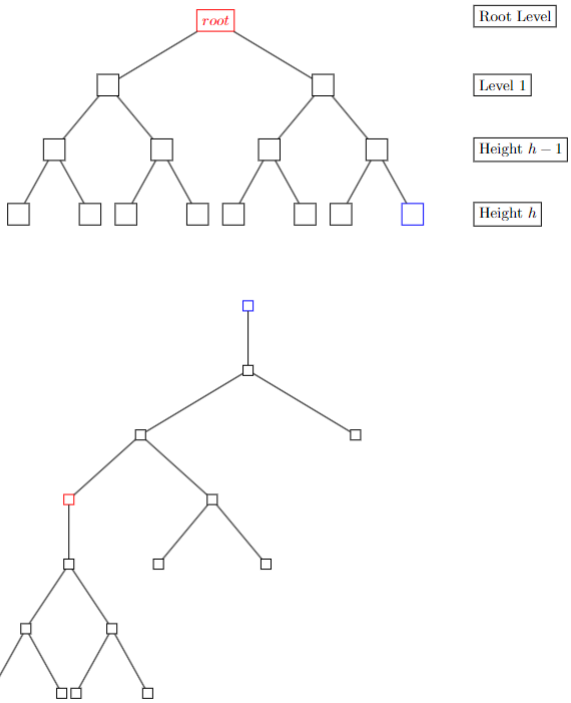
Re-rooting a binary tree tree is the process of making the leaf a root, while preserving the relationship between the nodes.

To understand how the height of the tree changes, we must understand how a tree is re-rooted first.

Suppose we have a tree of height h, which we can represent as :



Suppose we have a tree of height h which we want to re-root. This can be done by making the parent of the leaf the left node and the sibling the right node of the parent. This will be done for every subtree successively till the original root is reached. We can visualize this as *picking up* the leaf node and the dangling the binary tree as well.

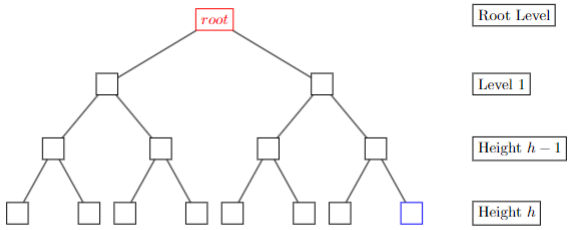


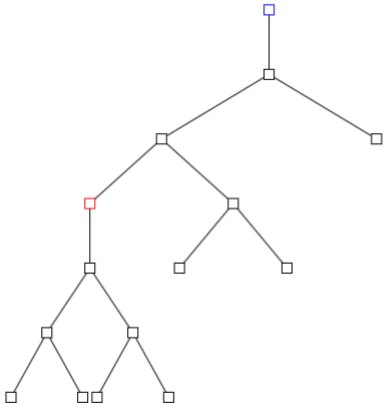
Now, considering that the height of the original leaf was h, the new height will be the distance from the new leaf to the new root. The height of the tree is the distance of the root to the most distant leaf. This can be divided into the distance between the new root and the old root + the distance from the old root to the leaf. Since both the distance of the new root (blue) and the old root (red) and the distance from old root (red) and leaves is h, the total height becomes $h + h = 2h$.

Hence the total height of the tree after re-rooting is $2h$.

1 (e) What is the diameter of a complete binary tree rerooted at one of its leaves?

A complete binary tree, upon re-rooting can be represented like this :

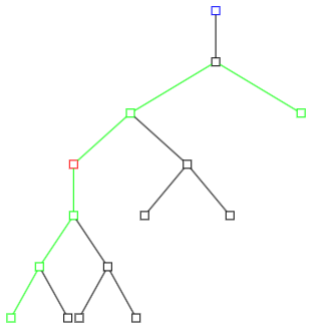




Where the original root is in red and the new root is in blue. Re-rooting is done when the leaf of the binary tree, in this case the rightmost leaf, is made the root.

To calculate the diameter, we must look at the diameter of a complete binary tree first. We know the expression for the diameter of a complete binary tree from our analysis in question **1(c)**.

Reiterating, the diameter of a complete binary tree is always equal to twice its height or $2h$ for a complete binary tree of height h. In the case of a re-rooted tree, the diameter will always pass through the previous root (red) since it is essential in connecting the leftmost node to the rightmost node.



Accordingly, the diameter can be seen here highlighted in green.

Hence again, the diameter can be divided in to two parts: the distance from the leftmost leaf to the original root, and the distance from the rightmost leaf to the root. Since the relationships between the nodes are undisturbed, as long as we are dealing with a complete(perfect) tree, the distance of the leaves from the original root will always be the same. Hence, both of these distances will be h, and combining them will give us $2h$, which is the diameter of the re-rooted tree.

2(a) The height of a heap is defined to the number of edges on the longest downward path from the root node to a leaf node. Thus, in the example above, the height of the heap is h = 3. If a (binary) heap has height h = 6, determine the minimum number and maximum number of elements that can be in this heap. Clearly justify your answer.

When a heap has the maximum number of elements, it forms a perfect binary tree. In a perfect binary tree, the number of leaves are equal to 2^d, where d is the depth level of the tree. The number of elements in a tree must be equal to the number of leaves of the first level + number of leaves of second level and so on till the number of leaves at the last level + the root, which we can mathematically depict as:

$$2^0 + 2^1 + 2^2 + \cdots + 2^d$$

As we can see, the first d terms form a geometric progression, which we can solve as :

$$2^0 + 2^1 + 2^2 + \cdots + 2^d = 2^{d+1} - 1$$

when $d = h$, this becomes:

$$2^{h+1} - 1$$

Hence, when $h = 6$, we have :

$$\begin{aligned} & 2^{6+1} - 1 \\ & = 2^7 - 1 \\ & = 128 \end{aligned}$$

Hence, a heap with height 6 has a maximum of 128 elements.

A heap with height 6 will have minimum number of elements when it has only 1 leaf in depth level 6.

Hence, the number of leaves will be equal to the leaf at height 6 + the number of elements of a tree of height 5:

$$= 2^{5+1} - 1 + 1$$
$$= 64$$

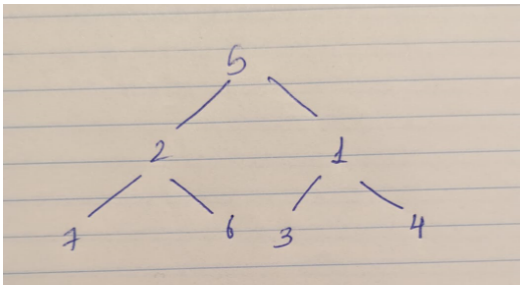
Hence, the minimum number of leaves a heap of height 6 can have is 64.

2 (b) Consider an unsorted array of n elements. Recall that the Heapsort Algorithm consists of two parts: first we run BUILD-MAX-HEAP to convert our input array into a max heap, and then we run MAX-HEAPIFY n times to generate the n elements of our sorted array. Demonstrate the Heapsort Algorithm on the input array [5, 2, 1, 7, 6, 3, 4]. Clearly show your steps.

Suppose we have the input array:

$$[5, 2, 1, 7, 6, 3, 4]$$

This can be written in a binary tree form:



First, we apply $BUILD - MAX - HEAP$.

For this, we compare the rightmost leaf of the tree with its sibling.

$$4 > 3$$

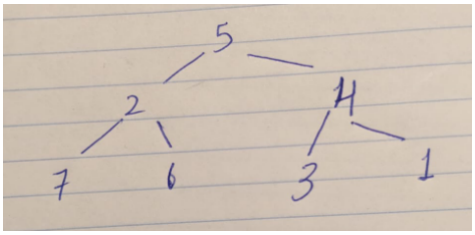
Since 4 is greater, we will use this to compare against the parent, 1. We can determine the parent using the formula $\frac{n-1}{2}$, where n is the index of the element. Here $n = 6$, so the parent is 1.

Now we compare 4 with 1:

$$4 > 1$$

Hence, we swap since we must satisfy the max heap condition.

The new tree will look like this:



And the new array will look like this :

$$[5, 2, 4, 7, 6, 3, 1]$$

Next, we move to the rightmost leaf in the left subtree. We will then compare it with the sibling element in that subtree. The sibling element is 7.

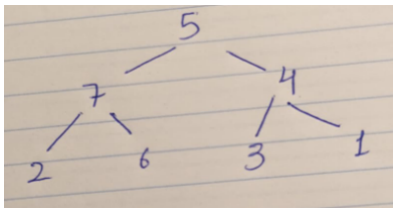
$$6 < 7$$

Hence, the sibling is greater. Now, we will use the sibling 7 to compare against the parent. The index of the sibling is 3, so according to the formula used earlier, the index of the parent will be 1, which is the element 2.

$$7 > 2$$

Since 7 is greater than 2, we will swap 2 and 7.

The tree will look like this:



The array will look like this after the swap:

$$[5, 7, 4, 2, 6, 3, 1]$$

Now, we move to the parents. We will compare the parents 7 and 4 to see which is greater.

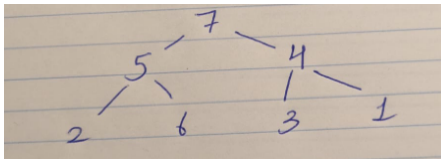
$$7 > 4$$

Hence, we will choose 7 to compare with root, which is index 0. The index 0 is the element 5.

$$5 < 7$$

Since 5 is less than 7, we will swap the root with the element 7 at index 1.

The resultant tree will look like this:



$$[7, 5, 4, 2, 6, 3, 1]$$

Next, we will check the left subtree if the newly substituted root and parent still satisfy the max heap condition. Hence, we will start by comparing the rightmost element of the left subtree with its sibling.

The rightmost element is 6, and its sibling is 2.

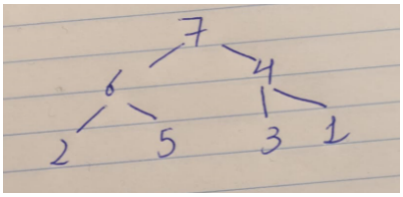
$$6 > 2$$

Now, we will compare element 6 with its parent, which is equal to 5 (index 1) according to the formula.

$$6 > 5$$

Since this does not satisfy max heap condition, we will swap 5 and 6.

The new subtree looks like this:



The new array looks like this:

$$[7, 6, 4, 2, 5, 3, 1]$$

Hence, the heap satisfies the max-heap condition.

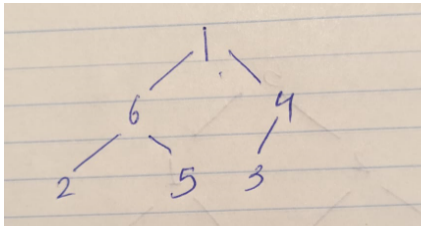
Now, we extract the element 7, which is guaranteed to be the maximum element in the entire array. This is done by swapping the root with the rightmost leaf and then removing the original root from the heap.

The array looks like this:

$$[1, 6, 4, 2, 5, 3|7]$$

Where $|$ is a divider.

The heap will look like this:



Next, we max-heapify our heap again, but this time we go in a top down fashion.

First we compare the child elements of the root, 6 and 4:

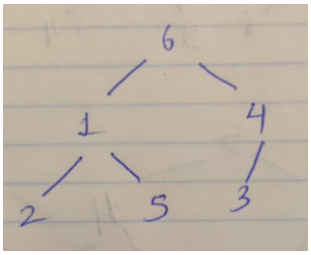
$$6 > 4$$

Hence, we choose 6 to compare with the root.

$$1 < 6$$

Since 6 is greater than 1, we swap 1 and 6 to satisfy max heap condition.

The heap looks like this:



The array looks like this:

$$[6, 1, 4, 2, 5, 3|7]$$

Next, we will check for heap condition on the left subtree. Here we pick the children of parent 1, which are 2 and 5 and compare them:

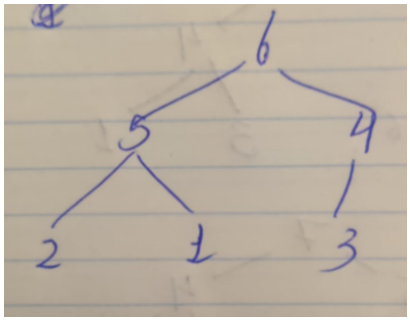
$$2 < 5$$

Hence, we will now compare 5 with the parent, 1

$$1 < 5$$

To satisfy max heap condition, we swap 1 and 5.

The heap looks like this:



The array looks like this:

$$[6, 5, 4, 2, 1, 3|7]$$

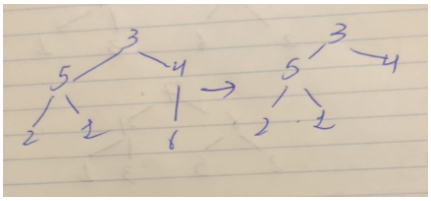
Now, we will check the right subarray. Since 3 does not have a sibling, we compare child with parent:

$$4 > 3$$

So the heap condition is still obeyed.

This means the max heap is again created. Now, the root is again the biggest element in the array. Hence, we will perform extraction and swap it with the leaf, and then remove it from the heap.

The heap will look like this:



The array will look like this:

$$[3, 5, 4, 2, 1|6, 7]$$

Next, we will max heapify our heap again.

For this, we compare the child of the root element with each other:

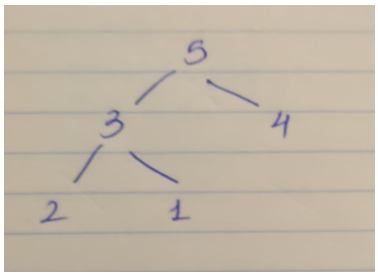
$$5 > 4$$

Hence, we will now compare 5 with the root element:

$$5 > 3$$

To retain max heap condition we swap 5 with 3.

The heap will look like this:



The array will look like this:

$$[5, 3, 4, 2, 1|6, 7]$$

Next, we will check the left subarray. We will compare the children of the left parent, which are 2 and 1:

$$2 > 1$$

Since 2 is greater, we will compare 2 with 3.

$$2 < 3$$

Hence, max heap condition is satisfied.

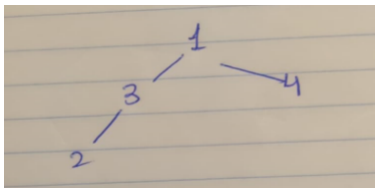
NOTE: Here we reach iteration 2 for part 2(c). The array is :

$$[5, 3, 4, 2, 1|6, 7]$$

Therefore 5 is the largest element in the heap, which is also at the root.

We now extract 5 by swapping it with the furthest and rightmost leaf, 1.

The heap will look like this:



The array will look like this:

$$[1, 3, 4, 2, |5, 6, 7]$$

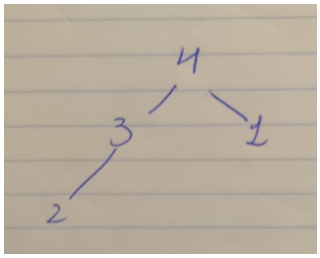
Next, we will compare the children of the root:

$$3 < 4$$

Since 4 is greater, we will compare 4 with the root, 1.

$$1 < 4$$

Since max heap condition is not satisfied, we swap 4 and 1:



The array will look like this:

$$[4, 3, 1, 2, |5, 6, 7]$$

Next, we will compare the children in the left subtree. Since there is only 2 in the subtree, we will compare it with 3 directly:

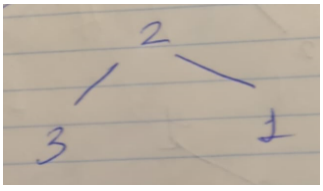
$$2 < 3$$

Hence, max heap is satisfied.

Therefore, 4 at the root is the biggest element in the heap.

We will extract 4 by swapping it with the furthest leaf, 2.

The heap will look like this:



The array will look like this:

$$[2, 3, 1|4, 5, 6, 7]$$

We will max-heapify again. Now, we compare the children of the root 2, that is 3 and 1:

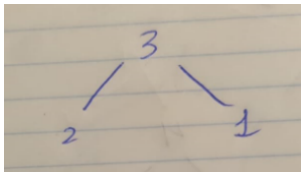
$$3 > 1$$

Since 3>1, we will compare the root with 3:

$$2 < 3$$

Since max heap condition is not satisfied, we swap 3 and 2.

The heap will look like this:

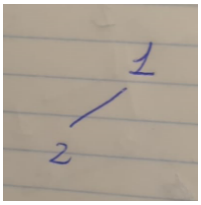


The array will look like this:

$$[3, 2, 1|4, 5, 6, 7]$$

Now, again, since we already compared the parents, the max heap is built again.

We now extract 3, leaving us with the heap:



And the array:

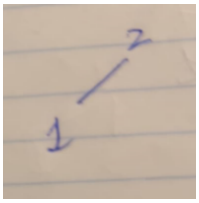
$$[1, 2|3, 4, 5, 6, 7]$$

Next we compare 1 and 2.

$$1 < 2$$

Since max heap is not satisfied, we will swap 2 and 1.

The max heap looks like this:



Next, since the max heap condition is satisfied, 2 is the largest element in the heap since it is at the root. We extract 2 by swapping 2 and 1 again and then remove 2 from the heap.

$$[1|2,3,4,5,6,7]$$

We are left with a singular element 1, which we extract again since it is the largest element in the heap automatically:

$$[1,2,3,4,5,6,7]$$

Hence, the array is sorted.

2(c) : In part (b) above, you should have noticed that after the i = 2 iteration of MAX-HEAPIFY, your heap was [5, 3, 4, 2, 1, 6, 7]. Notice how the first n − i elements form a max heap, and the last i elements are sorted and are the i largest elements of the array. Show that this property holds for any max heap with n elements. Specifically, prove that for all $1 \leq i \leq n$, after the i^{th} iteration of MAX-HEAPIFY, the first $n - i$ elements form a max heap, and the last i elements are sorted and are the i largest elements of the array.

Solution:

We noticed in part (b) of the second question that at the end of i=2, the heap was :

$$[5,3,4,2,1,6,7]$$

Which coincides with the question.

Here, the last 2 elements, 6 and 7 are sorted elements. This happens because we perform max-heapification before we extract the next largest element from the root.

We will attempt to prove this by using a loop invariant, which would prove the correctness of the algorithm and in turn, prove that at the end of any iteration, the subarray $i - n$ is always sorted for all arrays and the subarray 0 to $n - i$, is always a max heap.

Proof:

Loop Invariant: At the end of each iteration, the subarray $A[0 : ni]$ is always a max heap, and the subarray $A[n - i : n]$, where n is the number of elements, is always sorted.

Initialization: Before the start of the loop, since BUILD-MAX-HEAP is run, we have a max heap for the subarray $A[0 : ni]$. Since i = 0, the subarray $A[n : n]$ is empty. Hence, the invariant holds.

Maintenance: Since the root is the highest element at the start of the first iteration, we extract it. Hence, the largest element is at the end of the subarray $A[n - i : n]$ We then run MAX-HEAPIFY and at the end of the new iteration, we have a new max heap for $A[0 : ni]$. Then, we increment i for the next iteration. Hence, the loop invariant holds well.

Termination: Now, after the loop has ended, i = n, the subarray $A[0 : 0]$ is empty hence it is a max heap. Now, since the element was extracted, the subarray $A[0 : n]$ is sorted, hence the entire array is sorted. Hence the loop invariant holds.

Therefore, we have proved that the loop invariant holds through the algorithm. This means that at the end of each loop, the last i elements are sorted, and the first $n - i$ elements are a max heap.

2(d) Let P be a permutation of the first 7 positive integers. Sometimes this permutation is a max heap; examples include [7, 6, 5, 4, 3, 2, 1], [7, 6, 4, 2, 5, 1, 3], [7, 5, 6, 2, 4, 3, 1], and [7, 3, 6, 2, 1, 4, 5].

If P is a randomly-chosen permutation of [1, 2, 3, 4, 5, 6, 7], determine the probability that it is a max heap. Clearly and carefully justify your answer.

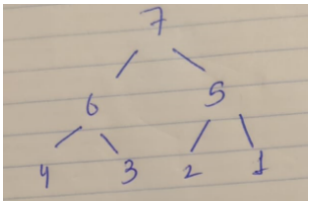
Solution:

Let us first calculate the total number of permutations or cases of the array that may exist. This can be given by $n!$. Since $n = 7$, the number of permutations without repeating numbers are $7! = 5040$.

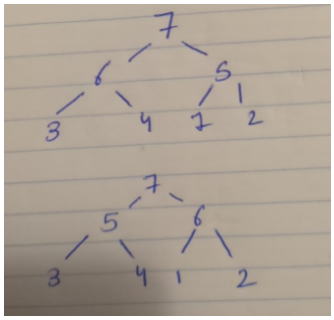
Now, we must check which arrays qualify for a max heap. Hand checking the arrays through brute force is not a viable solution, so we will try to understand what all the max heap arrays have in common.

Again, the conditions of the max heap detail that *for every node i other than the root, the value of the node is at most the value of its parent.*

Let us consider the simplest definable max heap from the array:



We can also consider other valid max heaps:



We notice that for the parents to be always greater than the children, we must have 7 at the root (index 0), and we may keep 5 and 6 at either index 1 or 2, and the rest of the elements can be switched around as leaves.

Case 1

Hence, first we will calculate all the instances where the order of the array is:

$$7, 5, 6, -, -, -, -$$

Where $-$ is a leaf node. This can be described by 4!, which is 24.

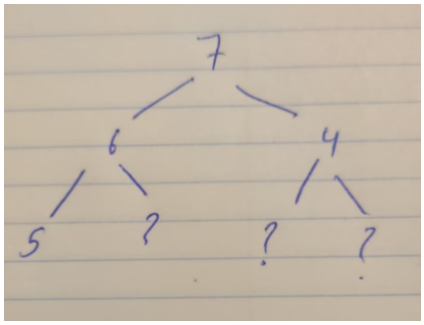
Next, for the configuration:

$$7, 6, 5, -, -, -, -$$

Again, this is equal to 4!, which is 24.

Case 2:

We can also produce a max heap where 5 is under the subtree of 6:



Here, again, the number will be equal to $2 * 2 * 3!$ because the subtrees can be swapped left and right and 5 can be swapped with the sibling. Hence it is equal to 24.

Case 3:

Here, we can put 5 and 4 under the subtree of 6 and 2 and 1 under the subtree of 3, where 5 and 4 can be swapped, or 2 and 1 can be swapped. The left and right subtrees can also be swapped, which leads to $2 * 2 * 2!$ configurations, equaling 8.

Now,

Total configurations for a max heap: $48 + 24 + 8 = 80$ configurations.

Total probability of choosing a max heap on random: $\frac{80}{5040} = 0.0158$

Hence, the probability of choosing a max heap on random is incredibly low, about 0.0158.