Heaps and Heapsort

Department of Electrical and Computer Engineering
McMaster University

January 2020

Textbook Reading

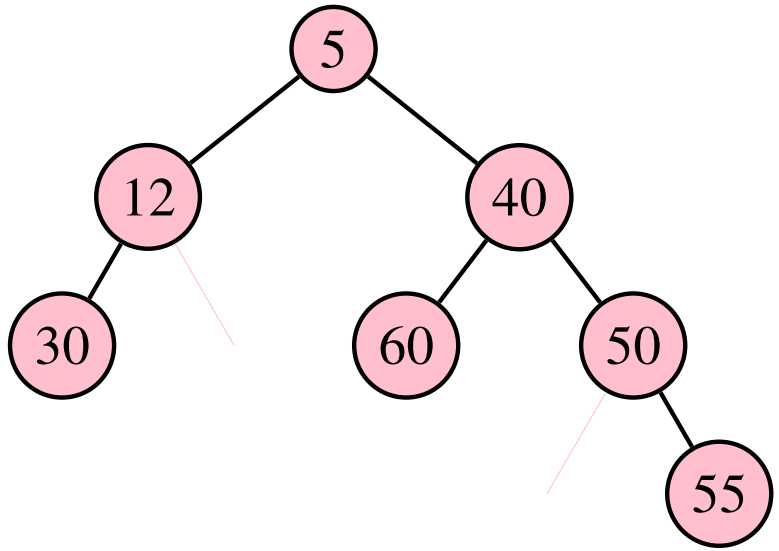
- Chapter 21: Sections 1-3, 5

Heaps

- A **(binary) heap** is the classic method used to implement priority queues. Here we use the term **heap** for a **binary heap**.
- A min/max heap supports **insert** and **deleteMin/deleteMax** in $\Theta(\log n)$ worst-case time.
- A heap is a binary tree where nodes store keys, with two additional properties:
  - an ordering property
  - a structure property

Heaps

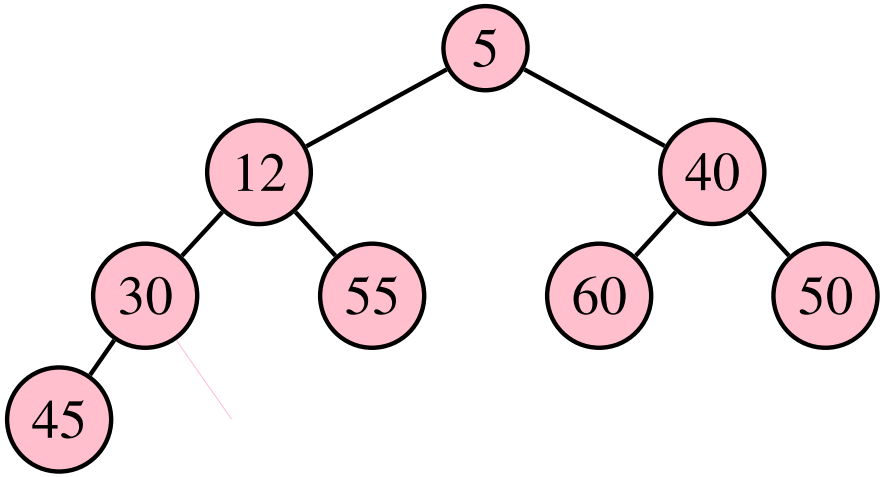
- A **binary min tree** is a binary tree in which for every node the key stored at that node is smaller than the keys stored at its children. Clearly, the key at each node is then smaller then the keys at all its descendants.



- Notice that the minimum key is **always** at the root of the binary min tree.

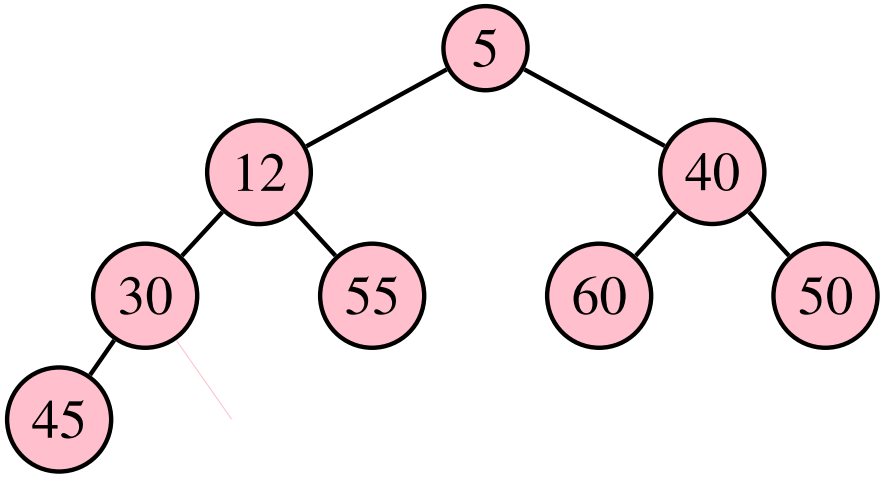
Heaps

- A **min heap** is a binary min tree (**ordering property**) which is completely filled on all levels, except possibly the lowest level, which is filled from left to right (**structure property**).



Heaps

- A **min heap** is a binary min tree (**ordering property**) which is completely filled on all levels, except possibly the lowest level, which is filled from left to right (**structure property**).



- A **max heap** is defined similarly, requiring the key at each node to be larger than keys at children.
- A heap can be efficiently implemented using an **array** - tree nodes are stored in level order: $\times$, 5, 12, 40, 30, 55, 60, 50, 45, $\times$, $\times$, $\cdots$
- Notice: no empty array spots between root and last leaf.

Heaps

- The **height** h of a heap with n nodes equals $\lfloor \log_2 n \rfloor$, i.e., the largest integer smaller than or equal to $\log_2 n$.
- Proof:
  - $2^h \leq n \leq 2^{h+1} - 1$
  - Thus, $\log_2 n - 1 < \log_2(n+1) - 1 \leq h \leq \log_2 n$
  - which implies that $h = \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) \rceil - 1$.

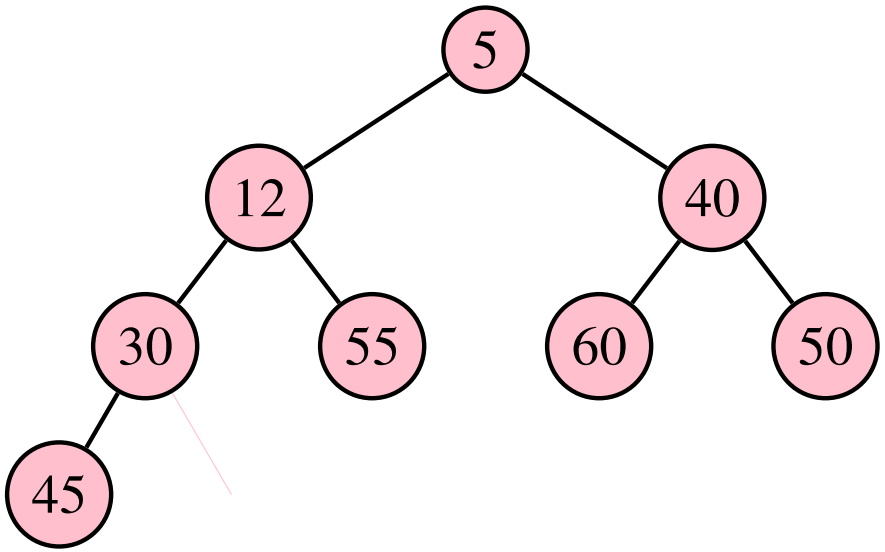
Heaps

- The **height** h of a heap with n nodes equals $\lfloor \log_2 n \rfloor$, i.e., the largest integer smaller than or equal to $\log_2 n$.
- Proof:
  - $2^h \leq n \leq 2^{h+1} - 1$
  - Thus, $\log_2 n - 1 < \log_2(n+1) - 1 \leq h \leq \log_2 n$
  - which implies that $h = \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) \rceil - 1$.
- Heap operations insert and deleteMin (for min heap) take an amount of time proportional to the height, in the worst-case: $\Theta(h) = \Theta(\log n)$.
- All operations on the heap must make sure to preserve the heap ordering and structure properties.

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the value stored in the new node violates the heap-order property, **percolate** the value upward until it fits.

Example: Insert 10

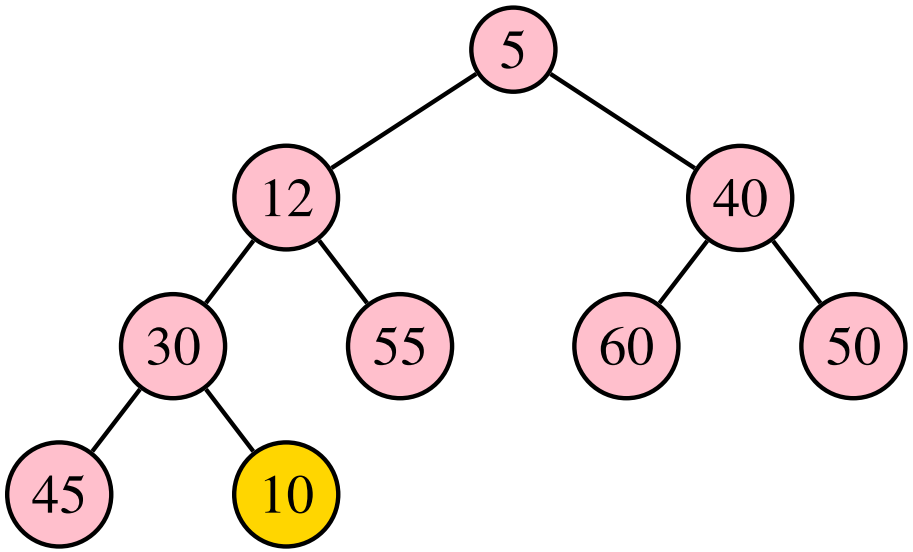


$\times, 5, 12, 40, 30, 55, 60, 50, 45, \times, \times, \cdots$

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the key stored in the new node violates the heap-order property, **percolate** the key upward until it fits.

Example: Insert 10 – the next open entry in the array is the leftmost leaf

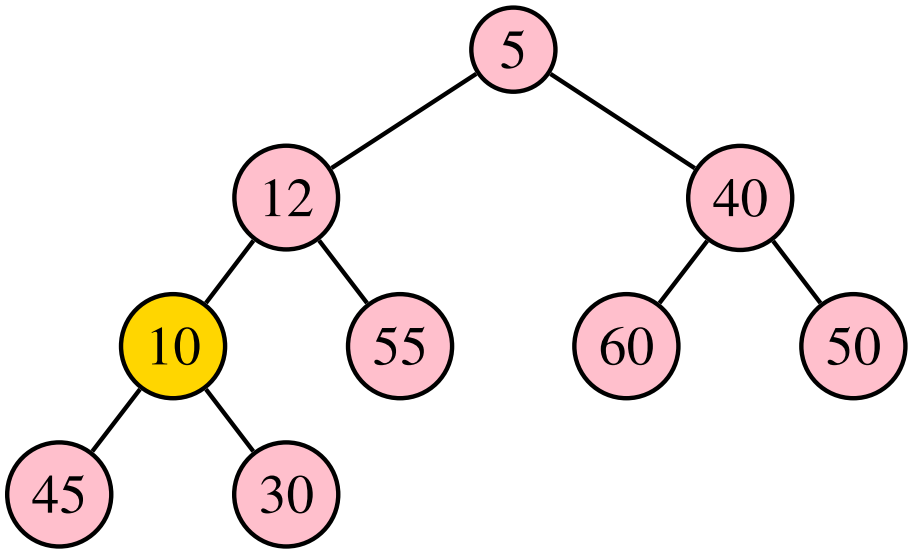


$\times$, 5, 12, 40, 30, 55, 60, 50, 45, **10**, $\times$, $\cdots$

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the key stored in the new node violates the heap-order property, **percolate** the key upward until it fits.

Example: Insert 10 – swap child with parent until heap order is satisfied

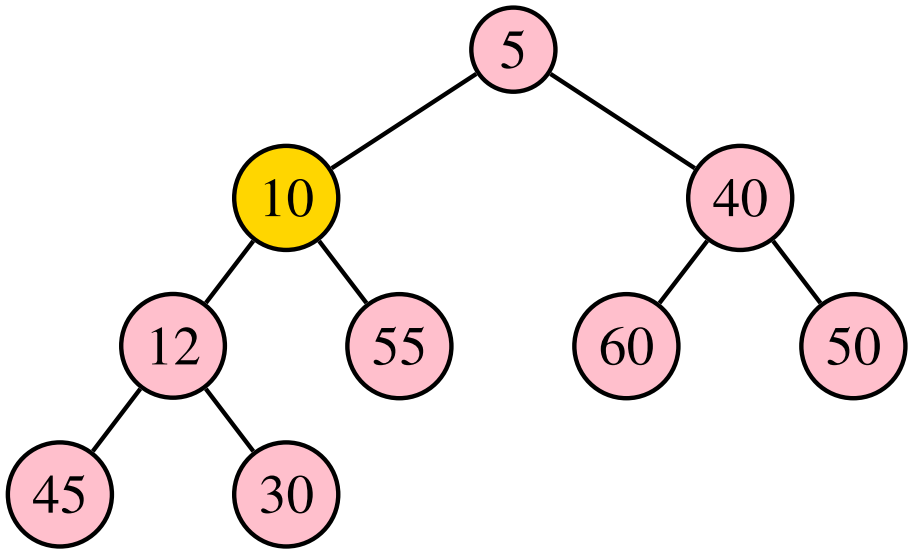


$\times$, 5, 12, 40, **10**, 55, 60, 50, 45, 30, $\times$, $\cdots$

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the key stored in the new node violates the heap-order property, **percolate** the key upward until it fits.

Example: Insert 10 – swap child with parent until heap order is satisfied

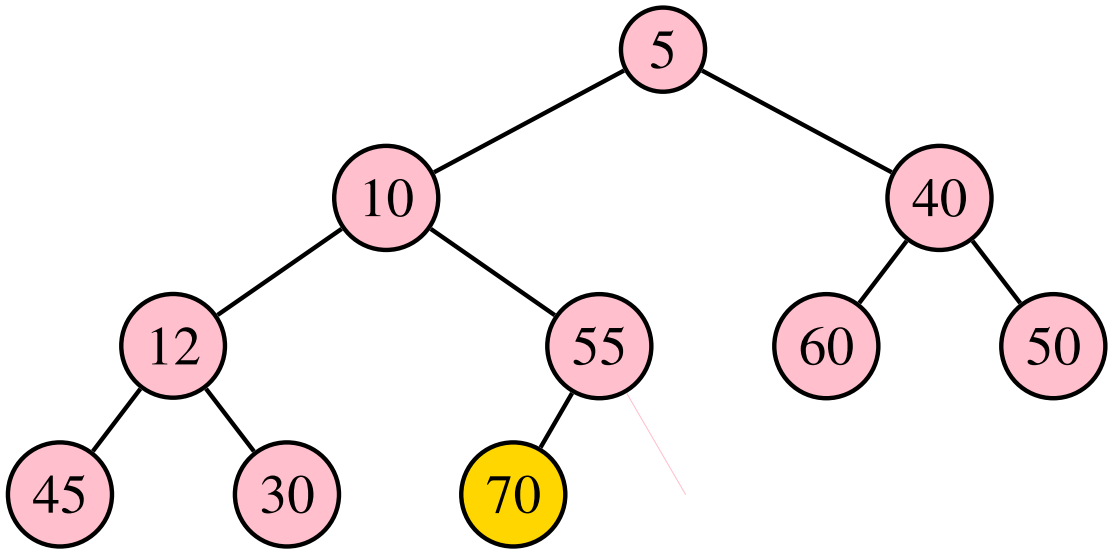


$\times$, 5, **10**, 40, 12, 55, 60, 50, 45, 30, $\times$, $\cdots$

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the key stored in the new node violates the heap-order property, **percolate** the key upward until it fits.

Example: Now, Insert 70.

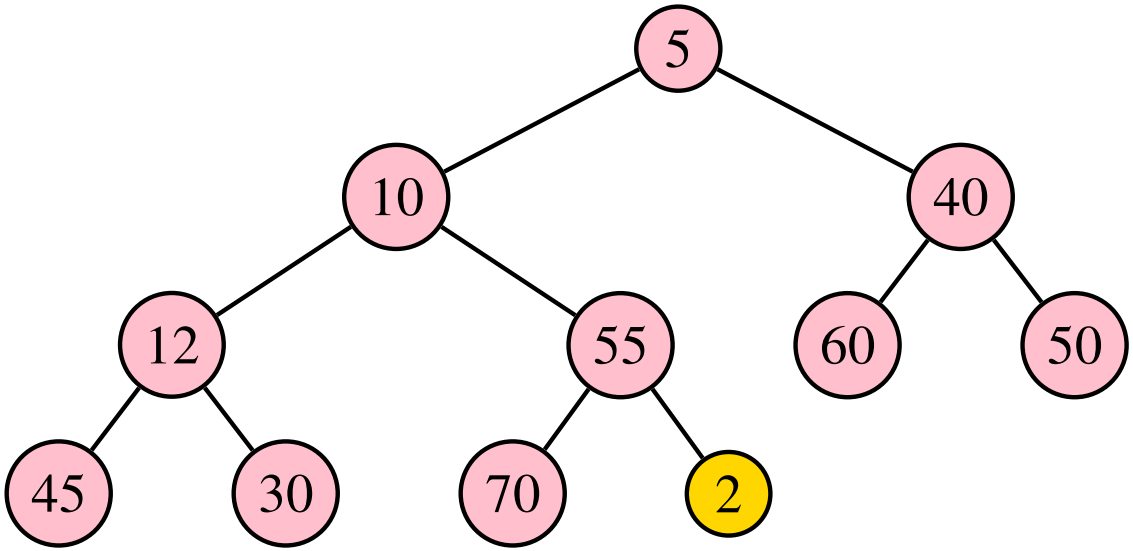


$\times$, 5, 10, 40, 12, 55, 60, 50, 45, 30, **70**, $\cdots$

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the key stored in the new node violates the heap-order property, **percolate** the key upward until it fits.

Example: Now, Insert 2.

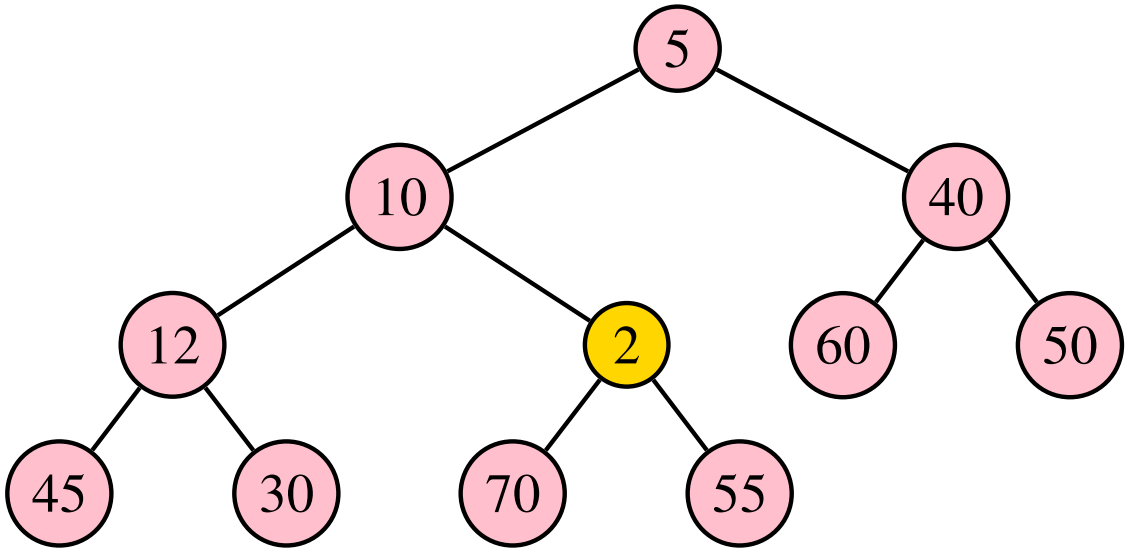


$\times$, 5, 10, 40, 12, 55, 60, 50, 45, 30, 70, **2**, $\cdots$

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the key stored in the new node violates the heap-order property, **percolate** the key upward until it fits.

Example: Now, Insert 2.

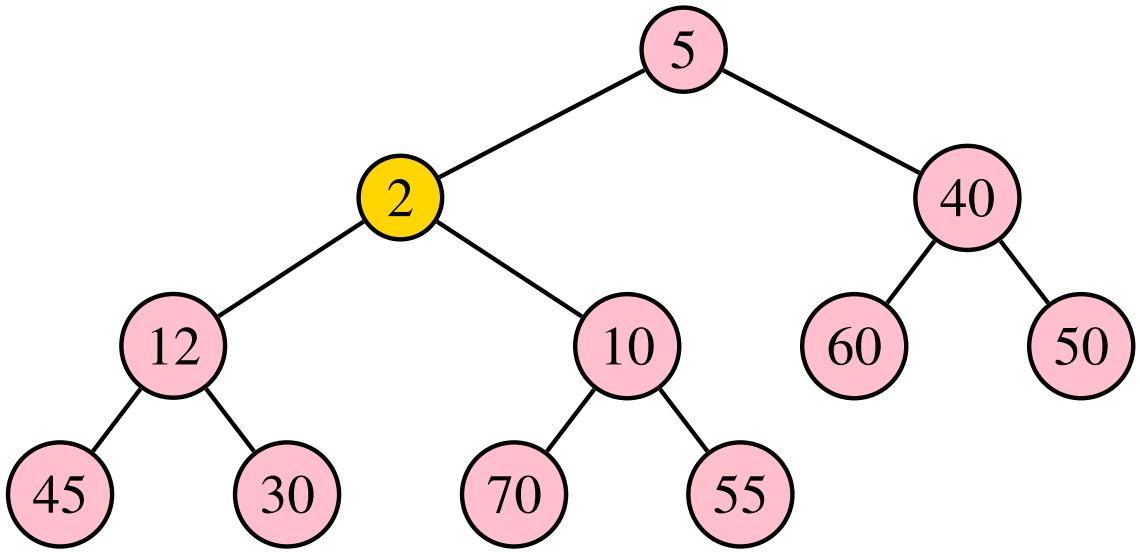


$\times$, 5, 10, 40, 12, **2**, 60, 50, 45, 30, 70, 55, $\cdots$

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the key stored in the new node violates the heap-order property, **percolate** the key upward until it fits.

Example: Now, Insert 2.

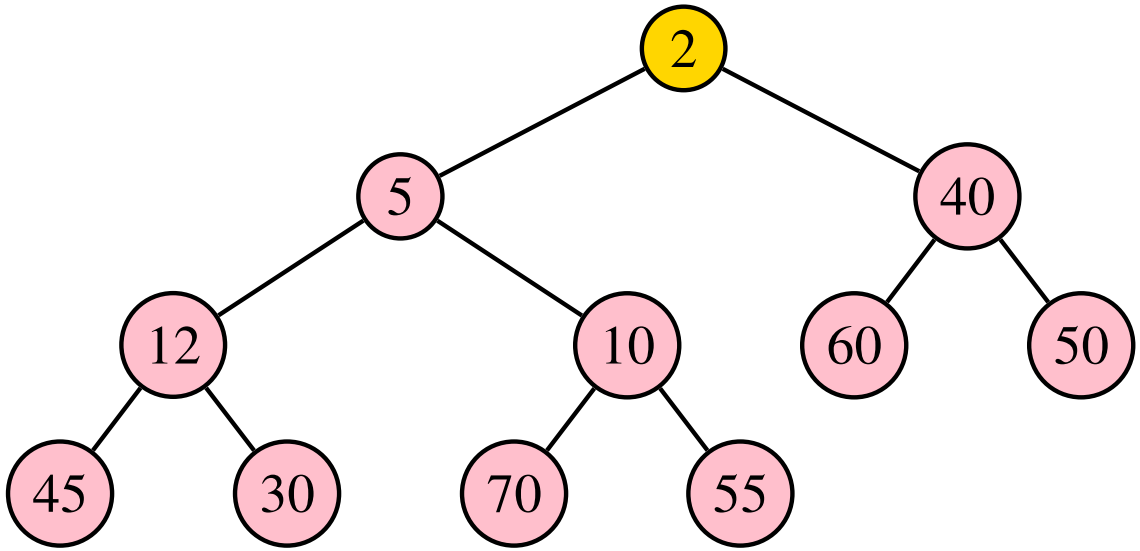


$\times$, 5, **2**, 40, 12, 10, 60, 50, 45, 30, 70, 55, $\cdots$

Min Heap – Insertion

- Insert a new node as the **next leaf** at the bottom level - next available array location.
- If the key stored in the new node violates the heap-order property, **percolate** the key upward until it fits.

Example: Now, Insert 2. **Done**



$\times$, **2**, 5, 40, 12, 10, 60, 50, 45, 30, 70, 55, $\cdots$

Min Heap – Insertion

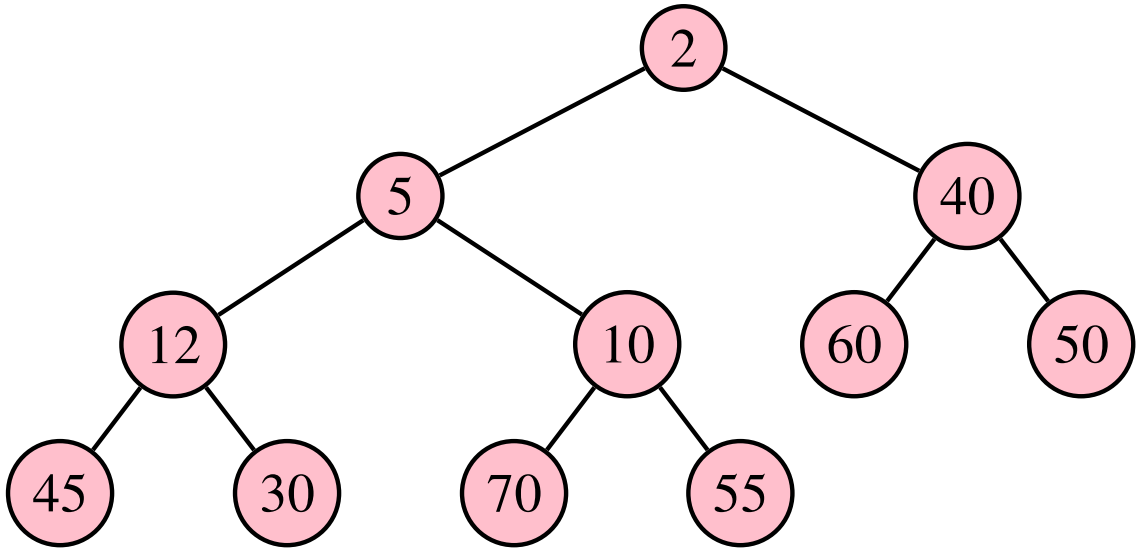
- Inserting into a min heap is simple due to array implementation
  - → Leaves are at the end of the array
- First insert at next available spot after the last occupied array location - $\Theta(1)$.
- Then percolate up ...
  - → To find the array spot where parent is stored - $\Theta(1)$.
  - → At worst you need to percolate the node up to the root $\rightarrow \Theta(\log n)$ worst-case running time.
  - → On average, this percolation terminates early
    - ☞ On average insert requires the node to move up approximately 1.6 levels $\rightarrow \Theta(1)$ average running time.

Min Heap – DeleteMin

- The item to delete is **always** at the root of the tree.
- The heap must be adjusted in order to maintain the ordering property.
- To maintain the heap structure property, need to promote the last leaf (i.e., the rightmost on lowest level) to the root.
- Then, you need to re-order the heap in order to maintain the heap ordering.

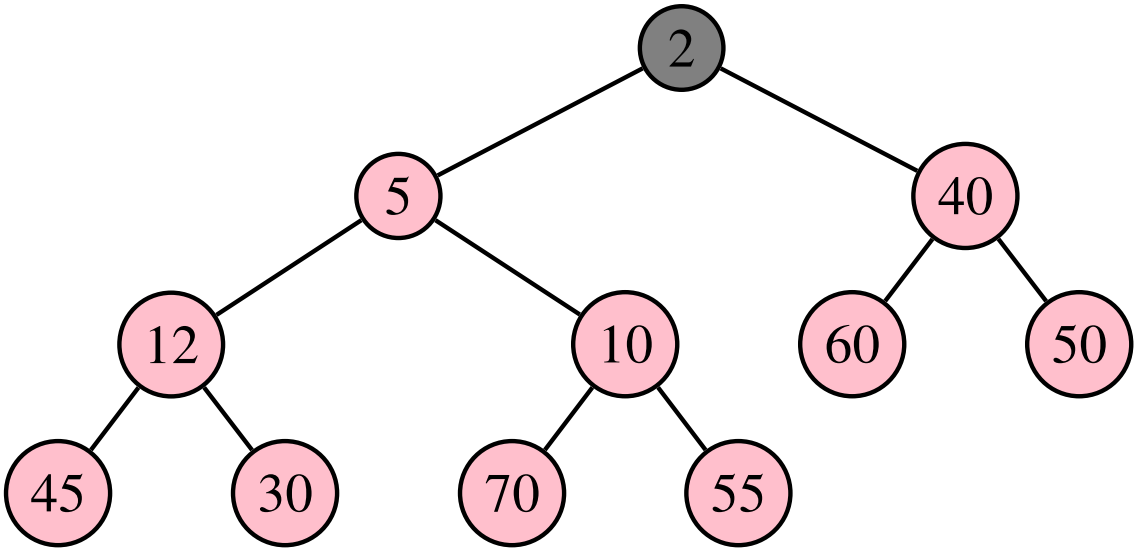
Min Heap – DeleteMin

Example: `DeleteMin`



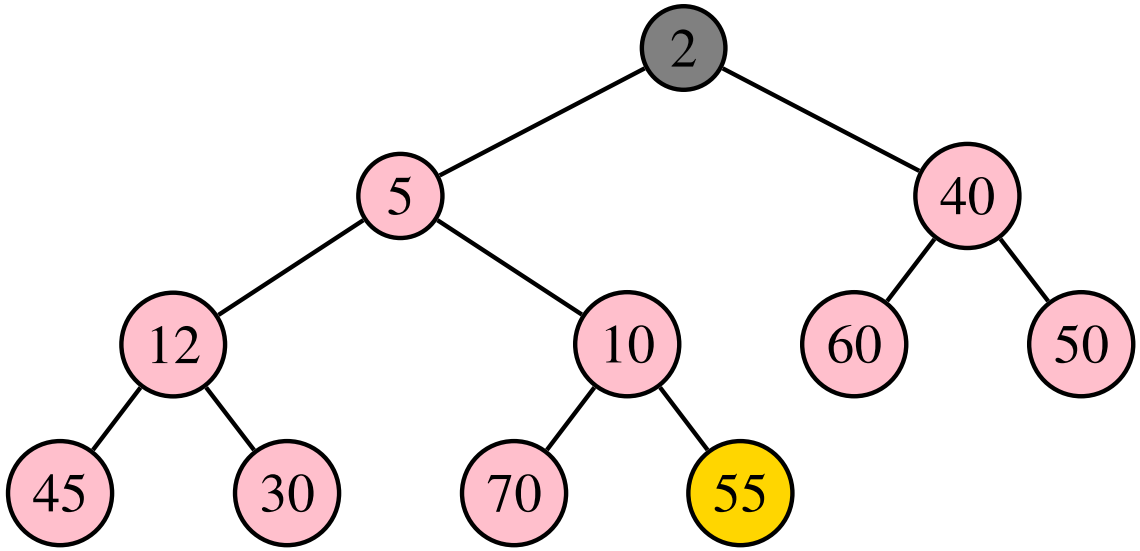
Min Heap – DeleteMin

Example: `DeleteMin` – remove the key stored at the root



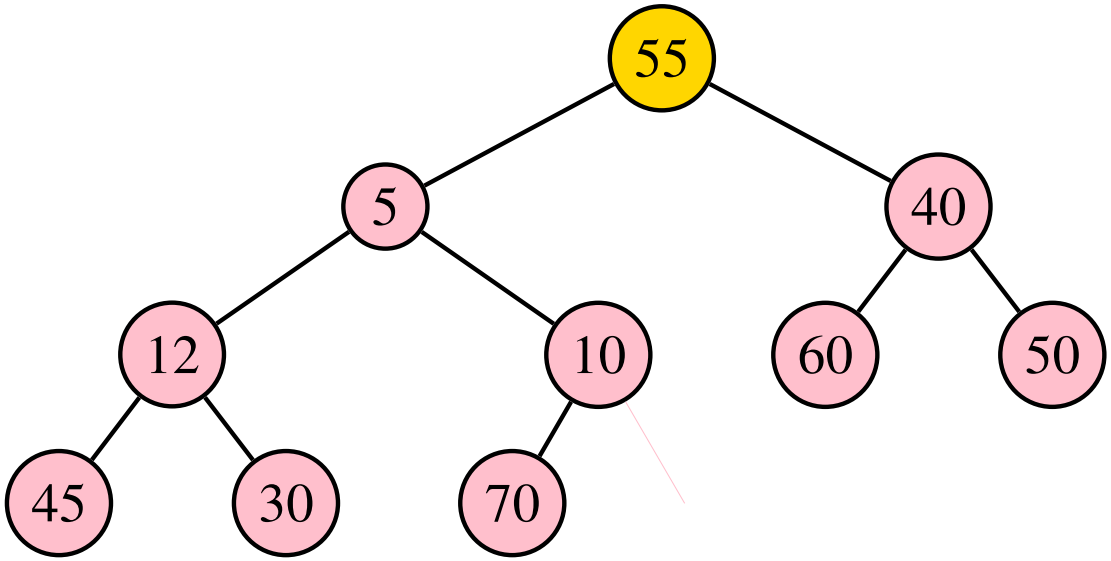
Min Heap – DeleteMin

Example: `DeleteMin` – remove the "last" leaf and place its key at the root.



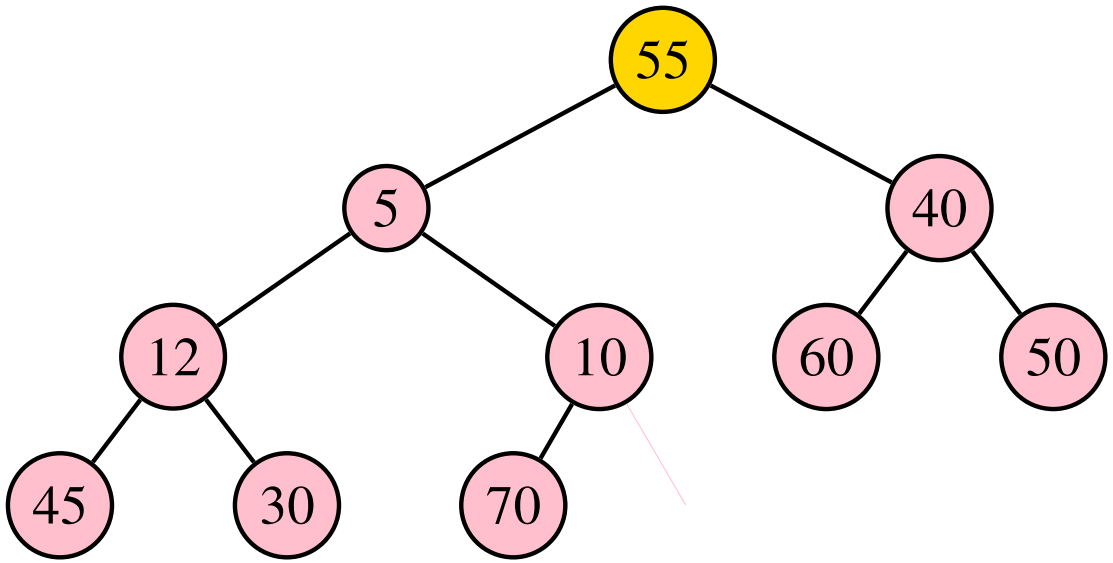
Min Heap – DeleteMin

Example: `DeleteMin` – remove the "last" leaf and place its key at the root.



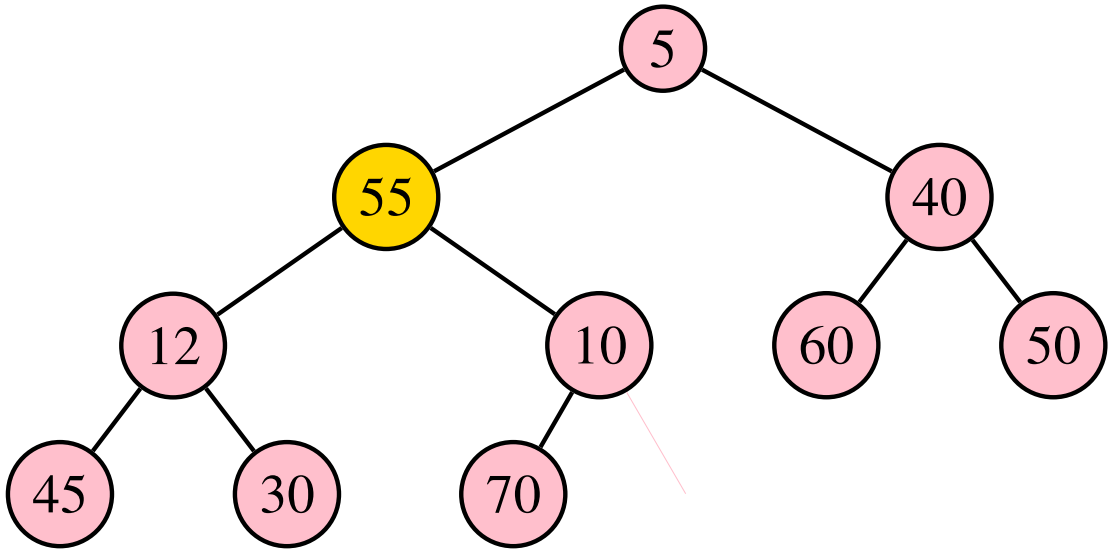
Min Heap – DeleteMin

Example: `DeleteMin` – re-organize heap to ensure ordering is correct - percolate down until it is placed correctly.



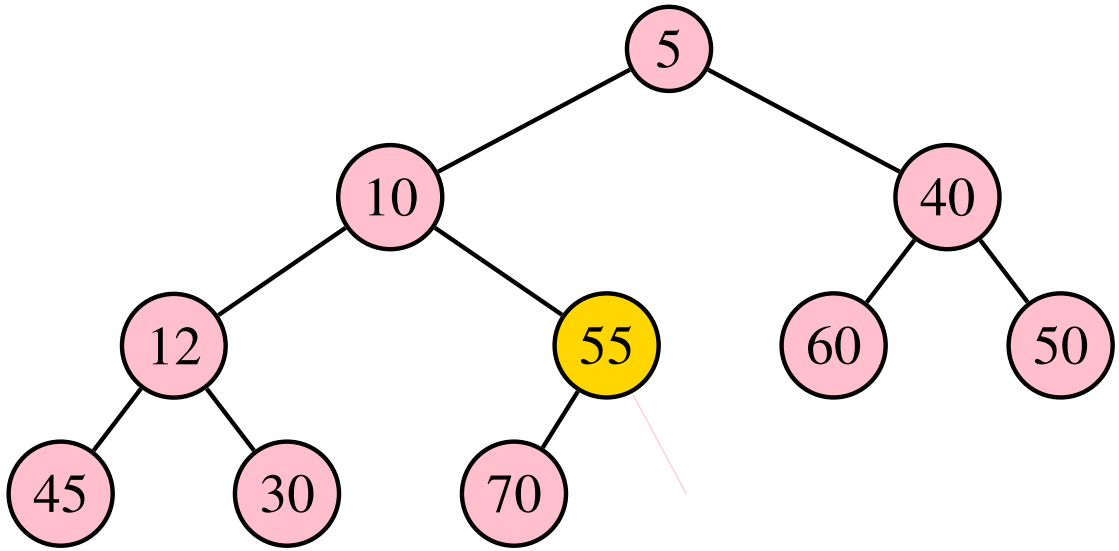
Min Heap – DeleteMin

Example: `DeleteMin` – re-organize heap to ensure ordering is correct.



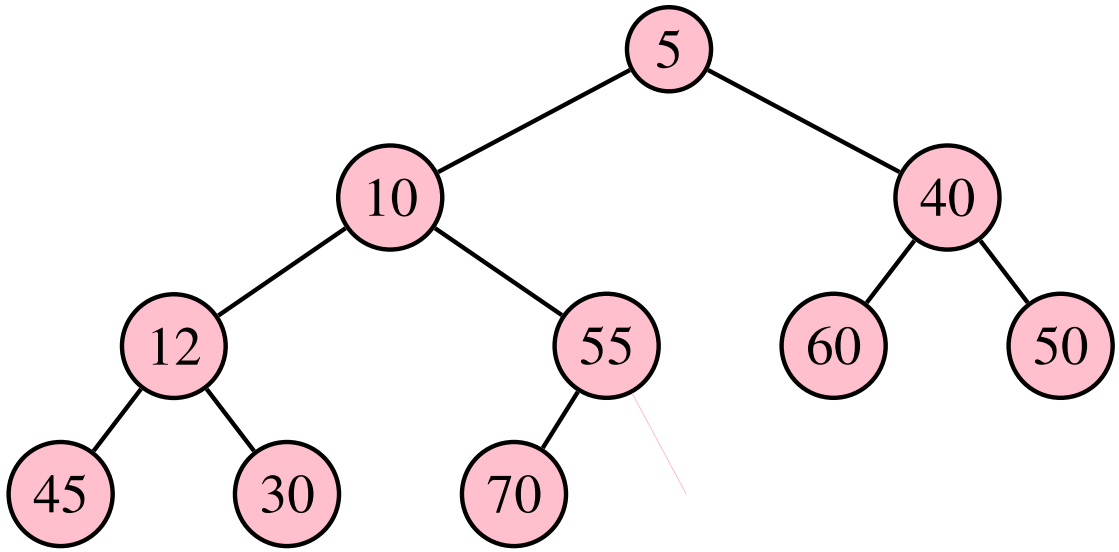
Min Heap – DeleteMin

Example: `DeleteMin` – re-organize heap to ensure the ordering property is satisfied.



Min Heap – DeleteMin

Example: `DeleteMin` – **Done**



Min Heap – DeleteMin

- In the worst case,
  - → Takes $\Theta(1)$ to remove the root element
  - → Takes $\Theta(\log n)$ comparisons in the worst-case to ensure that the heap ordering property is satisfied

Min Heap – DeleteMin

- In the worst case,
  - → Takes $\Theta(1)$ to remove the root element
  - → Takes $\Theta(\log n)$ comparisons in the worst-case to ensure that the heap ordering property is satisfied
- On average, need to perform $\Theta(\log n)$ comparisons to re-order the heap
  - → On average, insertion requires a smaller number of comparisons since it usually finishes prematurely. `DeleteMin` requires the promoted leaf to sink back to near the bottom of the tree.

Heap Sort

- Input: a list of n integers
- **Heap Sort Algorithm**:
  - Store the list elements in a min heap (build heap) -
  - Repeatedly remove minimum element from input list and insert at end of sorted list (remove min performed n times).
    - → Build a heap – $\Theta(n)$ on average and at worst.
    - → Delete the minimum n times – $\Theta(n \log n)$ on average and at worst.

Heap Sort

- Input: a list of n integers
- **Heap Sort Algorithm**:
  - Store the list elements in a min heap (build heap) -
  - Repeatedly remove minimum element from input list and insert at end of sorted list (remove min performed n times).
    - → Build a heap – $\Theta(n)$ on average and at worst.
    - → Delete the minimum n times – $\Theta(n \log n)$ on average and at worst.
- If the input list is stored in an array
  - → The heap can be built in place (in the same array).
  - → The sorted list is constructed going backwards from the end of the array.
  - → After all `deleteMin` operations are performed the array is sorted in decreasing order from left to right - reverse the array to obtain increasing order.
  - → To avoid the time overhead due to reversing, use a **max heap** and `deleteMax`.

Min Heap – Build

- **Build heap**: Given n elements, construct a heap to store them.
- To build a heap with n elements, you can just do an insert operation n times
  - → $\Theta(n \log n)$ (worst case), $\Theta(n)$ (average case)

Min Heap – Build

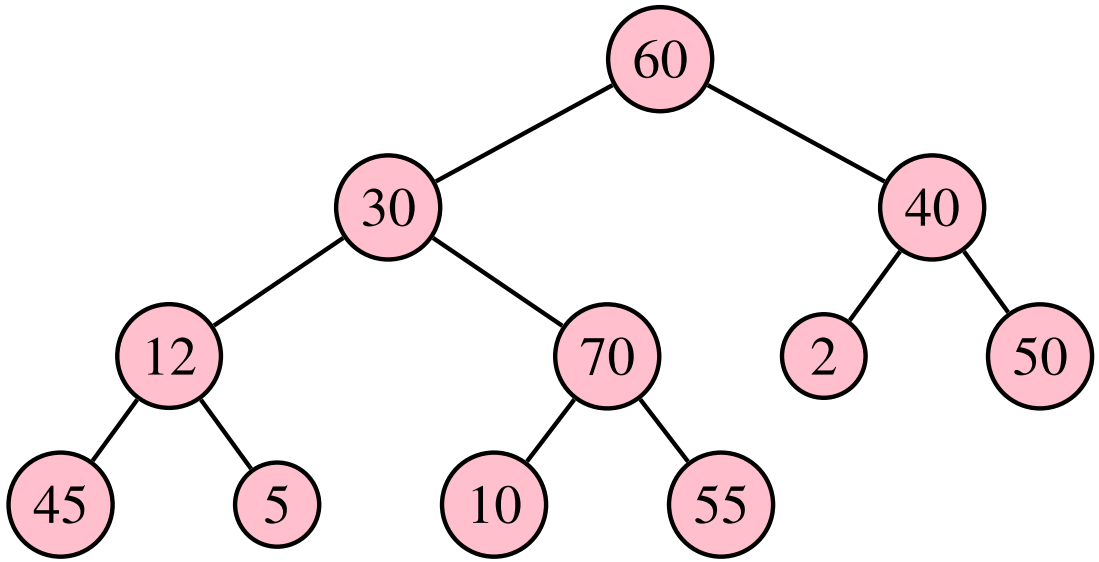
- **Build heap**: Given n elements, construct a heap to store them.
- To build a heap with n elements, you can just do an insert operation n times
  - → $\Theta(n \log n)$ (worst case), $\Theta(n)$ (average case)
- It is possible to build a heap faster, however, if there are no intervening deletions
  - → $\Theta(n)$ (*worst case*)
  - → Insert n elements into the array (unsorted)
  - → Build the heap by doing repeated `percolateDown` operations starting at parents of leaves.
  - → see details in Weiss (`buildHeap` - linear time heap construction)

Min Heap – Build

- **Build heap**: Given n elements, construct a heap to store them.
- To build a heap with n elements, you can just do an insert operation n times
  - → $\Theta(n \log n)$ (worst case), $\Theta(n)$ (average case)
- It is possible to build a heap faster, however, if there are no intervening deletions
  - → $\Theta(n)$ (*worst case*)
  - → Insert n elements into the array (unsorted)
  - → Build the heap by doing repeated `percolateDown` operations starting at parents of leaves.
  - → see details in Weiss (`buildHeap` - linear time heap construction)
- ☞ The **build heap** operation can be done in at worst $\Theta(n)$ operations.

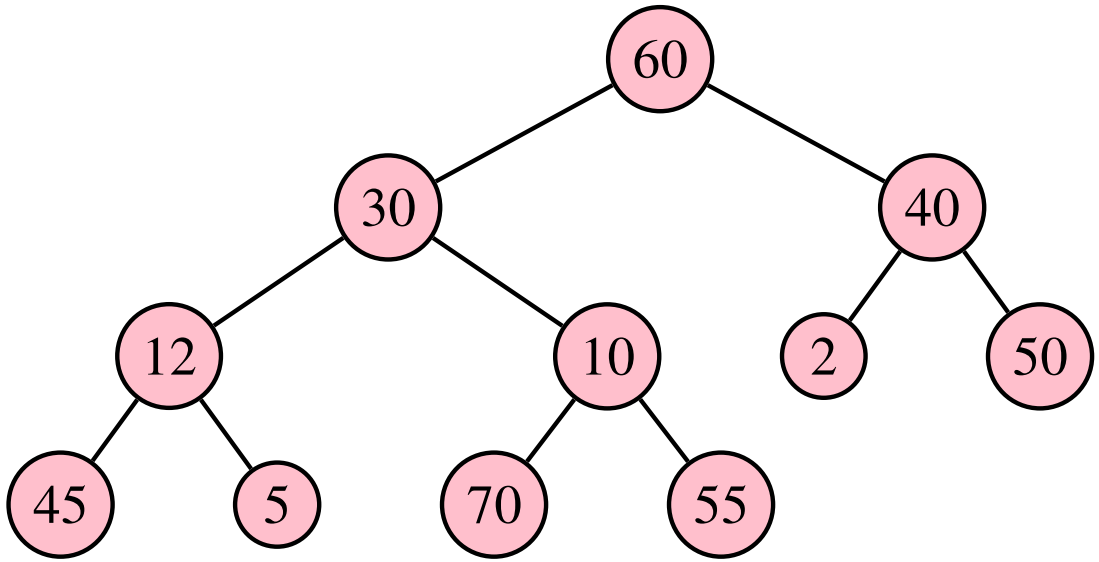
Example: Input array: [60, 30, 40, 12, 70, 2, 50, 45, 5, 10, 55]

Build Heap



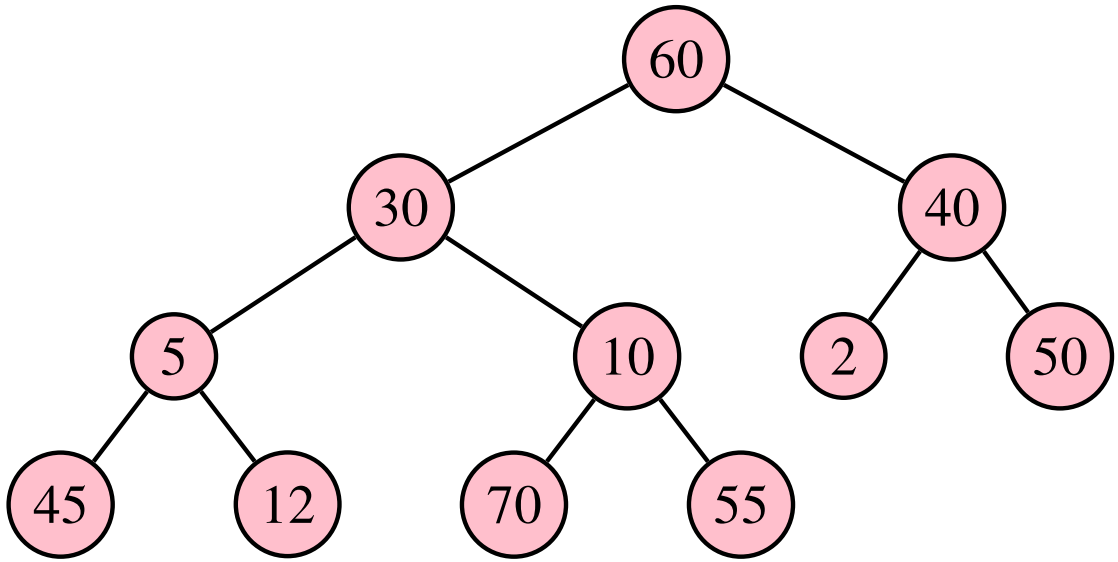
Example: Array: [60, 30, 40, 5, 10, 2, 50, 45, 12, 70, 55]

Build Heap



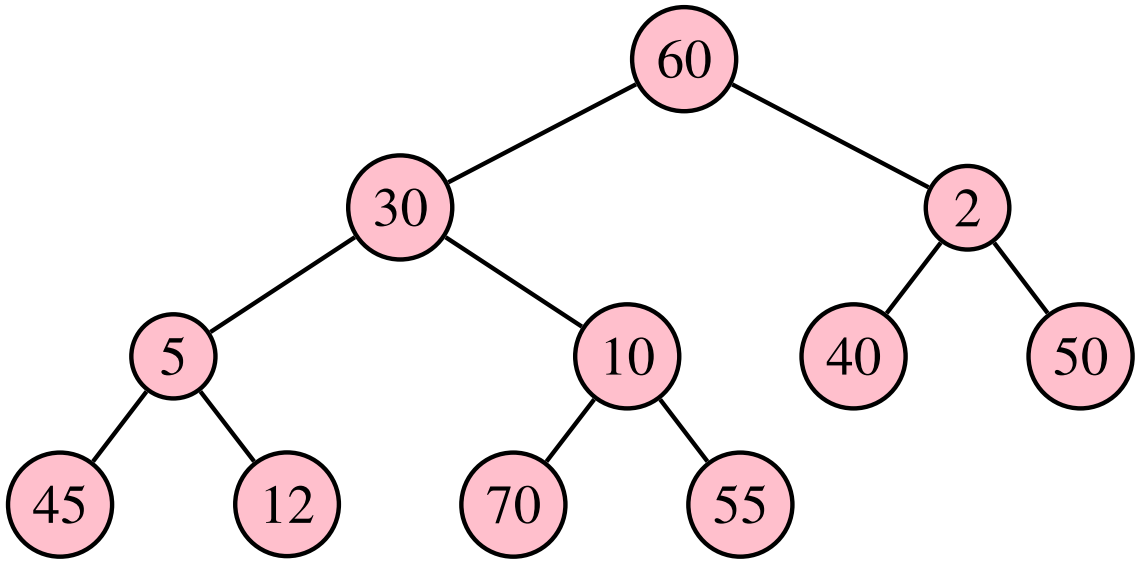
Example: Array: [60, 30, 40, 5, 10, 2, 50, 45, 12, 70, 55]

Build Heap



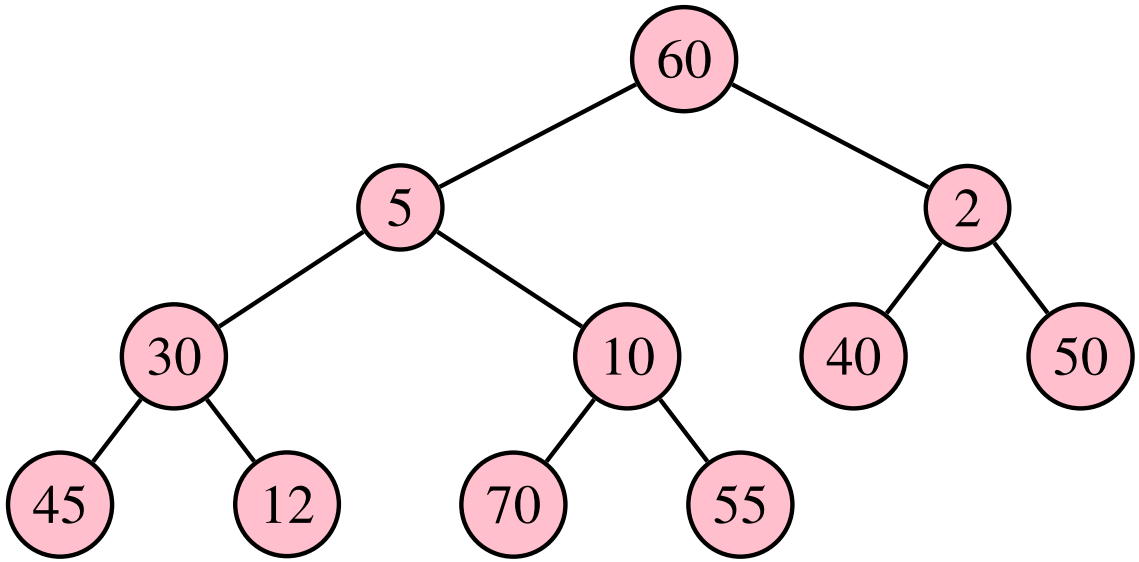
Example: Array: [60, 5, 2, 30, 10, 40, 50, 45, 12, 70, 55]

Build Heap



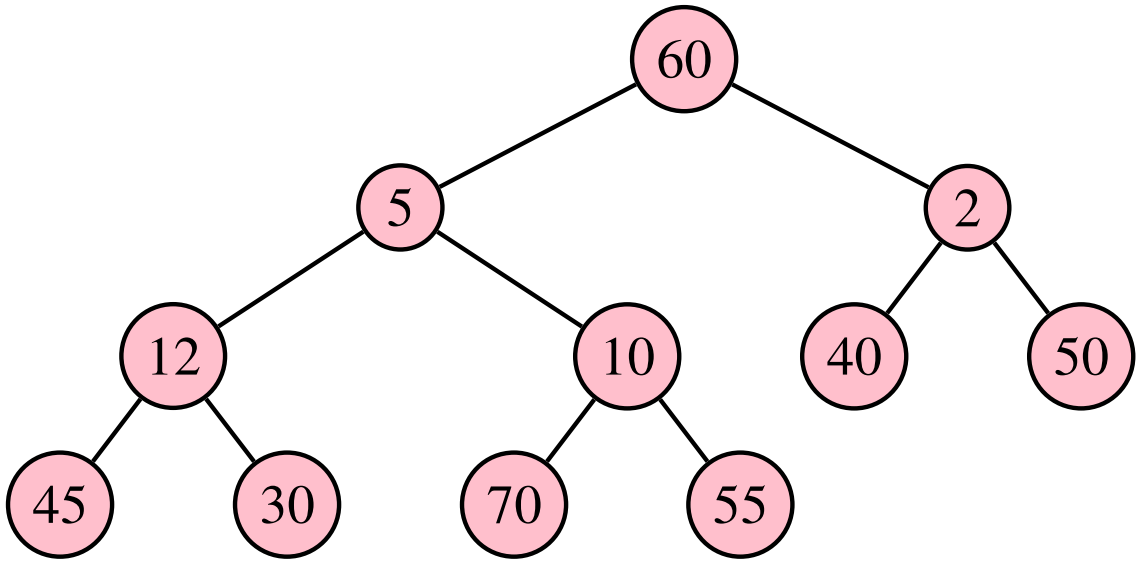
Example: Array: [60, 5, 2, 30, 10, 40, 50, 45, 12, 70, 55]

Build Heap



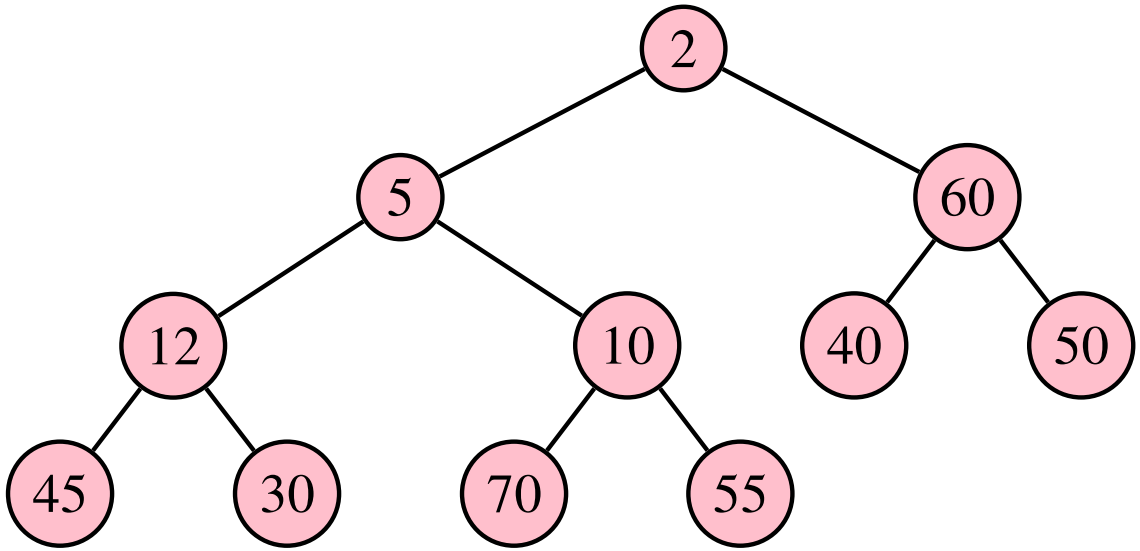
Example: Array: [60, 5, 2, 30, 10, 40, 50, 45, 12, 70, 55]

Build Heap



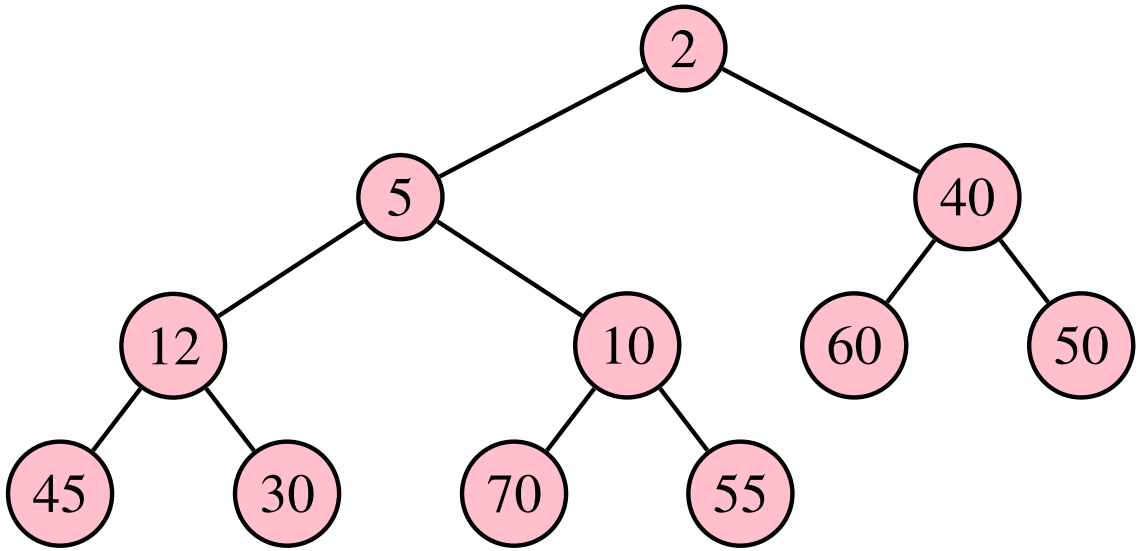
Example: Array: [2, 5, 60, 12, 10, 40, 50, 45, 30, 70, 55]

Build Heap



Example: Array: [2, 5, 40, 12, 10, 60, 50, 45, 30, 70, 55]

Build Heap - **Done**



Min Heap – Build

Maximum number of swaps required to build min heap (using multiple percolateDown operations) where h is height of the heap

$$\begin{aligned}\sum_{i=0}^{h-1}(h-i)2^i &= \sum_{i=0}^{h-1}h2^i - \sum_{i=0}^{h-1}i2^i \\ &= h(2^h-1) - ((h-2)2^h+2) \\ &= 2^{h+1} - h - 2\end{aligned}$$

Substituting $h = \log_2 n$ gives worst case runtime of

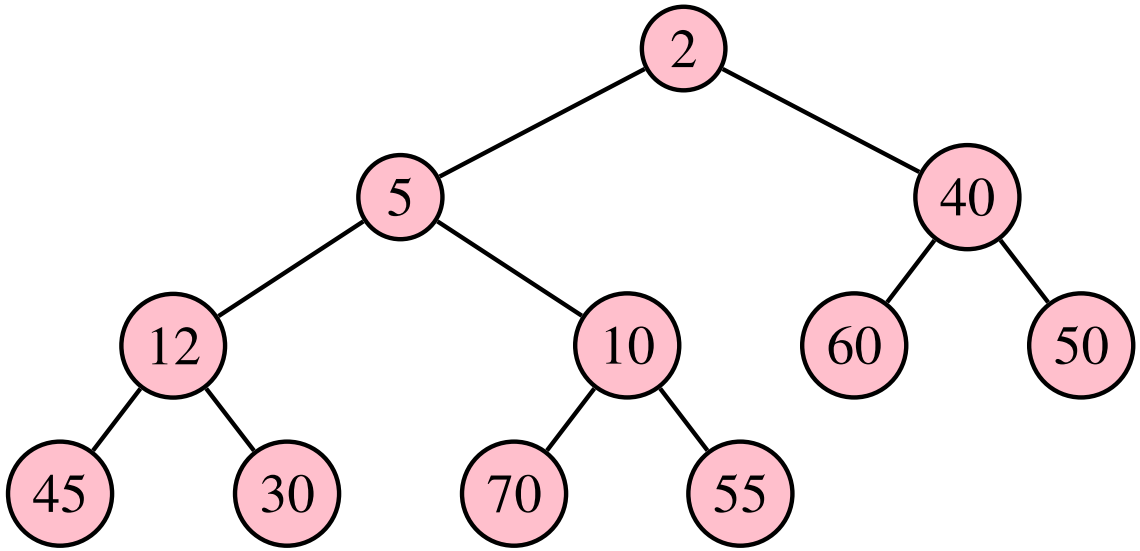
$$\Theta(n)$$

Heap Sort

Example: Array: [2, 5, 40, 12, 10, 60, 50, 45, 30, 70, 55 ||]
Heap Sort - To sort in place:

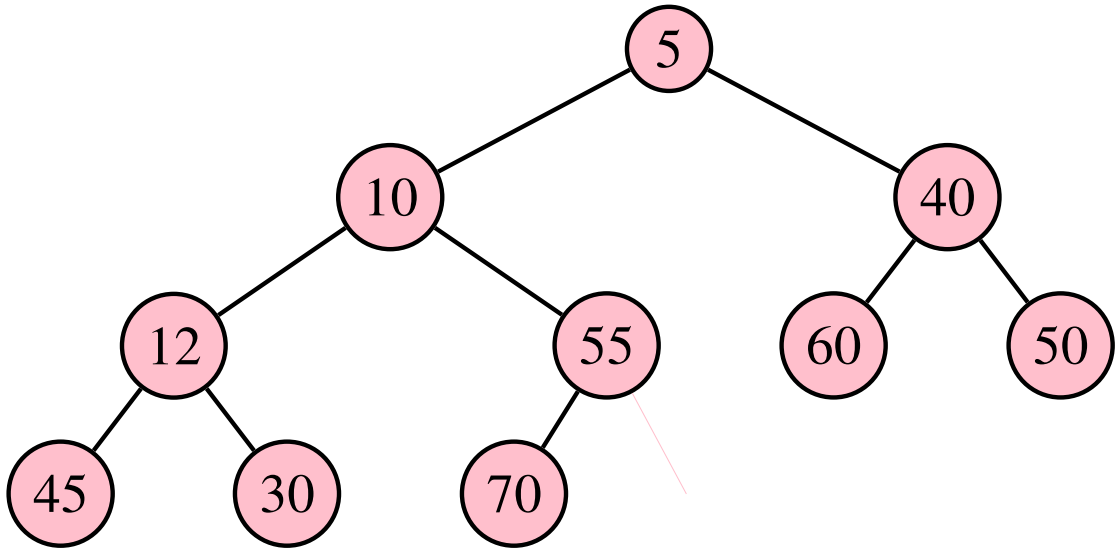
- → `deleteMin` and put value at end of array
- → Symbol || marks the end of the heap.
- → As the heap shrinks and the sorted list grows backwards from the end of the array || moves towards the beginning of the array.

Example: **Initial Heap**:
Array: [2, 5, 40, 12, 10, 60, 50, 45, 30,70, 55 ||]



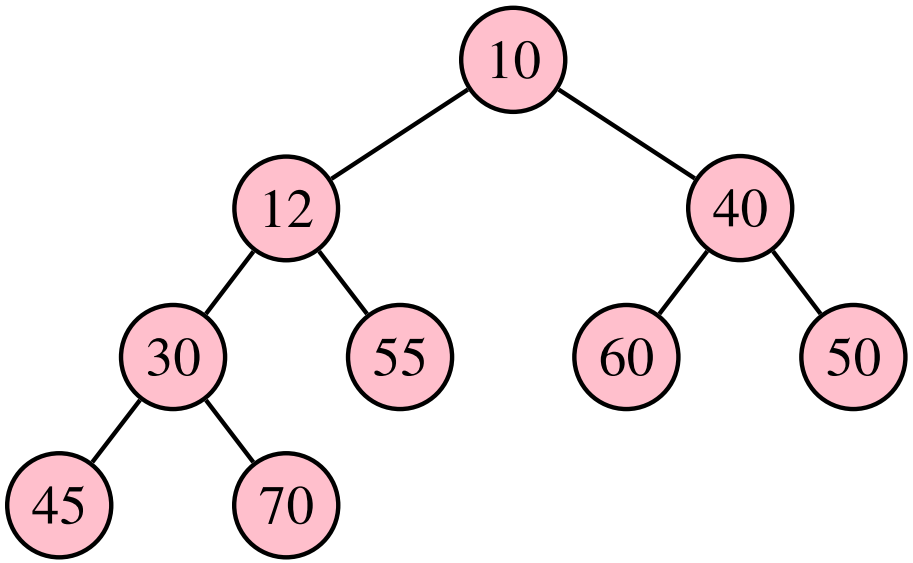
Example: **DeleteMin**: 2

Array: [5, 10, 40, 12, 55, 60, 50, 45, 30, 70 || 2]



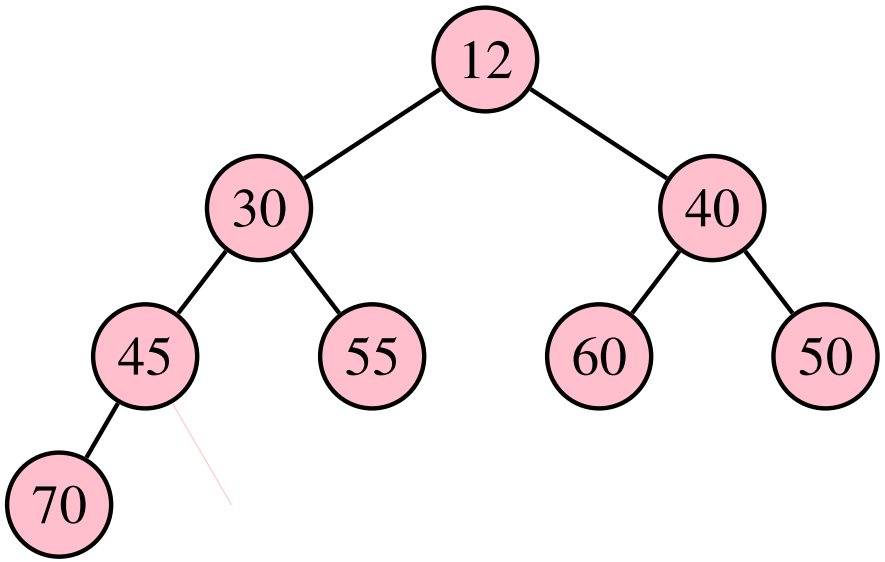
Example: **DeleteMin**: 5

Array: [10, 12, 40, 30, 55, 60, 50, 45, 70 || 5, 2]



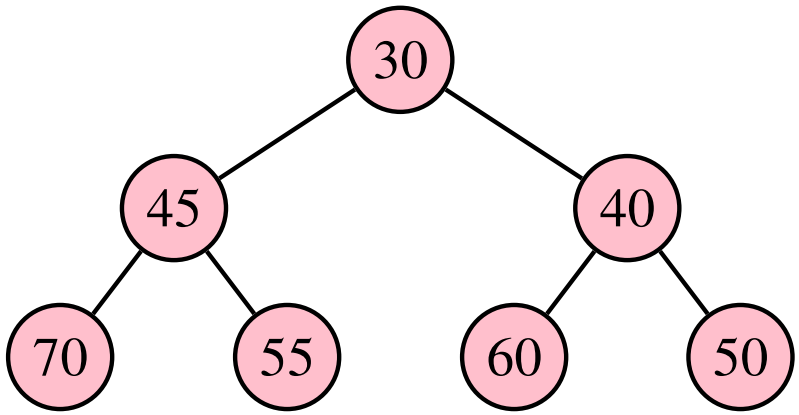
Example: **DeleteMin**: 10

Array: [12, 30, 40, 45, 55, 60, 50, 70 || 10, 5, 2]



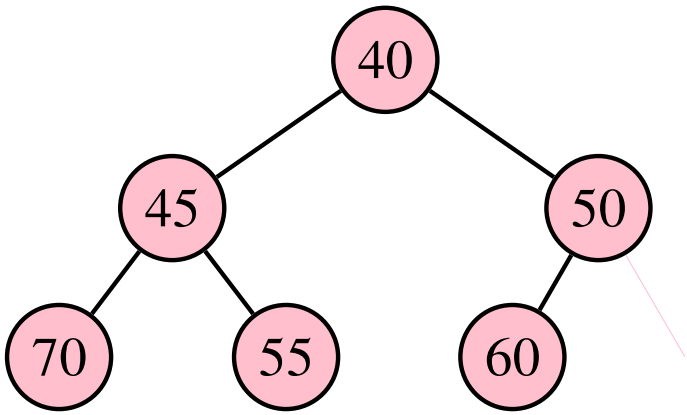
Example: **DeleteMin**: 12

Array: [30, 45, 40, 70, 55, 60, 50 || 12, 10, 5, 2]



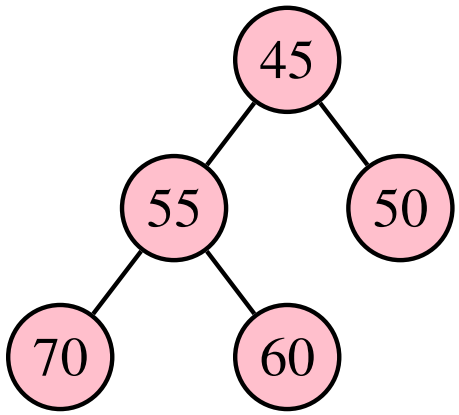
Example: **DeleteMin**: 30

Array: [40, 45, 50, 70, 55, 60 || 30, 12, 10, 5, 2]



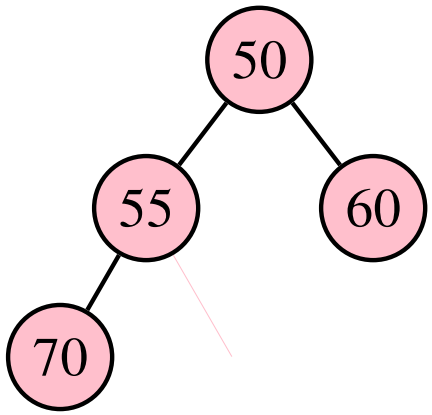
Example: **DeleteMin**: 40

Array: [45, 55, 50, 70, 60 || 40, 30, 12, 10, 5, 2]



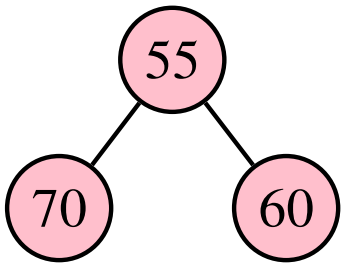
Example: **DeleteMin**: 45

Array: [50, 55, 60, 70 || 45, 40, 30, 12, 10, 5, 2]



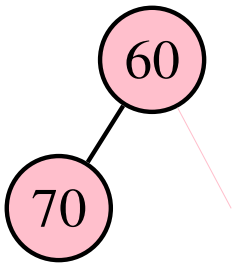
Example: **DeleteMin**: 50

Array: [55, 70, 60 || 50, 45, 40, 30, 12, 10, 5, 2]



Example: **DeleteMin**: 55

Array: [60, 70 || 55, 50, 45, 40, 30, 12, 10, 5, 2]



Example: **DeleteMin**: 60

Array: [70 || 60, 55, 50, 45, 40, 30, 12, 10, 5, 2]

Example: **DeleteMin**: 70

Array: [|| 70, 60, 55, 50, 45, 40, 30, 12, 10, 5, 2]

- The list is sorted in descending order. To get list is ascending order:
  1. The array can be reversed in $\Theta(n)$ time with $\Theta(1)$ space (how?)
  2. You can start initially with a **max heap** instead of a min heap.

Reversed array: [2, 5, 10, 12, 30, 40, 45, 50, 55, 60, 70]

Done

Heap Sort. Notes

- Heap Sort is selection sorting technique which exploits the features of heaps. For a given data set,
  - → Build a heap – $\Theta(n)$ on average and at worst.
  - → Delete the minimum n times – $\Theta(n \log n)$ on average and at worst.

Heap Sort. Notes

- Heap Sort is selection sorting technique which exploits the features of heaps. For a given data set,
  - → Build a heap – $\Theta(n)$ on average and at worst.
  - → Delete the minimum n times – $\Theta(n \log n)$ on average and at worst.

Key Points:

- This is a $\boxed{\Theta(n \log n)}$ sorting algorithm even in the worst case!
- In practice, quick sort is faster.
- Can sort arrays in place, i.e., do not require additional memory.
  - Can do this by adding the sorted elements at the end of the heap as the tree shrinks

Sorting. Concluding Remarks

- Sorting is a fundamental activity in a large number of computer systems.
- The sorting algorithm that is chosen depends on the size of the list and on the underlying environment
  - → Very small inputs - Insertion Sort is preferred.
  - → Large inputs - Mergesort, Quicksort, Heapsort are efficient
    - Mergesort, Heapsort - $\Theta(n \log n)$ running time in the worst case and on average.
    - Quicksort $\Theta(n \log n)$ running time on average, $\Theta(n^2)$ in the worst case; very fast in practice.
- In most cases, Quicksort is a good technique for sorting general lists. It was used in `Java` to sort arrays of primitive types and in `C`, `C++` for general arrays.

Sorting. Concluding Remarks (cont'd)

- Lately, **hybrid** sorting algorithms have gained popularity (e.g.,**Introsort, Timsort**). They combine various sorting techniques.
  - → **Introsort** starts with Quicksort and switches to Heapsort when the recursion depth exceeds some level. Used in Microsoft's C++ library.
  - → **Timsort** blends Mergesort and Insertion Sort and takes advantage of existing partial orderings in the data. Invented by Tim Peters. Used in Python.
  - → Java uses a variant of **Timsort** for arrays of objects and a variant of Quicksort (Dual-Pivot Quicksort) for arrays of primitive types.