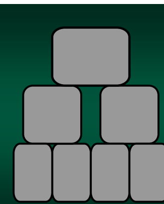
Heaps & Priority Queues

Section 2.4

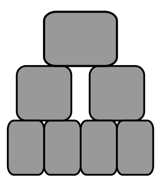
Heaps



Piles of data!

What is a heap?

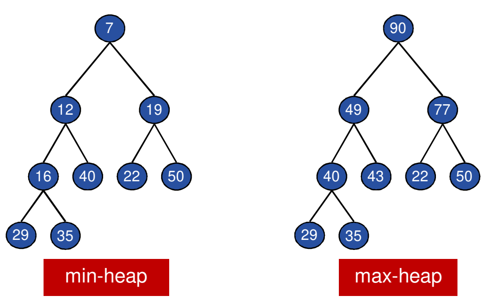
- A *heap* is a binary tree, but a notable format to the nodes
- The value of a node is smaller (or larger) than both of its children
- Every subtree is a heap



Min and max-heaps

- A *min-heap*
  - stores smaller items (minimal items) at the top of the tree
  - larger items are stored at the bottom
- A *max-heap*
  - stores larger items (maximum items) at the top of the tree
  - smaller items are stored at the bottom

Min and max-heaps

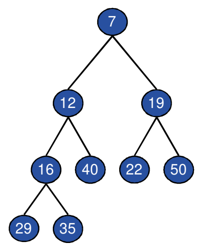


Terminology Warning

- The Heap ADT is not the same as the operating system's heap
- ADT is a tree that stores "heavier" objects at the bottom
- The system heap is a complex interwoven linked list

Heaps

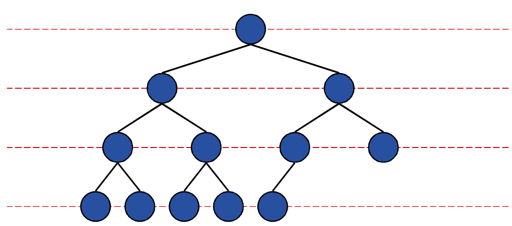
- Heaps are *complete* binary trees
- Nodes are added in breadth-first order
- The resulting tree is always optimal and *balanced*



Height of a Heap

- Let *h* be the height of the heap
- Let *i* be the depth of a node
- For all i = 0, … , h – 1
  - there are 2i nodes of depth i
  - at depth h - 1, the internal nodes are to the left of the external nodes
  - The last node of a heap is the rightmost internal node of depth h - 1

Height of a Heap



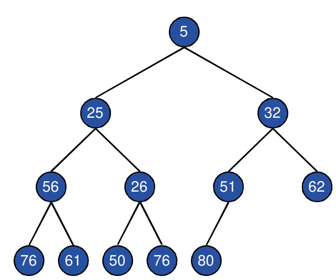
Why Use a Heap?

- Heaps are not as fast as B-Trees, but are far simpler for maintaining balance than AVL Trees
- We will cover these soon…
- Filling logic makes them a good idea for arrays
  - any array can be turned into a heap
  - … even if it *already* contains data
  - the data will need to be restructured to a proper heap
  - there is a sort algorithm based on this – *Heap Sort*

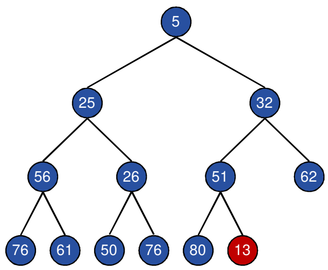
Adding an Node

- Begin at next available position for a leaf
- Now the item needs to be *up-heaped*
  - move the entry up depending on its value until a correct position is found
  - as this is done, nodes are swapped entries from parent to child change position
  - since a heap always has height O(log n), upheap runs in O(log n) time

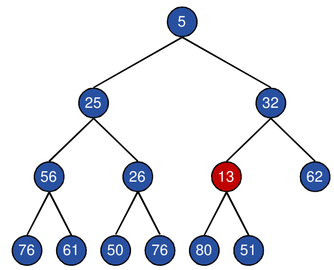
Minheap: Adding a Node 13



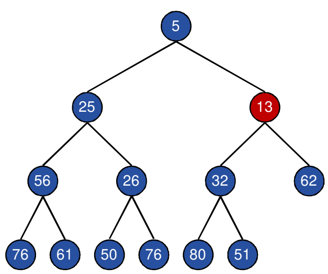
Upheaping 13



Upheaping 13



Upheaping 13: Done



Total Up-heap-val

- Just to make matters confusing, up-heap and down-heap are also known by various other terms – which are all valid
- These are some:
  - bubble-up
  - percolate-up
  - sift-up
  - heapify-up
  - cascade-up

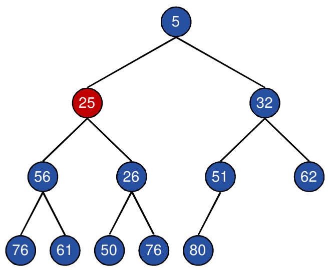
Deleting a Node

- Deleting a node is quite different from adding
- Remember, heaps must maintain *completeness*
- So, the right-most leaf will be involved
- Deletion:
  - remove the node and replace it will the right-most leaf
  - now, this node needs to *down-heaped* (moved down) to the correct location
  - since a heap always has height O(log n), down-heap runs in O(log n) time

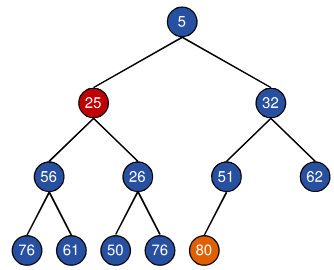
Downheap Algorithm

- With a heap, every node has two children
  - as you downheap, you swap nodes
  - so which one do you select?
- Preserve the heap structure ← **vital**
  - on a min-heap, swap with the smallest child
  - on a max-heap, swap with the largest child

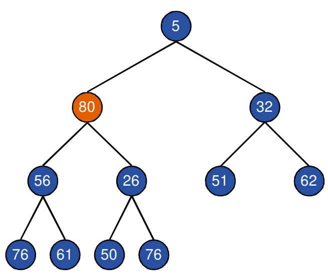
Minheap: Deleting 25



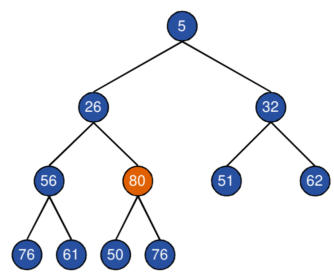
Deleting 25: Replace



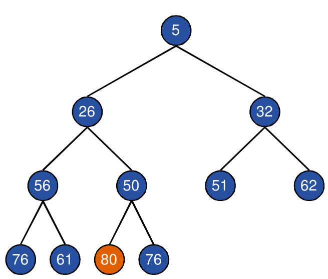
Deleting 25: Downheaping



Deleting 25: Downheaping



Deleting 25: Downheaping



As You Expected...

- Just like up-heap, down-heap has several other, completely valid, names
- These are some:
  - bubble-down
  - percolate-down
  - sift-down
  - heapify-down
  - cascade-down

Merging Heaps

Whoa, this is easy!

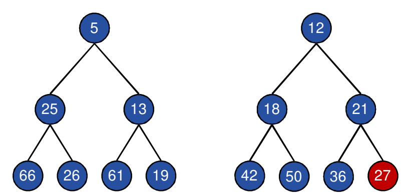
Merging Heaps

- Merging two heaps is actually quite easy
- One approach is to read one heap into another....
  - read all the data from one heap and add it to the second
  - requires O(n log n)

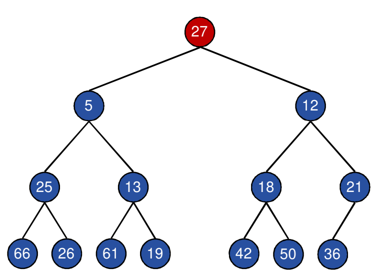
Merging Heaps

- Or, we can create a new root
  - remember: every subtree in a heap – is a heap
  - so, both trees can be added as a left / right subtree
  - just grab a node at the base of one, make it the root, and downheap
  - requires O(log n)

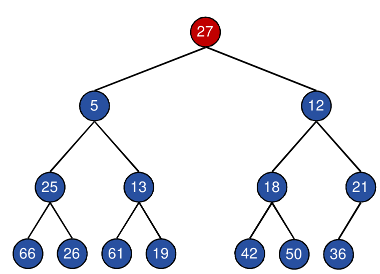
Merge: Create New Root



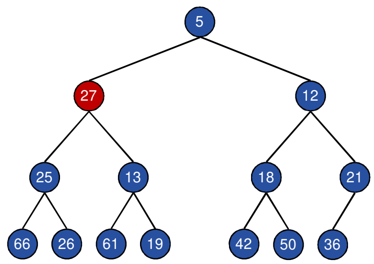
Merge: Create New Root



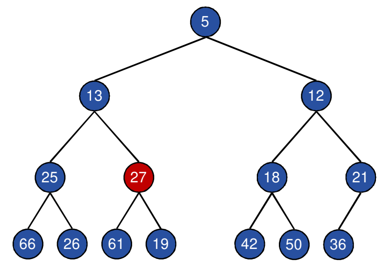
Merge: Downheap



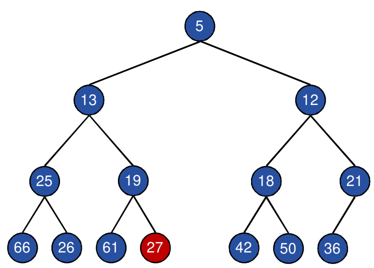
Merge: Downheap

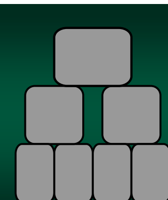


Merge: Downheap



Merge: Done



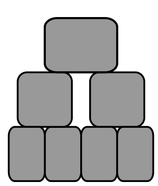


Heaps and Arrays

Not a Heap O' Trouble

Heaps and Arrays

- Heaps are *complete*, balanced, binary trees
- This rigid, predictable, structure...
  - lends itself to being stored in an array
  - each node has a pre-ordained location



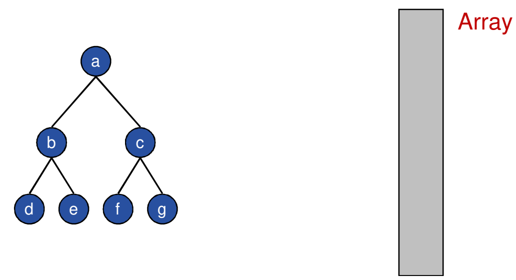
Heaps and Arrays

- Using an array, links between items…
  - are not explicitly stored
  - finding array index of an item's children and parent can be found using simple mathematics
- Heap ADTs only need to…
  - track the index of the end of the heap
  - all new items are added here – before upheap
  - and this is where the last item will be swapped for a deleted item (before it is downheaped)

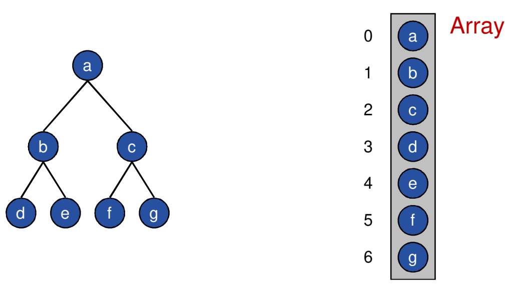
Heap Array Math

Find	0 Indexed Array	1 Indexed Array
Parent of node i	`(i – 1) / 2`	`i / 2`
Left child of node i	`(2 * i) + 1`	`2 * i`
Right child of node i	`(2 * i) + 2`	`(2 * i) + 1`

Heap in an Array



Heap in an Array



Priority Queues

Queues can play favorites

Priority Queues

- A stack is first-in-last-out
- A queue is first-in-first-out
- A priority queue is *first-in-least-out*

First-in-Least-Out

- "least" element is the first one removed
- If two items have the same "rank", items can be queued as normal
- The value that is used to determine an item's value is called a "key"

What is the "Least" Item?

- Meaning of "least" is defined by the ADT
- It is an abstract term and does not mean "minimal"
  - so, "least" can be any way of ranking items
  - ...if the items are mathematically transitive
  - "least" can be the largest value
- Examples
  - by the smallest / largest value
  - size of the data (e.g. files)

Typical Priority Queue ADT

- Main Operations:
  - add (object) : add the object to the priority queue
  - removeLeast : removes and returns the least item
  - getLeast : returns the least element
  - isEmpty : returns true if the PQ is empty

Implementation

- Before we select a data structure to implement a priority queue, we should look how data will be used
- The goal is to get the best time efficiency with as little overhead
- Know what data will be stored will influence how the priority queue is implemented and, importantly, *how*

Implementation

- We can use a basic data structure
  - array
  - linked-list
  - tree
- Or another ADT
  - B-Tree
  - Queue
  - Heap

Implementation with an Array

- Unsorted array
  - adding requires O(1)
  - removing requires O(n) – search and moving
- Sorted array
  - adding requires O(n) – find location, move rest
  - removing requires O(1) – if the head of the queue is at the array end
- Both approaches are inefficient

Implementation with a Linked List

- Unsorted linked list
  - adding requires O(1)
  - removing requires O(n) – find and delete node
- Sorted linked list
  - adding requires O(n) – find position and insert
  - removing requires O(1) – just remove the head/tail
- Just as inefficient as pure arrays

Implementation with a Binary Tree

- Unbalanced B-Tree
  - adding requires O(log n) to O(n)
  - removing requires O(log n) to O(n)
- Balanced binary tree
  - adding requires O(log n)
  - removing requires O(log n)
  - rebalancing requires an extra O(log n) time

Implementation with a Heap

- A priority queue can be implemented as a heap
- Remember...
  - in a heap, all the items below a node have a greater value
  - if that value is used as a priority key, the items on the top of the heap is the top of the queue

Implementation with a Heap

- *Heaps naturally implement a priority queue*
- To enqueue an item...
  - just add to it the heap
  - it will up-heap to the correct position
  - requires O(log n)
- To dequeue an item
  - just delete the root
  - requires O(log n) rebalance

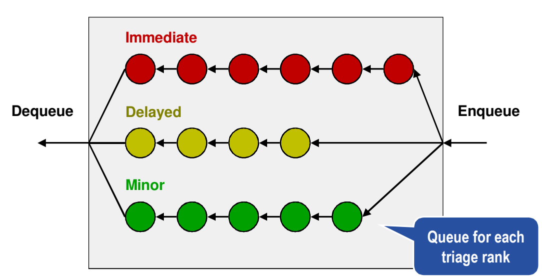
Hybrid Implementations

- In some cases, the key value can have a minor range of values – possibly just a few
- Examples:
  - hospital triage – immediate, delayed, minor
  - computer processes – OS, application, GUI
- We can make clever hybrid structures that maximize efficiency

Hybrid Implementations

- If the key contains a small number of values, you can use multiple queues – one for each key value
- Basically, the priority queue, internally, will have an array of queues
- Adding/removing items will always be O(1)
  - O(1) for the queue head
  - O(1) for enqueue/dequeue (using a linked list)

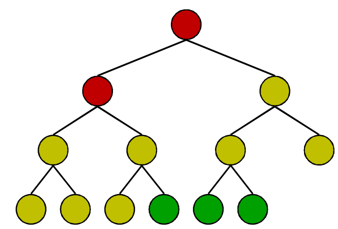
Hospital Triage – Array of Queues

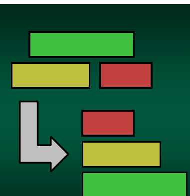


... But Heaps are Universal

- However, in most cases, the key values have large ranges
- For example, if the key is a 32-bit integer, do you want to create 4 million queues?
- Didn't think so….
- The pure Heap implementation works in all cases

Hospital Triage - Heap



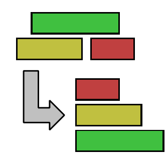


Heap Sort

Heap-a-rific algorithm!

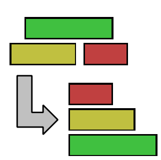
Heap Sort

- *Heap Sort* is an ingenious sort algorithm that uses a max-heap to sort an array of elements
- It takes advantage of ...
  - heap is a natural priority queue
  - heaps always add / remove from the right-most leaf



Heap Sort Works in Two Phases

- Phase 1: *Heapify*
  - the sort first converts the existing array into a heap
- Phase 2: Empty heap
  - removes all the nodes (treating it as priority queue)
  - sorted data is added to the end of the array



Implementation

- Both the "heap" and the remaining array can be used in memory at the same time
- The sorted array is stored at the empty space after the end of the heap
- This concept works for both Phase 1 and Phase 2

Phase 1: Array → Heap

- In Phase 1, we convert the array into a maxheap. This step is called *heapify*.
- Remember....
  - a heap can be stored in an array
  - so, we can just look at the array as a heap
  - ...but, its not quite a heap yet
  - data needs to be moved around until it is a heap

How do we convert it?

- First approach: *top-down*
  - start building the heap at the top of the array
  - iterate the variable *i* starting from the first element and build a heap above *i*
  - this is the easiest to conceptualize
- Second approach: *bottom-up*
  - fastest approach is to downheap all the leaves
  - leaves are always located at the second half of the array
  - so, we run the downheap, at the root, all he leaves

Phase 1: Heapify

```
n = count-1;  //last item

while (n >= 0)
{
    heapDown(array, n, count-1)
    n--;
}
```

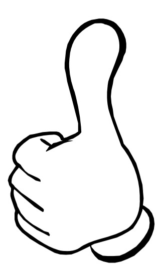

Phase 1: Heapify

```
heapify(array, count)
{
   last = count - 1;
   n = last;  //last item

   while (n >= 0)
   {
      downHeap(array, n, last)
      n--;
   }
}
```

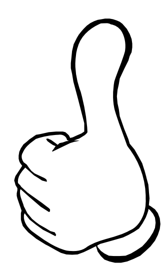

Phase 2: Root Deletion

- Now that the array is a maxheap, the root contains the maximum item
- If we remove the root, we have the last item in a sorted array!
- OMG! Sooooo, awesome!



Phase 2: Root Deletion

- Remember, when we remove the root...
  - right-most leaf is moved to the root and downheaped into the correct position
  - this leaf position is now empty



Phase 2: Root Deletion

- We can put the root, we just removed, in this new empty space
- *What a sec!* We just put the <u>largest</u> item in the <u>last</u> position in the array
- The value, <u>now</u> at the root of the heap, is the second largest item in the array

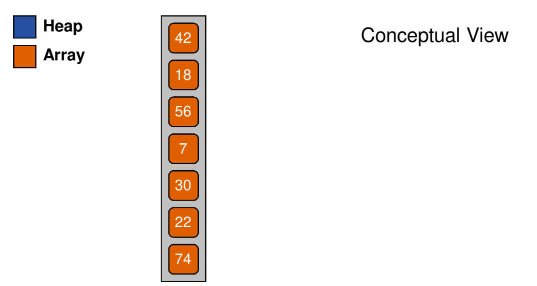
Phase 2: Root Deletion

- So, to sort the array....
  - so, we just keep removing the root and placing it position where the leaf was located
  - the "heap" section of the array shrinks as the sorted array grows from the bottom
  - wow, that's easy!

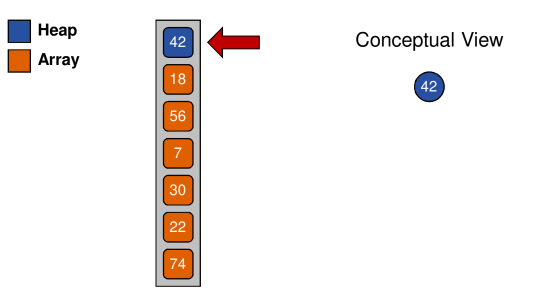
Heap Sort Algorithm

```
last = count - 1;
heapify(array, 0, last);
while (last > 0)
{
  // swap root and array[last]
  downHeap(0, last - 1);
  last--;
}
```

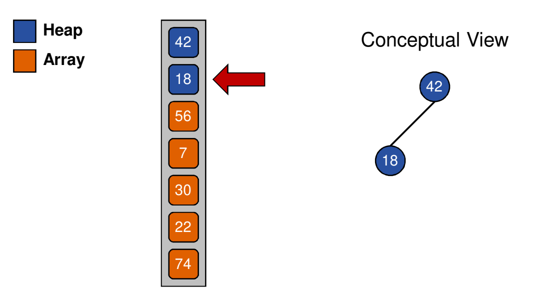

Phase 1 – Top-down Convert



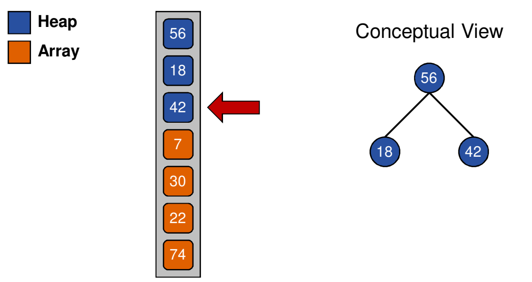
Phase 1 – Top-down Convert



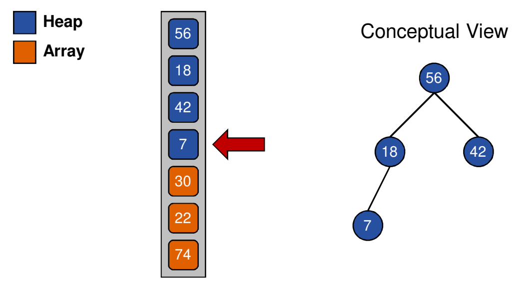
Phase 1 – Top-down Convert



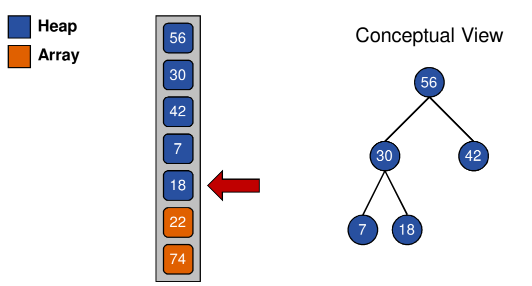
Phase 1 – Top-down Convert



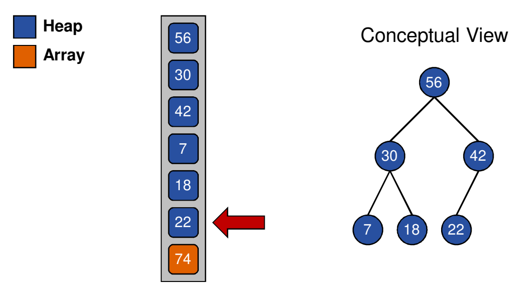
Phase 1 – Top-down Convert



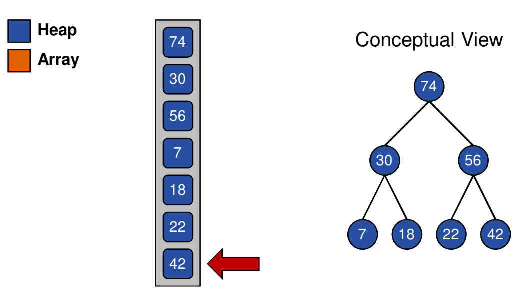
Phase 1 – Top-down Convert



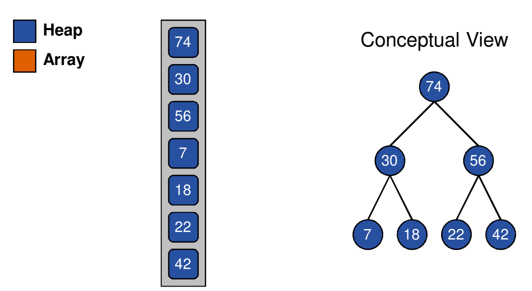
Phase 1 – Top-down Convert



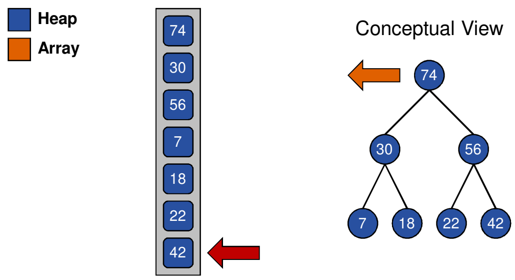
Phase 1 – Top-down Convert



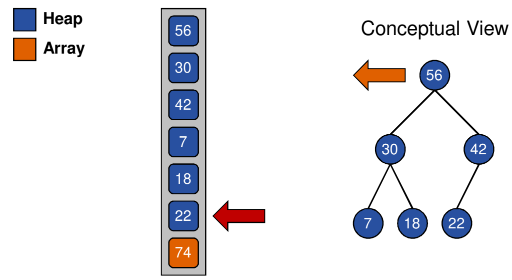
Phase 1 – Complete!



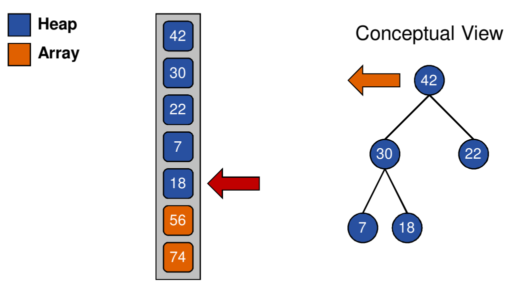
Phase 2 – Remove Roots



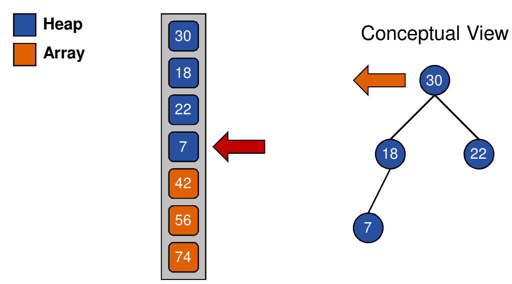
Phase 2 – Remove Roots



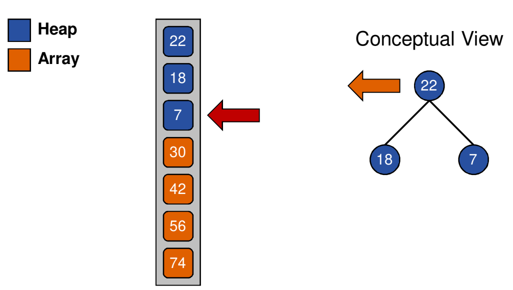
Phase 2 – Remove Roots



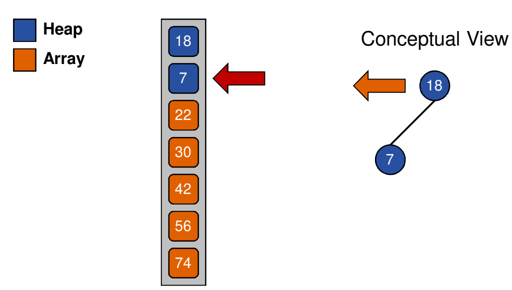
Phase 2 – Remove Roots



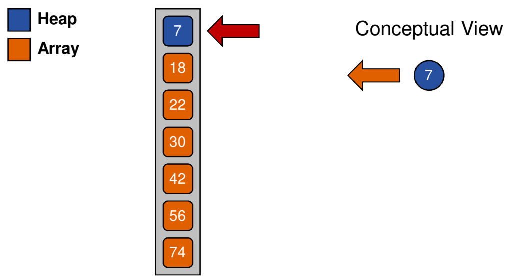
Phase 2 – Remove Roots



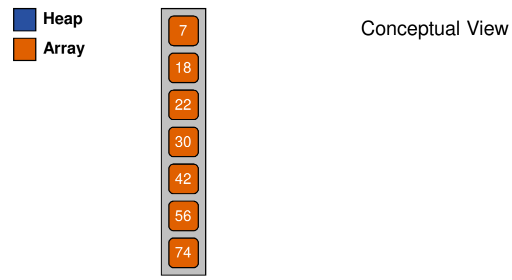
Phase 2 – Remove Roots



Phase 2 – Remove Roots



Phase 2 – Complete!



Merge Sort vs. Heap Sort

- Heap-Sort allows us to sort any array in O(n log n) just like Merge-Sort & Quicksort
- However, there is no overhead
  - Heap-Sort can be sorted in-place, meaning auxiliary storage is O(1)
  - Merge-Sort, however, requires O(n)
  - Quick-Sort can become $O(n^2)$

Merge Sort vs. Heap Sort

- However, in some cases, the recursive nature of Merge Sort is better
  - easy to distribute to multiple computers
  - Heap-Sort uses the entire array – not online
- But...in the Real World, it gets complex
  - you can cut an array into sub-lists, send them to different machines which Heap-Sort them
  - ... and then you Merge

Heap Sort Summary

Heap Sort	
Time Average	O(n log n)
Time Best	O(n log n)
Time Worst	O(n log n)
Auxiliary space	O(1)
Stable	No – Equal element order not preserved
Online?	No

Summary of Sorting Algorithms

Algorithm	Best	Average	Worst	Aux. Storage
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Shell Sort	O(n log n)	$O(n^{5/4})$	$O(n^{3/2})$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(1)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)