MS-CS Master Course Note (Non-Credit)

Mark Zhou

March 2024

This is my course note on "Algorithms for Sorting, Searching and Indexing" provided by Colorado University of Boulder. This is a non-credit prep course for an MS-CS degree.

Contents

1 Asymptotic Notation: Big O

1.1 Time and Space Complexity

We have some questions regarding how to figure out which algo is faster.

Q1: what input should we choose.

Q2: how to evaluate time.

When considering implementation details, that is under the realm of performance analysis. We don't want to be bogged down to this level and only wish to focus on the algorithm.

The runtime is the number of basic operations. For each sorting algorithm, there will be a range of possible runtime, due to the differences between inputs, the best case, average case, and worst case respectfully.

The average time cost is hard to analyze while may be more desirable than the worst case cost, which is too pessimistic in some cases. Never use best case cost.

In most of cases we will use worst case cost to evaluate.

Let's assume the cost of Insertion Sort is $f(n)$.

Hypothetically (just for examples) speaking:

$f(n) = 0.05n^2 + 1.5n + 70$

We assume that the different algorithms such as addition or multiplication have the same unit cost, that is to say, they cost the same under one unit run. But is that the real case?

Of course not. Different operations have different costs for a unit run. To change that, we can change the coefficient of the original equations.

We care about which algorithm is faster asymptotically. This is why

this kind of analysis is called asymptotic analysis.

The naming is inspired by one idea, that is the comparison should be within the realm of actual usage, for instance, when sorting 10 numbers, algo1 may be slower than algo2, but while sorting 10000 numbers, algo1 may outfast the opposite, that's why we compare them asymptotically.

And to be aware, we focus more on cases with large numbers or unit cases than small samples.

1.2 Big O

$f = O(g(n))$

Meaning: $f()$ **is ASUMPTOTICALLY upper bounded by** $g(n)$

Here are some properties:

When $g()$ **overtakes** $f()$**, meaning in the chart, g is above f after some time, and then f will remain overtaken forever.**

The constant factor must be disregarded.

> Note: In mathematics (logic), the symbol $\exists$ is read as "there exists" and the symbol $\forall$ is read as for all.

$\exists k > 0, N_0, \forall n \geq N_0, f(n) \leq k * g(n)$

$f(n) \leq k * g(n)$ **only happens beyond some point in time. When the above condition happens, we say:**

$f(n) = O(g)$

Let's take some examples:

$f = (1/2)n^2$

$g = 0.1n^3$

Assuming k=10, and N_0=1

Using the equation above $f(n) \leq k * g(n)$, then we will have $f(n) \leq g(n)$, which has got rid of the constant k;

Then we can say,

$f(n) = O(g)$

a.k.a.

$0.5n^2 = O(0.1n^3)$

As long as $g(n)$ will overtake $f(n)$ after K, $g(n)$ will be the cost of $f(n)$, aka $f(n) = O(g)$.

Under such circumstances, adding some constants to the equation of f and g, won't change the fact that $f(n) = O(g)$.

The $g(n)$ must always be less than or equal to $f(n)$ to make $f(n) = O(g)$ valid: that is to say, after a certain point N_0, the output of the function $f(n)$ is less than or equal to the output of the function $g(n)$.

2 Big Omega, Big Theta, and Examples

2.1 $f(n) = \Omega(g)$

f is asymptotically lower bounded by g (n): that is to say, after a certain point N_0, the output of the function f (n) is bigger or equal to the output of the function g (n).Since O () and $\Omega()$ are the opposite, if f (n) = O (g), then $g(n) = \Omega(f)$.

2.2 $f(n) = \Theta(g)$

f is asymptotically equal to g (n). This looks like a Bollinger band in technical analysis in some ways.

$\exists k_1, k_2, \forall n \geq N_0, k_1(g) \geq g \geq k_2(g)$

If $f = \Theta(g)$, it means f is asymptotically equal to $\Theta(g)$, which we can say,

$f = O(g), f = \Omega(g)$

When dealing with functions, we will typically get rid of constant figures in the function, however,

we cannot ignore the exponent constant such as the 2 and 4 in 2 to the power of 4x.

2^{4x}

Here is an example of big theta.

$f(n) = 2n + 3\log_2^n + 5$

$g(n) = 15n + 35\log_2^n + \sqrt{n} + 17$

In the above equations, n is the only thing that matters. Therefore we can ignore the log and square root elements by considering them as constants.

2.3 Examples of Asymptotic Notations

Here are five functions;

$f(n) = 2n^2 + 3\log_2^n + 4\sqrt{n} + 15$

$g(n) = 200\sqrt{n} + 15\log_2^n + 14n\sqrt{n}$

$h(n) = n^2 + 2n + 3\log^{logn}$

$l(n) = n^3 + 15n^2 + 2.5n$

$m(n) = 4n^2 + 13n^2\log_2^n$

Let's find out which element with n is the BIGGER one.

$n^2 = n * n > nlogn > \sqrt{n}$

Before we can compare these functions, we should determine the dominating term of each one:

$f : n^2$

$g : n^{1.5}$

$h : n^2$

$l : n^3$

$m : n^2 logn$

In this case, we can place these functions in such order, $l > m > f > h > g$

The smaller one is the Omega of the bigger one, and vice versa. For example, $l(n) = \Omega(m)$**,** $m = O(l)$**, and so on.**

3 Binary Search

3.1 what do we need to do in a binary search?

We need to check if a given element, say 6, is located in the given list. The prerequisite is that the list must be sorted to do a binary search.

Let's assume the list is sorted in ascending order. First, we need to find the middle element of the list by calculating the index number of left and right border elements, a.k.a. the first and last one in the list.

Here is how we do it:

$mid = (left + right)//2$

In the equation, // means floor divided, which will return exact the middle element with odd number list and the smaller one of the middle two elements in even number list.

Now we will check if the middle one equals to the given element. If so, problem solved! If not, and the given one is smaller than the middle, we will focus on the left side of the list, a.k.a. the elements located on the left of the middle one. In this case, if the first middle has index n, then the new "right" border element will have the index n-1.

If the middle element is smaller, of course we will focus on the right portion of the original list since this is an ascendibg ordered list. And in this case, if the first middle element's index is n, the new "left" will have the index n+1. Then we will conduct the calculation by finding the new middle on the left/right part of the list (a.k.a. the new range with the updated left or right border). We will compare the given value with the new middle.

We will repeat the process untill either find the element in the list, returning True, or not, which is a little bit complecated:

By the end of the process, the updated left border index will be bigger than the right border index in an ascending ordered list, which is not possible to find a valid range. This means the given element is not in the list.

For example, there is a list [1,4,9,15,20], and the given element is 8.

First we will find out the mid one by calculating the indexes:

$(0 + 4)//2 = 2$

The value of index 2 is 9, which is bigger than the given value 8. Then we will focus on the left part.

The updated right border will be index 1, therefore the new mid will be:

$(0 + 1)//2 = 0$

Now the equation gives us element value 1. This is smaller than 8. The current right border index is 1, and the updated left border is 1 as well. So the new range will gives us;

$(1 + 1)//2 = 1$

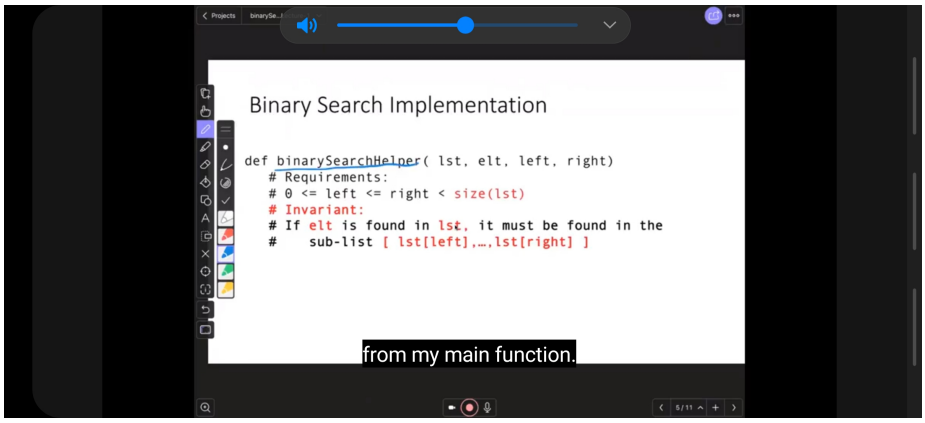
a.k.a. the value of index 1 is 4 which is smaller than 8.

And now we enter a crucial moment. The current right border index is 1, but since $4 > 8$, the updated left border index will be changed from 1 to 2.

Left index=2, right index =1, which is not valid.

Therefore, the given element 8 is not located in the given list. Problem solved.

3.2 Define a Func binarySearchHelper()



Here is the code implemented of the fuc binarySearchHelper().

This function will work as a recursion since it will continuously call a new binarySearchHelper() inside the current one until finished.

How to prove correctness?

The main property of binary search: if we can find the element in the list, it must be in the searching range.

3.3 Running Time Analysis of Binary Search Algo

For the running time analysis, we first assume this list has the length of:

$n = 2^k$

Each time when we check the middle element, we halve the list in half, then the updated list will have a new lengh of

$n/2 = 2^{k-1}$

By the end, the list will have a length of

$2^{k-k} = 1$

Now we narrow down the checking range to only one element and

check it with the given value. If they are not the same, we will try to find a new middle for more time, which will produce an invalid range as we mentioned before.

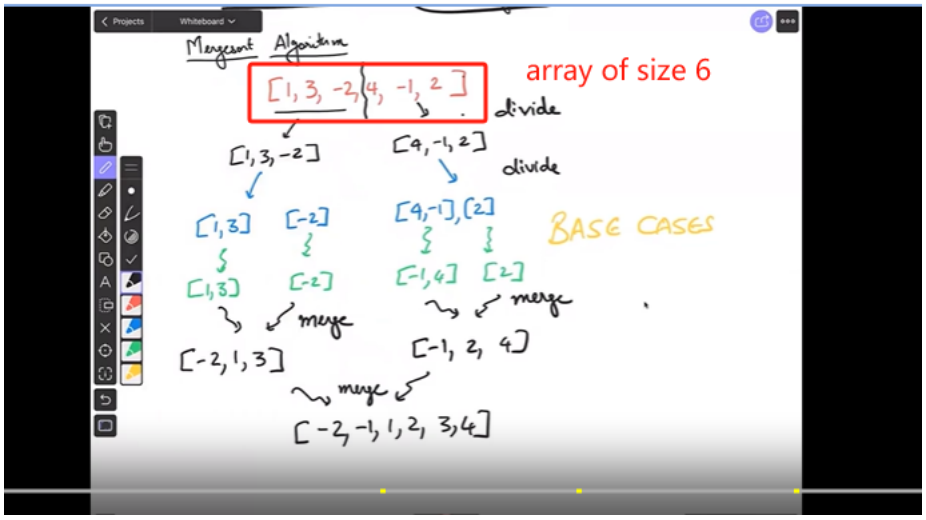
That is to say, we will at most run the program k+1 times until finished. So by the end, the largest running time will be:

$\log_2^n + 1$

Binary search is very efficient. If we have a list of 1 million values, we may only need to search for approximately 21 times.

4 Merge Sort

4.1 Mergesort: Correctness and Running Time



In order to do a mergesort, firstly we will divide the given array into two parts with the middle element as a separator.

For example, we have an array

$$1, 3, -3, 4, -1, 2$$

We will divide it into

$$1, 3, -2$$

and

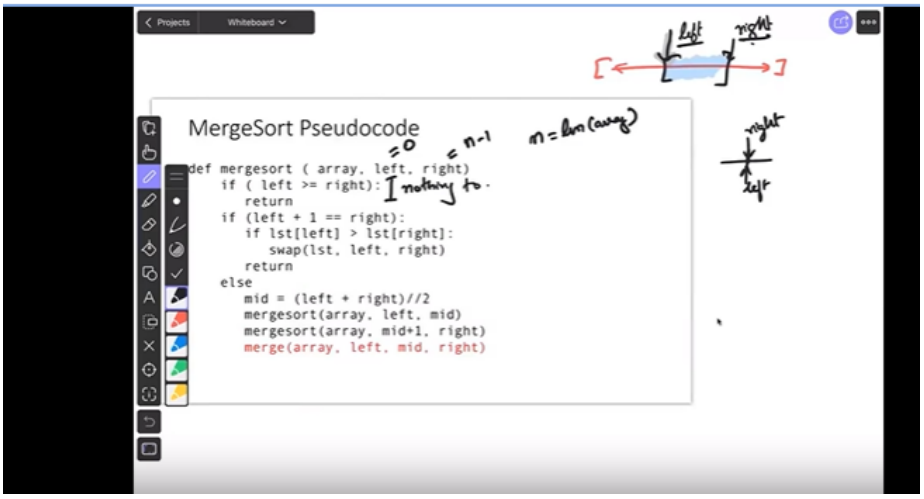
$$4, -1, 2$$

Then we will further divide these two into four parts. By now, we

have some array contain one one item while others contain two. For those with two items, we will do a swap if they are in the wrong order.

These kinds of very small divided array are called base cases, since they are somehow the base unit of this division process.

Then we will merge back all those sorted base case arrays, all the way to the top, aka one merged array.



Here is a pseudocode of mergesort. Let's get a closer look at how does it work.

At the very beginning, the algo will compare left and right element and check if they are in a valid form, that is to say, whether left (index of the value) is smaller than right (index of the value). If not, the range formed by the left and right index is invalid hence algo ended. Second, we will determine whether the array formed by left and right contains only two elements. If $left + 1 = right$, that means these are adjacent indices. If this is the case, we will swap the elements if they have the wrong order.

At last, we enter a position in which the range is larger than two elements array. Now we will simply find the mid element by performing a floor division and do merge sort on left side of the array which is ended by mid and right side which begins with mid+1. This procedure will sort this two subarrays. After that, we will MERGE all the subarray in the order of left, mid and right.

The merge procedure is the next big thing. After this step, all the elements of the array will be sorted.
First thing in Merge function, we will assign an initial value to two looping variables, i and j, which are the index of each subarray, by assigning left to i and mid+1 to j. This is to make sure the two loops will begin from the first item of each sub array, aka the left array and right array.
We will introduce a temp storage variable `tem_store` to maintain the sorted result and then write back to the original array. Although there are in-place algos that will do the work, it is much easier to just include a third array for storage.
The following is straight forward. While i and j are still within the range of each subarray, aka $i \leq mid$ and $j \leq right$, we will compare each i and j pair and store the smaller one to the `tem_store`. Then we add one to the index of the subarray in which the value just compared is smaller, that is to say, move on to the next item. In the meantime, the other array remain the same index, waiting to compare in the next round.
After all these steps and iterations, we will find a well sorted `tem_store` array. Then we simply write it back to the orginal array and problem

solved.

4.2 Running Time of Merge Procedure

The running time of merge procedure is precisely the length of the two arrays since each element will be added to the temp store once. We assume each step cost 1, then the total time cost will be len(array A+array B).

It can be written as $right - left + 1$**.**

Now let's look at it at higher level.

Suppose the original array has lenth of n.

The first split will be two n/2 arrays, then the split goes on until it reaches the base unit.

For each step, the cost will be the array's count mutiply by the base unit, for example, for the 1st level, the cost will be $2 * (n/2)$**, for the 2nd, it will be** $4 * (n/4)$ **and so on.**

Now we can try to conclude a general equation for the time cost of merge sort.

Let us assume that $n = 2^k$**;**

Since for each level, the number of level is $n, n/2, n/4, \ldots, 1$**;**

We can write it as $2^{k-1}, 2^{k-2}, \ldots, 2^{k-k}$**.**

For each level, the time cost is the base unit of that level mutiply by that level's amount of subarrays in that level. For example, the base unit of the 1st level after n is n/2. Then the time cost will be $2 * (n/2)$**. Therefore, the generalized time cost will be** $n * No.(level)$**. Now let's find out what's the number of level.**

We have already seen this above: $2^{k-1}, 2^{k-2}, \ldots, 2^{k-k}$**, corresponding to level 1, level 2, . . . , last level. We can derive the number of level**

from this array, $1, 2, \ldots, k$**.**

That is to say, the number of level is k. Ergo $n * No.(level) = n * k$**.**

Since $n = 2^k, k = \log_2^n$**.**

The final equation will be:

$Cost = n * \log_2^n$

If we using the Big O quatation to address to this problem, we get

$Mergesort = \Theta(n * \log_2^n)$

And we can get rid of the constant 2 and get

$Mergesort = O(n * \log^n)$**.**

5 Heap, Min Heap and Max Heap

5.1 Basic Understanding of Heap

Heap is a type of array that has certain properties. There are the concepts of min heap and max heap and we will go through these soon.

First here's a concept called left/right child. Let's say we have a heap of 9 elements. For element number 1, the element number 2 will be called the left child of element 1, and element 3 will be called the right child.

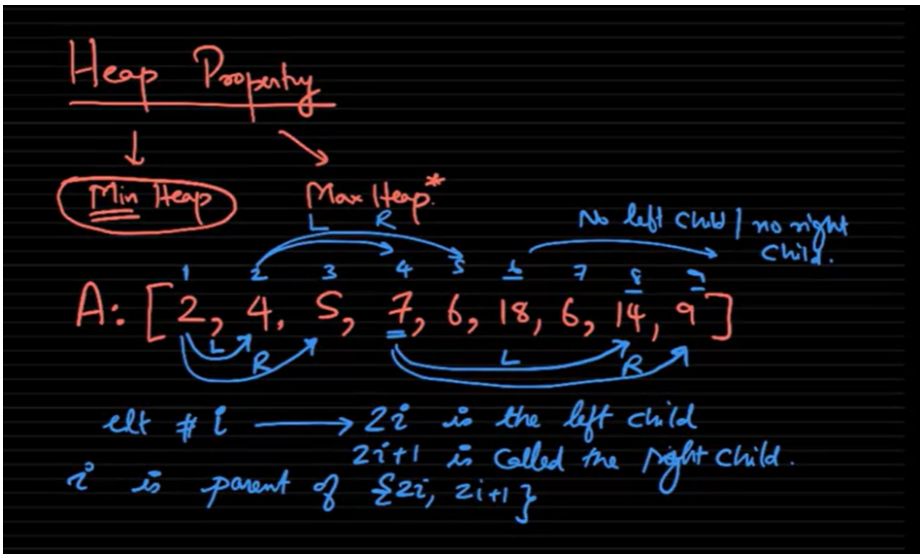
While for element 2, actually element 4 will be the left child and 5 the right child.

Therefore we have the equation of it:

$\forall i, leftchild = 2i, rightchild = 2i + 1$

For certain element in the array, If the left/right child element is not inside the current array, then the original one doesn't have left/right child.

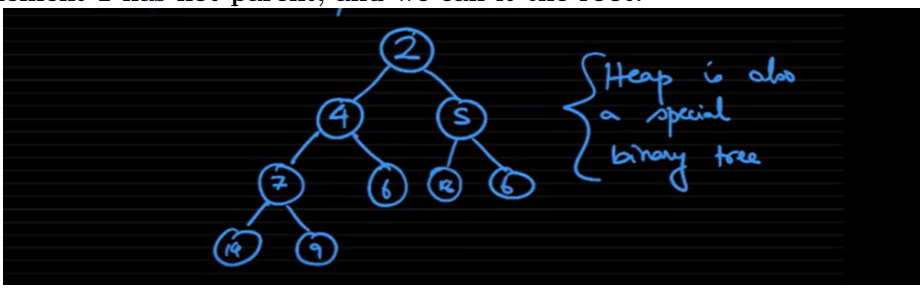
For example, element 6 has not such child since element 12 and 13 are not in the array of size 9.



We can say i element is the parent of 2i and 2i + 1.

for element j, j/2 is the parent. If the division gives a fraction of number then we just round it down (floor division).

Element 1 has not parent, and we call it the root.



If we write the heap array in a parent-child structure, it actually looks like a tree. Yes, heap is also a special binary tree (laid out as an array).

Here are some properties:

1. If a node in a heap has a right child, it must have a left child.

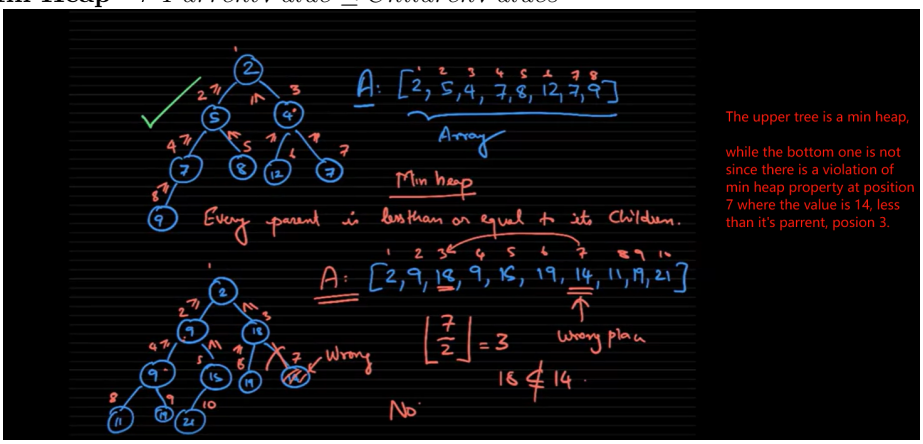
2. Heap is an array of size n. $\forall A[i], 2i \leq n, A[2i] = leftchild$

3. $\forall A[i], 2i+1 \leq n, A[2i+1] = rightchild$

And here is the Min Heap Property:

The value of the parent node must be less than or equal to the value of children nodes.

Min Heap $\rightarrow ParrentValue \leq ChildrenValues$



If we turn to the opposite side, we get the Max Heap Property: The value of the parent node must be larger than or equal to the value of children nodes.

Max Heap $\rightarrow ParentValue \geq ChildrenValues$

So for any array, we can simply check whether they meet min heap property or max heap property or not in order to determine their identity.

Why do we care about min heap or max heap? Here is a basic idea; For any min heap A, we can say A[1], which is the root, is the smallest element of the entire array.

We can solve this problem with induction.

The Base Case: The root element is the smallest one for Heap depth = 1.

And we have the induction hypothesis: Let the result from base case

hold for all $depth \leq d$, and then prove it for depth of d + 1.

Illustrated as a binary tree, we can see each triangle (consisting of three nodes) as a "sub-heap". The node in the upper level of the sub-heap is the root of it. For every level from 1 to d, the upper node is always the root of each sub-heap. Therefore the root of level 1 is the smallest in all level from 1 to d.

Then we can say that's the same case for level d+1.

That is to say, for any min heap A, we can say A[1], the root of it is the smallest element of the entire heap.

5.2 Heap Primatives: Bubble up

We are interested in the following operations on a heap:

1. Inserting elements into a heap

2. Deleting one from a heap

The main primitives of heap are called bubble up and bubble down. We will explain them later.

For now, lets assume there is a heap with only one point of failure. Here is an example of such min heap.

In the above array, the 6th element, which value is also 6, is smaller than its parrent element 3, with a value of 7. Besides that, the entire array fit the definition of min heap.

We can say in this case, the 6th element is in a wrong relation with its parent. It should be moved up in order to make this array a valid min heap.

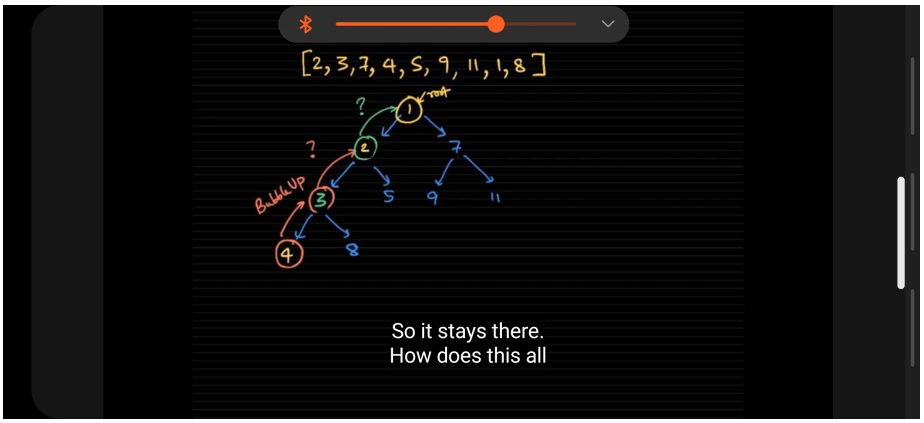
The operation that will acheive this is called bubble up.

Bubble Up: the swap of a value with its parrent value if they are in the wrong relation.

After we sway the misplaced value with its parent, now we have a new parrent of value 6 and its left child 7. Since the parrent is also less than or equal to its right child. The array is now a valid min heap.

Here is another example. We need to bubble up 3 times in order to make it a min heap.

$$2, 3, 7, 4, 5, 9, 11, 1, 8$$



In the above array, every elements except the last but one are in the right position. Element 8 with value of 1, is smaller than its parrent. Then we need to bubble up value 1 three times into position 4, 2, and 1 to make it work. Now value 1 become the root.

We can write a pseudocode for bubble up:

bubble up (A,j):

if $j \leq 1$:

return (Do nothing)

else if:

$A[j] < A[j//2]$: (// means floor divided)

swap (A[j], A[j//2])

bubble up (A,j//2) (Yes, this line makes this function recursive.)

return

This is a recursive program that will bubble up the misplaced element until it is in the right place.If we try to estimate the running time of bubble up, the worse case would be O (depth of heap), since the element will be a leaf (the one without child) and it lays on the bottom level of the heap.

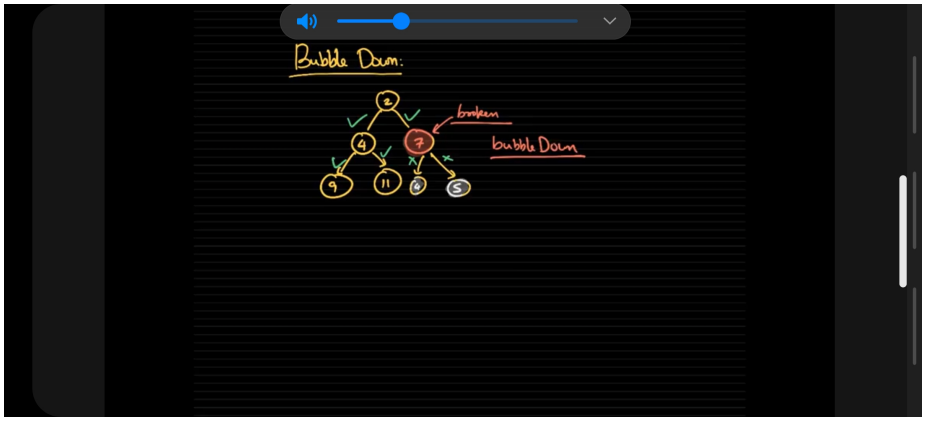
Then how about θ? Let's assume a heap has n element. If the leaf needed to be bubble up all the way to root position, the steps will be:

$n, n/2, n/4, \ldots, 1$

Then the steps will be $\log_2^n$, a.k.a. $\theta(\log_2^n)$.

If we have a heap of $n = 10^6$, which is 2^{19}, then the total operation required for bubble up is 19.

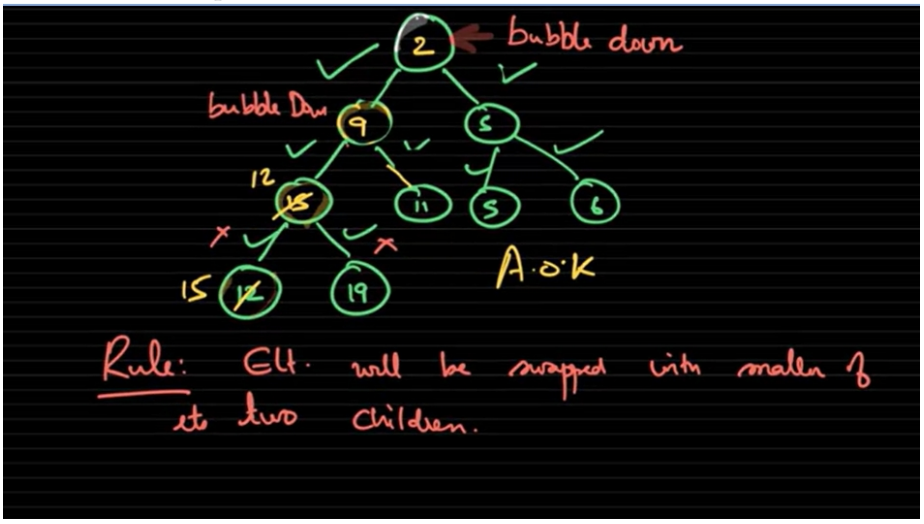
5.3 Bubble Down



In opposite to bubble up, bubble down is the operation that will move the root element to the right position. That is to say, if the root element is not the smallest one, we will swap it with the smallest child. So in the example above, the element 7, which is the parrent of 4 and 3, will be swapped with 3.

What if we cannot bubble down once and place the root element in the right position? Sometimes we need to do it multiple times.

Here is a min heap that needs to be bubbled down twice.



Now let's take a look at the pseudocode of bubble down.

bubble down (A,j):
if $2j > n$: (2j is actually the left child, n is the length of A.)
return (Do nothing since A has no children.)
else if: $2j \leq n$ and $2j + 1 \geq n$: (no right child)
if $A[j] > A[2j]$:
swap (A[j],A[2j])
bubble down (A,2j) (recusive)
else if: $2j \leq n$ and $2j + 1 \leq n$: (has both children)
if $A[2j] \leq A[2j + 1]$ and $A[j] > A[2j]$: (Left child is smaller, which will be swapped with the current parrent.)
swap (A[j],A[2j])
bubble down (A,2j) (recusive)
else if $A[2j] > A[2j + 1]$ and $A[j] > A[2j + 1]$: (right child is smaller and will be swapped.)
swap (A[j],A[2j+1])
bubble down (A,2j+1) (recusive)
return

We will first compare 2j, which is the left child of j, with n, the length of A. If $2j > n$, j has no children whatsoever.
Then we go into the second condition, parrent j has only one child, the left one of course. If the parrent is bigger than its child, in this case $A[j] > A[2j]$, we need to swap them.
At this moment, we will need to run the recursive procedure, that is to call bubble down again.

The third condition is that the parrent has both children. We will compare the left child with the right child. If the left child is smaller, we will compare it with the parrent. If the parrent is bigger, we will swap them. Of course, we will need to recursively run bubble down again.

By the end, if the right child is smaller, we will compare it with the parrent. If the parrent is bigger, we will swap them and call the main function recursively.

Problem solved.

How do we estimate the running time of bubble down?

The worst case will be the leaf node, which is the last level of the heap. That is to say, we need to bubble down the root element all the way to the bottom of the heap.

The time cost will be the depth of the heap, which is $\theta(\log_2^n)$.

How do we get this number? Let's do the math again.

In order to get the depth of the heap, we need to find out how many levels the heap has. The root element which needed to be bubbled down will move through the following steps;

$1, 2, 4, 8, 16, \ldots, n$

Which can be rewritten as $2^0, 2^1, 2^2, 2^3, 2^4, \ldots, 2^k$

k is the depth of the heap, and $k = \log_2^n$.

5.4 Reading Note on CLRS Chapter 6.3

Heapify is the process of reshaping a binary tree into a heap data structure.

The main idea of min heapify is to make sure that the root of the tree is the smallest element of the entire tree.

The min heapify procedure is a recursive one.

5.5 Priority Queue, Heapify and Heapsort

5.5.1 Inserting into a heap

First we need to understand how to insert an element into a heap.

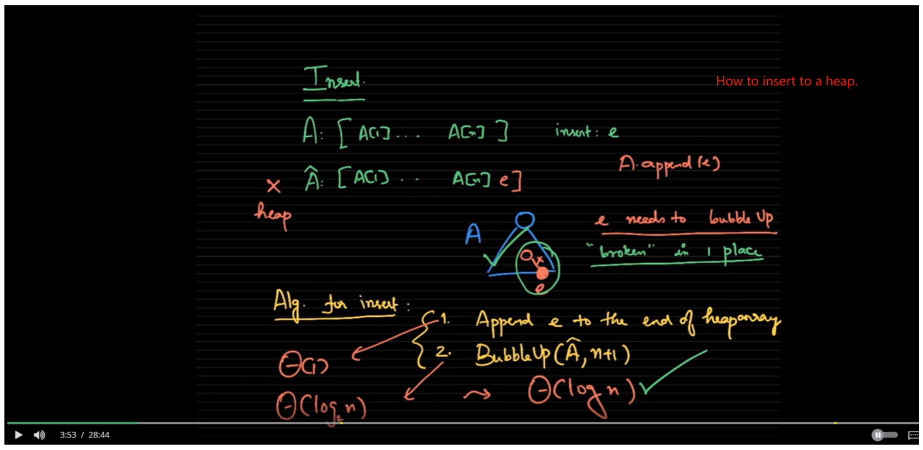
The main idea is to insert the element at the end of the array and then bubble up the element until it is in the right position.

That is to say, insertion = insert to the end + bubble up.

The time cost of inserting to the end of the heap is just as simple as $\theta(1)$**.**

Then we need to bubble up the element. The time cost is $\theta(\log_2^n)$**.**

So the total time cost is just $\theta(\log_2^n)$**.**



Priority Queues, Heapify, and Heapsort

5.5.2 Deleting from a heap

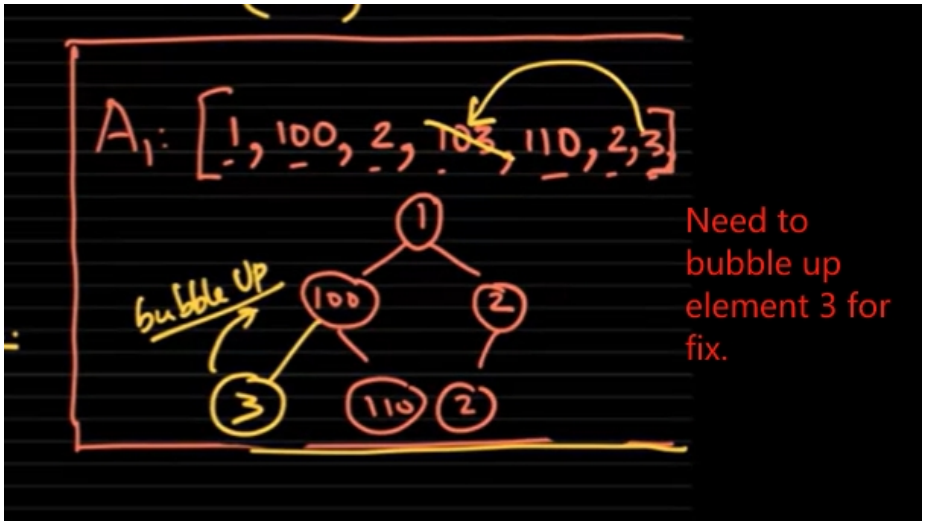


Figure 1: Bubble up for a fix.

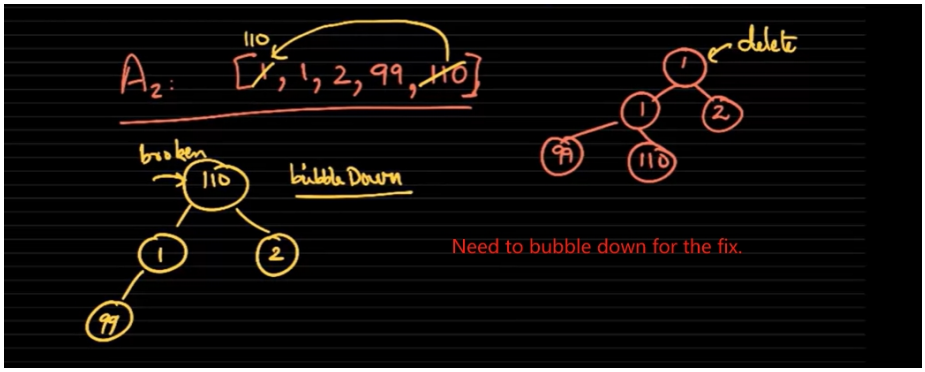


Figure 2: Bubble down for a fix.

Then we come to the concept of deletion of a heap element.

1. Replace the element that we want to delete with the last element of the heap.

2. Adjust the lenth of the heap to n-1, assuming the original length is n. Now we can simply delete the last element.

3. Bubble up or bubble down the element that we moved. The specific operation we use depends on the situation. We need only one operation for any cases.

Here is a question regarding the fix operation after deletion of a heap.

Question

Consider the heap $A = [a, b, c, d, e, f]$, and suppose we want to delete b from it. The first step is to move the last element, f, into b's position, giving $[a, f, c, d, e]$. The remaining step is to bubble f either up or down. How do we know that only one of these two directions is needed?

Try to answer in one or two sentences.

Because f is the only broken element. And it is either bigger or smaller than its parrent.

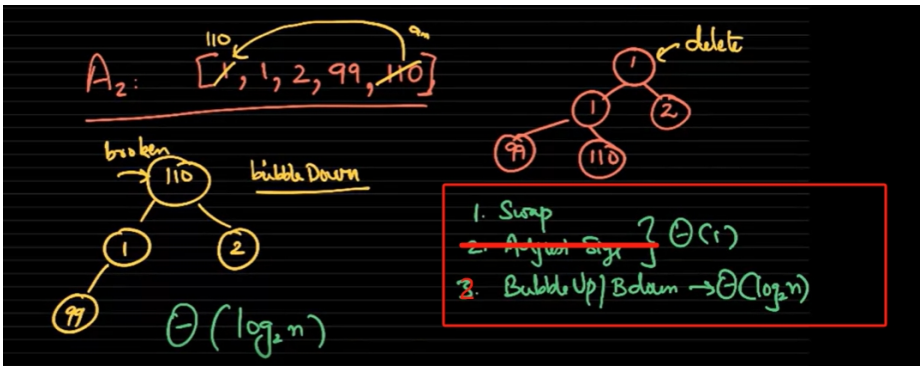
Correct

We only bubble f up if $a > f$, and we only bubble down if either $f > d$ or $f > e$ (or both). The min-heap property ensures that $a > d$ and $a > e$, and so f cannot be both less than a and greater than d or e.

The cost of time for deletion consists of two parts, 1 swap and then 2 bubble up/down.

The time of cost is $\theta(1)$ for step 1 and $\theta(\log_2^n)$ for step 2.

The total time cost is $\theta(\log_2^n)$.

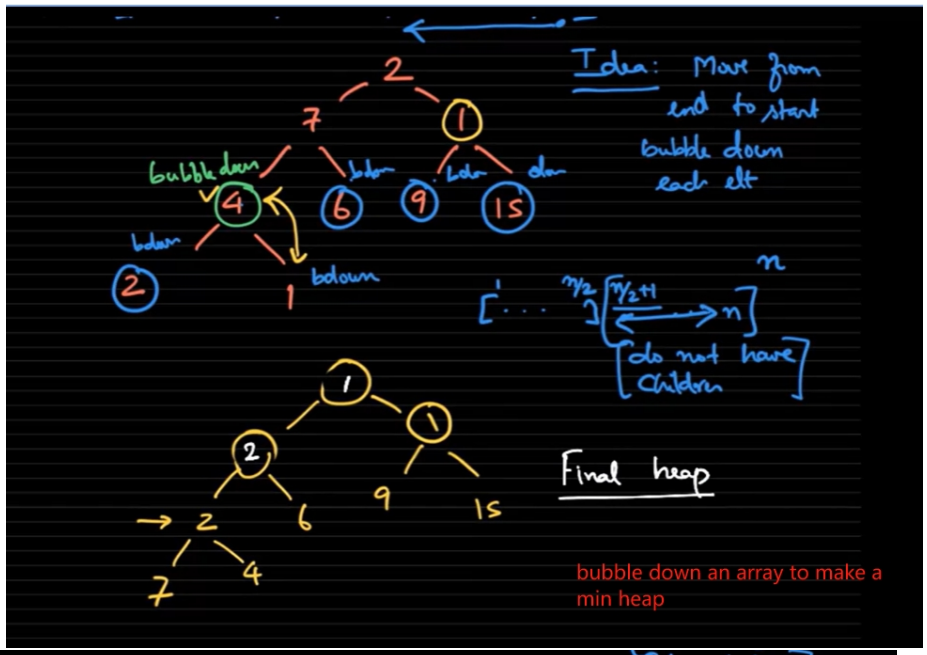


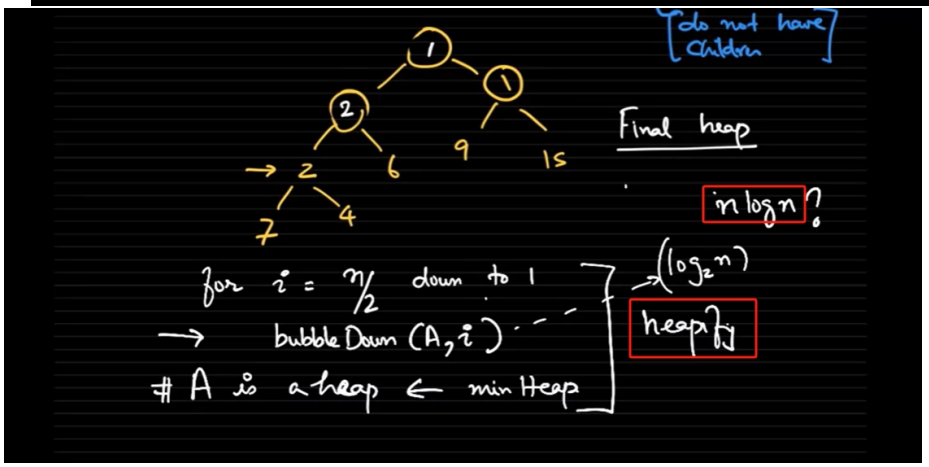
5.5.3 Finding the smallest element of a heap

The time cost of finding the smallest element of a min heap is $\theta(1)$. However, finding the largest element of a min heap is pretty hard and inefficient.

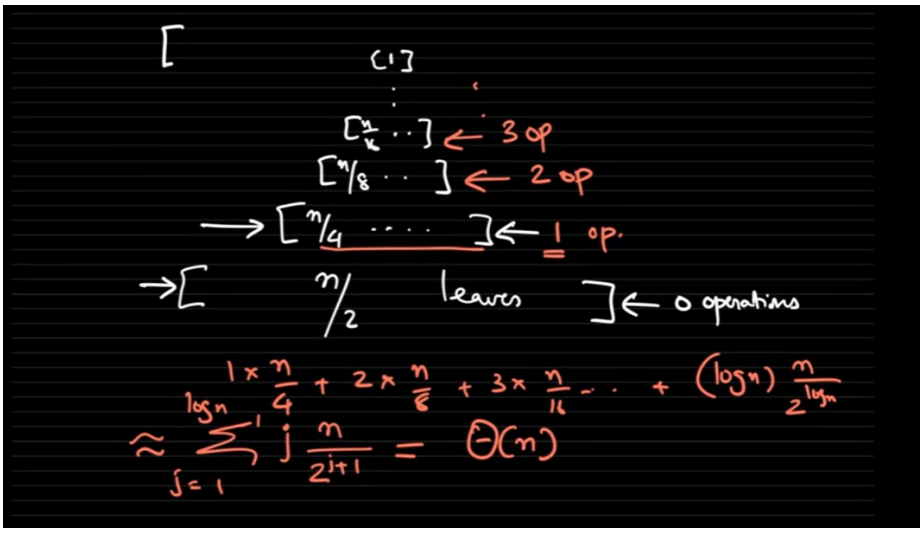
Now we are talking about priority queue.
Priority queue is a data structure that will allow us to insert elements and delete the smallest one.
The main idea of priority queue is to keep the smallest element at the root of the heap.
Priority queue has a priority setting.

We can bubble down elements that are in the wrong position and then get a min heap.
This is the main idea of heapify.
The running time of heapify is $O(n \log_2^n)$.

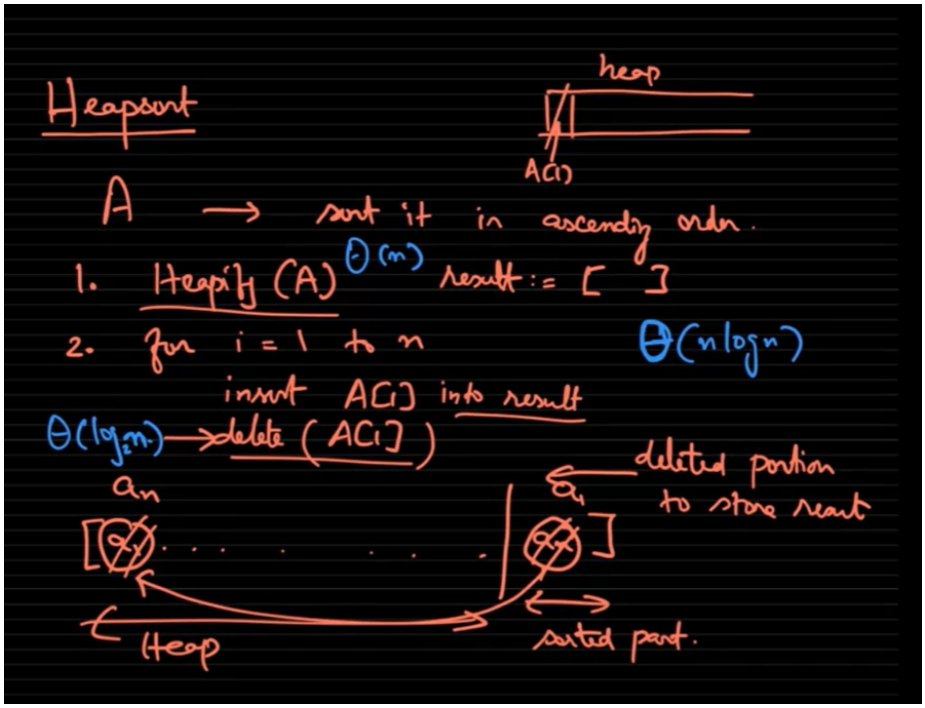




The overall running time of heapify is linear to the size of the array. That is to say, the running time is $\theta(n)$.


[
[1]
[n/k ..] ← 3 op
[n/8 ..] ← 2 op
→ [n/4 ....] ← 1 op.
→ [n/2 leaves] ← 0 operations
1 × n/4 + 2 × n/8 + 3 × n/16 .. + (log n) n/2^log n
≈ Σ_{j=1}^{log n} j n/2^{j+1} = Θ(n)

5.5.4 Heapsort



Question

Select all of the features below which accurately describe Heapsort.

- [x] $\Theta(n \log n)$ complexity.

  Correct

- [x] Easily parallelized.

  This should not be selected
  Unlike Mergesort, Heapsort's steps do not operate on totally separate sections of the array, and thus cannot be easily performed in parallel.

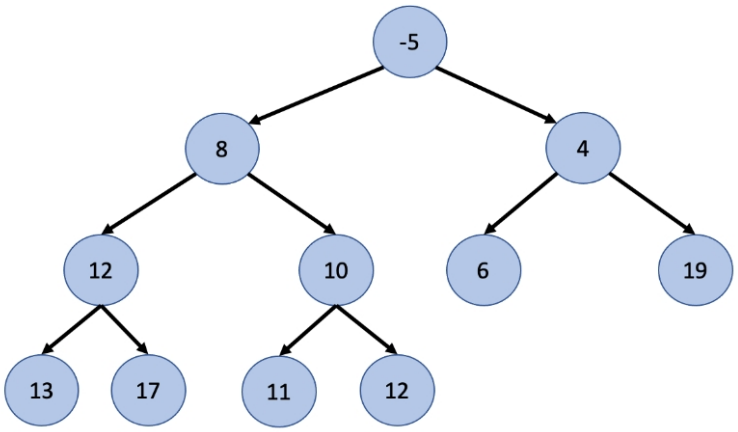
- [x] No need to allocate a new array to store the sorted elements ("in-place").

  Correct
  This is an advantage over Mergesort.

5.6 Quiz: Bubble-Up/Down, Insertion and Deletion

5.6.1 Question 1

1. Consider the following min-heap, in tree form: 6 / 6 points



Consider the following min-heap, in tree form:
Now suppose we perform a heap-insert to add a new element 1 to the heap.Select all the correct facts about this insertion process from the list below.

The first step is to place the new element at the beginning of the array, making it the root of the tree.

The first step is to place the new element at the end of the array, making it a child of the element.
Correct: After taking the first step, in which the new element is placed in the array, the heap property continues to hold.

In order to fix a heap property violation encountered during insertion, we perform a bubble-up operation, which swaps the new element and its parent

element.

Correct: In order to fix a heap property violation encountered during insertion, we perform a bubble-down operation which swaps the new element with one of its child elements.

When we insert 1 into the above min-heap, the first bubble-up step swaps element 1 with its parent 6. This fixes all the min-heap property violations.

When we insert 1 into the above min-heap, a second bubble-up step occurs which swaps element 1 with 4. This fixes all the min-heap property violations.

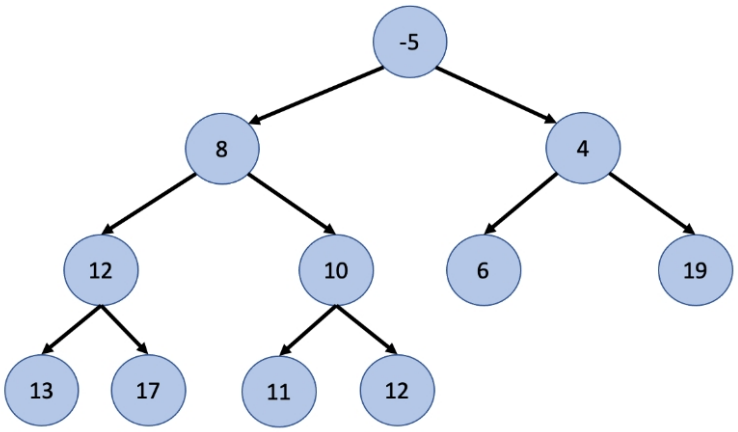
Correct: This is correct.

In the worst case, bubbling up will require $\theta(\log_2^n)$ swaps, moving the inserted element all the way from a leaf position in the heap to the root.

Correct: This is correct: note that the length of the longest path from any leaf of a heap to the root is at most $\log_2^n$.

5.6.2 Question 2

1. Consider the following min-heap, in tree form: 6 / 6 points



Consider again the min-heap example:
Suppose we heap-delete the element 8.

Select all correct facts about this deletion process from the list below.

The first step replaces the element to be deleted (8) with the very last element in the heap's array (12).

The first step replaces the element to be deleted (8) with the very first element in the heap's array (-5).

After the first step, in which we replace the element we are deleting (8), we are guaranteed that we have created at most one min-heap property violation. This should be selected as well.

To fix the heap property violation caused by the first step, we perform a bubble-up or bubble-down operation depending on whether the swapped-in element is larger than its children or smaller than its parent.

Deletion of an element in a heap with n elements requires at $O(\log_2^n)$ comparison and swap operations.

5.7 Quiz: Heapify, Priority Queue and Heapsort

5.7.1 Question 1

Suppose we wished to heapify an array into a minheap using the following algorithm (assume $a[0]$ is not used and assume that bubble up routine is implemented).

```
def heapify (a):
    n = len (a)
    for i in range (1, n):
        bubble_up (a, i)
    return a
```

The worst case complexity of this procedure will be $\theta(n \log_2^n)$.

The procedure above is computationally less efficient than the one presented in the lecture because $n/2$ of the elements in the array may potentially need to bubble up all the way to the root.
Correct: Indeed: each of these $n/2$ elements may incur a cost proportional to $\log(\frac{n}{2})$ since they may each bubble up from the leaf to the root of he heap.

Sorting the array using insertion sort will achieve the same complexity as the heapify procedure above.

Sorting the array using mergesort will achieve the same complexity as the heapify procedure above.

The worst case of the procedure above is realized if we tried to heapify an array that is already sorted in descending order.

The worst case of the procedure above is realized if we tried to heapify an array that is already sorted in ascending order.

5.7.2 Question 2

Select all the facts that are true about the heapsort procedure in comparison with the insertion and mergesort procedures we have studied thus far. Assume that all heaps are minheaps unless otherwise mentioned in the option.

Heapsort requires extra storage to store the heap unlike insertion sort that can sort an array in place.

Heapsort is asymptotically faster than mergesort.

If an array is already sorted in ascending order and the heapify procedure is run on it, then it does not modify the array in any way.

If a min-heap is in fact already sorted in ascending order, deleting the minimum element operation runs in constant time.

6 Hashtable