Unit 9, Part 3

Heaps and Priority Queues

Computer Science S-111
Harvard University

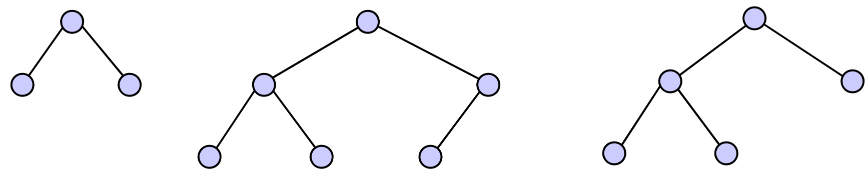
David G. Sullivan, Ph.D.

Priority Queue

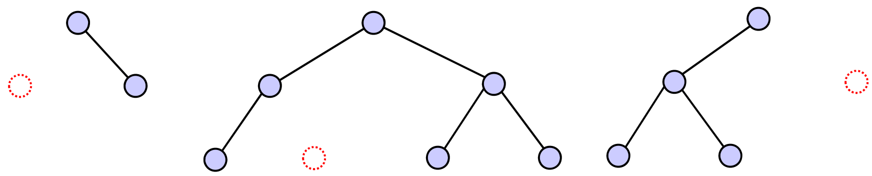
- A *priority queue* (PQ) is a collection in which each item has an associated number known as a *priority*.
  - ("Ann Cudd", 10), ("Robert Brown", 15), ("Dave Sullivan", 5)
  - use a higher priority for items that are "more important"
- Example application: scheduling a shared resource like the CPU
  - give some processes/applications a higher priority, so that they will be scheduled first and/or more often
- Key operations:
  - *insert:* add an item (with a position based on its priority)
  - *remove:* remove the item with the highest priority
- One way to implement a PQ efficiently is using a type of binary tree known as a *heap.*

Complete Binary Trees

- A binary tree of height *h* is *complete* if:
  - levels 0 through *h* - 1 are fully occupied
  - there are no "gaps" to the left of a node in level *h*
- Complete:

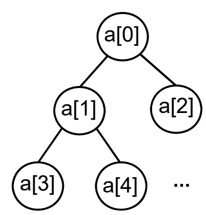


- Not complete (◌ = missing node):

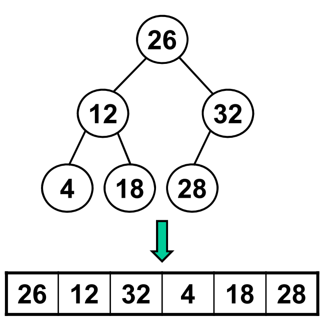


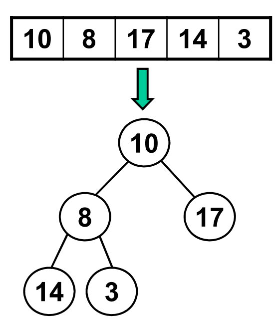
Representing a Complete Binary Tree

- A complete binary tree has a simple array representation.
- The tree's nodes are stored in the array in the order given by a level-order traversal.
  - top to bottom, left to right

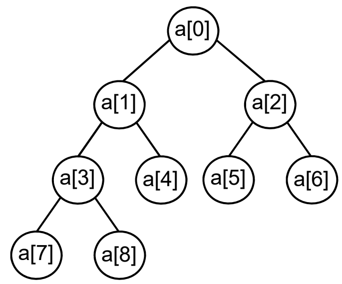


- Examples:





Navigating a Complete Binary Tree in Array Form



- The root node is in `a[0]`
- Given the node in `a[i]`:
  - its left child is in `a[2*i + 1]`
  - its right child is in `a[2*i + 2]`
  - its parent is in `a[(i - 1)/2]` (using integer division)
- Examples:
  - the left child of the node in `a[1]` is in `a[2*1 + 1]` = `a[3]`
  - the left child of the node in `a[2]` is in `a[2*2 + 1]` = `a[5]`
  - the right child of the node in `a[3]` is in `a[2*3 + 2]` = `a[8]`
  - the right child of the node in `a[2]` is in ________________
  - the parent of the node in `a[4]` is in `a[(4-1)/2]` = `a[1]`
  - the parent of the node in `a[7]` is in ___________________

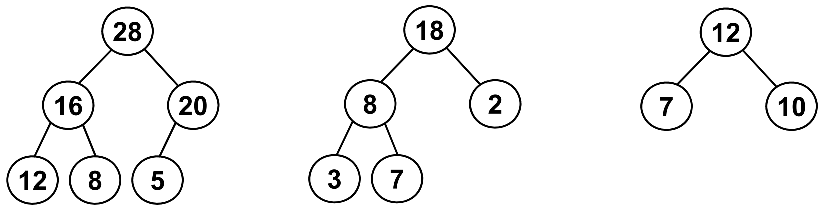
What is the left child of 24?

- Assume that the following array represents a complete tree:

0	1	2	3	4	5	6	7	8
26	**12**	**32**	**24**	**18**	**28**	**47**	**10**	**9**

Heaps

- Heap: a complete binary tree in which each interior node is greater than or equal to its children
  - examples:



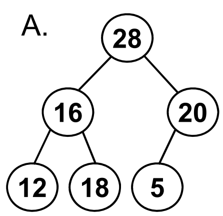
- The largest value is always at the root of the tree.

- The smallest value can be in *any* leaf node - there's no guarantee about which one it will be.

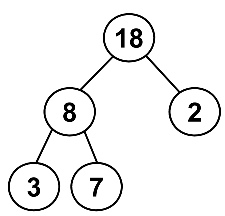
- We're using *max-at-top* heaps.
  - in a *min-at-top* heap, every interior node <= its children

Which of these is a heap?

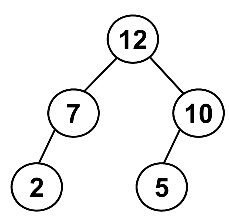
- A.



B.



C.

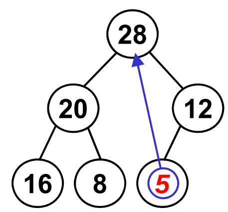


D. more than one (which ones?)

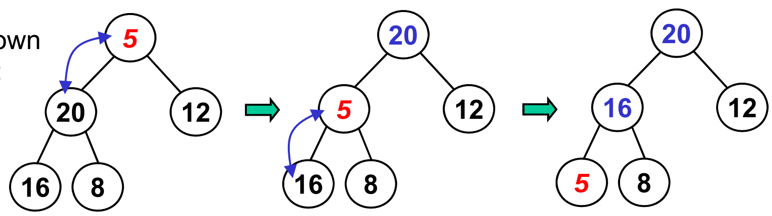
E. none of them

Removing the Largest Item from a Heap

- Remove and return the item in the root node.
- In addition, need to move the largest remaining item to the root, while maintaining a complete tree with each node >= children
- Algorithm:
  1. make a copy of the largest item
  2. move the last item in the heap to the root
  3. "sift down" the new root item until it is >= its children (or it's a leaf)
  4. return the largest item



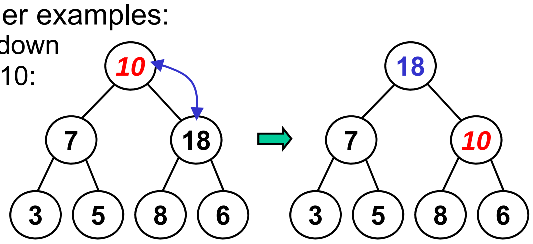
sift down the 5:



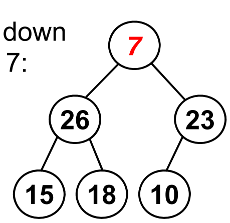
Sifting Down an Item

- To sift down item *x* (i.e., the item whose key is *x*):
  1. compare *x* with the larger of the item's children, *y*
  2. if *x* < *y*, swap *x* and *y* and repeat
- Other examples:

sift down the 10:



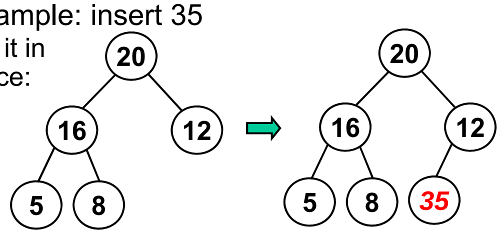
sift down the 7:



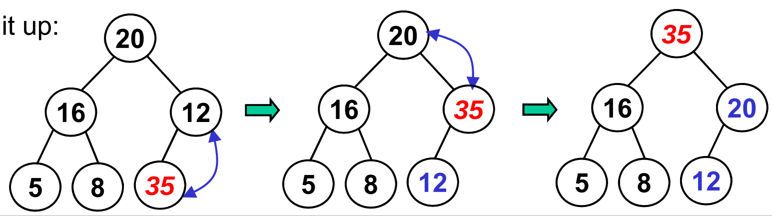
Inserting an Item in a Heap

- Algorithm:
  1. put the item in the next available slot (grow array if needed)
  2. "sift up" the new item
     until it is <= its parent (or it becomes the root item)
- Example: insert 35

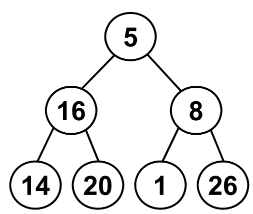
put it in place:



sift it up:



Time Complexity of a Heap



- A heap containing n items has a height <= $\log_2 n$. Why?

- Thus, removal and insertion are both $O(\log n)$.
  - remove: go down at most $\log_2 n$ levels when sifting down;
    do a constant number of operations per level
  - insert: go up at most $\log_2 n$ levels when sifting up;
    do a constant number of operations per level

- This means we can use a heap for a $O(\log n)$-time priority queue.

Using a Heap for a Priority Queue

- Recall: a *priority queue* (PQ) is a collection in which each item has an associated number known as a *priority*.
  - ("Ann Cudd", 10), ("Robert Brown", 15), ("Dave Sullivan", 5)
  - use a higher priority for items that are "more important"
- To implement a PQ using a heap:
  - order the items in the heap according to their priorities
    - every item in the heap will have a priority >= its children
    - the highest priority item will be in the root node
  - get the highest priority item by calling `heap.remove()`!

Using a Heap to Sort an Array

- Recall selection sort: it repeatedly finds the smallest remaining element and swaps it into place:

0	1	2	3	4	5	6
5	**16**	**8**	**14**	**20**	**1**	**26**

0	1	2	3	4	5	6
1	**16**	**8**	**14**	**20**	**5**	**26**

0	1	2	3	4	5	6
1	***5***	**8**	**14**	**20**	**16**	**26**

…

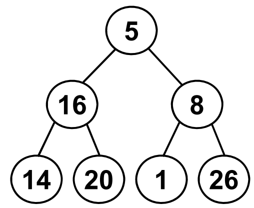
- It isn't efficient, because it performs a linear scan to find the smallest remaining element (O(n) steps per scan).
- Heapsort is a sorting algorithm that repeatedly finds the *largest* remaining element and puts it in place.
- It *is* efficient, because it turns the array into a heap.
  - it can find/remove the largest remaining in $O(\log n)$ steps!

Converting an Arbitrary Array to a Heap

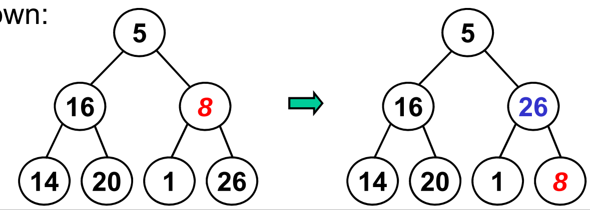
- To convert an array (call it `contents`) with n items to a heap:
  1. start with the parent of the last element:
     `contents[i]`, where `i = ((n – 1) – 1)/2 = (n – 2)/2`
  2. sift down `contents[i]` and all elements to its left

- Example:

0	1	2	3	4	5	6
5	16	8	14	20	1	26

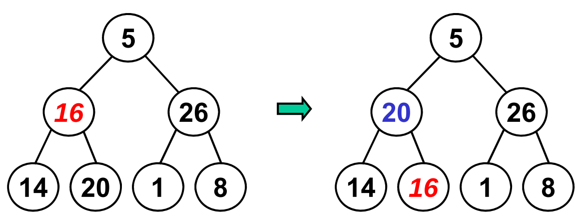


- Last element's parent = `contents[(7 – 2)/2]` = `contents[2]`.
  Sift it down:

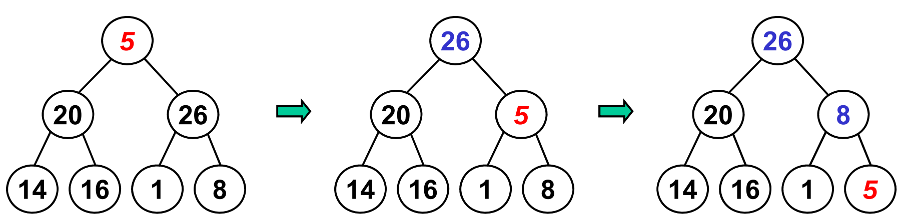


Converting an Array to a Heap (cont.)

- Next, sift down `contents[1]`:



- Finally, sift down `contents[0]`:



Heapsort

- Pseudocode:

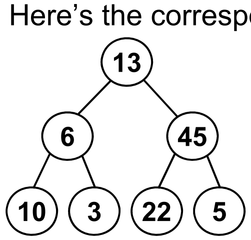
```
heapSort(arr) {
    // Turn the array into a max-at-top heap.
    heap = new Heap(arr);

    endUnsorted = arr.length - 1;
    while (endUnsorted > 0) {
        // Get the largest remaining element and put it
        // at the end of the unsorted portion of the array.
        largestRemaining = heap.remove();
        arr[endUnsorted] = largestRemaining;

        endUnsorted--;
    }
}
```

Heapsort Example

- Sort the following array:

0	1	2	3	4	5	6
13	**6**	**45**	**10**	**3**	**22**	**5**

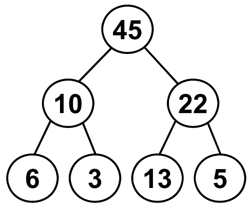
- Here's the corresponding complete tree:



- Begin by converting it to a heap:

Heapsort Example (cont.)

- Here's the heap in both tree and array forms:



0	1	2	3	4	5	6
45	10	22	6	3	13	5

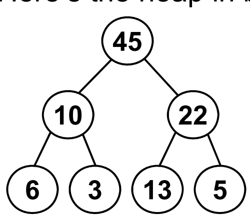
endUnsorted: **6**

- We begin looping:

```
while (endUnsorted > 0) {
    // Get the largest remaining element and put it
    // at the end of the unsorted portion of the array.
    largestRemaining = heap.remove();
    arr[endUnsorted] = largestRemaining;

    endUnsorted--;
}
```

Heapsort Example (cont.)

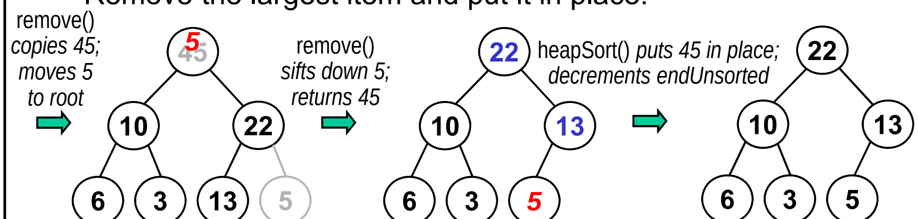
- Here's the heap in both tree and array forms:



0	1	2	3	4	5	6
45	10	22	6	3	13	5

endUnsorted: **6**

- Remove the largest item and put it in place:



toRemove: **45**

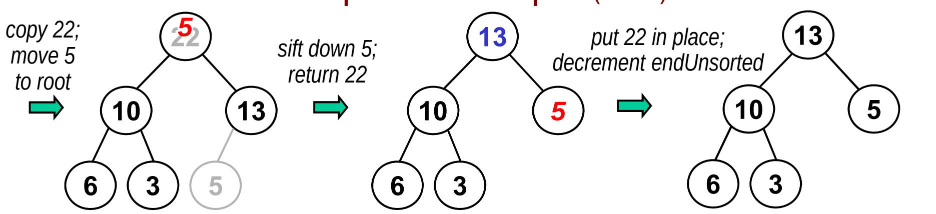
0	1	2	3	4	5	6
22	10	13	6	3	*5*	5

endUnsorted: **6**
largestRemaining: **45**

0	1	2	3	4	5	6
22	10	13	6	3	5	45

endUnsorted: **5**

Heapsort Example (cont.)



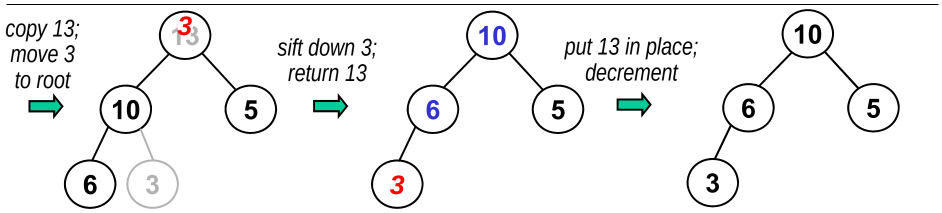
toRemove: **22**

0	1	2	3	4	5	6
13	10	5	6	3	5	45

endUnsorted: **5**
largestRemaining: **22**

0	1	2	3	4	5	6
13	10	5	6	3	22	45

endUnsorted: **4**



toRemove: **13**

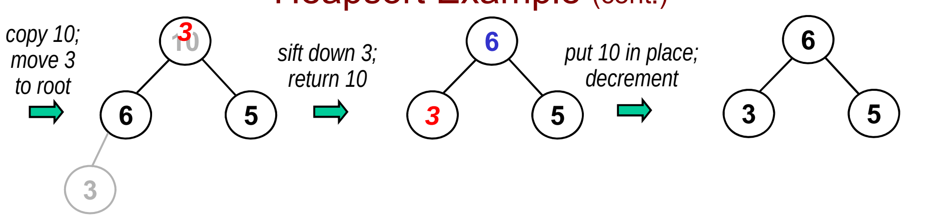
0	1	2	3	4	5	6
10	6	5	3	3	22	45

endUnsorted: **4**
largestRemaining: **13**

0	1	2	3	4	5	6
10	6	5	3	13	22	45

endUnsorted: **3**

Heapsort Example (cont.)



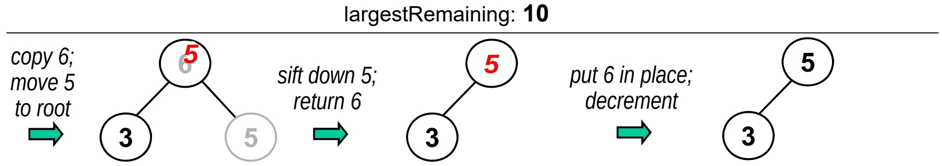
toRemove: **10**

0	1	2	3	4	5	6
6	3	5	3	13	22	45

endUnsorted: **3**
largestRemaining: **10**

0	1	2	3	4	5	6
6	3	5	10	13	22	45

endUnsorted: **2**



toRemove: **6**

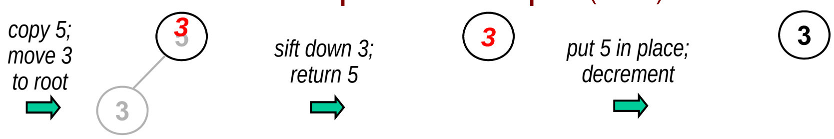
0	1	2	3	4	5	6
5	3	5	10	13	22	45

endUnsorted: **2**
largestRemaining: **6**

0	1	2	3	4	5	6
5	3	6	10	13	22	45

endUnsorted: **1**

Heapsort Example (cont.)



toRemove: **5**

0	1	2	3	4	5	6
3	3	6	10	13	22	45

endUnsorted: **1**
largestRemaining: **5**

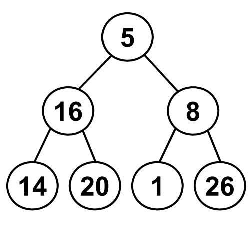
0	1	2	3	4	5	6
3	5	6	10	13	22	45

endUnsorted: **0**

- And now we terminate the loop:

```
while (endUnsorted > 0) {
    // Get the largest remaining element and put it
    // at the end of the unsorted portion of the array.
    largestRemaining = heap.remove();
    arr[endUnsorted] = largestRemaining;

    endUnsorted--;
}
```

Time Complexity of Heapsort



- Time complexity of creating a heap from an array?
- Time complexity of sorting the array?
- Thus, total time complexity = ?

How Does Heapsort Compare?

algorithm	best case	avg case	worst case	extra memory
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	$O(n \log n)$	$O(n^{1.5})$	$O(n^{1.5})$	$O(1)$
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ worst: $O(n)$
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

- Heapsort matches mergesort for the best worst-case time complexity, but it has better space complexity.
- Insertion sort is still best for arrays that are almost sorted.
  - heapsort will scramble an almost sorted array before sorting it!
- Quicksort is still typically fastest in the average case.