Binary Heaps

Dante Barbieri

Data Structures and Algorithms, Texas A&M University, College Station, TX

November 19, 2021

Section 1

Binary Heap Conceptually

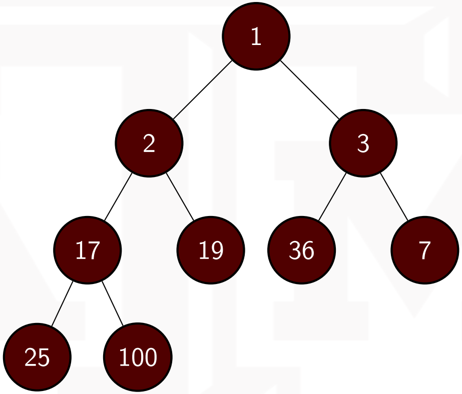


Figure: Tree Representation of a Binary Min-Heap

Binary Heap Structural Property

- The structure of a Binary Heap is a particular form of a Binary Tree.

Complete Binary Tree Property

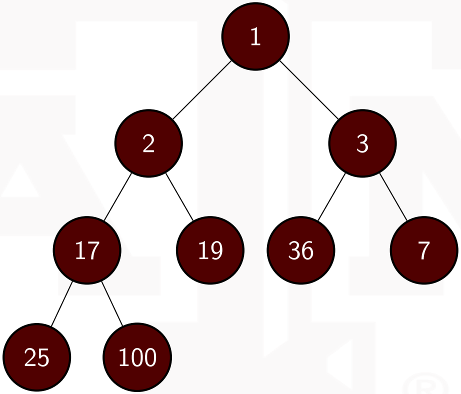
A heap T with height h is a **complete** *binary tree, that is, levels $0,1,2,\ldots,h-1$ of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \le 1 \le h-1$) and the nodes at level h fill this level from left to right.*

- All Binary Heaps must always follow the Complete Binary Tree Property.

The above definition comes from *Data Structures & Algorithms* by Goodrich et al.

Preferred Method to Store Binary Heaps

We take advantage of the Structural Property in order to store the Binary Heap in an Array.



0	1
1	2
2	3
3	17
4	19
5	36
6	7
7	25
8	100

Figure: Array Representation of the Binary Min-Heap from Concept

Specifics of Array Representation

Array Index	0	1	2	3	4	5	6	7	8
Heap Value	1	2	3	17	19	36	7	25	100

$$\text{Parent Index} = \left\lfloor \frac{\text{Index} - 1}{2} \right\rfloor$$

$$\text{Left Child Index} = 2 \times \text{Index} + 1$$

$$\text{Right Child Index} = 2 \times \text{Index} + 2 = 2\,(\text{Index} + 1)$$

$$n = \text{Array Length}$$

$$h = \lfloor \log_2 n \rfloor$$

$$\text{Root Index} = 0$$

$$\text{Leaf Index} \Rightarrow \text{Left Child Index} \geq n$$

Remember: `vector` is a good way to use Arrays that grow in C++.

Binary Heap Ordering

- The second qualification to be considered a Binary Heap is the satisfaction of an ordering property that organizes the locations of nodes in a heap relative to one another.
- This is similar to the Binary Search Tree ordering property.

Recall – Binary Search Tree Order Property

In a binary search tree T, for every node v other than the leaves (external nodes), the following are true:

1. *the value associated with v is greater than or equal to the value associated with v's left child,*
2. *the value associated with v is less than or equal to the value associated with v's right child, and*
3. *any values equivalent to the value associated with v lie either entirely in the left subtree of v or entirely in the right subtree of v for all v with minimum depth in T.*

Binary Min-Heap Ordering

- The "key" in the below definition is the value compared between two nodes to determine their order.
- This is similar to the practice for a Binary Search Tree, but allows us to compare only parts of an object.
  e.g.
  - Compare a students's GPA but not their name
  - Compare a CPU job's priority but not its length or its ID

Min-Heap Order Property

In a heap T, for every node v other than the root, the key associated with v is greater than or equal to the key associated with v's parent.

The above definition comes from *Data Structures & Algorithms* by Goodrich et al.

Binary Max-Heap Ordering

- This is identical to the Min-Heap Order Property, except the comparison goes the other direction.

Max-Heap Order Property

In a heap T, for every node v other than the root, the key associated with v is less than or equal to the key associated with v's parent.

The above definition comes from *Data Structures & Algorithms* by Goodrich et al.

Section 2

Insertion

First we need to ensure that the structural property of the Heap is maintained.

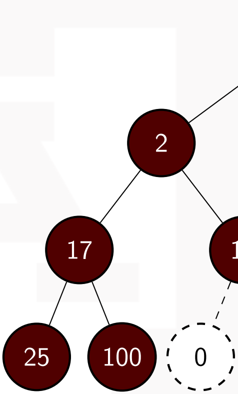


Figure: Insert '0' to the Binary Heap from before

Upheap

Once ‘0’ is inserted structurally, we need to fix the ordering. One way to do this is to continue up the heap until the order is correct or the new node becomes the root. This process is called Upheap.

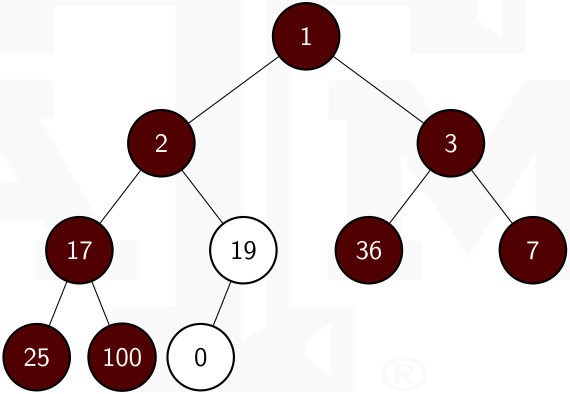


Figure: Compare ‘19’ and ‘0’

Upheap

We swapped '0' and '19' because they were out of order. Now we continue up the tree.

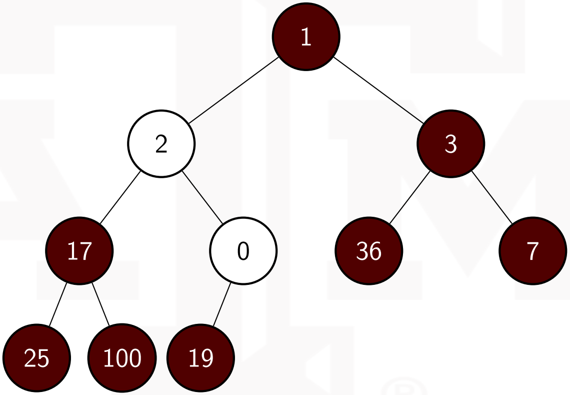


Figure: Compare '2' and '0'

Upheap

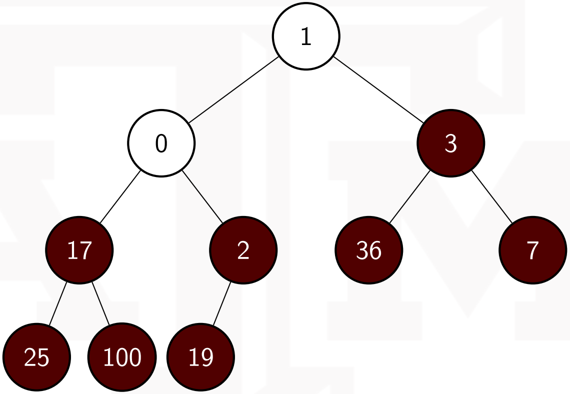


Figure: Compare ‘2’ and ‘0’

Upheap

We stop here because our node has become the root.

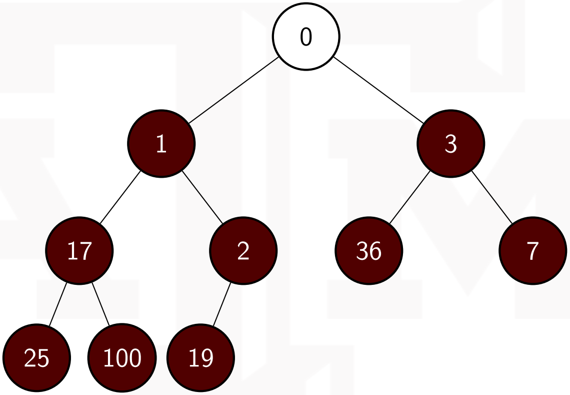


Figure: Base case as ‘0’ is the new root

Upheap

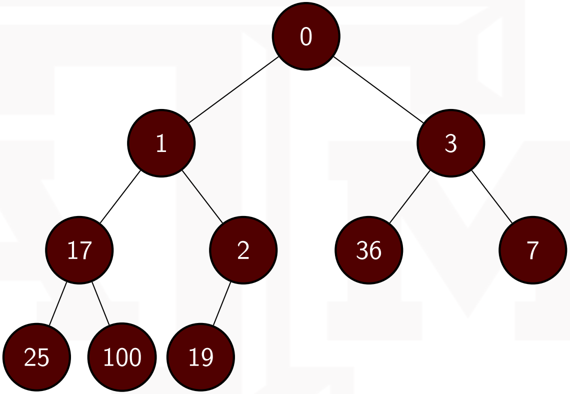


Figure: After inserting ‘0’ to the Binary Min-Heap

Insertion Summary

Algorithm 1 Insert with Included Upheap

```
Require: $T$ is a binary min-heap, $x$ is to be inserted into $T$
  $T$.insert_last($x$)          ▷ This preserves the structure of $T$
  $i \leftarrow T$.get_node($x$)
  while $i$ is not $T$.root and $i$.value $<$ $i$.parent.value do
      swap($i$, $i$.parent)          ▷ Upheap $x$
  end while
```

Removal

- We will discuss the removal of the root, which is a common operation.
- Removing an arbitrary node in a heap is possible, but this is uncommon.
    - A discussion of this can be found at the end of this section.

Removal

As with insertion, first, ensure the structure of the heap is maintained. Copy the last node into the root, and remove the last node.

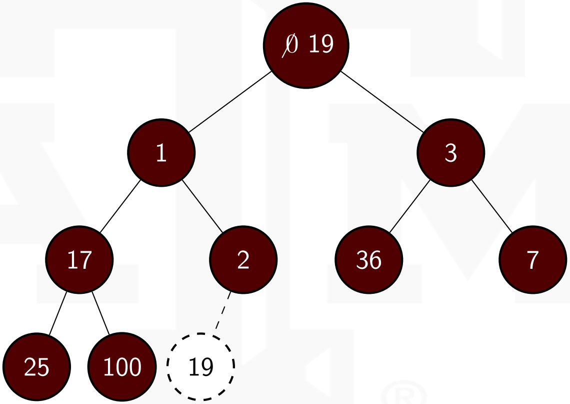


Figure: Copy '19' into '0' and erase the last node

Removal

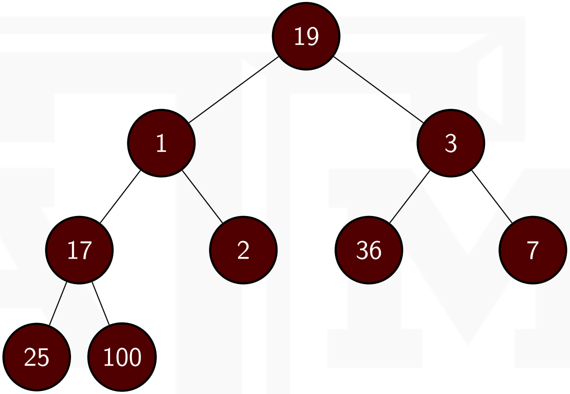


Figure: After deleting last node

The root is now out of order, so we need to perform a similar operation to Upheap. We call this operation Downheap.

Downheap

Compare the root with its children and swap with the most *extreme* child. For a min-heap, this is the smallest child.

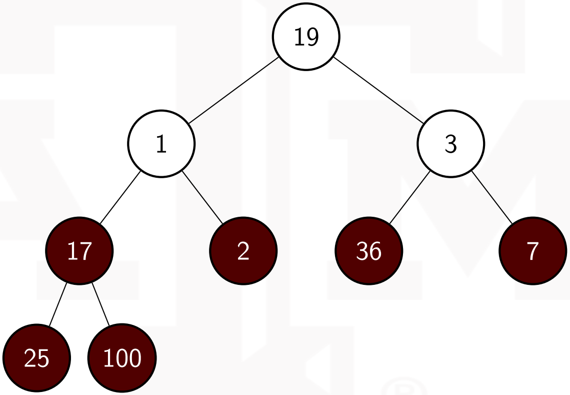


Figure: Comparing 19 with its children

Downheap

Swap ‘1’ with ‘19’ to correct the order. Because ‘1’ was the smaller child, it is guaranteed to be in order with respect to ‘3’, so we can continue downwards.

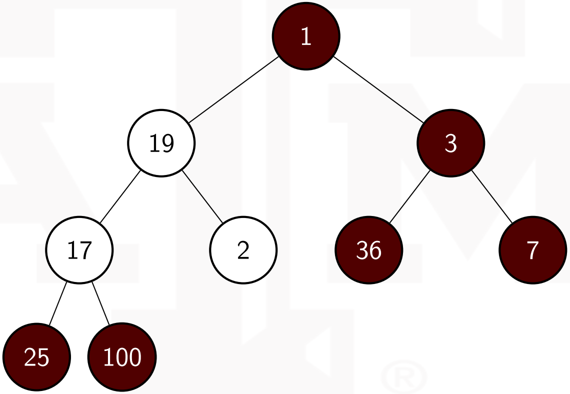


Figure: Comparing 19 with its children

Downheap

Swap ‘2’ with ‘19’ to correct the order. As ‘19’ is now a leaf, we can stop. If the node is ever in the correct order before becoming a leaf, we can stop then too.

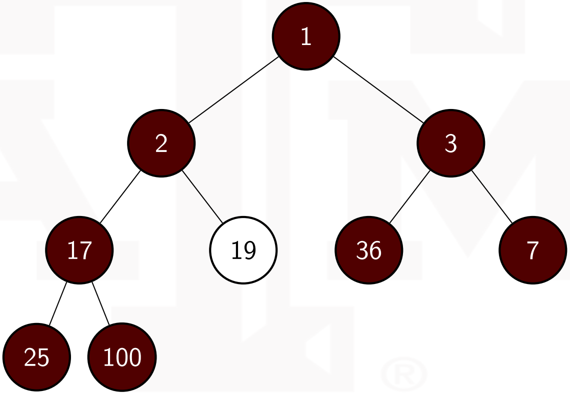


Figure: 19 is now a leaf

Downheap

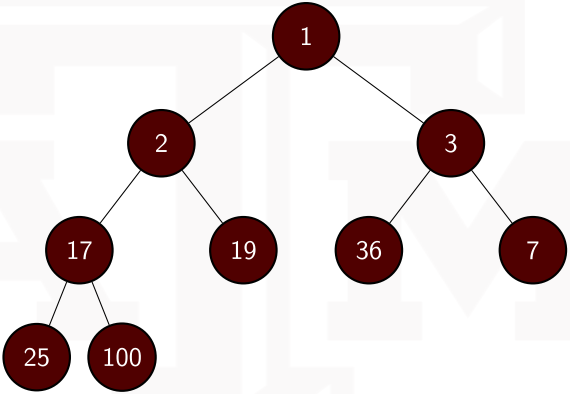


Figure: After removing the root from the heap

Removal Summary

Algorithm 2 Remove with Included Downheap

```
Require: $T$ is a binary min-heap
  $T$.root.value $\leftarrow$ $T$.get_last().value          $\triangleright$ Copy last into root
  $i \leftarrow T$.get_node($T$.root)                          $\triangleright$ $i$ refers to root
  while $i$ is not leaf and ($i$.value $>$ $i$.left_child.value or $i$.value $>$ $i$.right_child.value) do
      min_child $\leftarrow$ child of $i$ with minimum value
      swap($i$, min_child)                                   $\triangleright$ Downheap root
  end while
```

This algorithm assumes that if i.`right_child` does not exist, its value is ∞. Therefore, it cannot be the `min_child`.

Arbitrary Removal

You do not need to know this. That said, the process is similar. We take in the node we wish to remove and copy the last node into it. Then we perform Downheap from that node until it terminates. This task becomes difficult if we are asked to find the node we wish to remove. Searching in a Binary Heap is a painful experience, as you'll soon see in the next section.

Search

- Binary Heaps are not Binary Search Trees
- The search performance is really bad $O(n)$
  - This is *despite* the ordering property.
- So why use heaps?
  - Getting minimum (or maximum for max-heaps) is $O(1)$
  - Removing minimum (or maximum) is $O(\log n)$
  - Inserting is $O(\log n)$
  - Constructing a Heap from Scratch (Heapify/Bottom Up) is $O(n)$
- Heaps are only bad at search, so use a Search Tree for that!
- Heaps are great for Priority Queues and other Priority Systems.

Search

Search can be done

- linearly on the array (if array representation) or
- through a tree traversal (if tree representation).

Either approach yields $O(n)$ time, so we are stuck.

Heapify a.k.a. Bottom-Up Heap Construction

- Given an array of values, we need to convert it into a Heap.
  - Well, structurally, it is already a heap because all heaps are Arrays.
  - Therefore, we have to rearrange the data in the array.
- We have discussed Sorting before, which is an $O(n \log n)$ problem in the best of times for a Comparison-based sorting algorithm.
  - Thankfully, Heapify is not exactly sorting though it does make the array somewhat more sorted.
- Because Heapify does not require thoroughly sorting the array, it has a better Big-O bound of $O(n)$.
  - Thus, this process improves upon blindly calling Heap Insert for the n elements, which is $O(n \log n)$.
- We can use Heapify and Remove to sort data. We call this, Heapsort $O(n \log n)$.

Heapify

Algorithm 3 Bottom Up Heap Construction

```
Require: $V$ is a vector
  for $i \leftarrow V$.size()$/2 - 1$ to 0 do        ▷ includes 0
    Downheap($i$)        ▷ perform the Downheap loop on $i$
  end for
```

This assumes that you took the loop from `remove` and made it into a `Downheap` function, which is probably a *good* idea to do anyway. The following algorithm is recursive and can better illustrate what is happening. It is a bit less efficient, though, so the above algorithm is good to use.

Heapify

Algorithm 4 Bottom Up Heap Construction

```
Require: $L$ is a list (or other linked list), though a vector works too
  if $L$.empty() then
    return an empty heap
  end if
  $e \leftarrow L$.front()
  $L$.pop_front()
  Split $L$ into two lists, $L_1$ and $L_2$, each of size $(n-1)/2$
  $T_1 \leftarrow$ Heapify($L_1$)
  $T_2 \leftarrow$ Heapify($L_2$)
  Create binary tree $T$ with root $r$ storing $e$, left subtree $T_1$, and right
  subtree $T_2$
  Perform a down-heap bubbling from the root $r$ of $T$, if necessary
```

From: *Data Structures & Algorithms* by Goodrich et al.

Heapify

We now perform Heapify on the following data:

Array Index	Array Value
0	14
1	9
2	25
3	16
4	15
5	5
6	4
7	12
8	8
9	11
10	6
11	7
12	27
13	23
14	20

Heapify

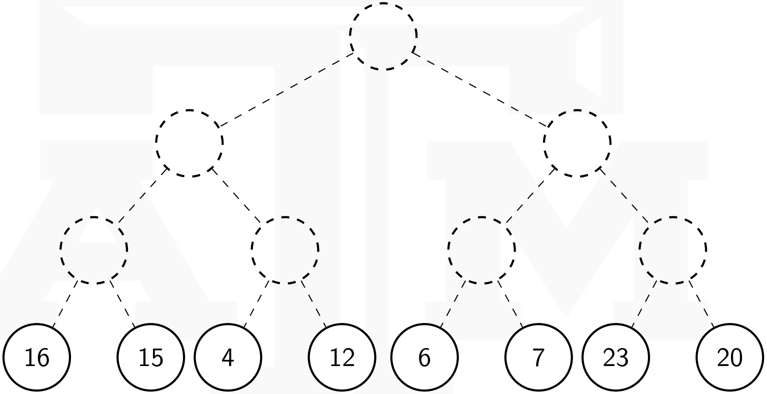


Figure: Construct trivial heaps

Heapify

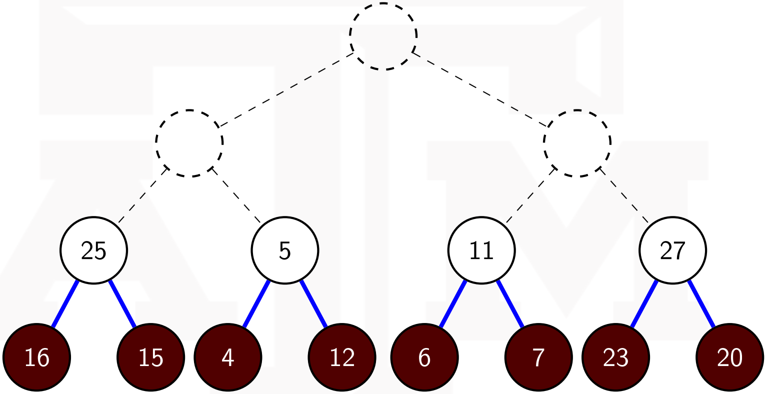


Figure: Insert third node

Heapify

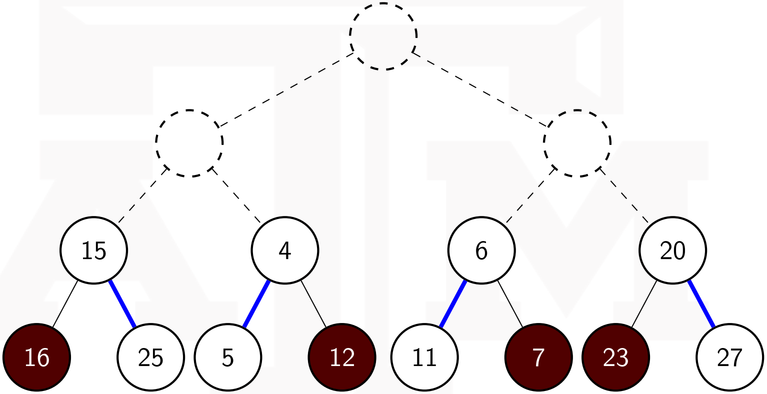


Figure: Downheap as necessary

Heapify

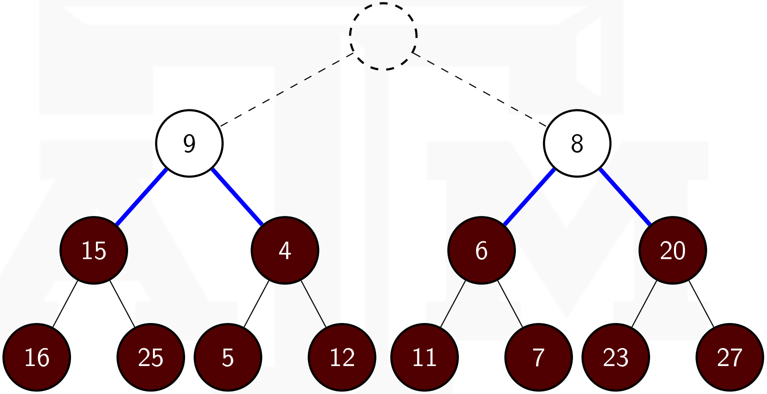


Figure: Combine heaps into bigger heaps adding root

Heapify

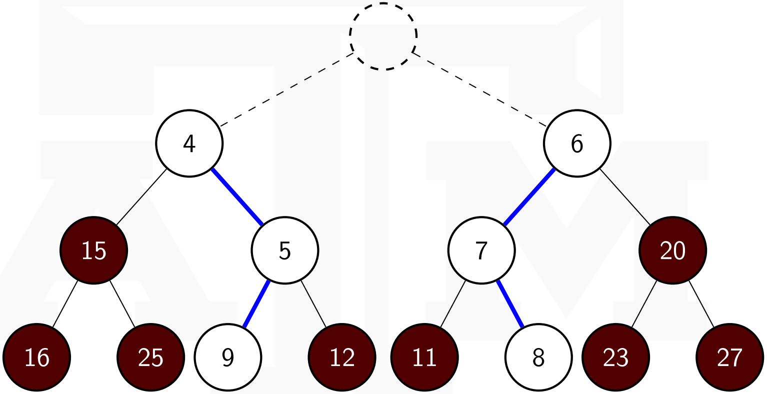


Figure: Downheap as necessary

Heapify

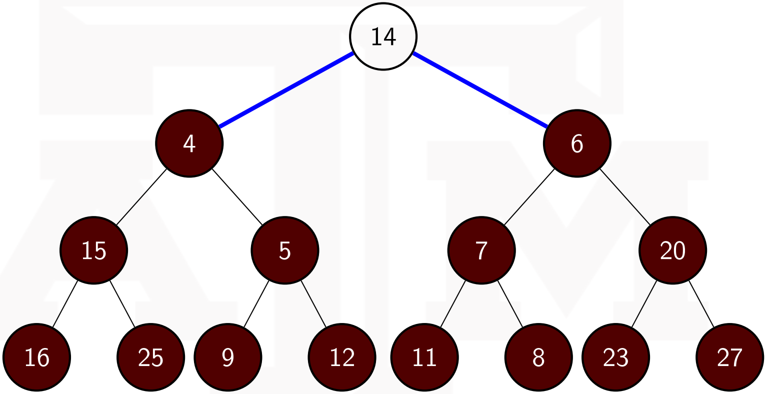


Figure: Insert final root

Heapify

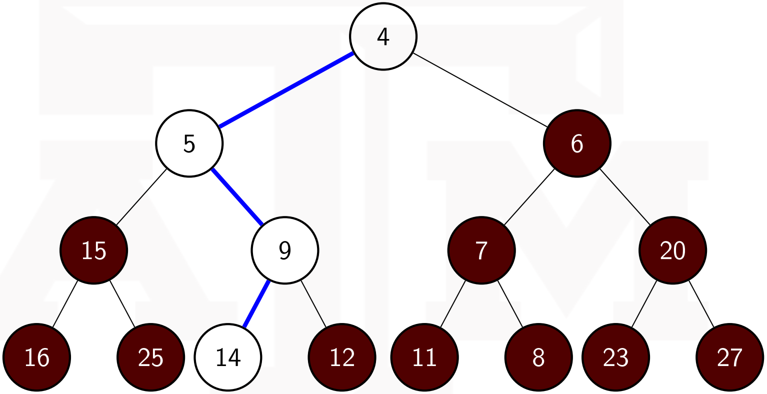


Figure: Downheap as necessary

Heapify

The final Heap Array:

Array Index	Array Value (originally)	Heap Value
0	14	4
1	9	5
2	25	6
3	16	15
4	15	9
5	5	7
6	4	20
7	12	16
8	8	25
9	11	14
10	6	12
11	7	11
12	27	8
13	23	23
14	20	27

Heap Sort

Algorithm 5 Heap Sort

```
Require: $V$ is a container (vector or list)
  $S \leftarrow \{\}$                          ▷ $S$ is the return vector which will be sorted
  Heapify($V$)                                ▷ Reorder $V$ to heap ordering $O(n)$
  while $V$ is not empty do                   ▷ loops $n$ times
      $S$.push_back($V$.remove_min())         ▷ Heap remove min $O(\log n)$
  end while
```

So the function is
$f(n) = O(n) + n \cdot O(\log n) = O(n) + O(n \log n) = O(n \log n)$.

Section 3

Conclusion

- Binary Heaps are useful for Priority Queues
- Don't search in Binary Heaps (unless you have to)
- Heapify lets us construct faster than repeatedly inserting
- Heap Sort builds a heap and removes one by one to get sorted order
    - No direct comparisons (heap does them)
    - Not recursive, but $O(n \log n)$ which is very nice
    - Can relatively quickly get subset of values in sorted order

References

Michael T. Goodrich, Roberto Tamassia, David Mount
Data Structures & Algorithms
Second Edition Chapter 8.

Thank You!