Yogendra Pokhrel
617585

1. Illustrate the operation of Max_Heapify(A, 3) using the given array.

A = (27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0)

Answer:

Initial Array (1-based indexing):

Index: 1 2 3 4 5 6 7 8 9 10 11 12 13 14

A = 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0

Step 1: Max_Heapify(A, 3)

We start at index i = 3

- left = 2 * 3 = 6
- right = 2 * 3 + 1 = 7
- A[3] = 3, A[6] = 10, A[7] = 1
- Among A[3], A[6], A[7], the largest is A[6] = 10

Swap A[3] and A[6]

Array after first swap:

A = (27, 17, 10, 16, 13, 3, 1, 5, 7, 12, 4, 8, 9, 0)

Step 2: Max_Heapify(A, 6)

Now, recurse at index i = 6

- left = 2 * 6 = 12
- right = 2 * 6 + 1 = 13
- A[6] = 3, A[12] = 8, A[13] = 9
- Largest is A[13] = 9

Swap A[6] and A[13]

Array after second swap:

A = (27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0)

Step 3: Max_Heapify(A, 13)

Now, recurse at index i = 13

- left = 26, right = 27 → both out of bounds (since heap size = 14)
- No further action needed.

Final Heap Array:

A = (27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0)

2. Show the operation of Build-Heap using the given array.

A = (5, 3, 17, 10, 84, 19, 6, 22, 9)

Answer:

Input Array (1-based indexing):

Index: 1 2 3 4 5 6 7 8 9

A = 5, 3, 17, 10, 84, 19, 6, 22, 9

Heap Type:

Max-Heap

Build-Heap works in bottom-up fashion, calling Max_Heapify() on all non-leaf nodes: from $\lfloor n/2 \rfloor$ down to 1.

Here, n = 9 ⇒ start from index $\lfloor 9/2 \rfloor$ = 4 down to 1.

Step: Build-Heap(A)

Step 1: Max_Heapify(A, 4)

- A[4] = 10, left = 8 (A[8] = 22), right = 9 (A[9] = 9)
- Max among A[4], A[8], A[9] is A[8] = 22
  Swap A[4] and A[8]

Array:

A = (5, 3, 17, 22, 84, 19, 6, 10, 9)

Step 2: Max_Heapify(A, 3)

- A[3] = 17, left = 6 (A[6] = 19), right = 7 (A[7] = 6)
- Max is A[6] = 19

  Swap A[3] and A[6]

Array:

A = (5, 3, 19, 22, 84, 17, 6, 10, 9)

Step 3: Max_Heapify(A, 2)

- A[2] = 3, left = 4 (A[4] = 22), right = 5 (A[5] = 84)
- Max is A[5] = 84

  Swap A[2] and A[5]

Array:

A = (5, 84, 19, 22, 3, 17, 6, 10, 9)

Recurse at A[5] = 3

- left = 10, right = 11 → out of bounds

  No further action.

Step 4: Max_Heapify(A, 1)

- A[1] = 5, left = 2 (A[2] = 84), right = 3 (A[3] = 19)
- Max is A[2] = 84

  Swap A[1] and A[2]

Array:

A = (84, 5, 19, 22, 3, 17, 6, 10, 9)

Recurse at A[2] = 5

- left = 4 (A[4] = 22), right = 5 (A[5] = 3)
- Max is A[4] = 22

Swap A[2] and A[4]

Array:

A = (84, 22, 19, 5, 3, 17, 6, 10, 9)

Recurse at A[4] = 5

- left = 8 (A[8] = 10), right = 9 (A[9] = 9)
- Max is A[8] = 10

Swap A[4] and A[8]

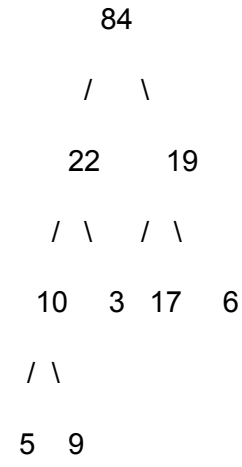
Array:

A = (84, 22, 19, 10, 3, 17, 6, 5, 9)

Final Max-Heap:

A = (84, 22, 19, 10, 3, 17, 6, 5, 9)

Binary Tree Representation:



3. Illustrate the operation of Heapsort using the array below:

(5, 13, 2, 25, 7, 17, 20, 8, 4).

Answer:

Heapsort Steps:

1. Build a Max-Heap from the input array.
2. Extract max element (root) and move it to the end.
3. Reduce the heap size and call Max_Heapify on root.
4. Repeat until the array is sorted.

Step 1:

Build Max-Heap

Initial array:

Index: 1 2 3 4 5 6 7 8 9

A = 5, 13, 2, 25, 7, 17, 20, 8, 4

We start from ⌊n/2⌋ = ⌊9/2⌋ = 4 → down to 1

Max_Heapify(A, 4)

- A[4]=25, children: A[8]=8, A[9]=4 → already max

No change.

Max_Heapify(A, 3)

- A[3]=2, children: A[6]=17, A[7]=20 → max is 20
- Swap A[3] ↔ A[7]

A = (5, 13, 20, 25, 7, 17, 2, 8, 4)

Max_Heapify(A, 2)

- A[2]=13, children: A[4]=25, A[5]=7 → max is 25
- Swap A[2] ↔ A[4]

A = (5, 25, 20, 13, 7, 17, 2, 8, 4)

Max_Heapify(A, 4): A[4]=13, children A[8]=8, A[9]=4 → no change

Max_Heapify(A, 1)

- A[1]=5, children: A[2]=25, A[3]=20 → max is 25
- Swap A[1] ↔ A[2]

A = (25, 5, 20, 13, 7, 17, 2, 8, 4)

Max_Heapify(A, 2): A[2]=5, children A[4]=13, A[5]=7 → max is 13

→ Swap A[2] ↔ A[4]

A = (25, 13, 20, 5, 7, 17, 2, 8, 4)

Max_Heapify(A, 4): A[4]=5, children A[8]=8, A[9]=4 → max is 8

→ Swap A[4] ↔ A[8]

A = (25, 13, 20, 8, 7, 17, 2, 5, 4)

Max-Heap built

Step 2: HeapSort — Extract max repeatedly

i = 9: Swap A[1]=25 with A[9]=4

A = (4, 13, 20, 8, 7, 17, 2, 5, 25)

→ Max_Heapify(A, 1) on size=8

→ Swap A[1] ↔ A[3] → then A[3] ↔ A[6]

A = (20, 13, 4, 8, 7, 17, 2, 5, 25)

A = (20, 13, 17, 8, 7, 4, 2, 5, 25)

i = 8: Swap A[1]=20 with A[8]=5

A = (5, 13, 17, 8, 7, 4, 2, 20, 25)

→ Max_Heapify(A, 1) on size=7

→ A[1]=5, children A[2]=13, A[3]=17 → max is 17

→ Swap A[1] ↔ A[3] → A[3]=5, A[6]=4, A[7]=2 → no swap

A = (17, 13, 5, 8, 7, 4, 2, 20, 25)

i = 7: Swap A[1]=17 with A[7]=2

A = (2, 13, 5, 8, 7, 4, 17, 20, 25)

→ Max_Heapify(A, 1) on size=6

→ A[1]=2, children A[2]=13, A[3]=5 → max is 13

→ Swap A[1] ↔ A[2] → A[2]=2, A[4]=8 → swap A[2] ↔ A[4]

A = (13, 2, 5, 8, 7, 4, 17, 20, 25)

A = (13, 8, 5, 2, 7, 4, 17, 20, 25)

i = 6: Swap A[1]=13 with A[6]=4

A = (4, 8, 5, 2, 7, 13, 17, 20, 25)

→ Max_Heapify(A, 1) on size=5

→ A[1]=4, children A[2]=8, A[3]=5 → max is 8

→ Swap A[1] ↔ A[2] → A[2]=4, A[4]=2, A[5]=7 → max is 7

→ Swap A[2] ↔ A[5]

A = (8, 4, 5, 2, 7, 13, 17, 20, 25)

A = (8, 7, 5, 2, 4, 13, 17, 20, 25)

i = 5: Swap A[1]=8 with A[5]=4

A = (4, 7, 5, 2, 8, 13, 17, 20, 25)

→ Max_Heapify(A, 1) on size=4

→ A[1]=4, children A[2]=7, A[3]=5 → max is 7

→ Swap A[1] ↔ A[2]

A = (7, 4, 5, 2, 8, 13, 17, 20, 25)

i = 4: Swap A[1]=7 with A[4]=2

A = (2, 4, 5, 7, 8, 13, 17, 20, 25)

→ Max_Heapify(A, 1) on size=3

→ A[1]=2, children A[2]=4, A[3]=5 → max is 5

→ Swap A[1] ↔ A[3]

A = (5, 4, 2, 7, 8, 13, 17, 20, 25)

i = 3: Swap A[1]=5 with A[3]=2

A = (2, 4, 5, 7, 8, 13, 17, 20, 25)

→ Max_Heapify(A, 1) on size=2

→ A[1]=2, A[2]=4 → swap

A = (4, 2, 5, 7, 8, 13, 17, 20, 25)

i = 2: Swap A[1]=4 with A[2]=2

A = (2, 4, 5, 7, 8, 13, 17, 20, 25)

Final Sorted Array:

A = (2, 4, 5, 7, 8, 13, 17, 20, 25)

4. **Show that in an n-element heap, there are at most [n/2^(h+1)] nodes of height h.**

To prove that in an n-element heap, there are at most [n / 2^(h+1)] nodes of height h, we need to focus on heap structure and binary tree properties.

Definitions

- A heap is a complete binary tree.
- In a complete binary tree, all levels are fully filled except possibly the last, which is filled from left to right.
- The height of a node is the number of edges on the longest downward path to a leaf.
- The height of the tree is $\lfloor \log_2(n) \rfloor$ for n nodes.

We need to prove:

For any heap of n nodes, the number of nodes with height h is at most:

[n / 2^(h+1)]

Intuition Behind the Formula

- At height 0 (leaf level): most nodes are present.
- At height 1: roughly half as many as at height 0.
- At height 2: roughly a quarter, etc.

Each level upward has about half as many nodes as the level below.

Let's use properties of binary trees:

- A complete binary tree with n nodes has:
    - At most n / 2 leaves.
    - At most n / 4 nodes at height ≥ 1.
    - At most n / 8 nodes at height ≥ 2.
    - And so on.

This is because in a complete binary tree:

- Half of the nodes are at the bottom level (height 0).
- One-quarter are at height 1.
- One-eighth are at height 2.

Thus, for any height h, the maximum number of nodes at that height is:

max nodes at height h = [n / 2^(h+1)]

Because:

- We are essentially counting how many nodes could exist at that level.
- The total number of nodes that can be parents to nodes at height h is ≤ n / 2^(h+1).

Example (n = 15):

Try with n = 31 (perfect binary tree of height 4):

- Height 0:

  31 / 2 ^ 1 = 15, Actual = 16

- Height 1:

  31 / 2 ^ 2 = 7, Actual = 8

- Height 2:

  31 / 2 ^ 3 = 3, Actual = 4

- Height 3:

  31 / 2 ^ 4 = 1, Actual = 2

- Height 4:

  31 / 2 ^ 5 = 0, Actual = 1

This shows: The actual number of nodes of height h is always ≤ floor(n / 2^{h+1}), validating the bound.

Conclusion

In a heap:

Max nodes at height h = [n / 2^(h+1)]

This bound follows directly from the structure of a complete binary tree which is a key property of heaps.