## **Build Heap - Heapify**

Heapify is an algorithm where we build a heap from scratch. There are generally two methods:

1. Use inserts with percolateUp
2. Build complete tree then percolateDown

Which is better?

## **Building a Binary Heap Using PercolateUp - buildHeap(E[] items)**

* A **binary heap** can be constructed if we have an initial collection of items.
* Suppose you have unordered array of n items you want to make into a priority queue.
  + We can perform a sequence of n insert() operations
  + best: O(N), average: O(N), worst(NlogN)
* Each add(E e) will take
  + O(1) average

and

* + O(log N) worst-case time
* Therefore, the total running time of buildHeap would be
  + O(N) average

and

* + O(N log N) worst-case
* The worst-case runtime for this algorithm is O(N/2 log N) = O(N log N) because we are percolating N/2 nodes up logN levels.
* This runtime is a result of most of the nodes in the heap being near the bottom, and we make them all go all the way up.
* What if instead we tried to percolate things down?

## **Floyd’s Build Heap – heapify(E[] items)**

* Recall our general strategy for working with the heap:
  + Preserve complete tree structure property
  + Break and restore heap ordering property

Floyd’s **buildHeap**:

1. Create a complete tree by putting the n items in array indices 1, . . .. N

(Requires having all the elements that we want to insert all at once!)

1. Treat the array as a heap and fix the heap-order property

(Exactly how we do this is where we gain efficiency)

* Main Idea: Percolate down all non-leaves (N/2 node from end of array).
  + Leaves are already in heap order
  + First non-leaf is at N/2
  + Work up towards the root one level at a time

## **Efficiency of Floyd’s Build Heap**

* There are two ways we could percolate down:

1. Percolate down from the **top** of the binary heap
2. Percolate down from the **bottom** of the binary heap

* Percolate Down starting from the top of the heap will fail.
* Percolate Down starting from the bottom will work.

**First Look: O(n log n)**

* **buildHeap** is *O*(*n* **log** *n*) where *n* is **size** 
  + **n/2** loop iterations: O(n)
  + **percolateDown** on each loop iteration: O(log n)

This is correct, but there is a more precise (“tighter”) analysis of the algorithm...

**Second Look: O(n)**

* **buildHeap** is *O*(*n*) where *n* is **size**
* Running percolate down from the bottom of the binary heap ensures that the **largest layer has the least distance to percolate**.
  + The bottom layer (leaves) has a N / 2 (half) of the total nodes in the heap. Therefore, 1/2 of the nodes percolate at most 0 levels.
  + The next layer up has N / 4 of the total nodes in the heap.

Therefore, 1/4 of the nodes percolate at most 1 level.

* + The next layer up has N / 8 of the total nodes in the heap.

Therefore, 1/8 of the nodes percolate at most 2 levels.

* + The next layer up has N / 16 of the total nodes in the heap.

Therefore, 1/16 of the nodes percolate at most 3 levels.

… and so on

* In summary:
  + 1/2 the loop iterations percolate at most 1 step
  + 1/4 the loop iterations percolate at most 2 steps
  + 1/8 the loop iterations percolate at most 3 steps
  + ... etc.
* ((**1**/2) + (**2**/4) + (**3**/8) + (**4**/16) + (**5**/32) + ...) = 2
* So at most 2(size/2) total percolate steps: O(n)

|  |  |  |
| --- | --- | --- |
| Height | Number of Nodes at This Level | Max Number of Levels Nodes Can Percolate |
| 0 | N / 16 | 3 |
| 1 | N / 8 | 2 |
| 2 | N / 4 | 1 |
| 3 | N / 2 | 0 |

Chart, line chart

Description automatically generatedChart, scatter chart

Description automatically generatedChart, scatter chart

Description automatically generatedChart, scatter chart

Description automatically generated

0

1

2

3

Chart, scatter chart

Description automatically generated

* Since we know the bottom layer of nodes will not percolate down any levels (because they are leaf nodes), and the total number of nodes in the bottom layer is N / 2, **we only have to percolateDown the upper half (first half of nodes) of the tree.**
* Therefore, we can start looping from the N / 2 node in the heap.

private void buildHeap() {

for(i = size/2; i>0; i--) {

hole = percolateDown(i);

}

}

**Floyd’s buildHeap is O(n) where n is array size:**

* n/2 total loop iterations: O(n)
  + 1/2 of the loop iterations percolate at most 1 level
  + 1/4 of the loop iterations percolate at most 2 levels
  + 1/8 of the loop iterations percolate at most 3 levels
* We know (1 + (**1**/2) + (**2**/4) + (**3**/8) + ...) = 2
* This means that nodes percolate down an average of 2 levels.
* Therefore, Floyd’s buildHeap is n/2 \* 2 = O(N) runtime

Shape

Description automatically generated

* In the following example, the size of this heap is 12. Node 2 is the 6th node in the tree.
* Therefore, we starting percolating down from Node 2.

Chart, scatter chart

Description automatically generated

**Summary**

* We can build a complete binary tree in linear time by simply making a copy of the items array and saving it into the heap object.
* Then, we restore heap-order by using percolateDown(int i) for each non-leaf node starting with the last node and ending with the root.
* After each percolation, the subtree rooted at the percolated node is a valid min-heap!
* This makes the complete binary tree into a heap-ordered tree once all nodes are percolated.

**Percolate Half of Nodes**

* Looking at Figure 6.14, we see that we perform size/2 percolateDown(int i) operations. But how do we know size/2 is the right place to start?
* We don’t need to worry about percolating any of the leaf nodes, because they have no children to percolate down to.
* Each level has a power of 2 number of nodes. Therefore, the first node before the last level is at size / 2.
* Let m = size/2. The mth node is guaranteed to have children at 2m and (possibly) 2m+1.
* In either case, the mth node’s children are are the last nodes in the tree, since 2m = size (or size – 1 using integer division)

**Code**

Text

Description automatically generated

**Example - heapify**

* The following figures are steps in the heapify process to build a heap-ordered tree from an unordered complete tree.
* Each dashed line corresponds to a percolateDown
  + One comparison to find the smaller child

and

* + One comparison to compare the smaller child with the current node
* There are a total of seven percolateDowns that we perform, because there are a total of 2h+1-1 = 15 nodes in this tree (perfect tree).
* Notice that there are only 10 dashed lines in the entire algorithm (there could have been an 11th—where?) corresponding to 20 comparisons.

1. The first tree in Figure 6.15 is the unordered tree.

15 / 2 = 7, which means we perform the first percolateDown at the 7th node with value 110.

A picture containing text, watch

Description automatically generated

**Runtime of heapify**

* To bound the running time of heapify, we must bound the number of dashed lines.
* This can be done by computing the sum of the heights of all the nodes in the heap, which is the maximum number of dashed lines. What we would like to show is that this sum is O(N).