# How can building a heap be O(n) time complexity?

Can someone help explain how can building a heap be O(n) complexity?

Inserting an item into a heap is `O(log n)`, and the insert is repeated n/2 times (the remainder are leaves, and can't violate the heap property). So, this means the complexity should be `O(n log n)`, I would think.

In other words, for each item we "heapify", it has the potential to have to filter down once for each level for the heap so far (which is log n levels).

What am I missing?

algorithm heap complexity-theory construction

edited Apr 21 '16 at 12:14

gsamaras
41k ● 21 ● 72 ● 137

asked Mar 18 '12 at 3:15

GBa
5,548 ● 15 ● 42 ● 66

what precisely do you mean by "building" a heap? – mfrankli Mar 18 '12 at 3:29

As you would in a heapsort, take an unsorted array and filterdown each of the top half elements until it conforms to the rules of a heap – GBa Mar 18 '12 at 3:29

2 Only thing i could find was this link: The complexity of Buildheap appears to be Θ(n lg n) – n calls to Heapify at a cost of Θ(lg n) per call, but this result can be improved to Θ(n) cs.txstate.edu/~ch04/webtest/teaching/courses/5329/lectures/… – GBa Mar 18 '12 at 3:30

1 @Gba watch this video from MIT: He explains well on how we get O(n), with a lil bit of math youtube.com/watch?v=B7hVxCmfPtM – CodeShadow Jul 7 '16 at 19:35

1 Direct link to the explanation @CodeShadow mentioned: youtu.be/B7hVxCmfPtM?t=41m21s – sha1 Nov 14 '16 at 2:22

## 13 Answers

Your analysis is correct. However, it is not tight.

It is not really easy to explain why building a heap is a linear operation, you should better read it.

A *great analysis* of the algorithm can be seen here.

---

The main idea is that in the `build_heap` algorithm the actual `heapify` cost is not `O(log n)` for all elements.

When `heapify` is called, the running time depends on how far an element might move down in tree before the process terminates. In other words, it depends on the height of the element in the heap. In the worst case, the element might go down all the way to the leaf level.

Let us count the work done level by level.

At the bottommost level, there are `2^(h)` nodes, but we do not call `heapify` on any of these, so the work is 0. At the next to level there are `2^(h - 1)` nodes, and each might move down by 1 level. At the 3rd level from the bottom, there are `2^(h - 2)` nodes, and each might move down by 2 levels.

As you can see not all heapify operations are `O(log n)`, this is why you are getting `O(n)`.

edited Apr 21 '16 at 12:19

gsamaras
41k ● 21 ● 72 ● 137

answered Mar 18 '12 at 3:39

emre nevayeshirazi
14k ● 9 ● 50 ● 66

14 This is a great explanation...but why is it then that the heap-sort runs in O(n log n). Why doesn't the same reasoning apply to heap-sort? – hba Mar 30 '13 at 3:11

38 @hba I think the answer to your question lies in understanding this image from this article. `Heapify` is `O(n)` when done with `siftDown` but `O(n log n)` when done with `siftUp`. The actual sorting (pulling items from heap one by one) has to be done with `siftUp` so is therefore `O(n log n)`. – The111 Aug 23 '13 at 0:12

2 I really like your external document's intuitive explanation at the bottom. – Lukas Greblikas Oct 6 '13 at 17:29

1 @hba the Answer below by Jeremy West addresses your question in more fine, easy-to-understand detail, further explaining The111's comment answer here. – cellepo Oct 21 '15 at 16:16

A question. It seems to me that the # comparisons made for a node at height `i` from the bottom of a tree of height h must make `2* log(h-i)` comparisons as well and should be accounted for as well @The111. What do you think? – Sid Nov 17 '16 at 15:55

---

The accepted answer gives the correct analysis to show that `buildHeap` runs in *O(n)* time. But the question was raised:

> "This is a great explanation...but why is it then that the heap-sort runs in *O(n log n)*. Why doesn't the same reasoning apply to heap-sort?"

An answer was given in the comments that suggests that the difference lies in `siftDown` vs. `siftUp` that I think somewhat misses the point (or is just too brief to make it obvious). Indeed, you don't need to use `siftUp` at all when implementing a heap sort (although it is required for a priority queue, for example, to implement the insert operation).

**EDIT:** I've modified my answer to describe how a max heap works. This is the type of heap typically used for heap sort or for a priority queue where higher values indicate higher priority. A min heap is also useful; for example, when retrieving items with integer keys in ascending order or strings in alphabetical order. The principles are exactly the same, you just need to switch the sort order.

Remember that the heap property specifies that each node must be at least as large as both of its children. In particular, this implies that the largest item in the heap is at the root. Sifting down and sifting up are essentially the same operation in opposite directions: move an offending node until it satisfies the heap property:

- `siftDown` swaps a node that is too small with its largest child (thereby moving it down) until it is at least as large as both nodes below it.
- `siftUp` swaps a node that is too large with its parent (thereby moving it up) until it is no larger than the node above it.

The number of operations required for each operation is proportional to the distance the node may have to move. For `siftDown`, it is the distance from the bottom of the tree, so `siftDown` is expensive for nodes at the top of the tree. With `siftUp`, the work is proportional to the distance from the top of the tree, so `siftUp` is expensive for nodes at the bottom of the tree. Although both operations are *O(log n)* in the worst case, in a heap, only one node is at the top whereas half the nodes lie in the bottom layer. So **it shouldn't be too surprising that if we have to apply an operation to every node, we would prefer `siftDown` over `siftUp`.**



The `buildHeap` function takes an array of unsorted items and moves them until it they all satisfy the heap property. There are two approaches one might take for `buildHeap`. One is to start at the top of the heap (the beginning of the array) and call `siftUp` on each item. At each step, the previously sifted items (the items before the current item in the array) form a valid heap, and sifting the next item up places it into a valid position in the heap. After sifting up each node, all items satisfy the heap property. The second approach goes in the opposite direction: start at the end of the array and move backwards towards the front. At each iteration, you sift an item down until it is in the correct location.

Both of these solutions will produce a valid heap. The question is: which implementation for `buildHeap` is more efficient? Unsurprisingly, it is the second operation that uses `siftDown`. If h = log n is the height, then the work required for the `siftDown` approach is given by the sum

```
(0 * n/2) + (1 * n/4) + (2 * n/8) + ... + (h * 1).
```

Each term in the sum has the maximum distance a node at the given height will have to move (zero for the bottom layer, h for the root) multiplied by the number of nodes at that height. In contrast, the sum for calling `siftUp` on each node is

```
(h * n/2) + ((h-1) * n/4) + ((h-2)*n/8) + ... + (0 * 1).
```

It should be clear that the second sum is larger. The first term alone is *hn/2 = 1/2 n log n*, so this approach has complexity at best *O(n log n)*. However, the sum for the `siftDown` approach can be bounded by extending it to a Taylor series to show that it is indeed *O(n)*. If there is interest, I can edit my answer to include the details. Obviously, *O(n)* is the best you could hope for.

The next question is: if it is possible to run `buildHeap` in linear time, why does heap sort require *O(n log n)* time? Well, heap sort consists of two stages. First, we call `buildHeap` on the array, which requires *O(n)* time if implemented optimally. The next stage is to repeatedly delete the largest item in the heap and put it at the end of the array. Because we delete an item from the heap, there is always an open spot just after the end of the heap where we can store the item. So heap sort achieves a sorted order by successively removing the next largest item and

putting it into the array starting at the last position and moving towards the front. It is the complexity of this last part that dominates in heap sort. The loop looks likes this:

```
for (i = n - 1; i > 0; i--) {
    arr[i] = deleteMax();
}
```

Clearly, the loop runs O(n) times (*n - 1* to be precise, the last item is already in place). The complexity of `deleteMax` for a heap is *O(log n)*. It is typically implemented by removing the root (the largest item left in the heap) and replacing it with the last item in the heap, which is a leaf, and therefore one of the smallest items. This new root will almost certainly violate the heap property, so you have to call `siftDown` until you move it back into an acceptable position. This also has the effect of moving the next largest item up to the root. Notice that, in contrast to `buildHeap` where for most of the nodes we are calling `siftDown` from the bottom of the tree, we are now calling `siftDown` from the top of the tree on each iteration! *Although the tree is shrinking, it doesn't shrink fast enough*: The height of the tree stays constant until you have removed the first half of the nodes (when you clear out the bottom layer completely). Then for the next quarter, the height is *h - 1*. So the total work for this second stage is

```
h*n/2 + (h-1)*n/4 + ... + 0 * 1.
```

Notice the switch: now the zero work case corresponds to a single node and the *h* work case corresponds to half the nodes. This sum is *O(n log n)* just like the inefficient version of `buildHeap` that is implemented using siftUp. But in this case, we have no choice since we are trying to sort and we require the next largest item be removed next.

In summary, the work for heap sort is the sum of the two stages: *O(n) time for buildHeap and* ***O(n log n) to remove each node in order****, so the complexity is O(n log n)*. You can prove (using some ideas from information theory) that for a comparison-based sort, *O(n log n)* is the best you could hope for anyway, so there's no reason to be disappointed by this or expect heap sort to achieve the O(n) time bound that `buildHeap` does.

edited May 20 '16 at 18:26

answered Sep 11 '13 at 13:22

Jeremy West

**2,673** ● 1 ● 7 ● 21

6 Great answer, for those reading through this question in the future, I think this answer gives the most thorough and intuitive explanation. – JKillian Aug 15 '14 at 3:28

A good answer indeed. A minor comment though: @jeremy should better mention at early of his answer that this answer is talking about a min heap, not the textbook max heap. – RayLuo Nov 6 '14 at 4:35

1 This is what made it intuitively clear to me: "only one node is at the top whereas half the nodes lie in the bottom layer. So it shouldn't be too surprising that if we have to apply an operation to every node, we would prefer siftDown over siftUp." – Vicky Chijwani Dec 25 '14 at 18:42

1 honestly, this answer should be marked as correct one. – Jackson Tale Feb 10 '15 at 17:44

1 @aste123 No, it is correct as written. The idea is to maintain a barrier between the part of the array that satisfies the heap property and the unsorted portion of the array. You either start at the beginning moving forward and calling `siftUp` on each item or start at the end moving backward and calling `siftDown`. No matter which approach you choose, you are selecting the next item in the unsorted portion of the array and performing the appropriate operation to move it into a valid position in the ordered portion of the array. The only difference is performance. – Jeremy West Dec 16 '15 at 22:05

## Intuitively:

> "The complexity should be O(nLog n)... for each item we "heapify", it has the potential to have to filter down once for each level for the heap so far (which is log n levels)."

Not quite. Your logic does not produce a tight bound -- it over estimates the complexity of each heapify. If built from the bottom up, insertion (heapify) can be much less than `O(log(n))`. The process is as follows:

**( Step 1 )** *The first* `n/2` *elements go on the bottom row of the heap.* `h=0`*, so heapify is not needed.*

**( Step 2 )** *The next* $n/2^2$ *elements go on the row 1 up from the bottom.* `h=1`*, heapify filters 1 level down.*

**( Step *i* )** *The next* $n/2^i$ *elements go in row* `i` *up from the bottom.* `h=i`*, heapify filters* `i` *levels down.*

**( Step *log(n)* )** *The last* $n/2^{\log_2(n)} = 1$ *element goes in row* `log(n)` *up from the bottom.* `h=log(n)`*, heapify filters* `log(n)` *levels down.*

**NOTICE:** that after step one, `1/2` of the elements `(n/2)` are already in the heap, and we didn't even need to call heapify once. Also, notice that only a single element, the root, actually

incurs the full `log(n)` complexity.

---

## Theoretically:

The Total steps `N` to build a heap of size `n`, can be written out mathematically.

At height `i`, we've shown (above) that there will be $n/2^{i+1}$ elements that need to call heapify, and we know heapify at height `i` is `O(i)`. This gives:

$$N = \sum_{i=0}^{log(n)} \frac{n}{2^{i+1}} i = \frac{n}{2} \left( \sum_{i=0}^{log(n)} i \left(1/2\right)^i \right) \leq \frac{n}{2} \left( \sum_{i=0}^{\infty} i \left(1/2\right)^i \right)$$

The solution to the last summation can be found by taking the derivative of both sides of the well known geometric series equation:

$$\frac{\partial}{\partial x} \left( \sum_{i=0}^{\infty} x^i \right) = \frac{\partial}{\partial x} \left( \frac{1}{1-x} \right) \quad \Rightarrow \quad \sum_{i=1}^{\infty} i x^i = \frac{x}{\left(1-x\right)^2}$$

Finally, plugging in `x = 1/2` into the above equation yields `2`. Plugging this into the first equation gives:

$$N \leq \frac{n}{2} \left(2\right) = n$$

Thus, the total number of steps is of size `O(n)`

edited Jan 15 at 17:51

answered Aug 18 '13 at 3:13

bcorso
23.8k ● 6 ● 42 ● 56

---

It would be O(n log n) if you built the heap by repeatedly inserting elements. However, you can create a new heap more efficiently by inserting the elements in arbitrary order and then applying an algorithm to "heapify" them into the proper order (depending on the type of heap of course).

See http://en.wikipedia.org/wiki/Binary_heap, "Building a heap" for an example. In this case you essentially work up from the bottom level of the tree, swapping parent and child nodes until the heap conditions are satisfied.

answered Mar 18 '12 at 3:36

mike__t
719 ● 4 ● 6

---

While building a heap, lets say you're taking a bottom up approach.

1. You take each element and compare it with its children to check if the pair conforms to the heap rules. So, therefore, the leaves get included in the heap for free. That is because they have no children.
2. Moving upwards, the worst case scenario for the node right above the leaves would be 1 comparison (At max they would be compared with just one generation of children)
3. Moving further up, their immediate parents can at max be compared with two generations of children.
4. Continuing in the same direction, you'll have log(n) comparisons for the root in the worst case scenario. and log(n)-1 for its immediate children, log(n)-2 for their immediate children and so on.
5. So summing it all up, you arrive on something like log(n) + {log(n)-1}*2 + {log(n)-2}*4 + ..... + 1*2^{(logn)-1} which is nothing but O(n).

answered Jul 3 '12 at 10:44

Jones
86 ● 3 ● 7 ● 20

---

As we know the height of a heap is **log(n)**, where n is the total number of elements.Lets represent it as **h**

When we perform heapify operation, then the elements at last level(**h**) won't move even a single step.

The number of elements at second last level(**h-1**) is $2^{h-1}$ and they can move at max **1** level(during heapify).

Similarly, for the $i^{th}$, level we have $2^i$ elements which can move **h-i** positions.

Therefore total number of moves=$S= 2^h*0+2^{h-1}*1+2^{h-2}*2+...2^0*h$

$S=2^h \{1/2 + 2/2^2 + 3/2^3+ ... h/2^h\}$ ----------

$S/2=2^h \{1/2^2 + 2/2^3+ ... h/2^{h+1}\}$ ----------

subtracting equation **2** from **1** gives

$S/2=2^h \{1/2+1/2^2 + 1/2^3+ ...+1/2^h+ h/2^{h+1}\}$

$S=2^{h+1} \{1/2+1/2^2 + 1/2^3+ ...+1/2^h+ h/2^{h+1}\}$

now $1/2+1/2^2 + 1/2^3+ ...+1/2^h$ is decreasing **GP** whose sum is less than **1** (when h tends to infinity, the sum tends to 1). In further analysis, let's take an upper bound on the sum which is 1.

This gives $S=2^{h+1}\{1+h/2^{h+1}\}$

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

$\sim 2^h+h$

as $h=\log(n)$, $2^h=n$

Therefore $S=n+\log(n)$

$T(C)=O(n)$

edited Apr 12 at 2:11
Shikhar Gupta
58 ● 1 ● 7

answered Feb 20 at 20:14
Tanuj Yadav
939 ● 5 ● 18

---

Successive insertions can be described by:

```
T = O(log(1) + log(2) + .. + log(n)) = O(log(n!))
```

By starling approximation, `n! =~ O(n^(n + O(1)))`, therefore `T =~ O(nlog(n))`

Hope this helps, the optimal way `O(n)` is using the build heap algorithm for a given set (ordering doesn't matter).

edited Jul 19 '14 at 12:09
Matt ♦
57.8k ● 15 ● 108 ● 151

answered Sep 14 '13 at 10:45
Tomer Shalev
31 ● 5

---

In case of building the heap, we start from height, **logn -1** (where logn is the height of tree of n elements). For each element present at height 'h', we go at max upto (logn -h) height down.

```
So total number of traversal would be:-
T(n) = sigma((2^(logn-h))*h) where h varies from 1 to logn
T(n) = n((1/2)+(2/4)+(3/8)+.....+(logn/(2^logn)))
T(n) = n*(sigma(x/(2^x))) where x varies from 1 to logn
 and according to the [sources][1]
function in the bracket approaches to 2 at infinity.
Hence T(n) ~ O(n)
```

answered Aug 20 '15 at 9:26
Kartik Goyal
311 ● 2 ● 11

---

@bcorso has already demonstrated the proof of the complexity analysis. But for the sake of those still learning complexity analysis, I have this to add:

The basis of your original mistake is due to a misinterpretation of the meaning of the statement, "insertion into a heap takes O(log n) time". Insertion into a heap is indeed O(log n), but you have to recognise that n is the size of the heap **during the insertion**.

In the context of inserting n objects into a heap, the complexity of the ith insertion is O(log n_i) where n_i is the size of the heap as at insertion i. Only the last insertion has a complexity of O (log n).

answered Jul 9 at 1:28
N.Vegeta

 46 ● 1 ● 3

---

I really like explanation by Jeremy west.... another approach which is really easy for understanding is given here
http://courses.washington.edu/css343/zander/NotesProbs/heapcomplexity

since, buildheap depends using depends on heapify and shiftdown approach is used which depends upon sum of the heights of all nodes. So, to find the sum of height of nodes which is given by S = summation from i = 0 to i = h of (2^i*(h-i)), where h = logn is height of the tree solving s, we get s = 2^(h+1) - 1 - (h+1) since, n = 2^(h+1) - 1 s = n - h - 1 = n- logn - 1 s = O(n), and so complexity of buildheap is O(n).

edited May 26 '14 at 14:31

answered May 26 '14 at 9:19


Nitish Jain
134 ● 4 ● 14

---

"The linear time bound of build Heap, can be shown by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines. For the perfect binary tree of height h containing N = 2^(h+1) – 1 nodes, the sum of the heights of the nodes is N – H – 1. Thus it is O(N)."

answered Jan 15 '15 at 1:17


sec3
1

---

Basically, work is done only on non-leaf nodes while building a heap...and the work done is the amount of swapping down to satisfy heap condition...in other words (in worst case) the amount is proportional to the height of the node...all in all the complexity of the problem is proportional to the sum of heights of all the non-leaf nodes..which is (2^h+1 - 1)-h-1=n-h-1= O(n)

answered May 31 '15 at 20:15


Shubham Jindal
1

---

think you're making a mistake. Take a look at this: http://golang.org/pkg/container/heap/
Building a heap isn'y O(n). However, inserting is O(lg(n). I'm assuming initialization is O(n) if you set a heap size b/c the heap needs to allocate space and set up the data structure. If you have n items to put into the heap then yes, each insert is lg(n) and there are n items, so you get n*lg(n) as u stated

answered Mar 18 '12 at 3:30


Mike Schachter
124 ● 7

1 no it is not tight. tighter analysis of build heap yields O(n) – emre nevayeshirazi Mar 18 '12 at 3:31

it looks like that's an estimation. The quote in thearticle he referenced is "The intuition is that most of the calls to heapify are on very short heaps" However this is making some assumptions. Presumably, for a large heap, the worst case scenario would still be O(n*lg(n)), even if usually you could get close to O(n). But I could be wrong – Mike Schachter Mar 18 '12 at 3:34

Yes that is my intuitive answer as well, but references such as wikipedia state "Heaps with n elements can be constructed bottom-up in O(n)." – GBa Mar 18 '12 at 3:35

1 I was thinking of a fully sorted data structure. I forgot the specific properties of a heap. – Mike Schachter Mar 18 '12 at 3:41

---