1. For the following array

40	6	3	11	2	4

a) create a max heap using the algorithm we discussed in class (BUILD-MAX-HEAP)

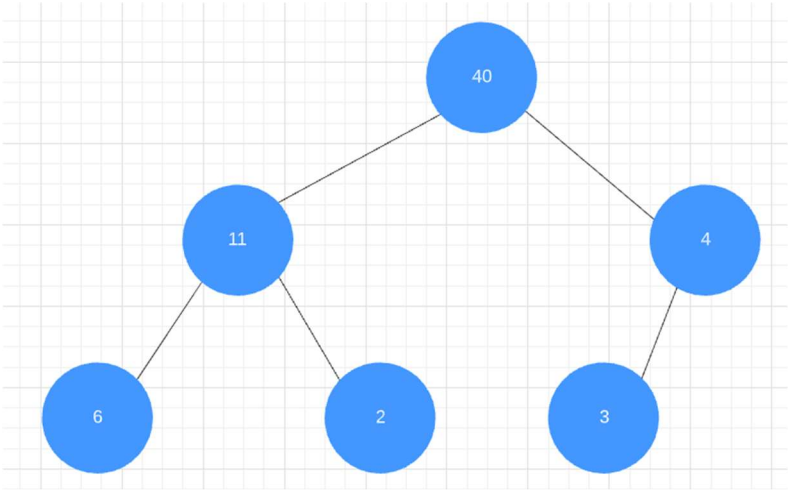
BUILD-MAX-HEAP(A)

A.heap-size = A.length

for i = $\lfloor$A.length/2$\rfloor$ downto 1

MAX-HEAPIFY(A, i)

The order of MAX-HEAPIFY would call (A, 3), (A, 6), (A, 40)



b) remove the largest item from the max heap you created in 1a, using the HEAP-EXTRACTMAX function from the book. Show the array after you have removed the largest item

Extracts 40 from the array by copying the last node in the array. 40 would be copied by 3.

Max = A[1]

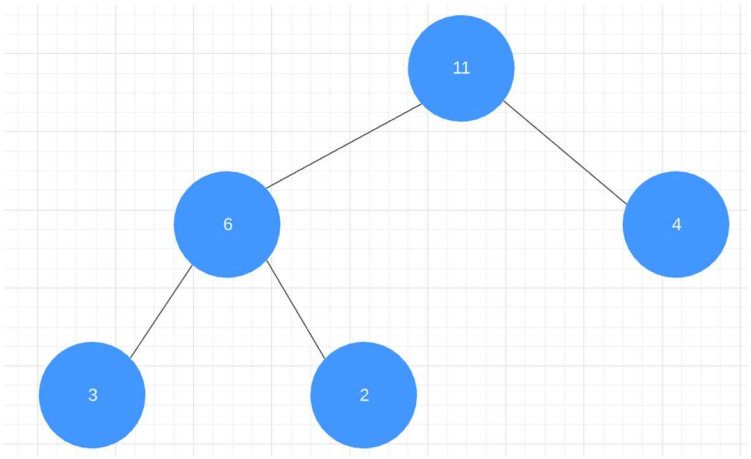
A[1]=A[A.heap-size]

3	11	4	6	2	3

A.heap-size = A.heap-size -1

3	11	4	6	2

MAX-HEAPIFY(A, 1) -> MAX-HEAPIFY 3 -> MAX-HEAPIFY A[3] or 3



11	6	4	3	2

c) Using the algorithm from the book, MAX-HEAP-INSERT, insert 56 into the heap that resulted from question 1b. Show the array after you have inserted the item.

Original heap array

3	11	4	6	2

MAX-HEAP-INSERT(A, key)

A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)

56 will be inserted at the end of the array as -∞

A[i] = key

while i > 1 and A[PARENT(i)] > A[i]

exchange A[i] with A[PARENT(i)]

i = PARENT(i)

11	6	4	3	2	56

A[3] : 4 < A[6]: 56

Exchange A[6] with A[3]

11	6	56	3	2	4

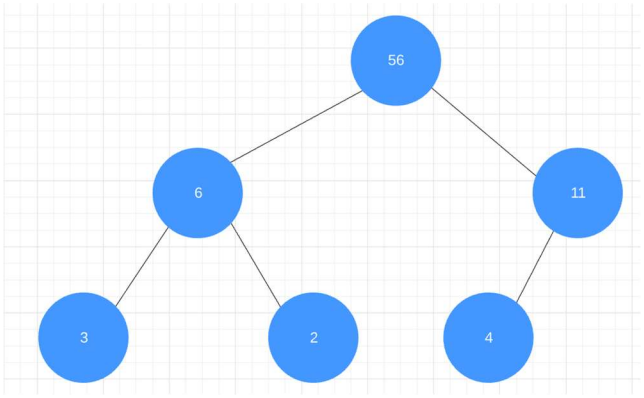
A[3]: 56 < A[1]: 11

Exchange A[3] with A[1]

Brandon Vo

Resulting array:

56	6	11	3	2	4



2. Consider of 3-ary max-heap. A 3-ary max-heap is like a binary max-heap, but instead of 2 children, nodes have 3 children.

(a) How would you represent a 3-ary max-heap in an array?

- The root would still begin at A[1].
- The left child of the parent would be located at $A[i * 3 - 1]$
- The middle child of the parent would be located at $A[i * 3]$
- The right child of the parent would be located at $A[i * 3 + 1]$
- For a child node, their parent would be located at $A[\lfloor \frac{i+1}{3} \rfloor]$

(b) What is the height of a 3-ary max-heap of n elements in terms of n and 3?

The height of a binary tree has two nodes for each parent which has $\log_2 n$. For 3 nodes, they would affect the modifier for the log n. With three nodes for each parent, that means 3 affects the log n implementation.

$$Height = \log_3 n$$

(c) Give an efficient implementation of HEAP-EXTRACT-MAX. Analyze its running time in terms of 3 and n.

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size -1
MAX-HEAPIFY(A, 1)
return max

MAX-HEAPIFY(A, i)
    l = LEFT(i)
    m = MIDDLE(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
```

else largest = i

if m ≤ A.heap-size and A[m] > A[largest]

largest = m

else largest = i

if r ≤ A.heap-size and A[r] > A[largest]

largest = r

if largest ≠ i

exchange A[i] with A[largest]

MAX-HEAPIFY(A, largest)

Runtime: The MAX-HEAPIFY function percolates upward from the leaf level of the 3-ary heap tree and can go up to the root level during the worst case-scenario, making it dependent on the size of the 3-ary heap tree.

$$O(\log_3 n)$$

(d) Give an efficient implementation of MAX-HEAP-INSERT. Analyze its running time in terms of 3 and n.

MAX-HEAP-INSERT(A, key)

A.heap-size = A.heap-size + 1

A[A.heap-size] = $-\infty$

HEAP-INCREASE-KEY(A, A.heap-size, key)

if key > A[i]

error "new key is larger"

HEAP-INCREASE-KEY(A, i, key)

A[i] = key

while i > 1 and A[PARENT(i)] < A[i]

exchange A[i] with A[PARENT(i)]

i = PARENT(i)

This procedure will insert the new element at the bottom of the heap tree then run HEAP-INCREASE-KEY. HEAP-INCREASE-KEY assigns the value to the inserted element and then repeatedly swaps the element upward until it passes the conditions for a max-heap subtree. This swap can repeat up until it reaches the root tree, meaning that the worse case time-complexity depends on the height of the binary tree.

$$O(\log_3 n)$$

(e) Give an efficient implementation of HEAP-INCREASE-KEY(A, i, k). Analyze its running time in terms of 3 and n.

HEAP-INCREASE-KEY(A, i, key)

A[i] = key

while i > 1 and A[PARENT(i)] < A[i]

    exchange A[i] with A[PARENT(i)]

    i = PARENT(i)

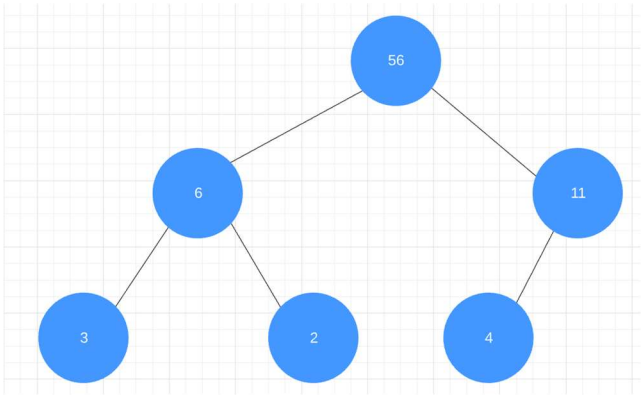
Because this swapping procedure will repeat as for each level in the heap tree, the worst case time complexity will last the height of the heap tree.

$$O(\log_3 n)$$

Brandon Vo

3. Show that the number of leaves in a heap is $\lceil \frac{n}{2} \rceil$. (See if you can find a simple answer based on the lecture.)

For a child node in a heap, a parent can be found at $\lfloor n/2 \rfloor$. The last node in the heap array will correspond to the last parent of the previous layer.



56	6	11	3	2	4

As shown above, the last leaf in the array is 4 which corresponds to the parent node of 11. A[6] leads to A[3]. A[3] is the last parent node of the layer, meaning that every node after that will be a leaf node.

Every node from A[n/2+1…A.length] are all leaf nodes which is the second-half of the heap array. This can be shown by how after A[3]: A[4], A[5], and A[6] which are 3, 2, and 4 are all leaf nodes. This shows that A[n/2] is the last parent node which means that $\lceil \frac{n}{2} \rceil$ nodes in the array represents the leaves found in the last layer of the heap tree.

4. Prove the correctness of HEAP-DECREASE-KEY using a loop invariant.

```
HEAP-DECREASE-KEY(A, i, key)
if key > A[i]
        error "new key is larger"
A[i] = key
while i > 1 and A[PARENT(i)] > A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

- Loop Invariant Property: The loop invariant condition is checking if the min heap property is valid for the entire tree when checking to swap A[i] with A[PARENT(i)].

- **Initialization**-When the loop begins, the entire array should be maintaining its heap property with the exception of A[i] and A[PARENT(i)] as our loop will be checking with A[i] maintains the max heap property with A[PARENT(i)].
    - When we start the loop, the entire heap should already have its min heap property pre-established due to the heap being preprocessed. A[i] may or may be maintaining its max-heap property as A[i] has been given a newly assigned key value and we have not checked its new placement yet.
- **Maintenance**-For each step, the element A[i] is checked with its parent at A[i/2].
    - If the min-heap property doesn't hold up for the subtree, then A[i] is swapped with A[i/2]. If there is a swap, A[i] takes the index position to become the parent while A[i/2] takes the position of being the child node. The loop must repeat as it is still uncertain if the node's new position validates the min-heap property or not. The entire heap tree, on the other hand, still maintains its heap property with the exception of the node with the new value.
    - If the min-heap property was found to be true, then there is no swap. In this case, it has been proven that the entire tree holds up the min-heap property, proving the loop invariant to be true.
- **Termination**-The loop terminates when the child node can no longer swap with its parent node or it has reached the root node. This means the max heap property is now validated and maintained by the heap tree, including the node with the new value as well. This means the loop invariant has been proven now that the entire tree's heap property is maintained.

5. For the following algorithm:
   a) Write the recurrence formula for the following function. Each node should hold the cost of that recursive call when you don't consider the cost to compute the recursive subproblems.[1]
   b) Draw the recursion tree for the following function and show the running time for each level. In your recursion tree, show the top 3 levels of the recursion tree and the bottom level of the recursion tree. Put "..." (dots) to represent the levels you didn't show.

```
minimum(A, l, r)
1) if (r − l == 0)
2)         return A[r]
3)
4) lmin = minimum(A, l, b(l + r)/2c)
5) rmin = minimum(A, b(l + r)/2c + 1, r)
6) print(rmin, lmin)
7) if rmin < lmin
8)         return lmin
9) else
10)        return rmin
```

2T(n/2)

O(1)·C

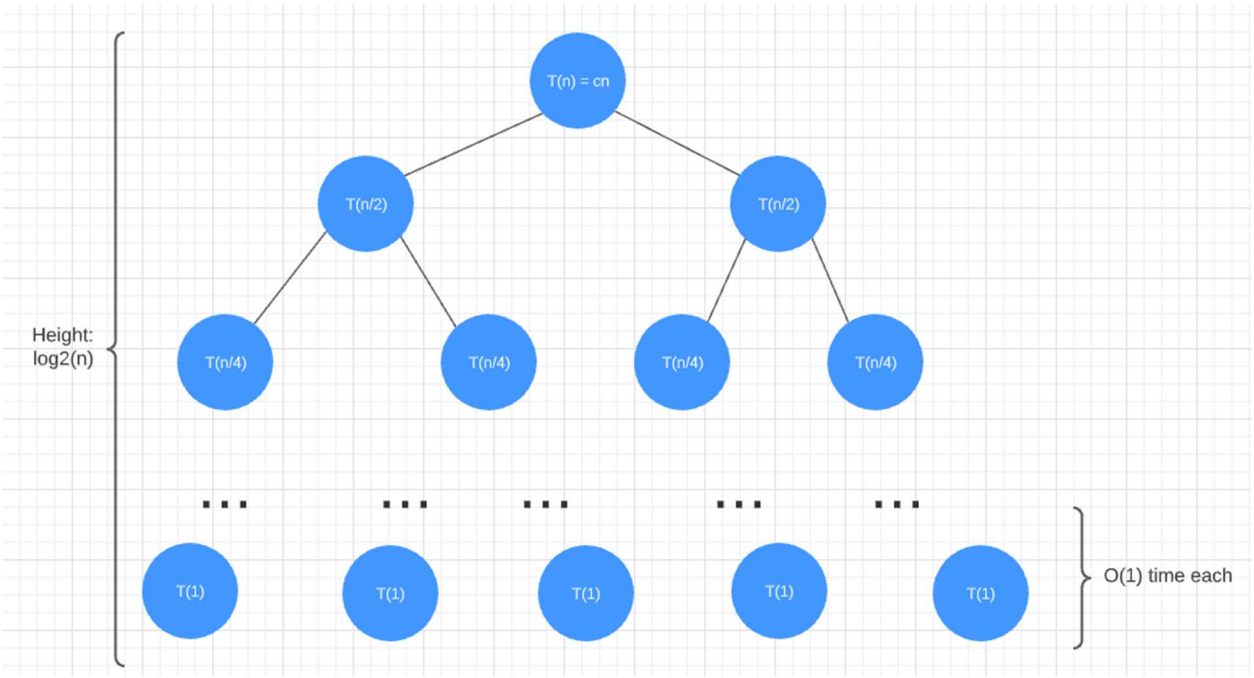
a)

- Each function calls upon two more recurrence functions until the base case is met: $2T\left(\frac{n}{2}\right)$.
- The root function costs O(1) time to do a comparison between lmin and rmin. This would take a constant time *c*.

$$T(n) = 2T\left(\frac{n}{2}\right) + c * O(1)$$

Brandon Vo

b)


T(n) = cn
T(n/2)
T(n/2)
T(n/4)
T(n/4)
T(n/4)
T(n/4)
Height:
log2(n)
T(1)
T(1)
T(1)
T(1)
T(1)
O(1) time each

6. Now that you're a graduate student, you are bombarded with things to do! You have to make sure you play FFXIV, re-clear Skyrim for the 300th time, and do your homework assignment for CS6033. Of course, as you are trying to finish all your tasks a new task might arise (your roommate might hit you up for a game of League, or you might remember you promised to call your parents last weekend) and you have to make sure it gets done.

   You decide to keep working until all the tasks are done because you want to enjoy your weekend. In an attempt to build momentum, you decide to complete the tasks in order of the time you estimated they would take.

   Devise an algorithm that will tell you the next task to do in O(log n) time, as well as integrate new tasks in O(log n).

   Justify the running time of your algorithm.

We can build a min-priority queue with the function to insert a new task, decrease-key, and extract min to utilize our schedule.

```
HEAP-EXTRACT-MIN (A)
if A.heap-size < 1
        error "heap underflow"
min= A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MIN-HEAPIFY(A, 1)
        return min

MIN-HEAPIFY (A, i)
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] < A[i]
        smallest = l
else smallest = i
if r ≤ A.heap-size and A[r] < A.smallest
```

```
        smallest = r
if smallest ≠ i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY (A, smallest)
```

To acquire the next task to do, the priority queue can execute HEAP-EXTRACT-MIN and then reheapify the structure. HEAP-EXTRACT-MIN would take the task that requires the least amount of time for us to work on. Then, to prevent doing the same task again, the heap will remove the min root node and call MIN-HEAPIFY.

MIN-HEAPIFY would reorder the heap to maintain the heap tree structure and place the next task that requires the shortest amount of time to complete as the new root of the min-heap, providing us with the next task to do when needed. The process of MIN-HEAPIFY would recursively check and swap from the root node down to the leaves of a tree, meaning that MIN-HEAPIFY takes O(log n) time.

This means that the entire process of HEAP-EXTRACT-MIN takes O(log n) time to take the next task and prepare the next task for us to choose.

```
MIN-HEAP-INSERT(A, key)
        A.heap-size = A.heap-size + 1
        A[A.heap-size] = ∞
        HEAP-DECREASE-KEY(A, A.heap-size, key)
```

```
HEAP-DECREASE-KEY(A, i, key)
A[i] = key
while i > 1 and A[PARENT(i)] > A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

To integrate new tasks, we can use implement MIN-HEAP-INSERT. MIN-HEAP-INSERT places a new task at the bottom of the heap then calls on HEAP-DECREASE-KEY. The process of inserting the new task at the bottom of the heap by itself would be O(1). When HEAP-DECREASE-KEY is called, it will assign the actual time value needed to complete the task then recursively swap the new task with its parent node until the new node is larger than its parent at which point it will be at its proper position in the list of tasks to complete.

The process of checking and swapping the new task with its parent begins from the leaf level and may percolate upward to the root node if needed, costing O(log n) time.

7. Last week when you asked for advice, you didn't think you would get so many suggestions. Due to not wanting to lose most of your money in broker fees,[2] you don't want to invest less than 100$ per stock, so you decide to buy the B < N stocks that did the best in the past 6 months time. Since the advice keeps flowing in and you decide to invest your money next Monday, you want to quickly be able to determine the best stocks to buy in O(B) time and insert a new stock suggestion in O(log N) time. Partial credit for determining the top B stocks in O(B log N) time and inserting a new stock in O(log N) time.[3]

   For this question, you may assume the stock names you have already removed any duplicate names.

Assuming that the set of N stocks are arranged such that their value is tied to performance, we can create a preprocessed max-heap for the set of N stocks. The value for each node representing the stocks represents the amount of value and performance those stocks have gained over the 6-month time period.

Pseudocode:

- There will be two stock heaps:
  - Stock heap B will contain all the B stocks that were performing the best within the past 6 months.
    - Heap B will be a min heap.
    - Will also be assuming that B will be a fixed size that's already full during preprocessing.
  - Stock heap A will contain the rest of the stocks from N.
    - Heap A will be a max heap.

```
INSERT-NEW-STOCK(A, B, key)
    INSERT-B(B, key)                    //Insert it into B first then see if it will be
                                        moved to Heap A
    Temp = HEAP-EXTRACT-MIN(B)          //Assuming B must be a fixed size,
                                        To maintain B's size, we take the smallest
                                        value from B and put it in A. This costs
                                        O(log b) time.
    INSERT-A(A, Temp)       //The process of inserting a new node into Heap A costs
                            O(log n) due to A holding the remainder of the N stocks
```

```
INSERT-A(A, key)
        A.length = A.length + 1
        A[length] = -∞
        HEAP-INCREASE-KEY(A, A.length, key)  //A is a max-heap.  Using the same function
                                              from the previous question
INSERT-B(B, key)
        B.length = B.length + 1
        B[length] = ∞
        HEAP-DECREASE-KEY(B, B.length, key)  //B is a min-heap
                                              /Using the same function from the previous
                                              question


HEAP-EXTRACT-MIN (B)
if B.heap-size < 1
        error "heap underflow"
min= B[1]
B[1] = B[B.heap-size]
B.heap-size = B.heap-size - 1
MIN-HEAPIFY(B, 1)
        return min


HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size -1
MAX-HEAPIFY(A, 1)
```

return max

```
ACQUIRE-STOCK-INVESTMENT-LIST(B)        //Get every stock value in the B heap
    Total = 0
    For i = 1 to B.length               //This goes through the entire B heap
                                        array costing O(B) time.
        Total = Total + B[i]
    Return Total
```

Preprocessing: We can create two heaps: One heap to contain the list of B stocks and another heap containing the list of N stocks where B < N. Heap B would hold onto the list of B stocks taken from N as its own separate heap. The stocks in Heap B would be the best performing stocks as a Min-Heap. Heap A would hold onto the rest of the (N-B) stocks as a max-heap.

Finding the best stocks: Due to the preprocessing step, the best-performing stocks are already contained within the B heap. To determine the best stocks, we can simply traverse the entirety of the B heap and collect the names and values of all the stock options found. Because this heap would be of size B and we would have to travel through the entire heap of B, this would take O(B) time.

Inserting a new stock suggestion: To insert a new stock, we would have to decide whether to insert it into the heap A or heap B. What we can do is insert the element into Heap B then extract the minimum value and place it in A.

- Heap B would be the min-heap which means that if the new stock option has a value greater than Heap B's root, then it's large enough to be a part of the list of B stocks to invest into. Otherwise, the new stock option will be placed at the root where it will be extracted and placed into Heap A.
- If we're adamant about B maintaining a fixed size, then we would have to insert the new element and then move the smallest element in heap B to place in Heap A.
    - We can insert an object into Heap B, reheapify the structure to mark all node's positions then extract the smallest element from B and move it to Heap A. The process of inserting a new element, extracting the min, and reheapifying Heap B would cost O(log b) time because we will be percolating upward from Heap B's tree
- The root element being extract will be inserted into Heap A which will call the reheapify process on Heap A and cause the element to percolate upward on Heap A's tree.
    - Because this process depends on Heap A's size, this would mean going up the tree of size (N-B).

    - This would mean inserting the element into Heap A would cost O(log n) time.
- This leaves us with O(log b) and O(log n) time to insert a new element into Heap B and move an element from B to A. Because B < N, asymptotically speaking, O(log n) will cost more in terms of time complexity and will overtake O(log b).
- This process of inserting a new element will take O(log n) time.

8. Just out of curiosity, you would like to know the average return of the top 1/4 stock picks every week (based on how well they performed in the past 6 months as of last Friday). You are impatient and want the answer in O(1) time. The time to insert a new stock suggestion must run in O(log N) time, in order to keep the costs of recomputing the existing picks in the system over the weekend and the costs of inserting new entries during the week low. For this question, you may assume the stock names you have already removed any duplicate names.

Preprocessing: We can establish two heaps: Heap A will carry the top ¼ stock picks while the Heap B will carry the remaining ¾ stock options that weren't performing as well.

- Heap B containing the rest of the ¾ stock picks will be constructed as a max-heap.
- Heap A containing the ¼ top stock picks will be a min-heap.
- In addition, Heap A will hold an additional node which would hold the average values of all the stock options. This node containing the average values will be placed at the root node.

Acquiring the average of the top ¼ stocks: Because we have a pre-processed heap holding the top ¼ stock options, we will also pre-prepare the average values of all the stock options and use it as the root node. The stock options themselves will be the children nodes of the node holding the averages.

To get the averages of the top ¼ stock options, we will read the root value of the heap, costing O(1) time.

```
STOCK-AVERAGE(A)

    Return A[1]
```

Inserting a new stock suggestion: To insert a new stock, we would first need to check if the stock needs to be placed into the ¾ stock heap or the ¼ stock heap. It will be compared with the root of the ¾ stock heap.

- If the new stock value is larger than Heap B's root, then it will be placed into the 1/4 stock heap.
- We would also have to compare the new stock option with the two children of the root node since the root node is holding the averages of the entire heap. If the new stock option is greater than the root's children, then it will be placed in the ¼ stock heap.
- In either case, we will insert it into the heap by placing it at the bottom of the heap and calling the MIN-INCREASE-KEY or MAX-INCREASE-KEY to determine where the stock option should be placed in the array.
- During the process, it will percolate up one of the two tree which would cost O(log n) time complexity for both trees on average.
- This marks the process of inserting a new stock option into one of the two heaps to cost O(log n) time.

```
STOCK-INSERT (A, B, key):
    If key > B[1]   //Place it in the ¼ stock heap.  B is a max-heap.
        INSERT_A(A, key)
    Else
        INSERT_B(B, key)

INSERT_A(A, key)
    A.heap-size = A. heap-size + 1
    A[A. heap-size] = ∞   //A heap is a min-heap
    HEAP-DECREASE-KEY(A, key)
    If A.heap-size is > ¼ of the total of the stock options  //Must maintain the ¼ size
        Temp = EXTRACT-MIN-SIZE(A)
        INSERT-B(B, Temp)                    //We must take an element from A
                                             and put it in B to maintain the ¼ size of A
    RECALCULATE-AVERAGE(A, key)

INSERT-B(B, key)                    //B is a max-heap
    B.heap-size = B. heap-size + 1
    B[B.heap-size] = -∞
    HEAP-INCREASE-KEY(B, key)
```

```
HEAP-EXTRACT-MIN(A)                    //Because the root holds the total average, we must check the child nodes

if A.heap-size < 1
        error "heap underflow"

min = 0
 if A[2] < A[3]                        //Assuming A[2] and A[3] will always exist as well
        min = A[2]                     //Extract the lesser of the two children
        A[2] = A[A.heap-size]
        A.heap-size = A.heap-size -1
        MIN-HEAPIFY(A, 2)              //Min-HEAPIFY percolates down, so it will not see the root
else
        min = A[3]
        A[3] = A[A.heap-size]
        A.heap-size = A.heap-size -1
        MIN-HEAPIFY(A, 3)              //Min-HEAPIFY percolates down, so it will not see the root
return min

RECALCULATE-AVERAGE(A, key)
        A[1] = A[1] * A. heap-size -2  //A.heap-size – 2 because we do not include the node containing the average or the new node
        A[1] = (A[1] + key)/A. heap-size -1   //A. heap-size – 1 because the node containing the average does not count
                                       //This operation should still run in O(1) time because we aren't traversing the heap tree
```