1. For the following array

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 40 | 6 | 3 | 11 | 2 | 4 |

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

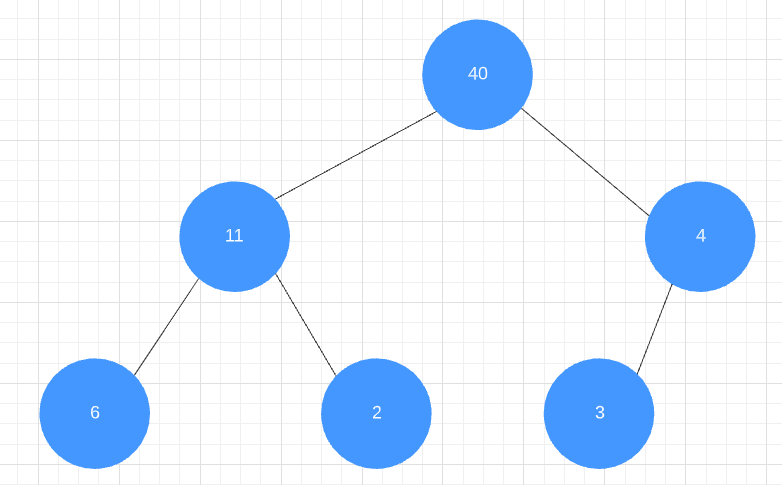
BUILD-MAX-HEAP(A)

A.heap-size = A.length

for i = ⎣A.length/2⎦ downto 1

MAX-HEAPIFY(A, i)

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



1. 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]

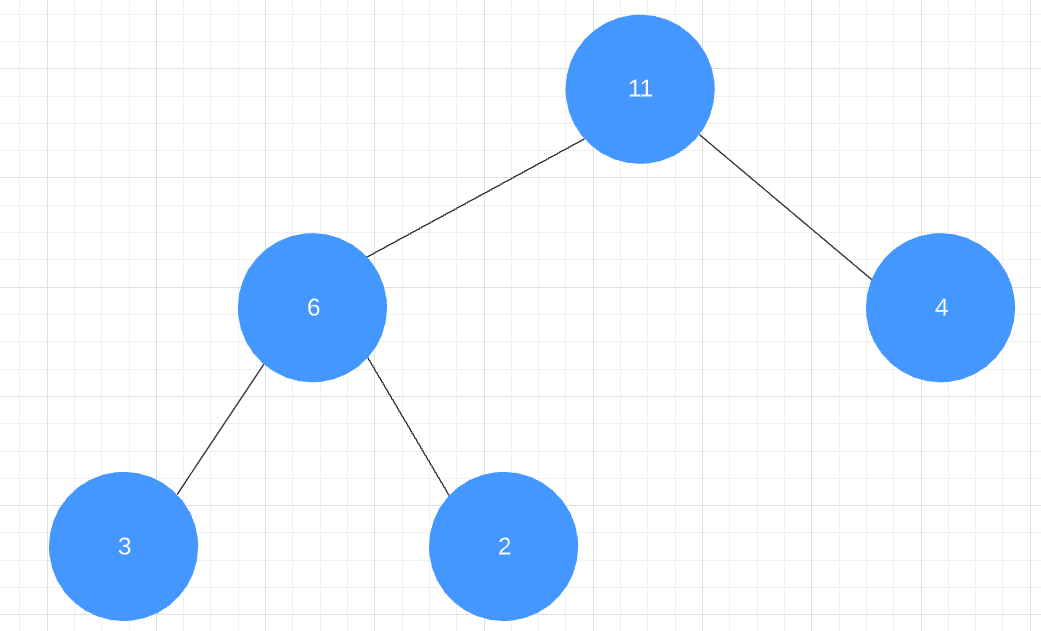
A[1]=A[A.heap-size]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 11 | 4 | 6 | 2 | 3 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 11 | 4 | 6 | 2 |

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



1. 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 |

MIN-HEAP-INSERT(A, key)

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

A[A.heap-size] = ∞

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

56 will be inserted at the end of the array

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

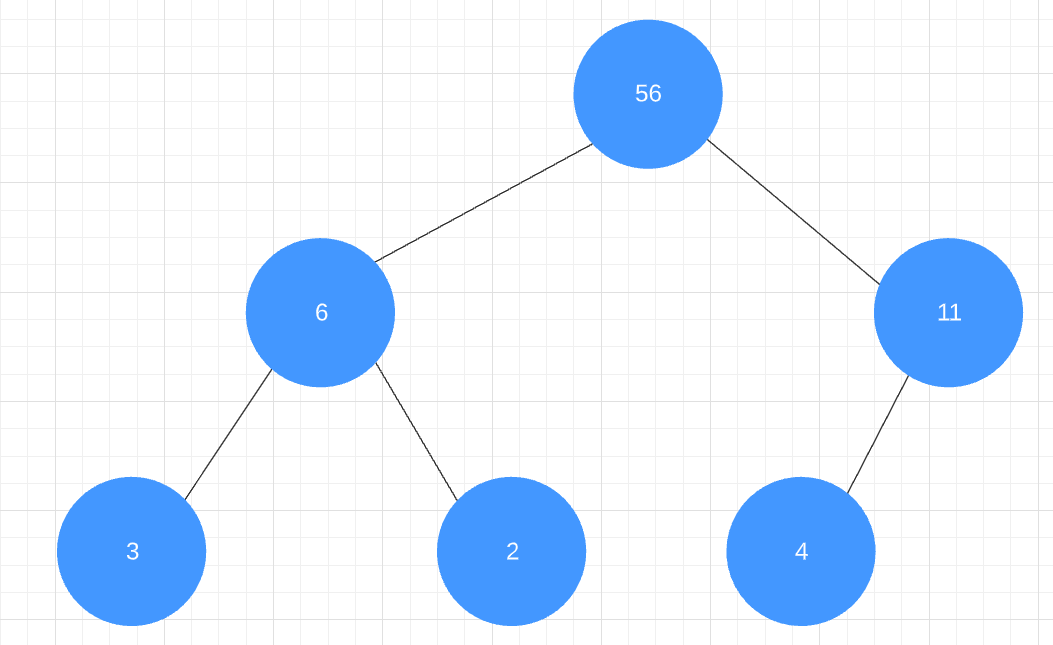
Exchange A[6] with A[3]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 11 | 6 | 56 | 3 | 2 | 4 |

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

Exchange A[3] with A[1]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 56 | 6 | 11 | 3 | 2 | 4 |



1. Consider of 3-ary max-heap. A 3-ary max-heap is like a binary max-heap, but instead of 2 children, nodes have 4 children.
2. 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
* The middle child of the parent would be located at
* The right child of the parent would be located at
* For a child node, their parent would be located at

1. 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 . For 3 nodes, they would affect the modifier for the log n.

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

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)

m = MIDDLE(i)

r = RIGHT(i)

if l <= A.heap-size and A[l] < A[i]

smallest = l

else smallest = i

if m <= A.heap-size and A[m] < A[i]

smallest = lm

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)

**Runtime:**

1. 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] = ∞

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

if key > A[i]

error “new key is larger”

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.

1. 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]

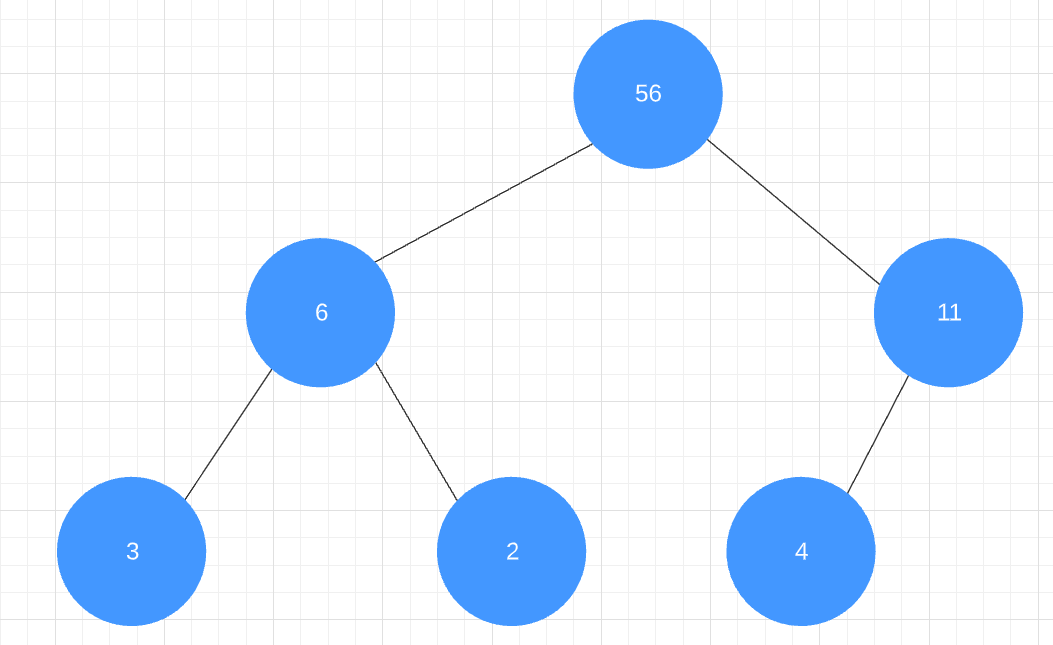
exchange A[i] with A[PARENT(i)]

i = PARENT(i)

Nothing needs to be changed. This works with a 3-ary heap as well as a 2-ary heap as long as the code can be adjusted to find A[PARENT(i)]. Because this swapping procedure will repeat as for each level in the heap tree, the worse case time complexity will last the height of the heap tree.

1. Show that the number of leaves in a heap is . (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 . 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…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 nodes in the array represents the leaves found in the last layer of the heap tree.

1. 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 for is to preserve the property of its heap tree. Since HEAP-DECREASE-KEY is used for min-heaps, the property being that a A[i] is the child of A[i/2].
* **Initialization**-Before the loop begins, the element A[i] is marked with a value given by the key. A[i] remains in the heap tree where its parent can be found at A[i/2].
* **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, 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. In this case, the property holds up because the new A[i/2] will be the parent of the new A[i].
  + If the min-heap property was found to be true, then there is no swap. In this case, the property still holds up because A[i/2] remains the parent of A[i/2]
* **Termination**-The loop terminates when the child node can no longer swap with its parent node. This means the position of A[parent(i)] and A[i] are now set. The property still stands as A[i/2] is the parent of A[i] even if their values may have been swapped.

1. For the following algorithm:
2. 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
3. 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