COMP1002 DATA STRUCTURES AND ALGORITHMS

LECTURE 7: HEAPS

Curtin University
Discipline of Computing

Last updated: September 22, 2019

Copyright Warning

This Week

- Priority queues
- Heaps
  - Array representation of binary trees
  - Analysis of Heap efficiency
  - MaxHeap vs MinHeap
- HeapSort and comparison to other O(N log N) sorts

Priority Queues

- We've talked about FIFO queues in earlier lectures
- But there is another type of queue that is also fairly common: the Priority Queue
  - Two operations: add and remove
  - Items added to priority queue with an associated priority
    - Priority indicates how quickly the item must be dealt with
    - Highest priority item in the queue is always removed first
  - Priority-based processing is quite common. Examples:
    - Task scheduling for CPU execution by an operating system
    - Inventory ordering: low-stock and/or popular items are the most important (highest priority) to order
    - Preferential treatment for loyal and/or large customers

Priority Queues – Priority Definition

- The priority value is usually an integer
  - `void add(int priority, Object value)`
    - Could be a float, but that's less common
- But what constitutes "high priority"? Two options:
  - Higher integer values = higher priority
    - *e.g.,* bigger vs smaller
  - Lower integer values = higher priority
    - *e.g.,* first, second, third: like a race
  - These lectures will assume high value = high priority
    - Just makes it easier to keep it straight in your head!

Priority Queues – Implementation

- So how can we implement a priority queue ADT?
- So far, we only know of arrays and linked lists. Both have a fairly similar priority queue implementation:
  - Add: add them in sorted order according to priority
    - Requires searching through array/list to find insertion point
    - Averages N/2 steps, ie: Add = O(N)
  - Remove: simply take from the rear (since highest-priority will be at the end when in sorted order)
    - Fast: Remove = O(1)
    - Note: We never remove anything but the highest-priority item

Priority Queues – Implementation

- An alternative is to avoid sorting the data and instead make it remove's problem to find the highest priority
  - Add: Append the item to end of array/list
    - Fast: Add = O(1)
  - Remove: Search through list to find highest-priority item
    - Must go through all N items just in case highest is last item
    - *i.e.*, Remove = O(N)
- Whichever alternative is taken, you cannot avoid having one of add or remove being O(N)
  - Can we do better than O(N)? Fortunately, yes: that's what a Heap data structure is for

Heaps

- The heap data structure is *not* the same as “the heap” used in programming languages to denote the area of memory used to allocate objects
- Heaps are organised in a binary tree (but NOT as a binary *search* tree) where the highest priority item is at the root, and lower-priority items are below
  - Requirement: children are always smaller than their parent
    - *i.e.,* a heap organises items (weakly sorted) from top to bottom
  - Thus it is *NOT* organised like a binary search tree, which requires that leftChild < parent < rightChild
    - *i.e.,* a binary search tree organises items from left to right

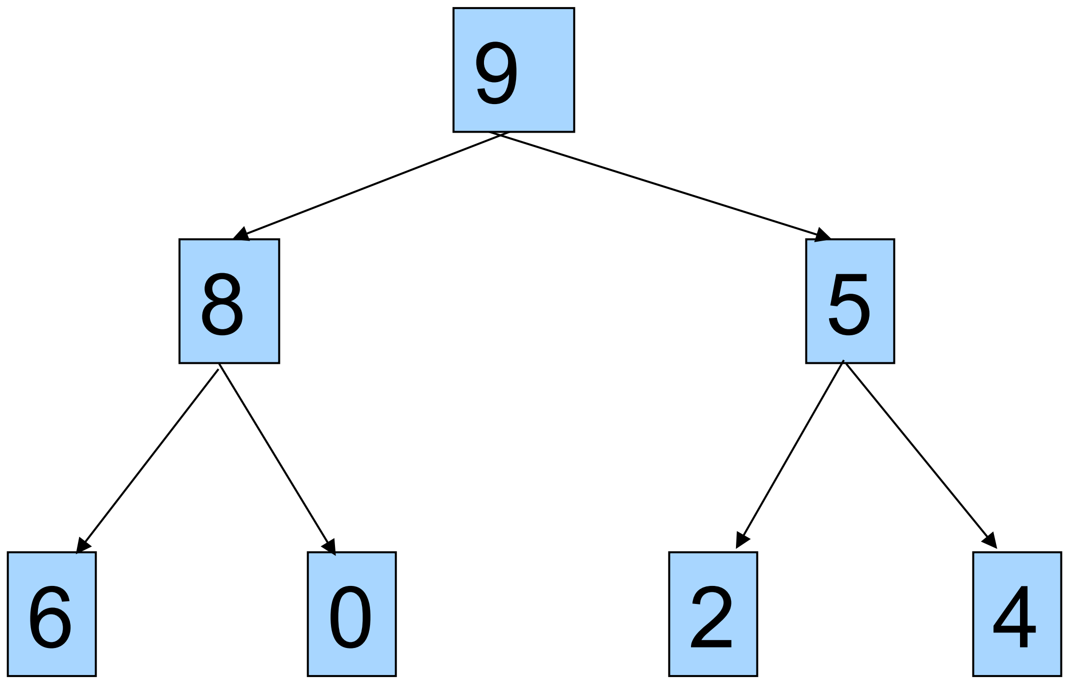
Heaps and Priority

- Since heaps are so closely associated with priority queues, they also explicitly define priority order
  - A max heap is a heap where a larger priority value is considered a higher priority
  - A min heap is a heap where a smaller priority value is considered a higher priority
- For the remainder of this lecture we will be working with max heaps (unless otherwise stated)

Heap Binary Tree – Properties

- The main constraint that a heap tree has is that each child must be of lower priority than its parent
  - This guarantees that the highest priority item is the root
  - It doesn't matter if the left child is larger or equal to the right child, or vice-versa
- By a little bit of clever algorithm design, the heap is also guaranteed to be always almost-complete
  - Thus always guaranteeing O(log N) access time
  - We will see how this is guaranteed when we discuss how add() and remove() work in a heap

(Max) Heap – Example



Heaps – Some Notes

- A heap mandates that children nodes are always of lower priority than their parents
  - This is enough to guarantee that the root is the highest priority item, which is enough for a priority queue
  - Ordering is vertical, but higher-priority items only *tend* to be higher up in the tree
    - Different subtrees may contain much different priorities
    - *e.g.,* 6 is lower in the tree than 5, but is of higher priority
  - Thus a heap is only *weakly* ordered
- Heaps can also contain duplicate priority items
  - Priority is *not* a unique key for lookup!

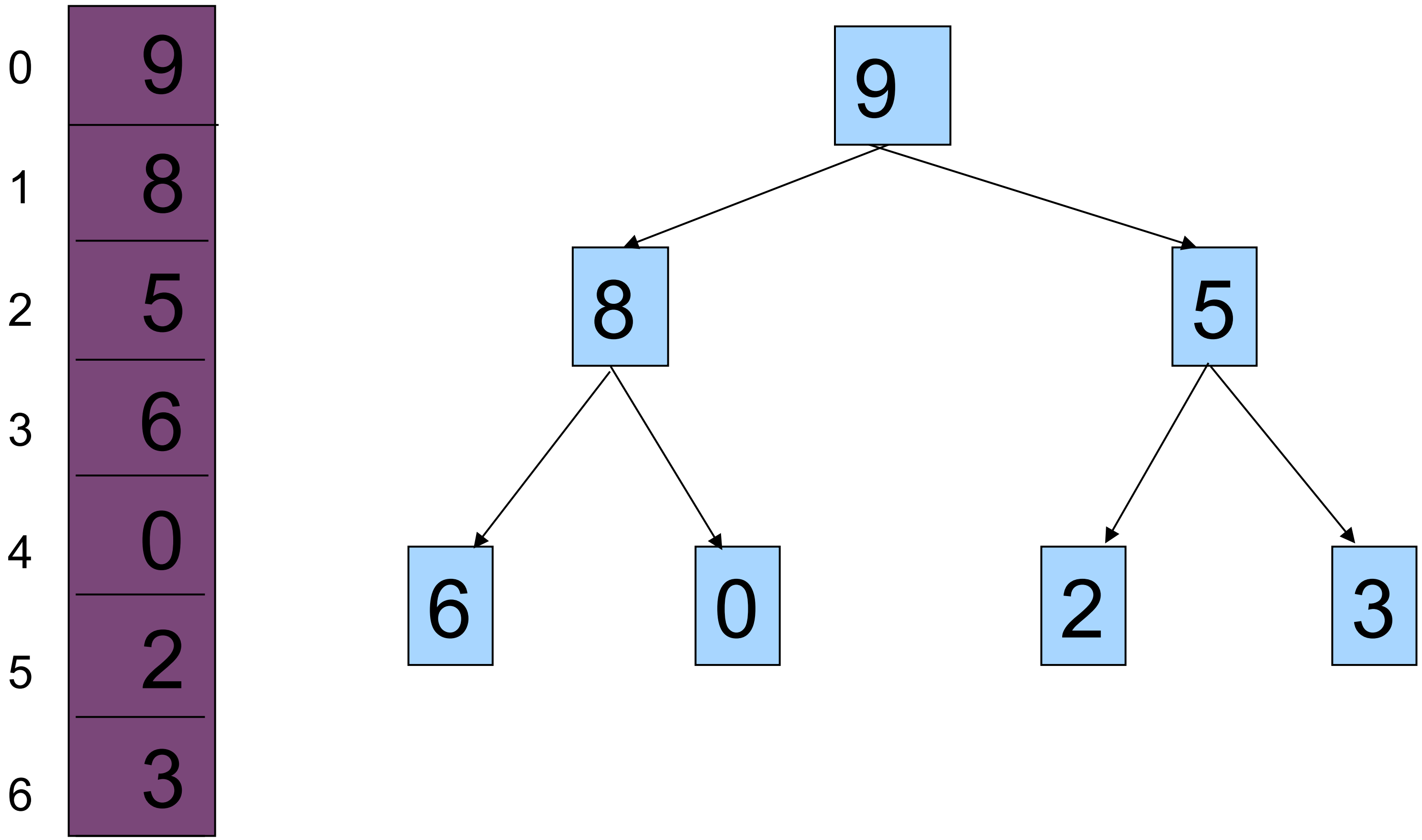
Array Representation of Binary Trees

- Let's take a small detour and discuss different ways of representing a binary tree
- Normally, trees are represented (implemented in memory) via tree nodes pointing at other tree nodes
  - Nodes are scattered about in memory (ie: non-contiguous)
  - Each node has left child and right child pointers
- But it is also possible to represent a binary tree with an array

Array Representation of Binary Trees

- There are a few ways to go about representing a tree in an array form
- Heaps use a form that has certain desirable properties
    - Other forms just complicate things
  - Heaps consider the tree as a set of levels, and ‘pack’ the levels into an array, one level after the other
  - This works *only* because it is almost-complete
- Converting a heap’s binary tree to array form is easy:
  - Simply read off the tree level-by-level and build the array in that order

Heap Array – Example



Heap Arrays

- This array form has a crucial benefit: it allows us to *calculate* how to go up and down the tree via arithmetic
  - The root is at element [0] in the array
  - All siblings are beside each other in the array
  - Thus if we are at node [currIdx], then:

$$\text{leftChildIdx} = (\text{currIdx} * 2) + 1$$

$$\text{rightChildIdx} = (\text{currIdx} * 2) + 2$$

$$\text{parentIdx} = (\text{currIdx} - 1) / 2$$

  - The * 2 comes about since we have a *binary* tree
  - parentIdx is derived by inverting equation for leftChildIdx
    - Inversion of rightChildIdx is equivalent since / 2 is DIV 2 and the right child index is always an even number

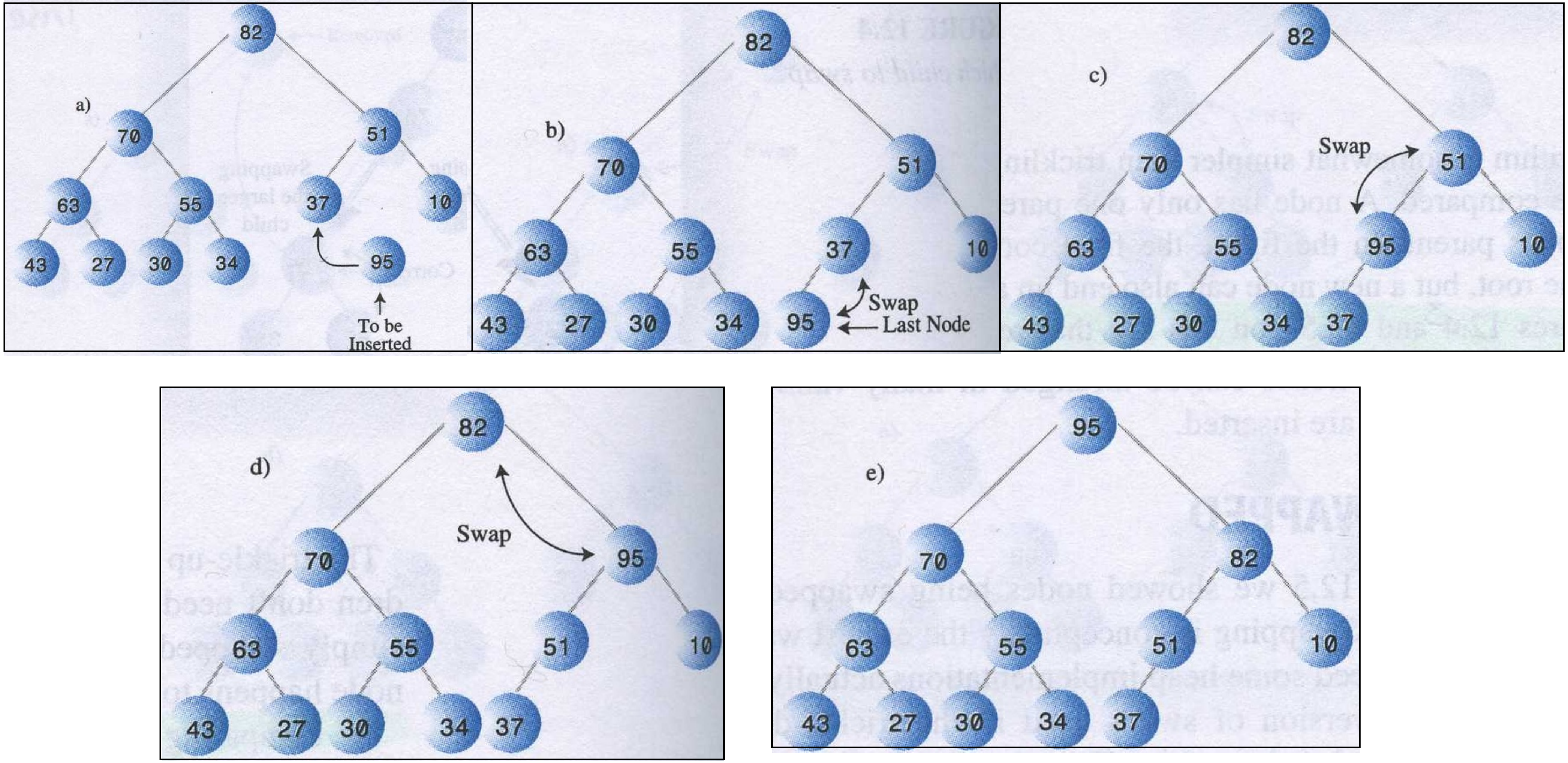
Heap Arrays

- Why does it matter to use arithmetic for traversal?
  - Because as we will see later, a heap needs to be able to traverse up *and* down the tree
  - In a tree form, this would require the addition of a 'parent' pointer in each node – extra memory overhead
  - With the arithmetic-based traversal, we can even do away with the left/right child pointers: no memory overhead!
    - ie: we only need to store the priority+data in the array
- BUT: the arithmetic only works for [almost-]complete trees
  - All levels are full (*i.e.,* exactly 2x larger than parent level), except for the last level which is filled from the left

Heap – Add

- Strategy: Initially place a new item in the next slot of the almost-complete tree
  - This guarantees the tree will remain almost-complete
  - The ‘next slot’ is easy to find: it’s at the end of the used portion of the array!
- Then ‘trickle’ the new item up through the tree until it meets a parent of equal or higher priority
  - Trickle-up = swapping based on priority checks vs parent
  - Essentially, we promote the new item until it reaches the place where it should be at (according to priority)
  - It doesn’t matter what branch it starts in: remember, heaps are only weakly ordered

Add Example



From Lafore, p474

Details of Add's Trickle-Up

- add() is essentially a loop that swaps the new node up the tree (trickle-up) while the following conditions hold true:
  - The new node has NOT made it to the root, AND
  - The parent's priority is lower than the new node
- Trickle-up can be done iteratively or recursively.

Iterative Trickle-Up

```
IMPORT heapArray, curIdx
EXPORT heapArray
Assertion: WHILE cur NOT root AND cur > parent DO
              Swap cur with parent, then try again

parentIdx = (curIdx-1)/2
WHILE curIdx > 0 AND heapArr[curIdx] > heapArr[parentIdx]
   temp = heapArr[parentIdx]
   heapArr[parentIdx] = heapArr[curIdx]
   heapArr[curIdx] = temp
   curIdx = parentIdx
   parentIdx = (curIdx-1)/2
ENDWHILE
```

Recursive Trickle-Up

```
IMPORT heapArray, curIdx
EXPORT heapArray
Assertion: IF cur NOT root AND cur > parent THEN
              Swap cur with parent, then try again

parentIdx = (curIdx-1)/2
IF curIdx > 0 THEN
   IF heapArr[curIdx] > heapArr[parentIdx] THEN
      temp = heapArr[parentIdx]
      heapArr[parentIdx] = heapArr[curIdx]
      heapArr[curIdx] = temp
      trickleUp <- heapArray, parentIdx
```

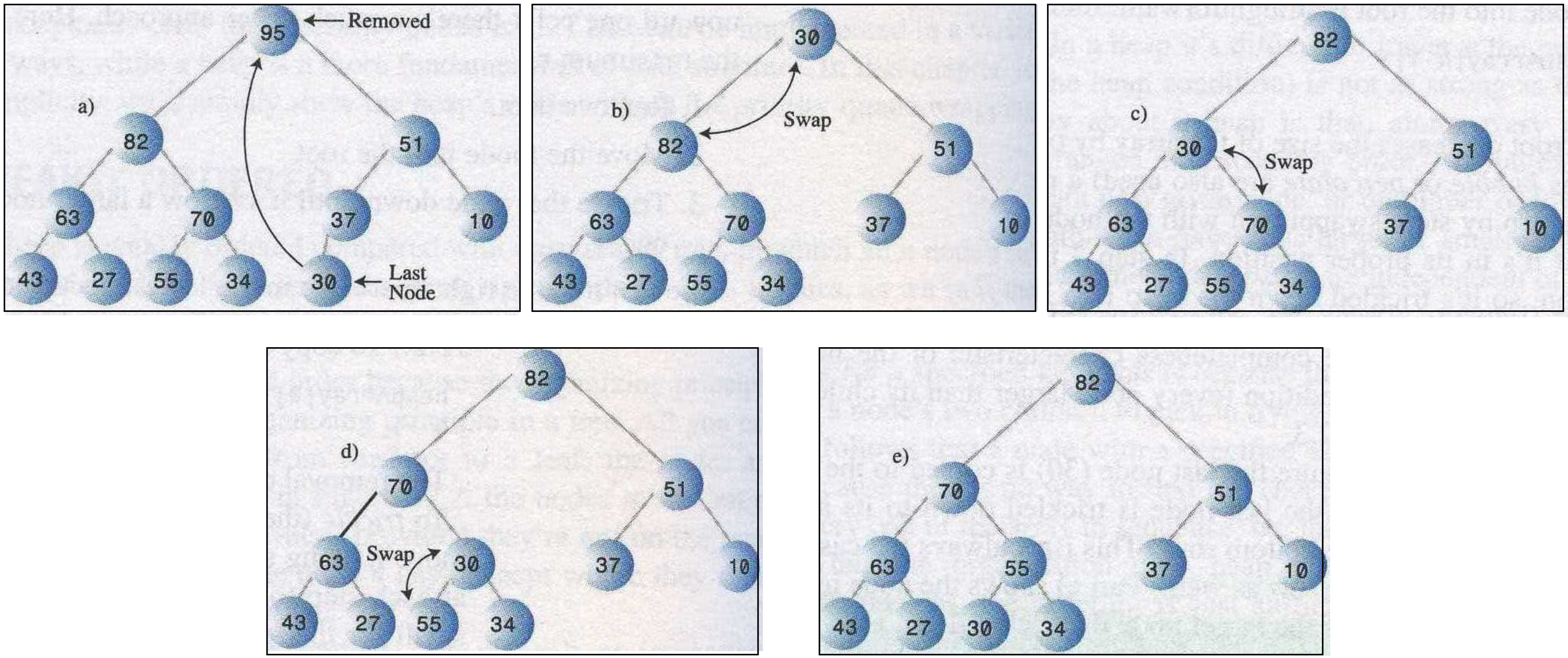
Heap – Remove

- Since the heap is a way of implementing a priority queue, ie: we always want to remove the item with the highest priority
  - And the heap is organised such that the highest priority item is *always* the root node
- It then follows that we will always remove the root node
  - Of course now we have a problem – we have lost our root node and our tree is not almost-complete anymore

Heap – Remove

- Strategy: Take a copy of the root at element [0], and move the last element to replace the root
  - Since the last element is at the final almost-complete position, removing it will maintain almost-complete tree
  - But now the root is going to be a low-priority item
    - *i.e.,* the heap's rule that parent >= children is being violated
- So 'trickle' this incorrect root node <u>down</u> through the tree until it finds its correct position
  - *i.e.,* swap down until *neither* child is higher priority
  - This often involves swapping to the bottom of the tree

Remove Example



From Lafore, p472

Details of Remove's Trickle-Down

- Removing the root and moving the last node into the root's position is pretty easy
  - Just copy the root to a temp variable, and copy the last-used element in the array to the root at [0]
- After that, trickle-down is quite similar to trickle-up: keep trickling-down the node while:
  - The node still has children (ie: currIdx < count/2) AND
  - *Either* children's priority is <u>higher</u> than the node

Details of Remove's Trickle-Down

- However, unlike add(), remove() has two possibilities for swapping:
  - Swap with left child OR Swap with right child
- We must swap with the higher-priority child to maintain that "all parents are higher priority than children"
  - To simplify the code: before the swap, compare the two children *first* and choose the highest-priority child
  - *Then* compare the trickling node with that child and swap if the child that has higher priority

Iterative Trickle-Down

```
IMPORT heapArray, curIdx, numItems
EXPORT heapArray

lChildIdx = curIdx * 2 + 1
rChildIdx = lChildIdx + 1
keepGoing = true
WHILE keepGoing AND lChildIdx < numItems     //is a left child
   keepGoing = false
   largeIdx = lChildIdx
   IF rChildIdx < numItems                   //is a right child
      IF heapArr[lChildIdx] < heapArr[rChildIdx]
         largeIdx = rChildIdx                //find largest child
   IF heapArr[largeIdx] > heapArr[curIdx]
      swap <- heapArr, largeIdx, curIdx
      keepGoing = true
   curIdx = largeIdx
   lChildIdx = curIdx * 2 + 1
   rChildIdx = lChildIdx + 1
ENDWHILE
```

Recursive Trickle-Down

```
IMPORT heapArray, curIdx, numItems
EXPORT heapArray

lChildIdx = curIdx * 2 + 1
rChildIdx = lChildIdx + 1

IF lChildIdx < numItems                             //is a left child
   largeIdx = lChildIdx
   IF rChildIdx < numItems                          //is a right child
      IF heapArr[lChildIdx] < heapArr[rChildIdx]
         largeIdx = rChildIdx                       //find largest child
   IF heapArr[largeIdx] > heapArr[curIdx]
      swap <- heapArr, largeIdx, curIdx
      trickleDown <- heapArray, largeIdx, numItems
```

Heaps – Complexity Analysis

- **Add**: Best = O(1), Average/Worst = O(log N)
  - Best case: occurs when adding a very low priority item
    - It won’t be trickled up since it is already in the right spot
    - Thus O(1)
  - Worst case: occurs when the added item has highest priority and must be trickled up all the way to the root
    - Since the heap tree is *always* almost-complete, there are log N levels to trickle through, resulting in O(log N)
  - Average case: ½ the items are at the bottom of the tree.
    - Which is O(log N)

Heaps – Complexity Analysis

- **Remove**: O(log N) for all cases
  - Because we take the last node (which will be among the lowest priorities), place it at the root and trickle down
    - Even in the best case there must be some trickle-down since the node's correct place was at the very bottom of the tree
    - And in fact it will usually trickle *all* the way down!
  - Thus O(log N) for pretty much all cases, even best case

Heaps – Summary

- Data is stored in a weakly-ordered way
  - There is some order (parent larger than children), but nothing like in a BST, thus ordered traversal of the tree is not possible
- Only the first item (root) can be taken
  - Heaps are not useful for searching for a particular value
    - We aren't storing by key, we are storing by *priority*
- Stores the binary tree in an array form and uses arithmetic on element indexes to traverse the tree
  - And the tree is always in an [almost]-complete state
- *Both* add and remove are fast O(log N) operations
  - Plus add/remove are crafted to maintain almost-completeness

HeapSort

- A heap returning items in priority order is kind of like getting data in sorted order, just one at a time
- This implies we can use heaps to perform sorting
  - Take an array of unsorted data
  - Add all elements of the unsorted array into a heap, using the element value as the priority
    - This will organise the elements into a heap
  - Remove each element from the heap one at a time and place them back into the array
    - Since a heap returns highest-priority first, the elements will come out in sorted order (or reverse sorted order)

HeapSort

- Depending on whether you are using a max-heap or a min-heap will affect the order of the sort
  - Max-heaps will return larger values first, hence the heap is effectively providing data in reverse order
    - Not a big deal: simply populate the target array in *reverse* order (from back to front).
  - This is just as efficient as using a min-heap and populating the target array in forwards order
    - The only difference between the two is that you either loop from 0…N (min-heap) or loop from N…0 (max-heap)

HeapSort Time Complexity

- If a heap is available HeapSort is a particularly simple algorithm to implement:
  - An initial for loop to add all array values to the heap
  - A second for loop to take them all out one at a time
- But how efficient is it?
  - Add: O(log N) done N times = O(N log N)
    - OK, best case of O(1) * N = O(N), but that's rare!
  - Remove: O(log N) done N times = O(N log N)
  - Total = Add + Remove

    = O(N log N) + O(N log N) (or best case O(N) + O(N log N))

    = O(N log N)

In-Place HeapSort

- Hence HeapSort is scalable – O(N log N)
  - Unfortunately, it is unstable since we may get equal-priority values being swapped relative to each other
  - The simple approach outlined is also not in-place
    - The heap has an array that is the same size as the original array
  - However, it is possible to make HeapSort in-place by integrating it into the heap's code (more complicated!)
    - First organise the array into a heap incrementally by 'expanding' the heap one element at a time and adding that new element
      - This is termed to 'heapify' the array
    - Then every time the root is taken from the heap, add it to the array slot that has just been 'vacated' by the last node

heapify

```
IMPORT heapArray, numItems
EXPORT heapArray
ASSERTION: imported array will be random, exported will
           be a heap

   // start at last non-leaf, go backwards
   for ii = (numItems/2)-1 downto 0        //0 based array
      // put iith element in correct place in heap
      trickleDown <- heapArray, ii, numItems
```

heapSort (in-place)

```
IMPORT array, numItems
EXPORT sortedArray
ASSERTION: imported array will be random, exported will
           be the same array sorted

   heapify <- array, numItems
   for ii = numItems-1 downto 1          //0th item will be sorted
      swap <- array, 0, ii
      trickleDown <- heapArray, 0, (ii)     //ii is numItems--
```

heapSort example

Import

	0	1	2	3	4	5	6	7	8
Import	5	4	1	11	10	3	2	16	12

	0	1	2	3	4	5	6	7	8
After Heapify	16	12	3	11	10	1	2	5	4

	0	1	2	3	4	5	6	7	8
After 1st swap	4	12	3	11	10	1	2	5	**16**

	0	1	2	3	4	5	6	7	8
After trickleDown	12	11	3	5	10	1	2	4	**16**

	0	1	2	3	4	5	6	7	8
After 2nd swap	4	11	3	5	10	1	2	**12**	**16**

	0	1	2	3	4	5	6	7	8
After trickleDown	11	10	3	5	4	1	2	**12**	**16**

	0	1	2	3	4	5	6	7	8
After 3rd swap	2	10	3	5	4	1	**11**	**12**	**16**

	0	1	2	3	4	5	6	7	8
After trickleDown	10	5	3	2	4	1	**11**	**12**	**16**

After 4th swap

0	1	2	3	4	5	6	7	8
1	5	3	2	4	**10**	**11**	**12**	**16**

After trickleDown

0	1	2	3	4	5	6	7	8
5	4	3	2	1	**10**	**11**	**12**	**16**

After 5th swap

0	1	2	3	4	5	6	7	8
1	4	3	2	**5**	**10**	**11**	**12**	**16**

After trickleDown

0	1	2	3	4	5	6	7	8
4	2	3	1	**5**	**10**	**11**	**12**	**16**

	0	1	2	3	4	5	6	7	8
After 6th swap	1	2	3	**4**	**5**	**10**	**11**	**12**	**16**
After trickleDown	3	2	1	**4**	**5**	**10**	**11**	**12**	**16**
After 7th swap	1	2	**3**	**4**	**5**	**10**	**11**	**12**	**16**
After trickleDown	2	1	**3**	**4**	**5**	**10**	**11**	**12**	**16**
After 8th swap	1	**2**	**3**	**4**	**5**	**10**	**11**	**12**	**16**

HeapSort

- ☑ Can be an in-place sort if built into the Heap class
- ☑ Consistently O(N log N) for all cases
- ☑ By far the easiest O(N log N) algorithm to implement ***if*** you can make use of an *existing* Heap class
  - Just a couple of for loops: one to insert all the elements, another to extract them out in [reverse-]sorted order
  - Although with this approach it cannot be made in-place
- ☒ Unstable sort
- ☒ Poor use of a modern CPU's L2 cache
  - Trickle-up and trickle-down jump all over the array
- ☒ Requires implementing a Heap:

MergeSort

- ☑ Easy to make execute in parallel
  - Since different split and merge branches are independent, we can assign each branch to a different CPU
- ☑ Makes efficient use of a modern CPU's L2 cache
  - It merges two sub-arrays that are *beside* each other
    - AND goes through each array from left to right
  - So accesses are always close together: perfect for L2 caching
    - With a large L2 cache, MergeSort can become very fast:
L2 accesses are up to 5x faster than main memory accesses
- ☑ And: Stable, consistently O(N log N) for all cases
- ☒ But: Not an in-place sort

QuickSort

- ☑ Easy to make execute in parallel
  - Since different split+partition branches are independent, we can assign each branch to a different CPU
- ☑ Makes some use of a modern CPU's L2 cache
  - Splitting will mean that it eventually operates on sub-arrays that are small enough to fit into the L2 cache
  - But not as good as MergeSort at this
- ☑ And: in-place sort
- ☒ But: Unstable sort, recursive (stack overflows), $O(N^2)$ worst-case, fairly complicated to implement well

Next Week

- Advanced Sorting