Heaps and Priority Queues

Computer Science E-22
Harvard University

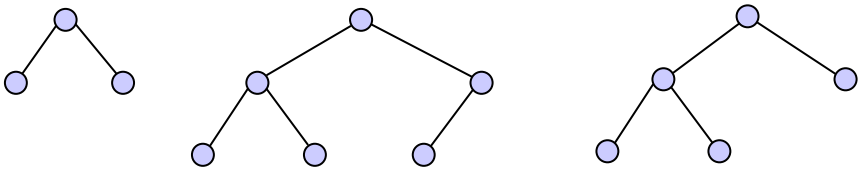
David G. Sullivan, Ph.D.

Priority Queue

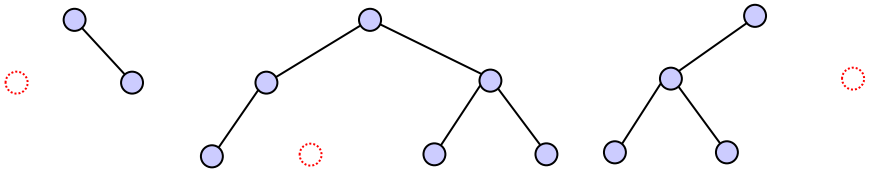
- A *priority queue* (PQ) is a collection in which each item has an associated number known as a *priority*.
  - ("Ann Cudd", 10), ("Robert Brown", 15), ("Dave Sullivan", 5)
  - use a higher priority for items that are "more important"
- Example application: scheduling a shared resource like the CPU
  - 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:

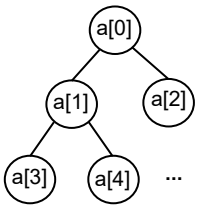


- Not complete (◌ = missing node):

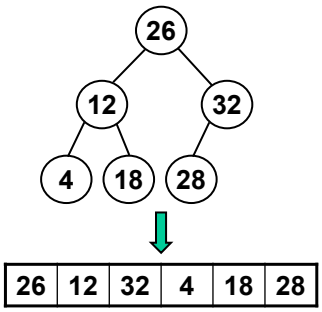


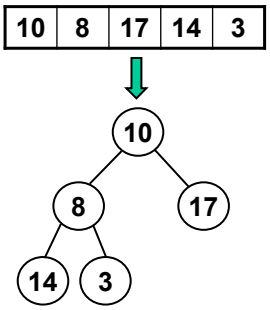
Representing a Complete Binary Tree

- 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



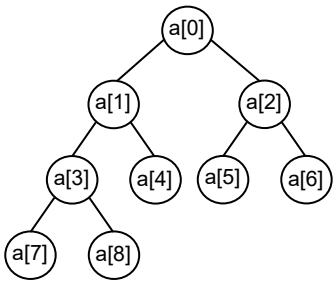
- Examples:





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 _____

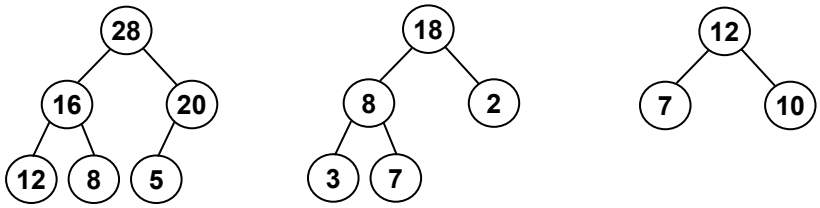
What is the left child of 24?

- 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

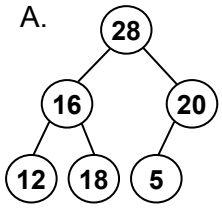
- Heap: a complete binary tree in which each interior node is greater than or equal to its children
  - examples:



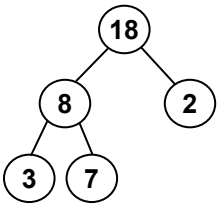
- The largest value is always at the root of the tree.
- The smallest value can be in *any* leaf node - there's no guarantee about which one it will be.
- We're using *max-at-top* heaps.
  - in a *min-at-top* heap, every interior node <= its children

Which of these is a heap?

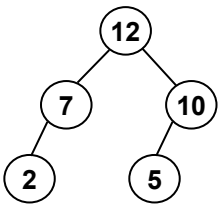
- A.



B.



C.



D. more than one (which ones?)

E. none of them

How to Compare Objects

- We need to be able to compare items in the heap.
- If those items are objects, we can't just do something like this:

```
if (item1 < item2)
```

  Why not?

- Instead, we need to use a method to compare them.

An Interface for Objects That Can Be Compared

- The `Comparable` interface is a built-in generic Java interface:

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- It is used when defining a class of objects that can be ordered.
- Examples from the built-in Java classes:

```
public class String implements Comparable<String> {
    ...
    public int compareTo(String other) {
        ...
}
public class Integer implements Comparable<Integer> {
    ...
    public int compareTo(Integer other) {
        ...
}
```

An Interface for Objects That Can Be Compared (cont.)

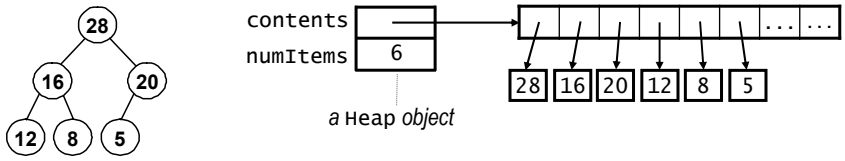
```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- `item1.compareTo(item2)` should return:
  - a negative integer if `item1` "comes before" `item2`
  - a positive integer if `item1` "comes after" `item2`
  - 0 if `item1` and `item2` are equivalent in the ordering
- These conventions make it easy to construct appropriate method calls:

numeric comparison	comparison using `compareTo`
`item1 < item2`	`item1.compareTo(item2) < 0`
`item1 > item2`	`item1.compareTo(item2) > 0`
`item1 == item2`	`item1.compareTo(item2) == 0`

Heap Implementation

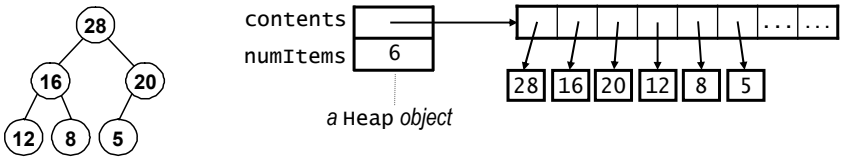
```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;

    public Heap(int maxSize) {
        contents = (T[])new Comparable[maxSize];
        numItems = 0;
    }
    …
}
```



- `Heap` is another example of a generic collection class.
  - as usual, `T` is the type of the elements
  - `extends Comparable<T>` specifies `T` must implement `Comparable<T>`
  - must use `Comparable` (not `Object`) when creating the array

Heap Implementation (cont.)

```
public class Heap<T extends Comparable<T>> {
    private T[] contents;
    private int numItems;

    …
}
```



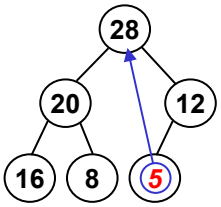
- The picture above is a heap of integers:

  `Heap<Integer> myHeap = new Heap<Integer>(20);`

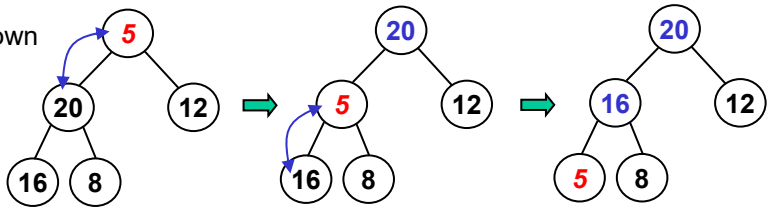
  - works because `Integer` implements `Comparable<Integer>`
  - could also use `String` or `Double`

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



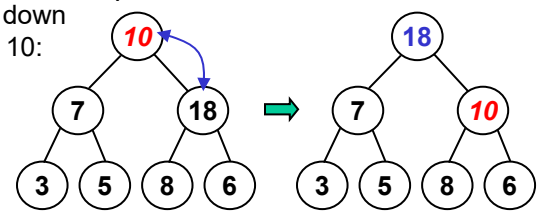
sift down the 5:



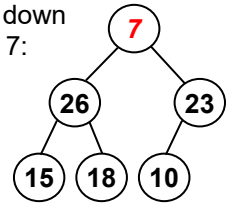
Sifting Down an Item

- 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:



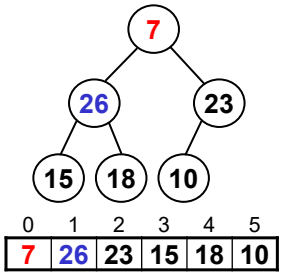
sift down the 7:



`siftDown()` Method

```
private void siftDown(int i) {    // assume i = 0
    T toSift = contents[i];

    int parent = i;
    int child = 2 * parent + 1;
    while (child < numItems) {
        // If the right child is bigger, set child to be its index.
        if (child < numItems - 1  &&
          contents[child].compareTo(contents[child + 1]) < 0) {
            child = child + 1;
        }
        if (toSift.compareTo(contents[child]) >= 0) {
            break;  // we're done
        }
        // Move child up and move down one level in the tree.
        contents[parent] = contents[child];
        parent = child;
        child = 2 * parent + 1;
    }

    contents[parent] = toSift;
}
```

- We don't actually swap items. We put the sifted item in place at the end.

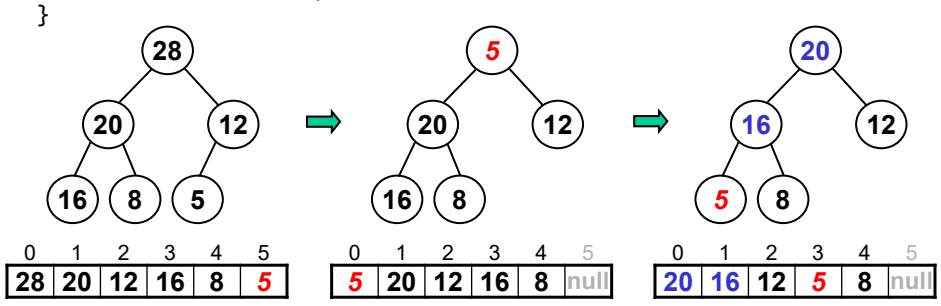


toSift: 7

parent	child
0	1

remove() Method

```
public T remove() {
    // check for empty heap goes here
    T toRemove = contents[0];

    contents[0] = contents[numItems - 1];
    contents[numItems - 1] = null;
    numItems--;
    siftDown(0);

    return toRemove;
}
```



numItems: **6**
toRemove: **28**

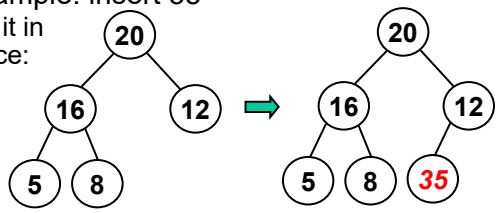
numItems: **5**
toRemove: **28**

numItems: **5**
toRemove: **28**

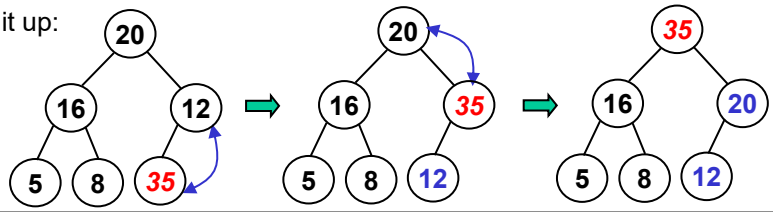
Inserting an Item in a Heap

- 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:

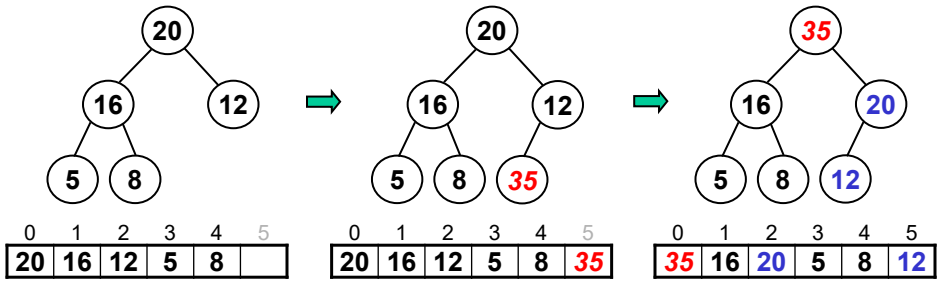


sift it up:



insert() Method

```
public void insert(T item) {
    if (numItems == contents.length) {
        // code to grow the array goes here…
    }
    contents[numItems] = item;
    siftUp(numItems);
    numItems++;
}
```

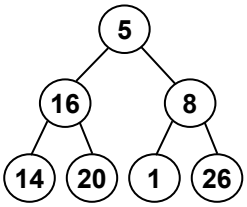


numItems: **5**
item: **35**

numItems: **5**
item: **35**

numItems: **6**

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

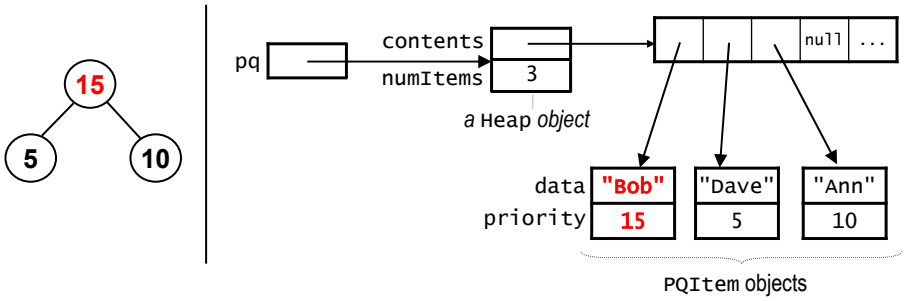
- Recall: a *priority queue* (PQ) is a collection in which each item has an associated number known as a *priority*.
  - ("Ann Cudd", 10), ("Robert Brown", 15), ("Dave Sullivan", 5)
  - use a higher priority for items that are "more important"
- To implement a PQ using a heap:
  - order the items in 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()`!
- For this to work, we need a "wrapper" class for items that we put in the priority queue.
  - will group together an item with its priority
  - with a `compareTo()` method that compares priorities!

A Class for Items in a Priority Queue

```
public class PQItem implements Comparable<PQItem> {
    // group an arbitrary object with a priority
    private Object data;
    private int priority;
    ...

    public int compareTo(PQItem other) {
        // error-checking goes here…
        return (priority - other.priority);
    }
}
```

- Example: `PQItem item = new PQItem("Dave Sullivan", 5);`
- Its `compareTo()` compares `PQItem`s based on their priorities.
- `item1.compareTo(item2)` returns:
  - a negative integer if `item1` has a lower priority than `item2`
  - a positive integer if `item1` has a higher priority than `item2`
  - 0 if they have the same priority

Using a Heap for a Priority Queue



- Sample client code:

```
Heap<PQItem> pq = new Heap<PQItem>(50);
pq.insert(new PQItem("Dave", 5));
pq.insert(new PQItem("Ann", 10));
pq.insert(new PQItem("Bob", 15));

PQItem mostImportant = pq.remove(); // will get Bob!
```

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**
1	**16**	**8**	**14**	**20**	**5**	**26**
1	**5**	**8**	**14**	**20**	**16**	**26**

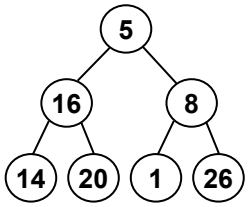
...

- 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!

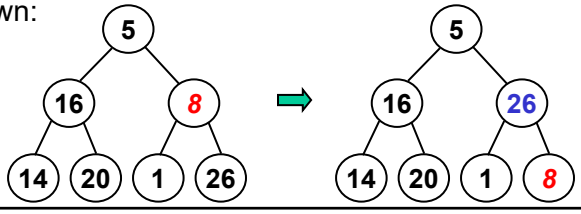
Converting an Arbitrary Array to a Heap

- 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

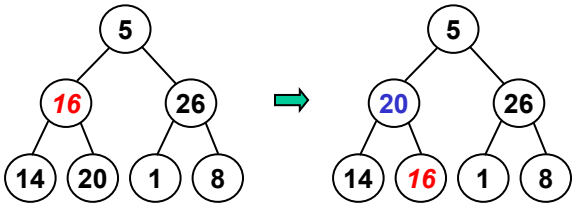


- Last element's parent = `contents[(7 – 2)/2]` = `contents[2]`. Sift it down:

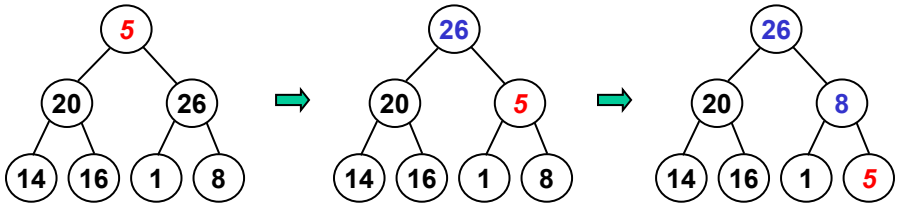


Converting an Array to a Heap (cont.)

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



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



Heapsort

- Pseudocode:

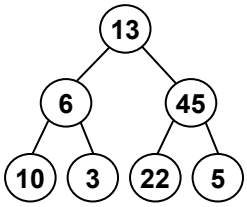
```
heapSort(arr) {
    // Turn the array into a max-at-top heap.
    heap = new Heap(arr);

    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

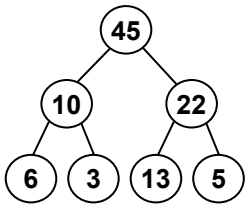
- Here's the corresponding complete tree:



- 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

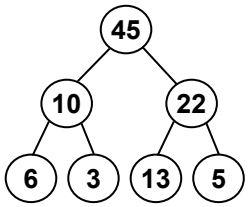
endUnsorted: **6**

- 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.)

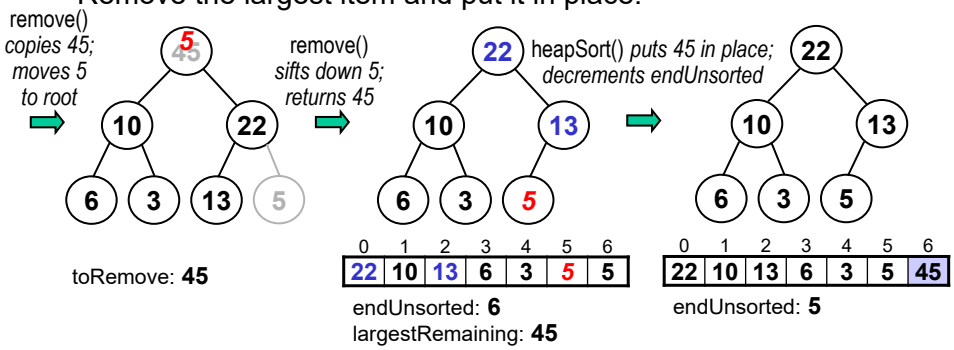
- Here's the heap in both tree and array forms:



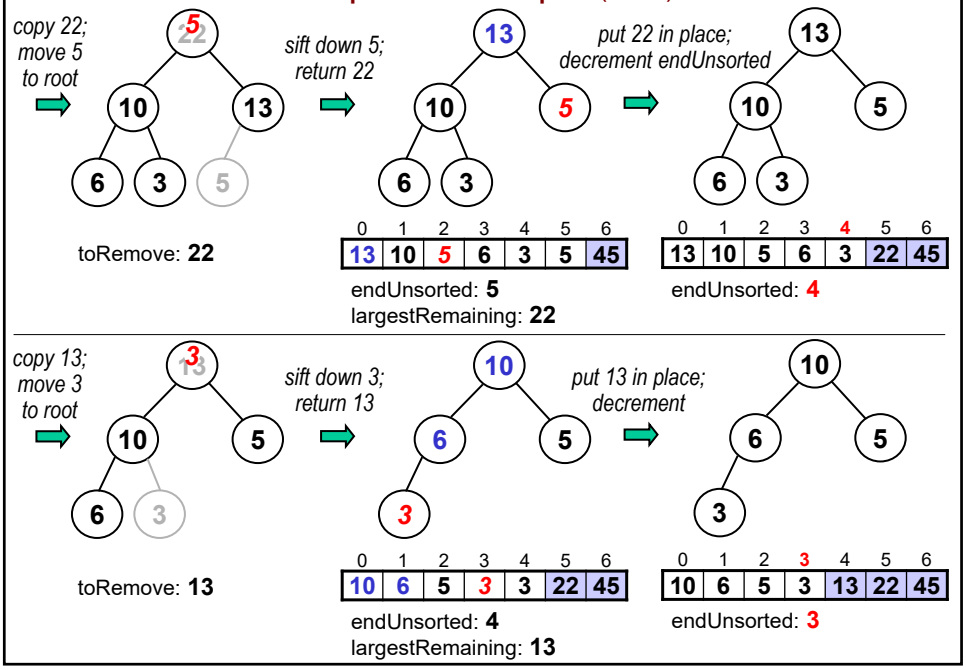
0	1	2	3	4	5	6
45	10	22	6	3	13	5

endUnsorted: **6**

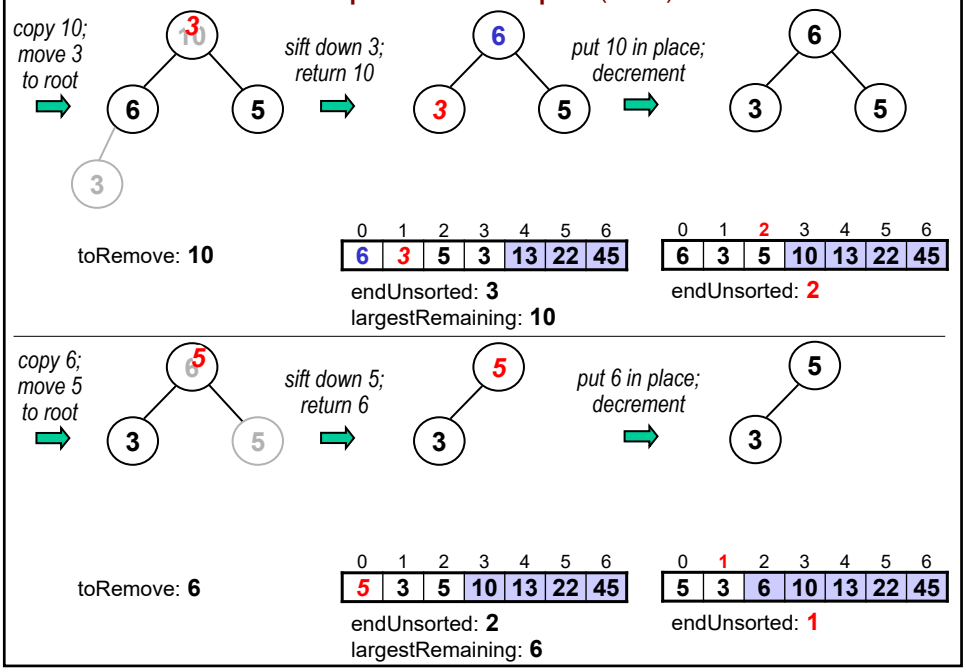
- Remove the largest item and put it in place:



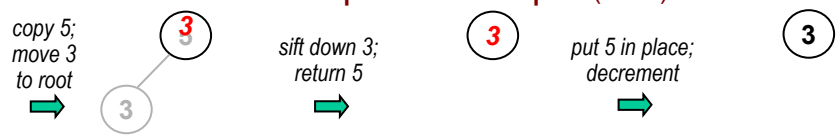
Heapsort Example (cont.)



Heapsort Example (cont.)



Heapsort Example (cont.)



toRemove: **5**

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

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

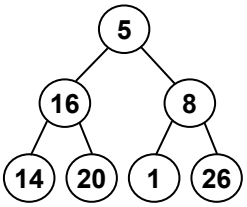
0	1	2	3	4	5	6
3	5	6	10	13	22	45

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--;
}
```

Efficiency of Heapsort



- Time complexity of going from a heap to a sorted array?

- It can be shown that turning an array into a heap takes O(n) steps.
  - even better than O(n log n)!
  - n/2 calls to siftDown(), most of which involve small subheaps
- 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	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(n \log n)}$	$\boldsymbol{O(1)}$

- 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.
- Quicksort is still typically fastest in the average case.