Priority Queues – Binary Heaps

Outline

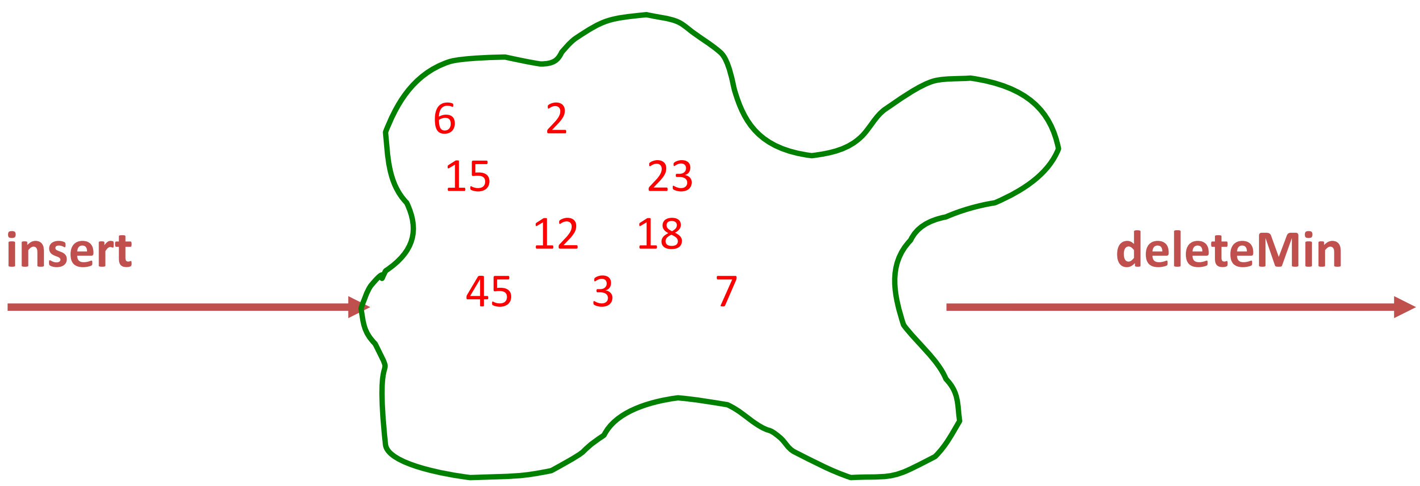
- Binary Heaps
  - Order
  - Structure
  - Insertion
  - Deletion
- Building Heaps
  - Floyd’s method
- Heap Sort

Recall Queues

- FIFO: First-In, First-Out
- Some contexts where this seems right?
- Some contexts where some things should be allowed to skip ahead in the line?

Queues that Allow Line Jumping

- Queue: First-In, First-Out (FIFO)
- Need a new ADT
- Operations: Insert an Item,
  Remove the "Best" Item



Applications of the Priority Queue

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Select most frequent symbols for compression
- Sort numbers, picking minimum first

- Anything ***greedy***

Priority Queue ADT

- In a Priority Queue, we always remove the item with the **highest priority.**

- “Highest Priority” is application-dependent, for example:
  - Item with minimum key value.
  - Items with maximum key value.

Potential Implementations

	insert	deleteMin
Unsorted list (Array)	O(1)	O(n)
Unsorted list (Linked-List)	O(1)	O(n)
Sorted list (Array)	O(n)	O(1)*
Sorted list (Linked-List)	O(n)	O(1)

Can we do better ?

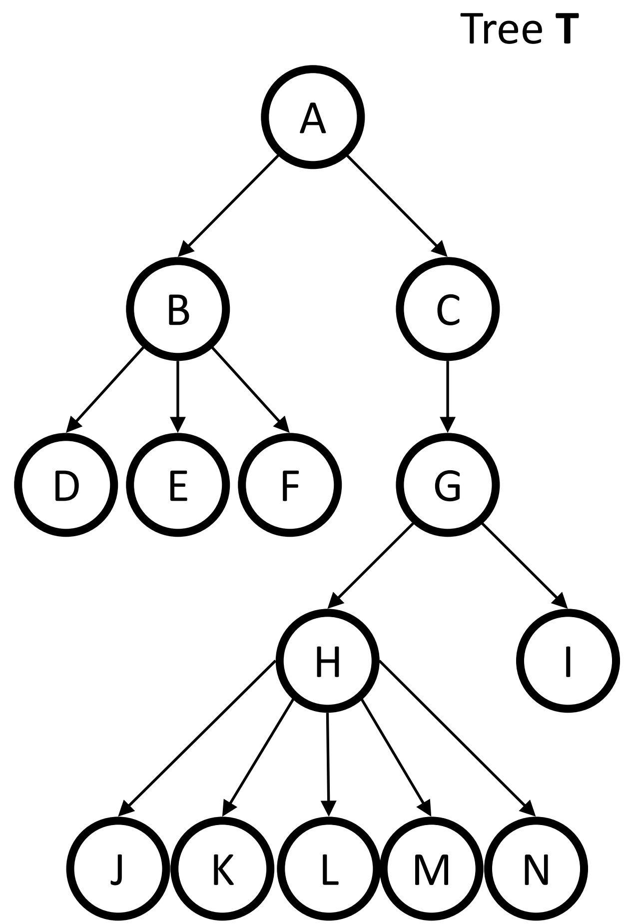
Heaps provide…

- Insert: O(log n) worst case, O(1) on average
- DeleteMin: O(log n) worst and average.

Binary Heap Properties

1. Structure Property
2. Ordering Property

Tree Review



root(**T**)*:*

leaves(**T**)*:*

children(B)*:*

parent(H)*:*

siblings(E)*:*

ancestors(F)*:*

descendents(G)*:*

subtree(C)*:*

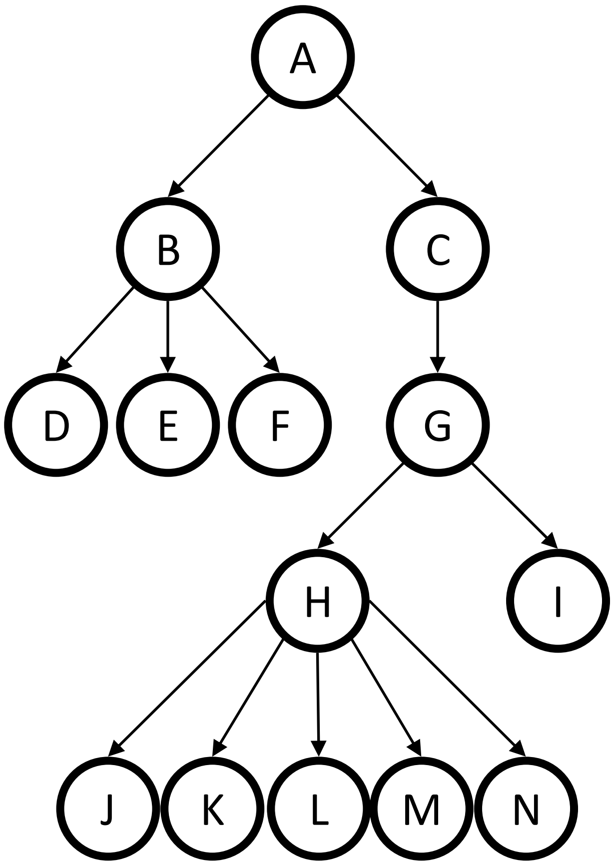
More Tree Terminology

depth(B):

height(G):

degree(B):

branching factor(**T**):

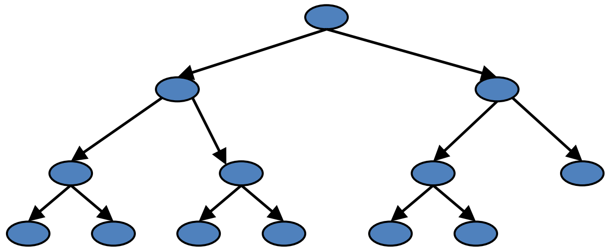


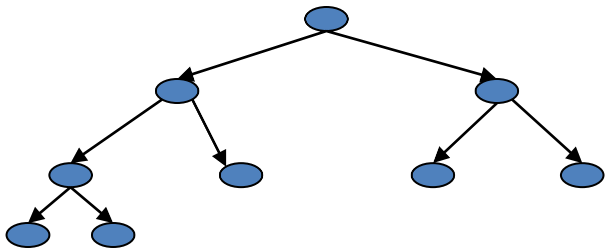
Heap **Structure** Property

- A binary heap is a ***complete*** binary tree.

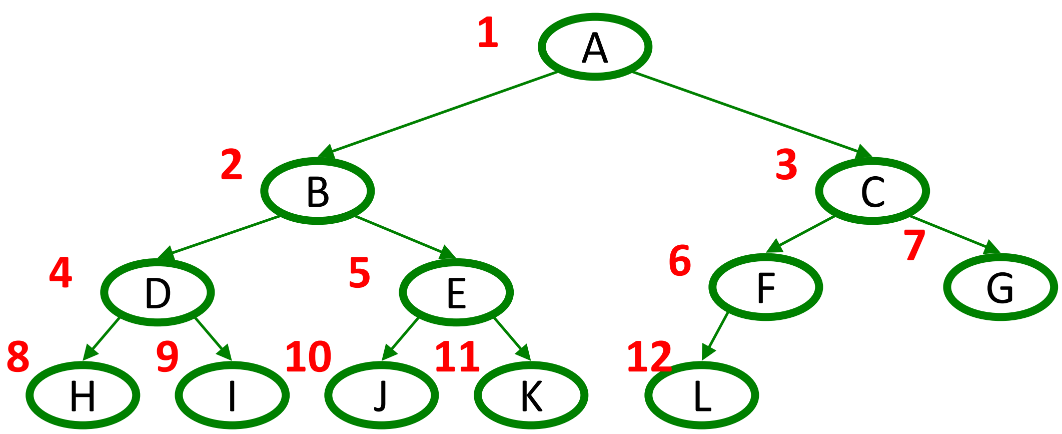
Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:





Representing Complete Binary Trees in an Array



From node **i**:

left child:

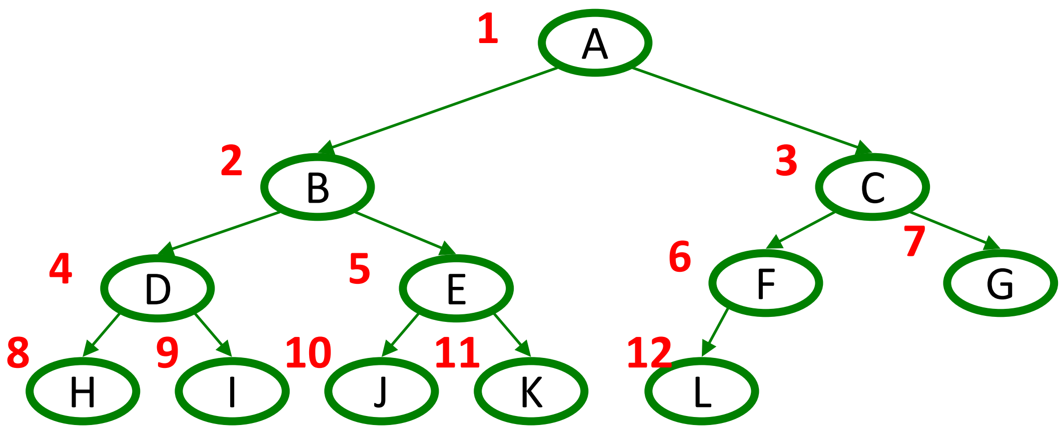
right child:

parent:

implicit (array) implementation:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Representing Complete Binary Trees in an Array



From node **i**:

left child: 2*i

right child: 2*i+1

parent: i/2

implicit (array) implementation:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Why use an array?

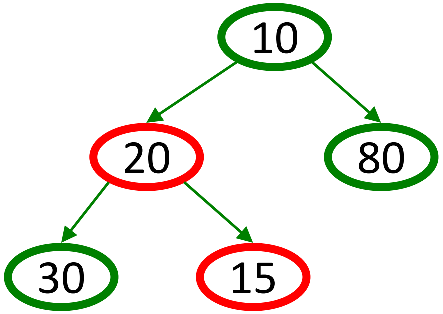
1. Space: No pointers. The arrays are packed.

2. *2, /2, + are faster operations than dereferencing a pointer. (Faster operations) but also, better locality.

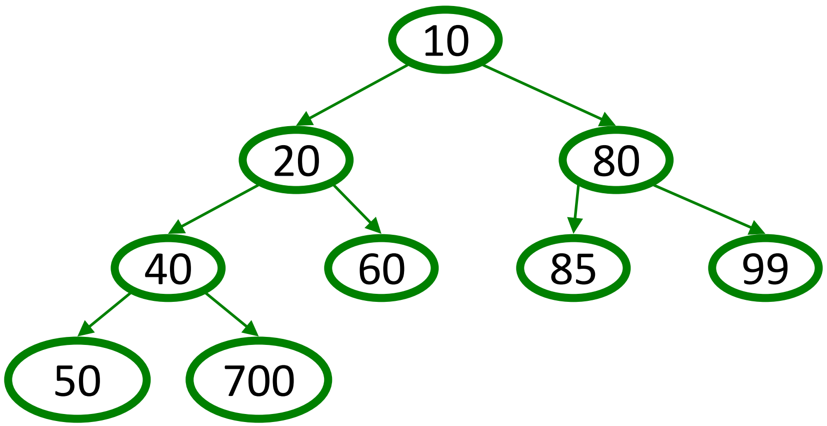
3. Finding the last node in the tree/array takes O(1) time.

Heap **Order** Property

Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X. (MinHeap)



not a heap

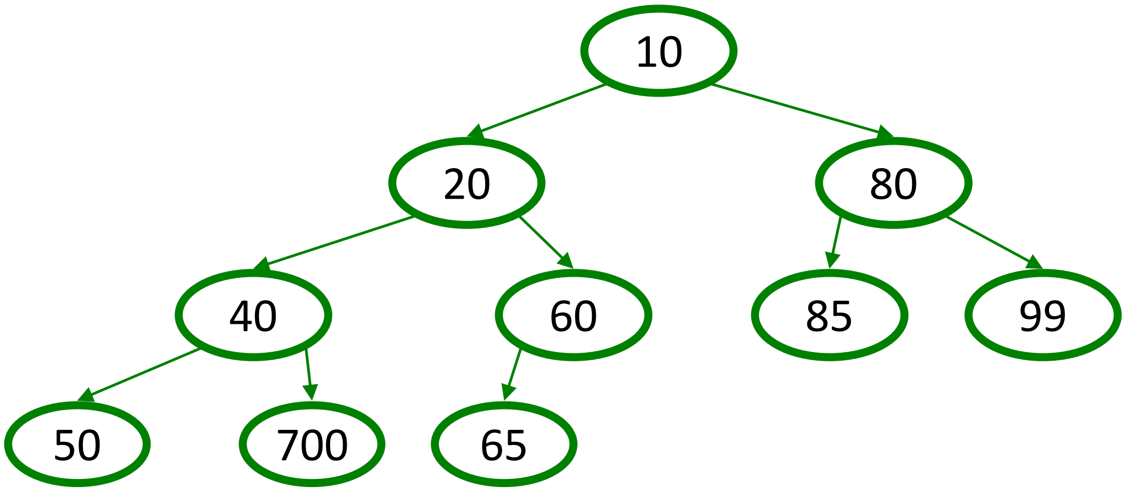


Heap Operations

- findMin:
- insert(val): bubble up.
- deleteMin: sink down.

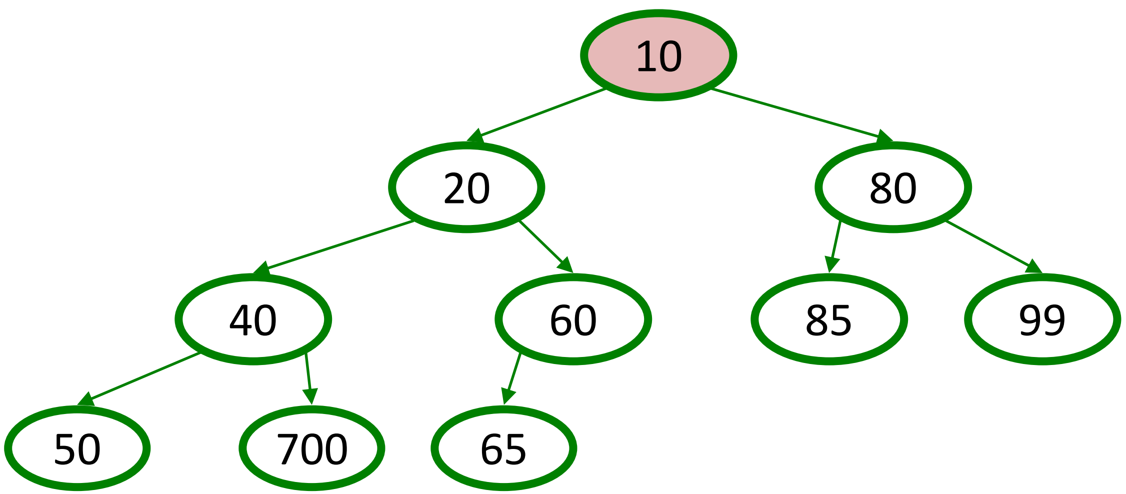
Heap - findMin

- Top/root of the tree



Heap - findMin

- Top/root of the tree
- arr[1]
- O(1)



Heap – Insert(val)

Basic Idea: Append to end (maintaining structure) and bubble up to maintain heap order

1. Put val at “next” leaf position
2. Bubble up by repeatedly exchanging node until no longer needed

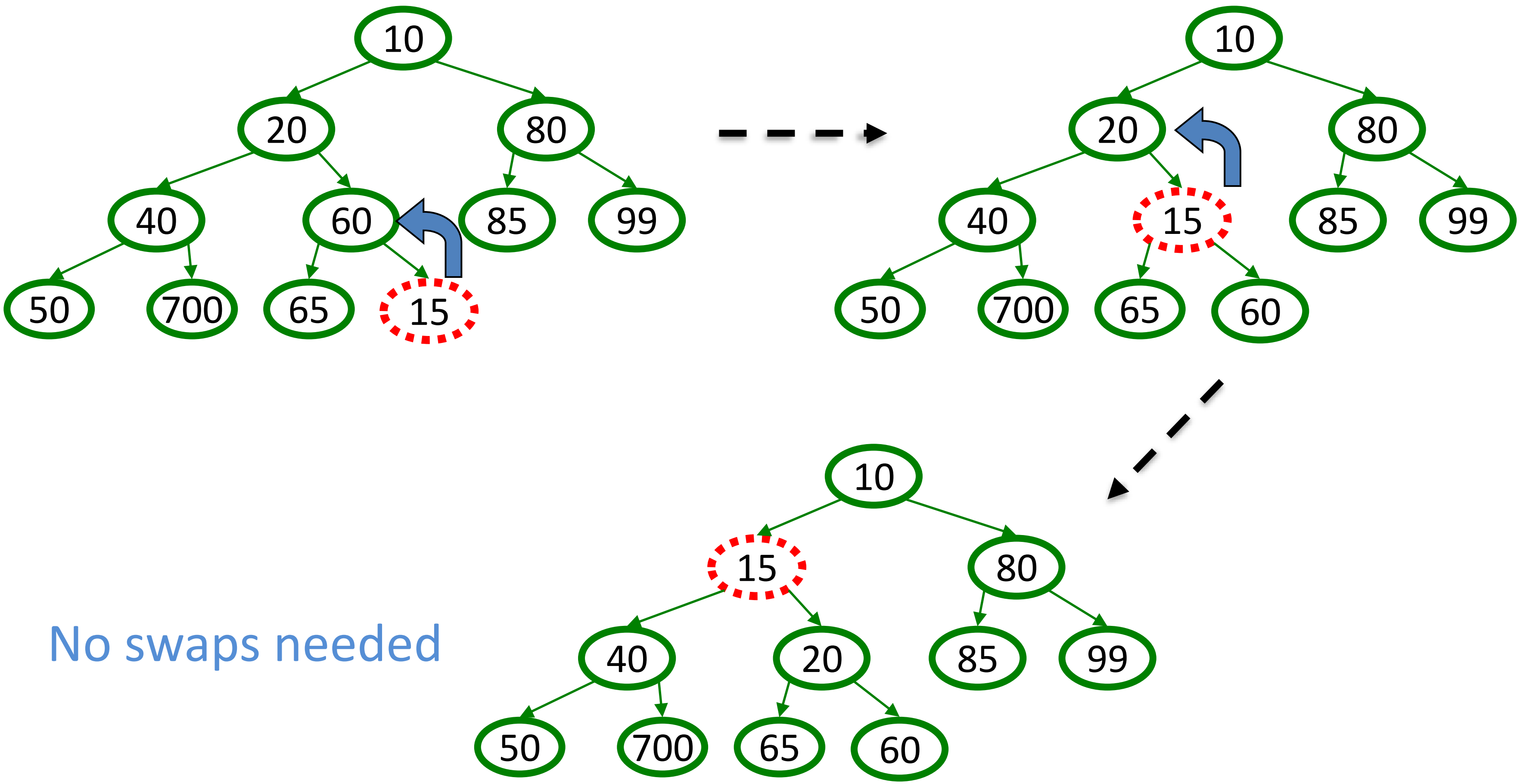
insert(val)

```
void insert(Object o) {
  assert(!isFull());
  size++;
  newPos =
    bubbleUp(size,o);
  Heap[newPos] = o;
}
```

runtime: O(logN) worst case
constant: on average

Heap – Example

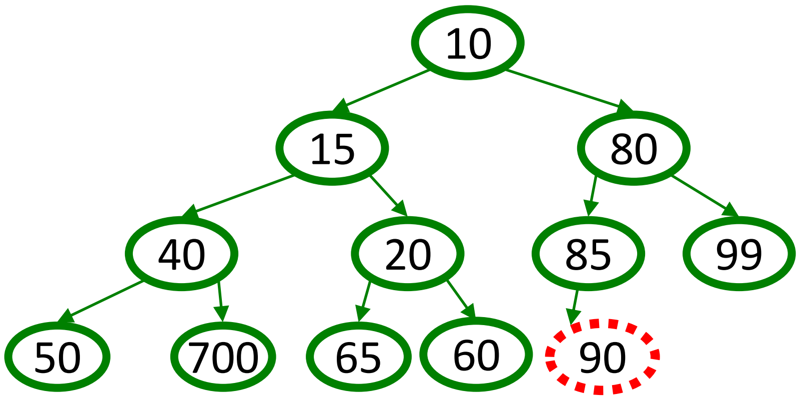
Insert 15



No swaps needed

Heap – Example

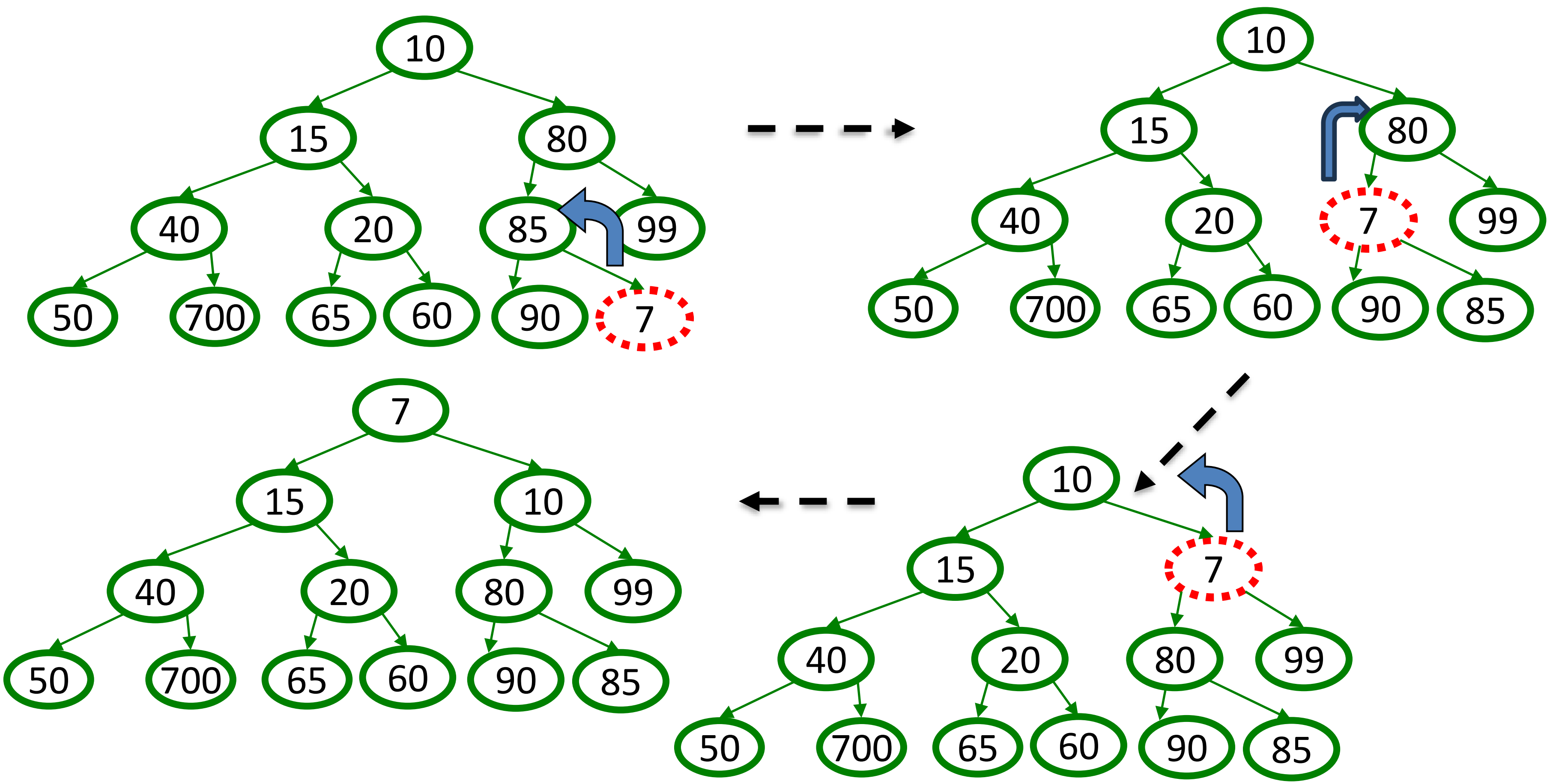
Insert 90



No swaps needed

Heap – Example

Insert 7



Insert Code (optimized)

```
void insert(Object o) {
  assert(!isFull());
  size++;
  newPos =
    bubbleUp(size,o);
  Heap[newPos] = o;
}
```

```
int bubbleUp(int hole,
               Object val) {
  while (hole > 1 &&
         val < Heap[hole/2])
    Heap[hole] = Heap[hole/2];
    hole /= 2;
  }
  return hole;
}
```

– bubble up an EMPTY space, and then do a swap (reduces the # of swaps).

runtime: O(logN) worst case

constant: on average

Heap – Deletemin

Basic Idea:

1. Remove root (that is always the min!)
2. Put “last” leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

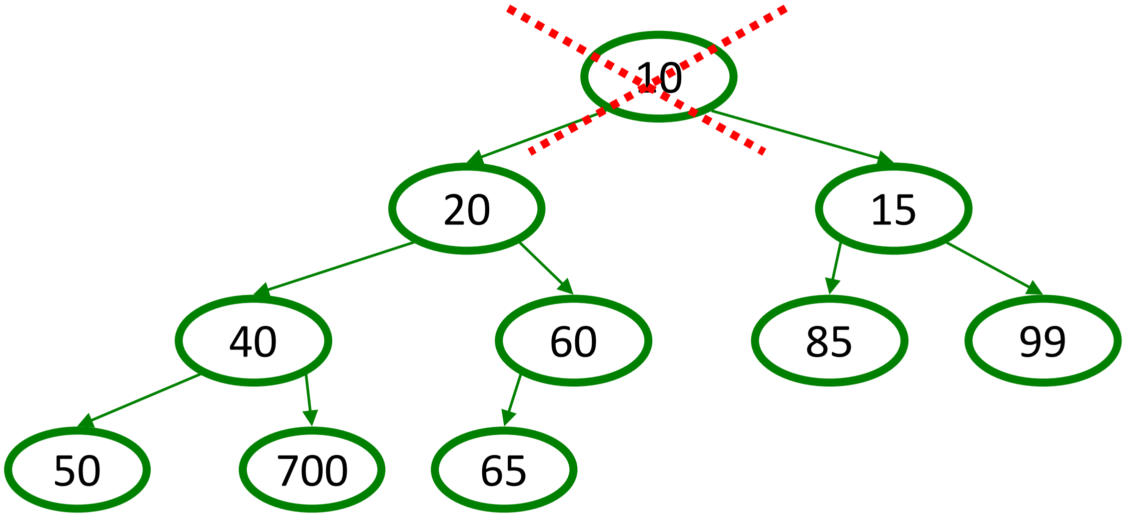
DeleteMin Code

```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[1];
  size--;
  newPos =
    sinkDown(1,
        Heap[size+1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

runtime: O(logN)

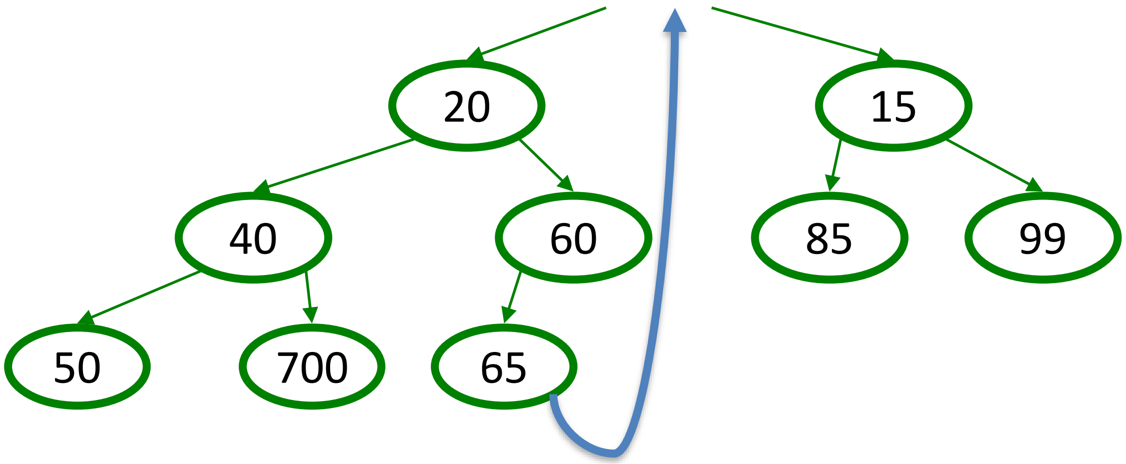
Heap – Deletemin

Delete 10



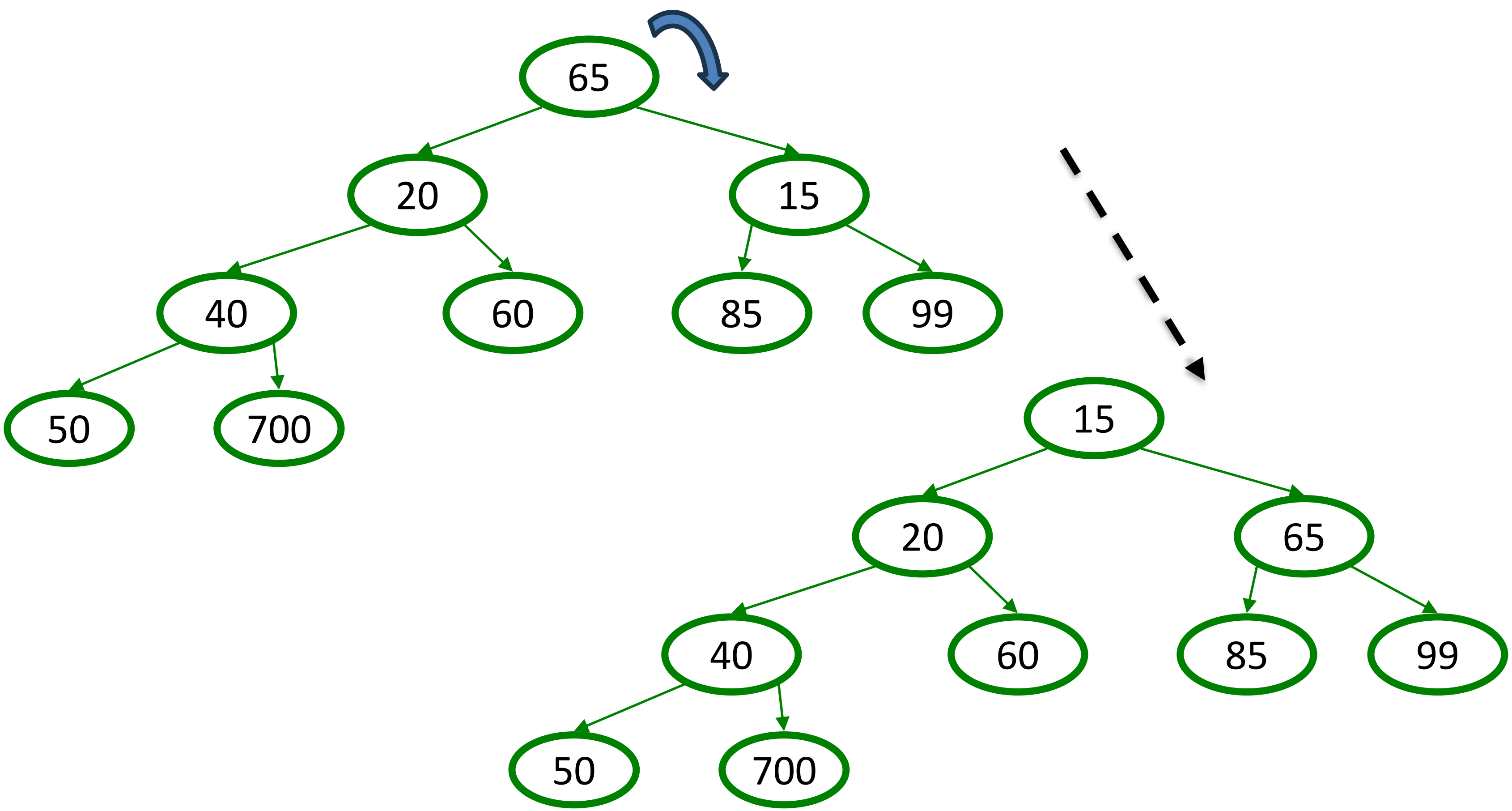
Heap – Deletemin

Delete 10



Heap – Deletemin

Delete 10



DeleteMin Code (Optimized)

```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[1];
  size--;
  newPos =
    sinkDown(1,
        Heap[size+1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

runtime: O(logN)

```
int sinkDown(int hole,
                  Object val) {
while (2*hole <= size) {
    left = 2*hole;
    right = left + 1;
    if (right ≤ size &&
        Heap[right] < Heap[left])
      target = right;
    else
      target = left;

    if (Heap[target] < val) {
      Heap[hole] = Heap[target];
      hole = target;
    }
    else
      break;
  }
  return hole;
}
```

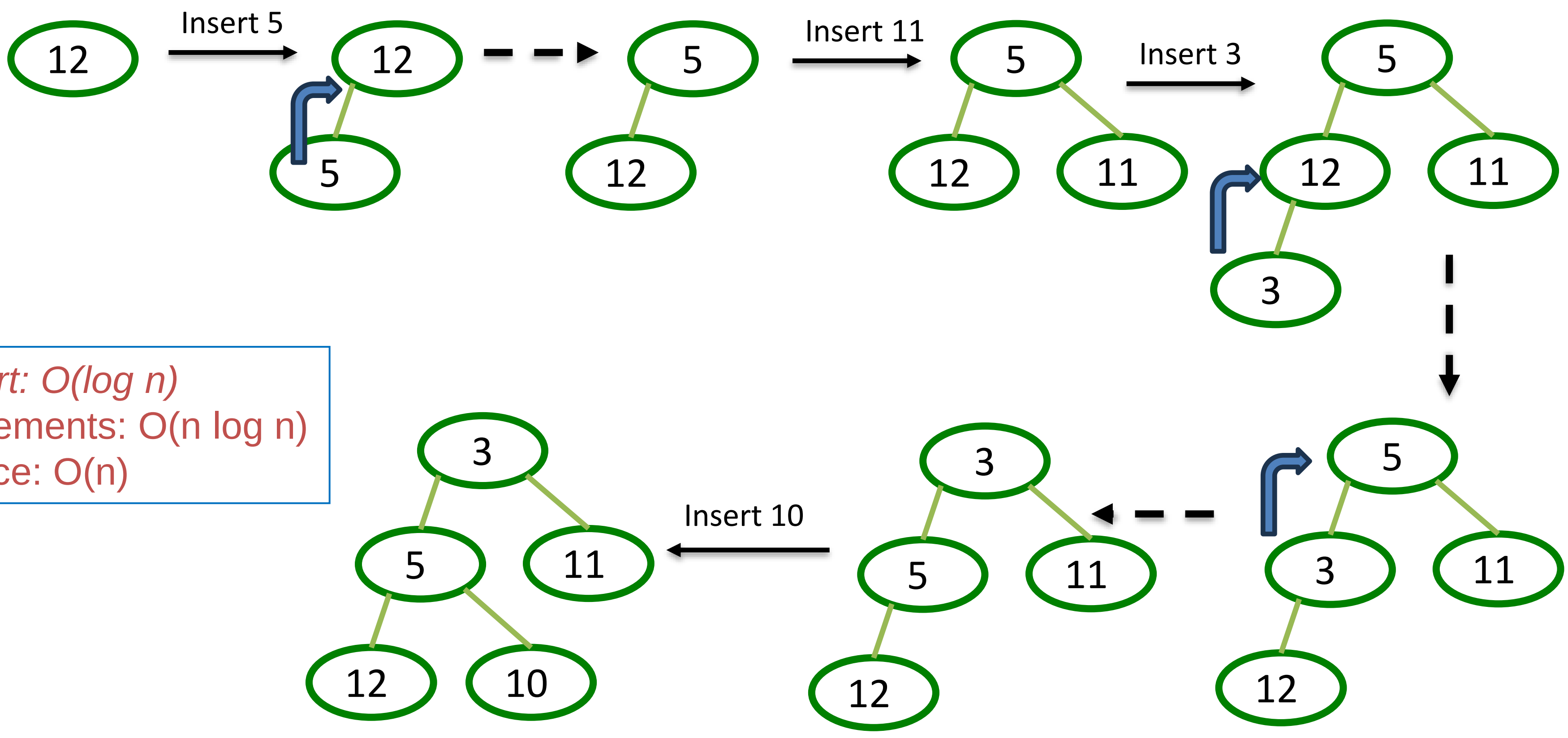
Heap – Update a key

Basic Idea: Given a ptr to value, change its “key” and bubble up/sink down to maintain heap order

```
void update(T *pval,int priority) {
  auto old_priority = pval->priority;
  pval->priority = priority;
  if(priority<old_priority)
    bubbleup(pval);
  else
    sinkdown(pval);
}
```

Create a heap

Insert 12, 5, 11, 3 , 10



Insert: O(log n)
N elements: O(n log n)
Space: O(n)

Building a Heap

- Adding the items one at a time is O(n log n) in the worst case

- Can we do it in O(n)?

Working on Heaps

- What are the two properties of a heap?
  - Structure Property
  - Order Property

- How do we work on heaps?
  - Fix the structure
  - Fix the order

Buildheap: Floyd’s method

Basic Idea: Construct a complete binary tree, sink down nodes from halfway, all the way to the root.

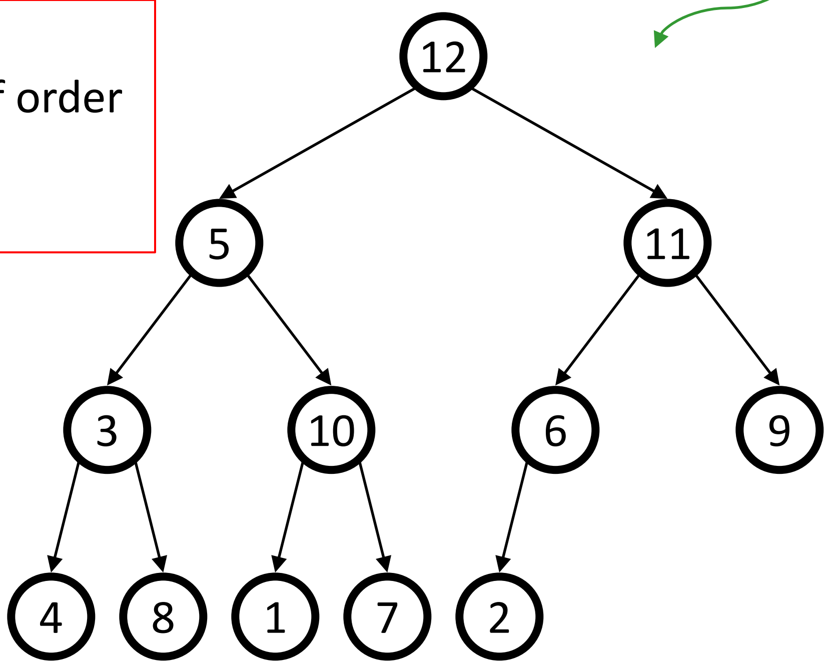
```
private void buildHeap() {
  for ( int i = currentSize/2; i > 0; i-- )
      sinkDown( i );
}
```

BuildHeap: Floyd's Method bottom up

	12	5	11	3	10	6	9	4	8	1	7	2

Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!

Question:
which nodes MIGHT be out of order in any heap?

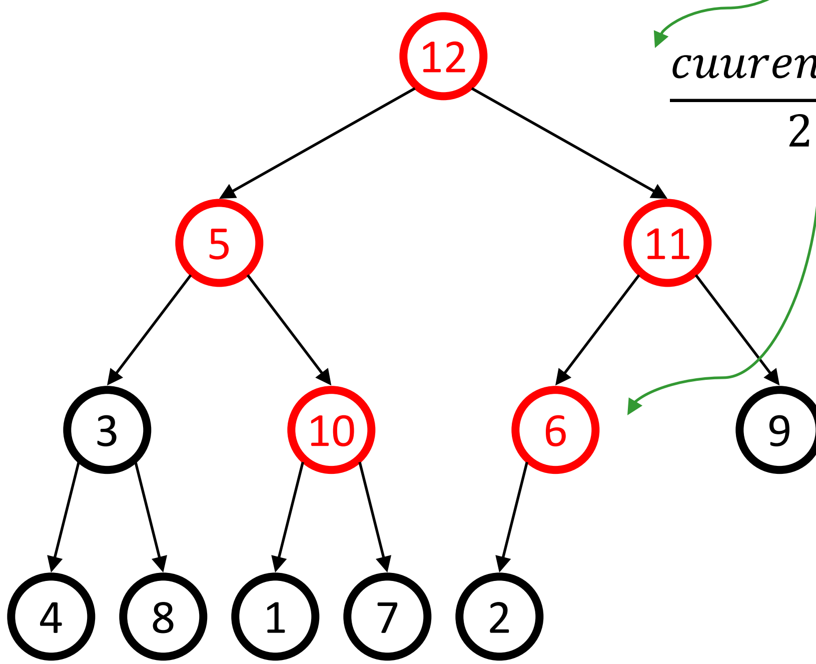


BuildHeap: Floyd's Method bottom up

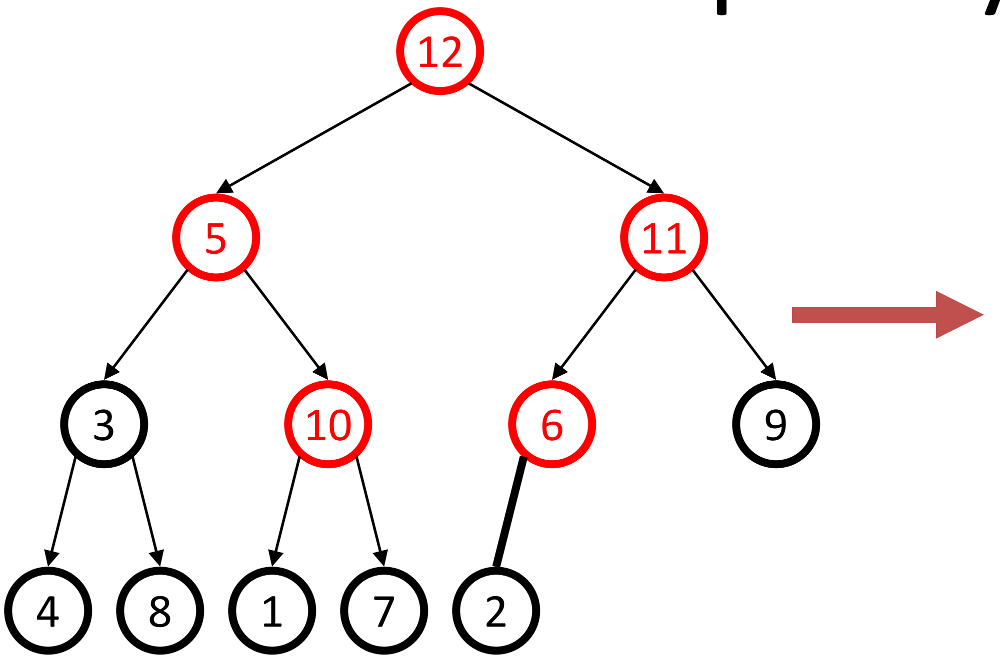
	12	5	11	3	10	6	9	4	8	1	7	2

Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!

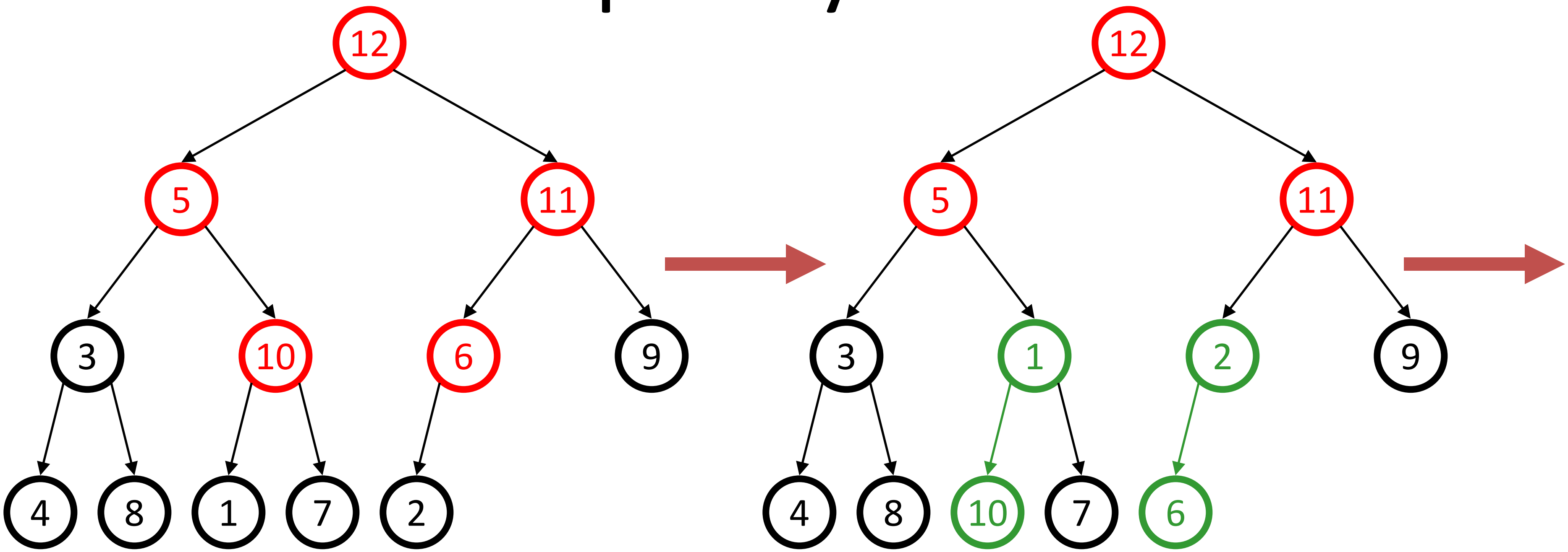
$$\frac{cuurentSize}{2} = \frac{13}{2} = 6$$



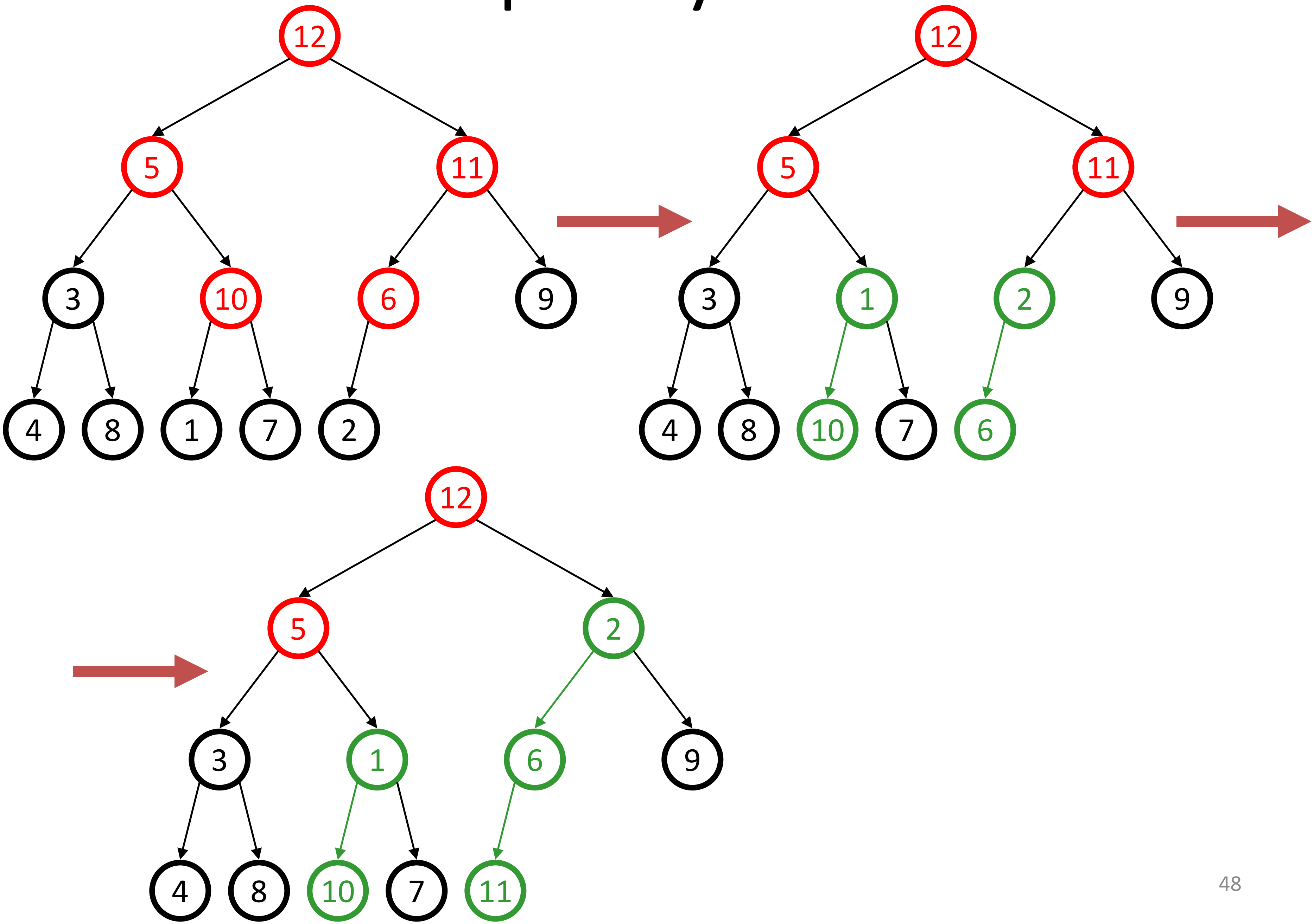
BuildHeap: Floyd's Method



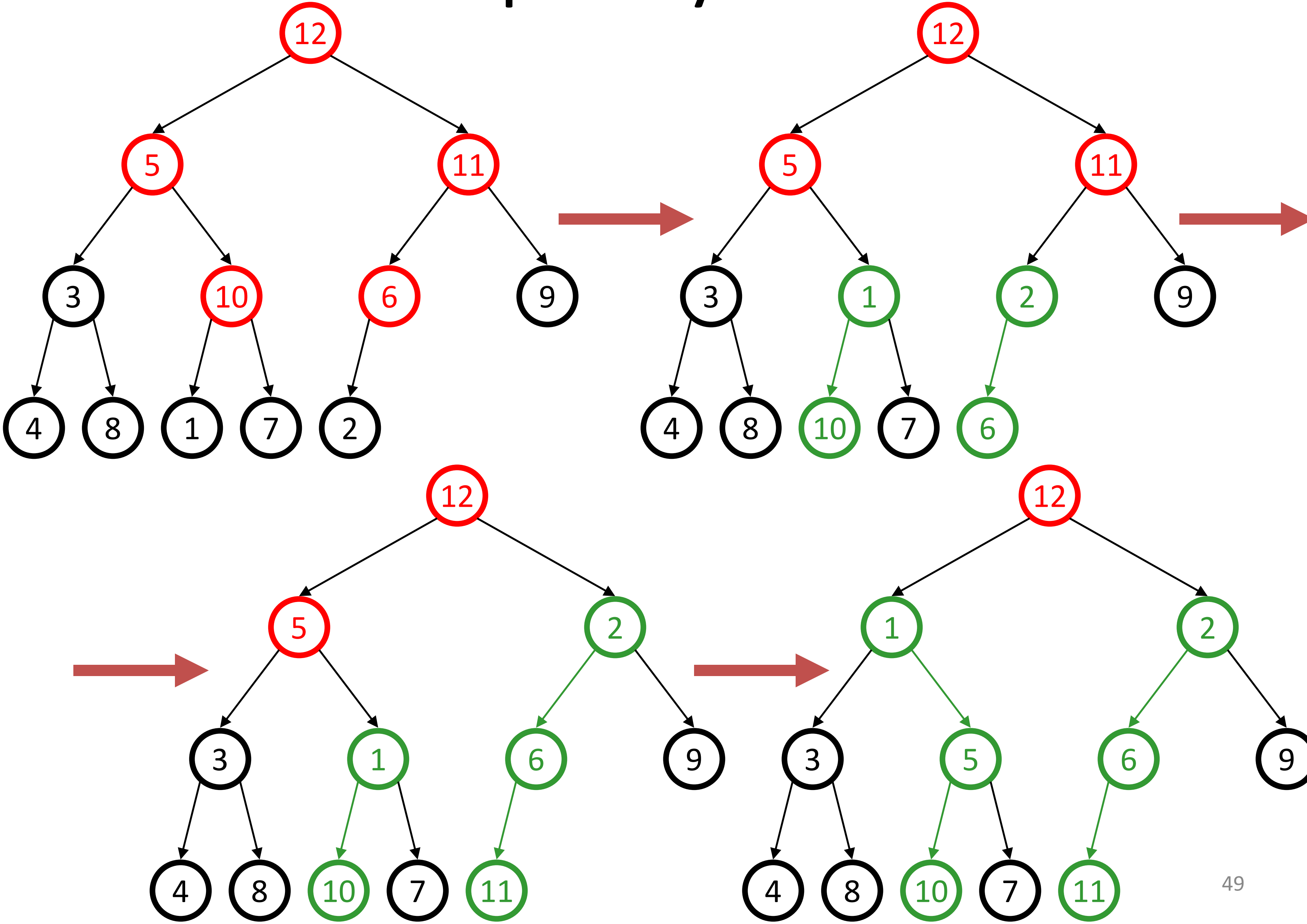
BuildHeap: Floyd's Method



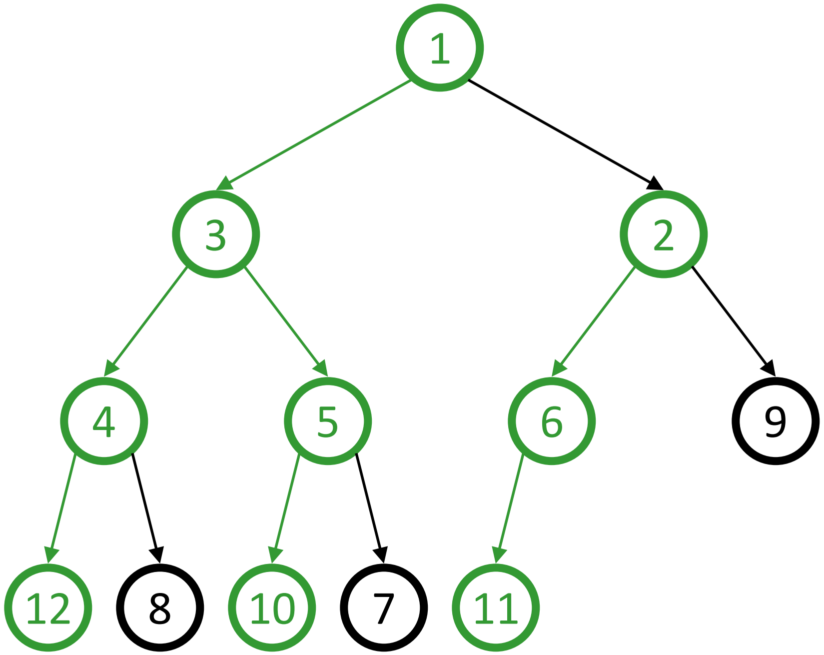
BuildHeap: Floyd's Method



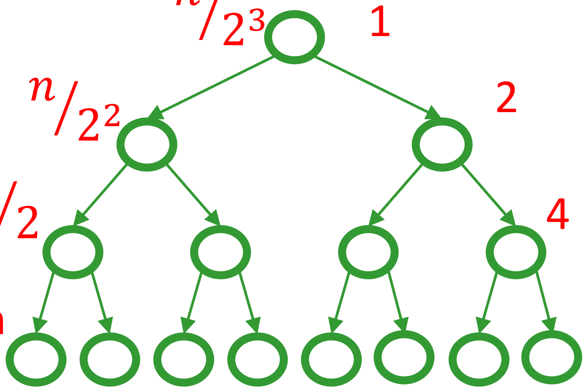
BuildHeap: Floyd's Method



Finally…



Floyd's method: Time complexity



How many sink down operations?

$$T(n) = \frac{n}{2} * 1 + \frac{n}{4} * 2 + \frac{n}{8} * 3 + \ \dots\dots.\log(n)$$

$$= n\,(\tfrac{1}{2} + \tfrac{1}{2^2} + \tfrac{1}{2^3} + \ \dots\dots.e^{\log(n)})$$

$$= n\,(\sum_{i=1}^{\log(n)} \frac{i}{2^i})$$

$$= n * 2$$

$$= O\,(n)$$

Facts about Heaps

Observations:

- Finding a child/parent index is a multiply/divide by two
- Operations jump widely through the heap
- Each bubble/sink step looks at only two new nodes
- Inserts are at least as common as deleteMins

Realities:

- Division/multiplication by powers of two are equally fast
- Looking at only two new pieces of data: bad for cache!
- With huge data sets, disk accesses dominate

Heapsort

HeapSort

- Can we find a way to avoid duplicating the data and still take advantage of the heapifying operation?

- HeapSort takes advantage of the fact that our heaps are implemented using arrays.

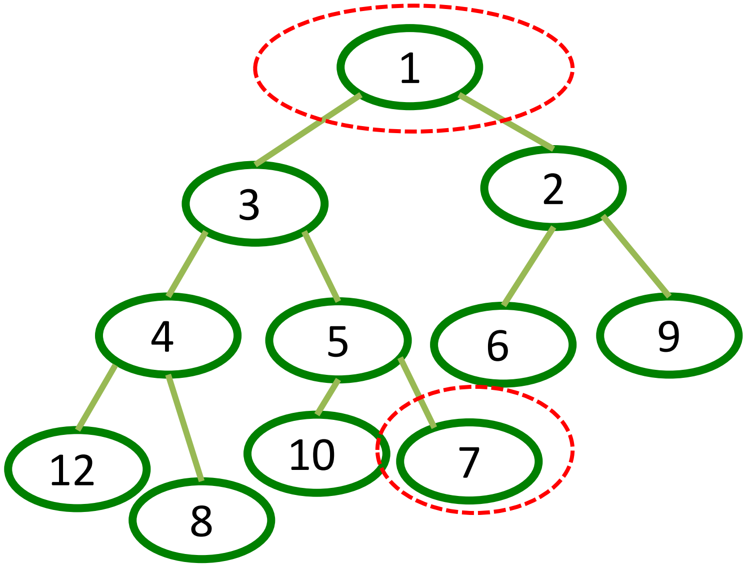
- Two step algorithm:
  1. Construct a heap within the passed array.
  2. While heap is not empty (size != 1)
     a. "extract" the largest item
     b. Heapify the remaining heap.

Sorting with a Priority Queue

```
void heapSort(int a[], int size){
   Heap<int> heap(size);
   for(int i = 0; i < size; ++i)
     heap.insert(a[i]);
   for(int i = size-1; i >= 0; --i)
     a[i] = heap.pop();
}
```

- What is the complexity of this sorting method?
- What is the disadvantage of it?
  - How much space is required to sort a[ ]?

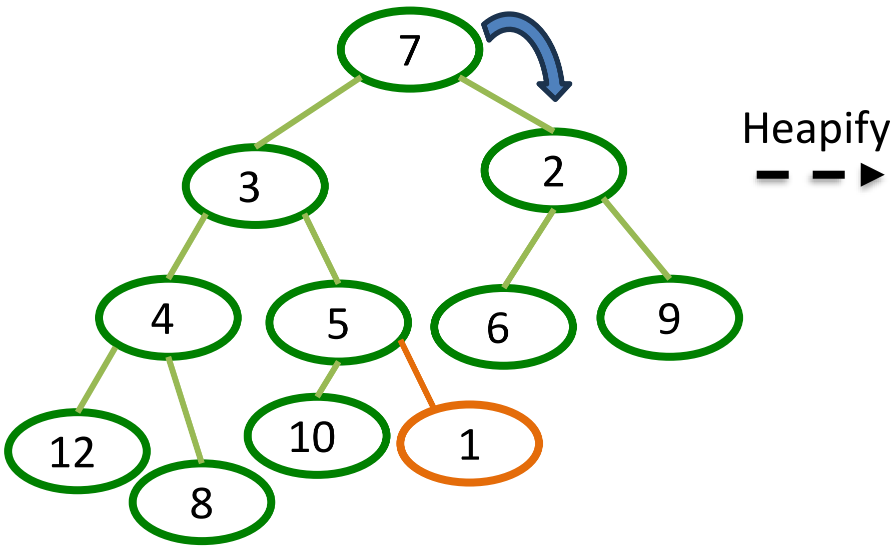
Heapsort: Example



Replace 1 and 7

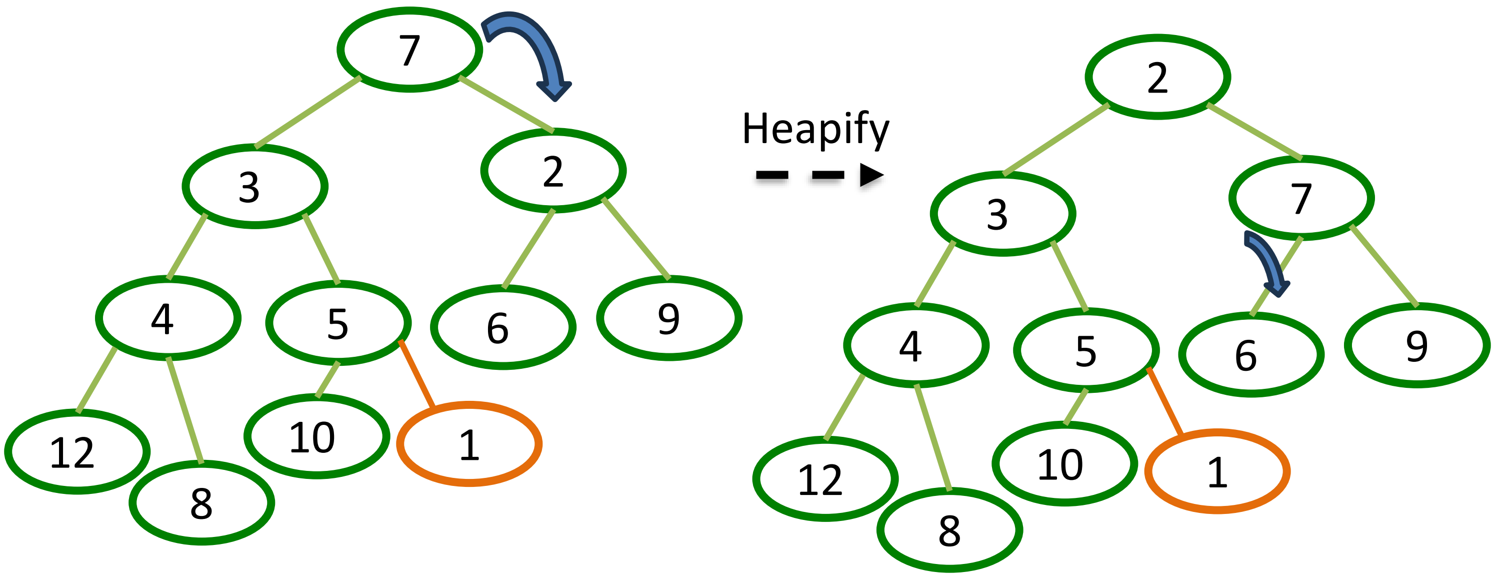
	1	3	2	4	5	6	9	12	8	10	7	

Heapsort: Example



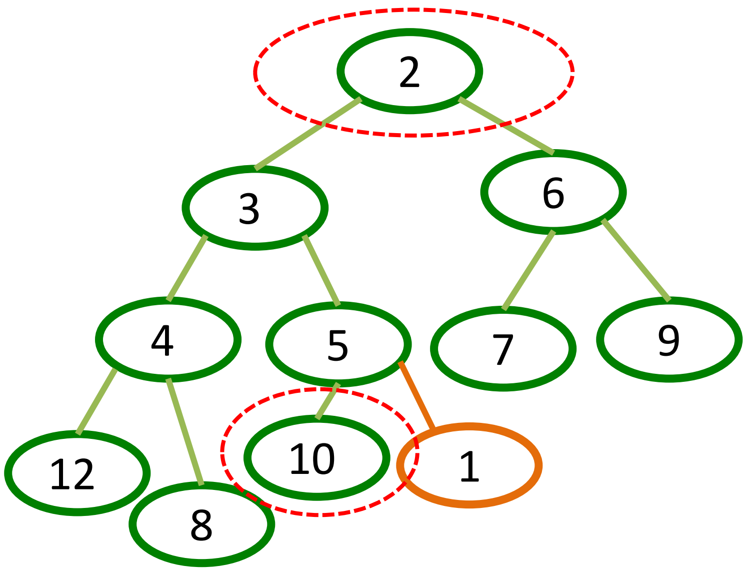
	7	3	2	4	5	6	9	12	8	10	1	

Heapsort: Example



	~~7~~2	3	~~2~~7	4	5	6	9	12	8	10	1	

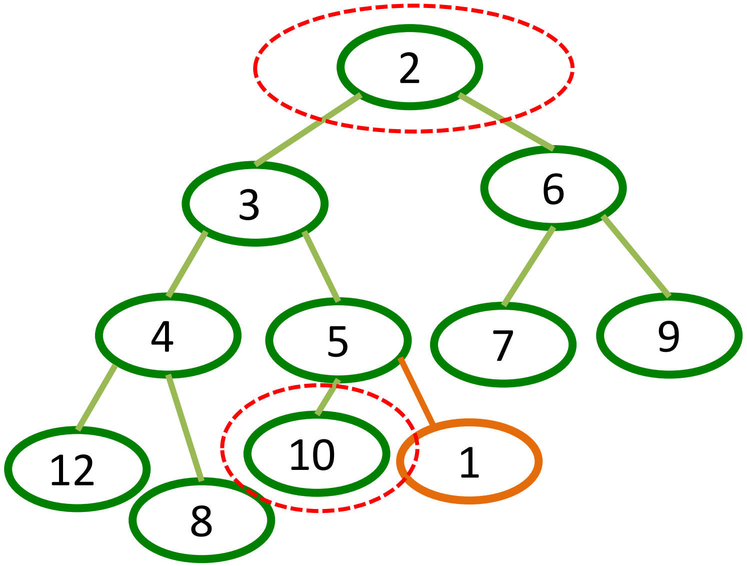
Heapsort: Example



Replace 2 and 10

	2	3	6	4	5	7	9	12	8	10	1	

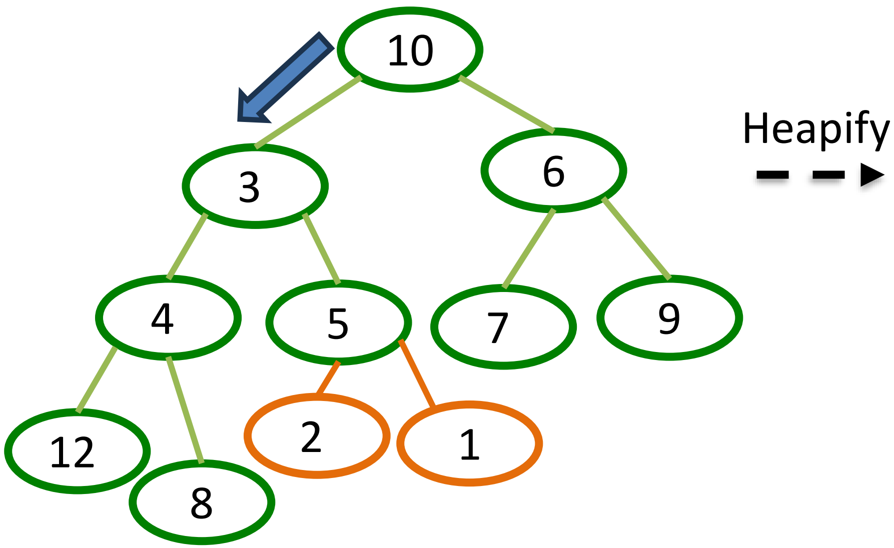
Heapsort: Example



Replace 2 and 10

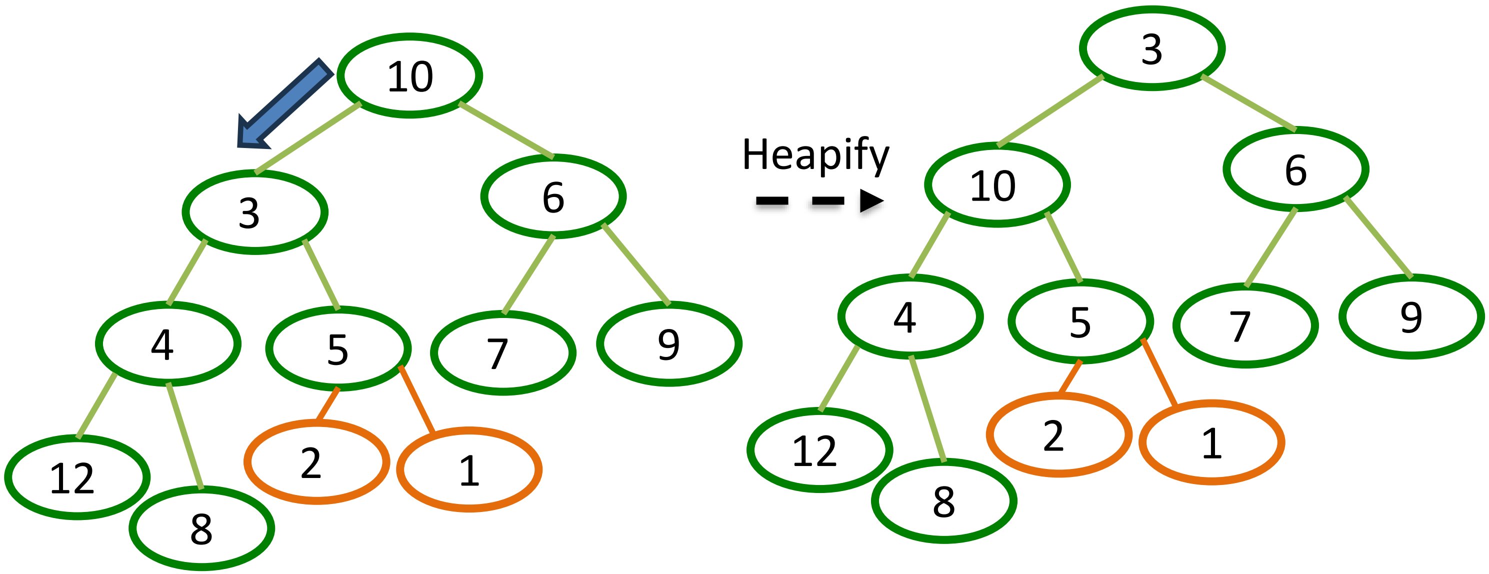
	2	3	6	4	5	7	9	12	8	10	1	

Heapsort: Example



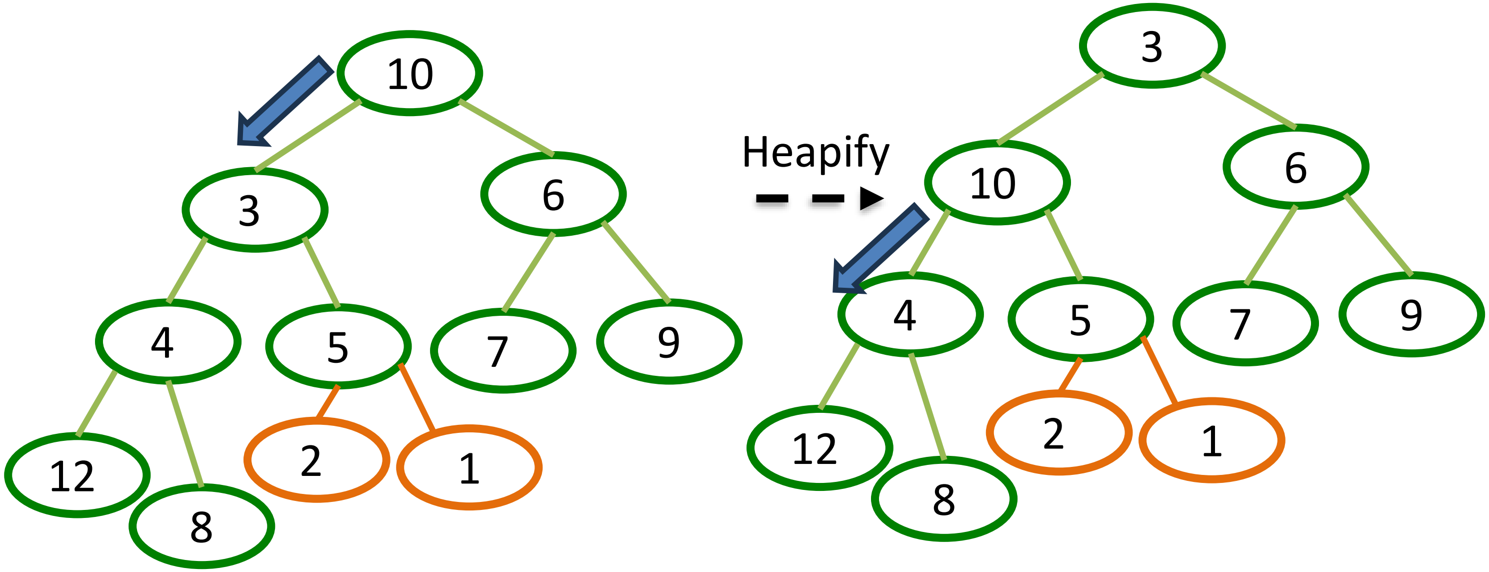
	10	3	6	4	5	7	9	12	8	2	1	

Heapsort: Example



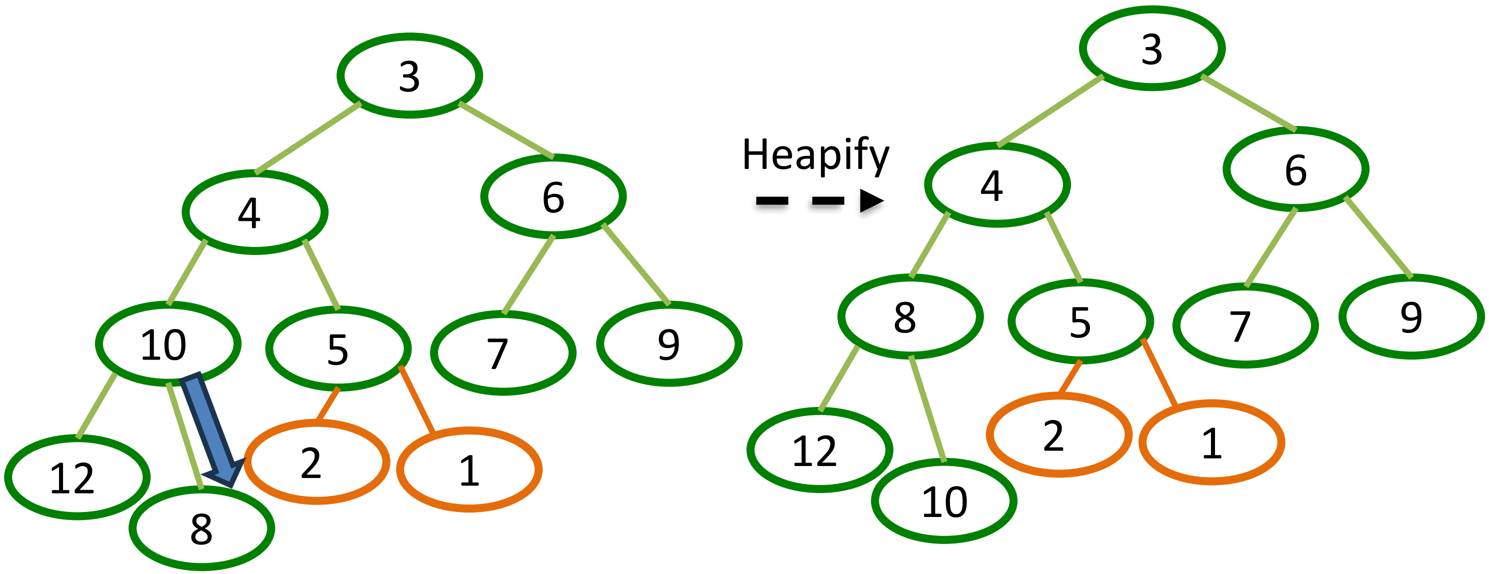
	~~10~~3	~~3~~10	6	4	5	7	9	12	8	2	1	

Heapsort: Example



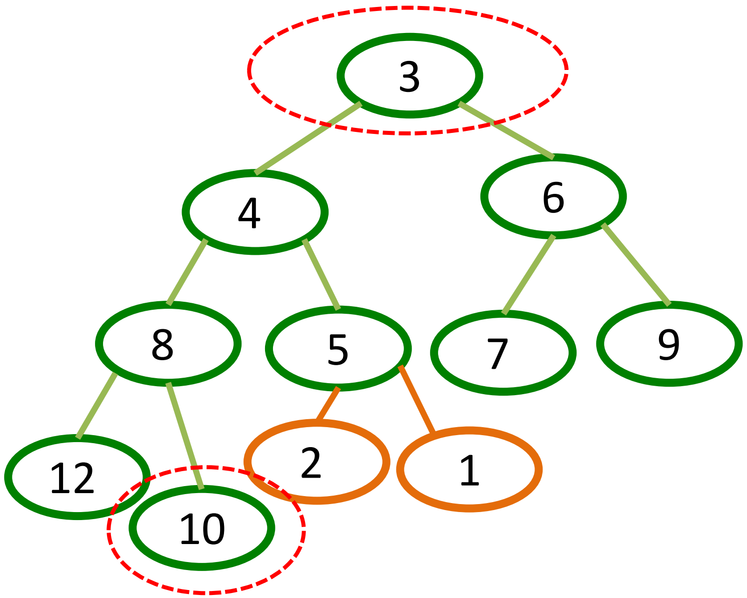
	~~10~~3	~~310~~4	6	~~4~~10	5	7	9	12	8	2	1	

Heapsort: Example



	~~10~~3	~~310~~4	6	~~410~~8	5	7	9	12	~~8~~10	2	1	

Heapsort: Example



Replace 3 and 10

	3	4	6	8	5	7	9	12	10	2	1	

Heap Sort: Complexity

- For each operation: log (n)
- Sorting n elements: O(nlogn)
- Space: O(n)