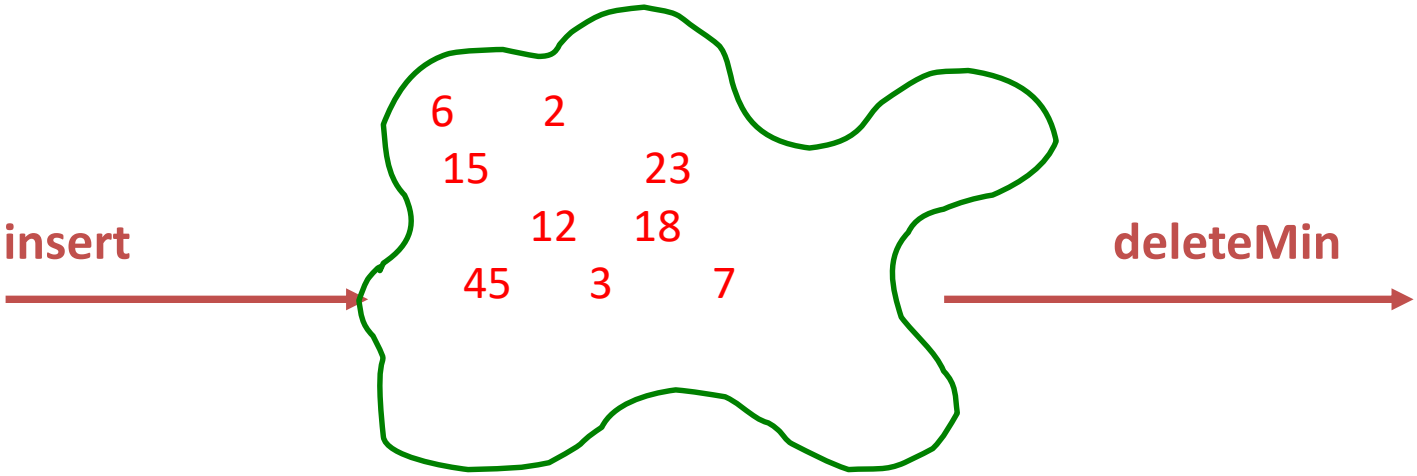
Priority Queues – Binary Heaps

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)*:*

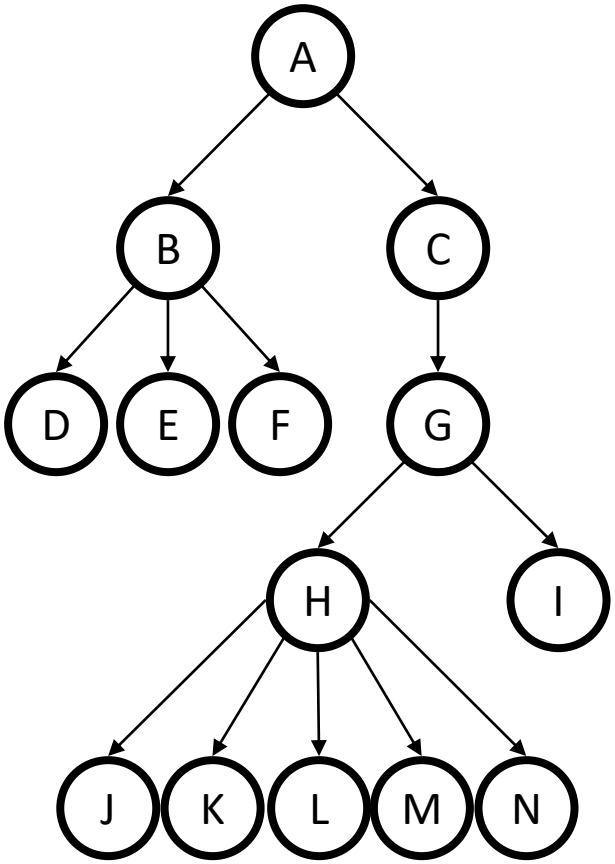
siblings(E)*:*

ancestors(F)*:*

descendents(G)*:*

subtree(C)*:*

Tree **T**



More Tree Terminology

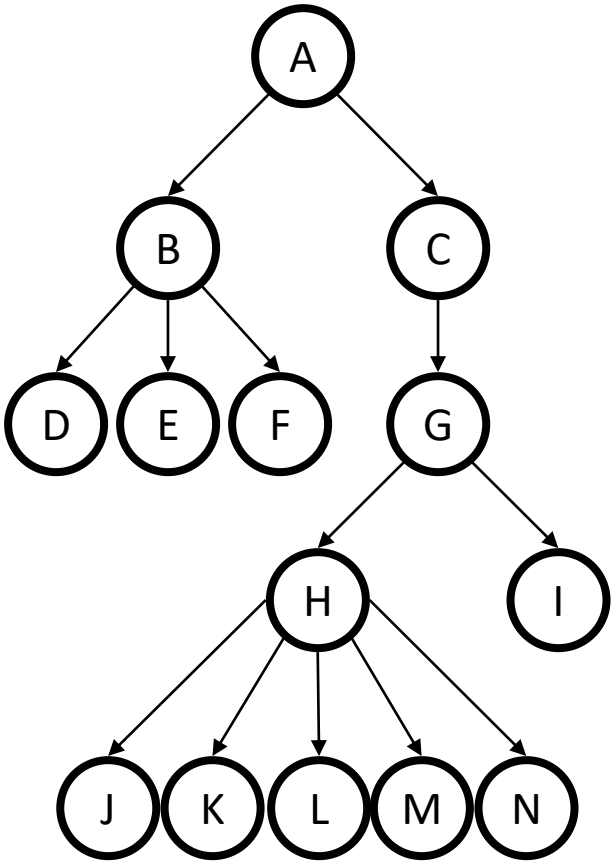
depth(B)*:*

height(G)*:*

degree(B)*:*

branching factor(**T**)*:*

Tree **T**

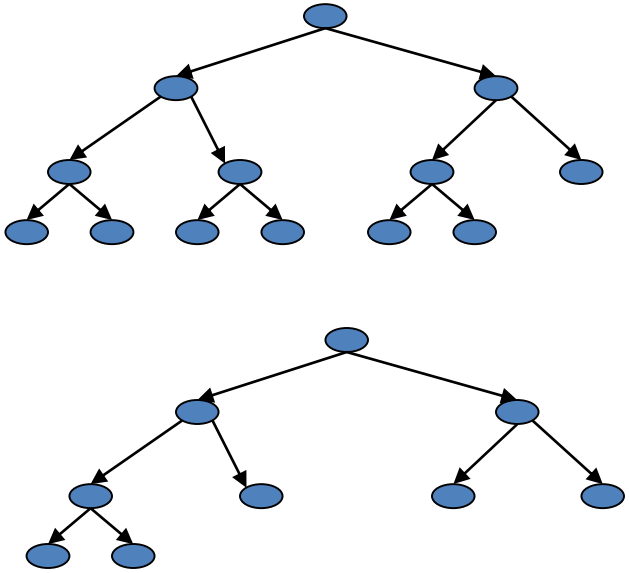


Heap **Structure** Property

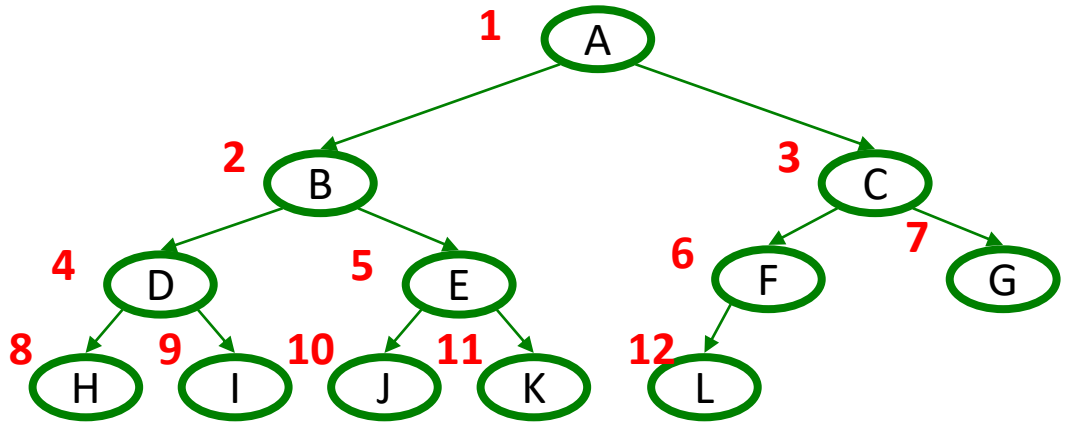
- A binary heap is a ***complete*** binary tree.

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:

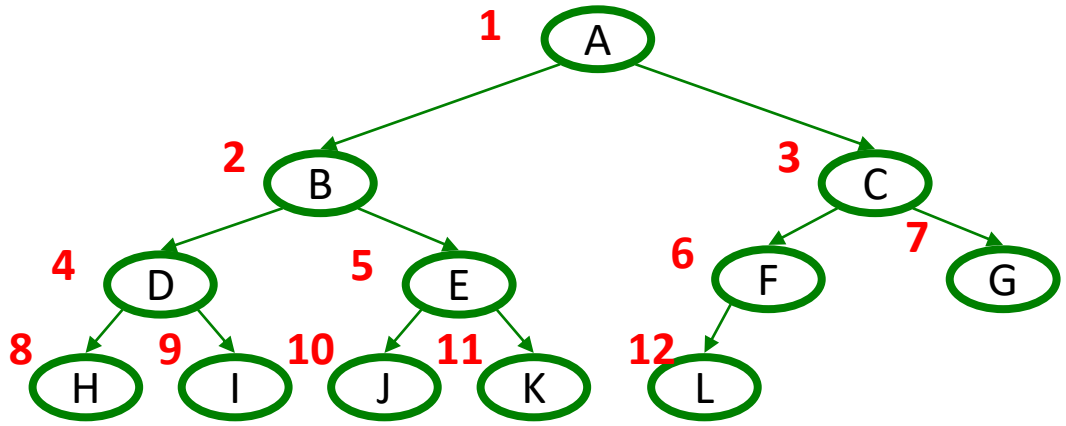
right child:

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?

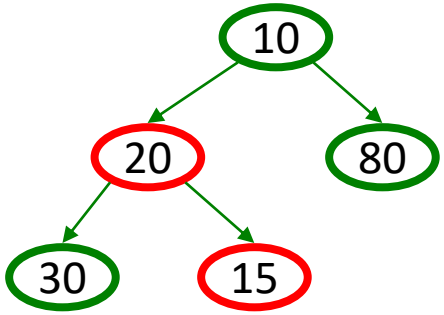
1. Space: No pointers. The arrays are packed.

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

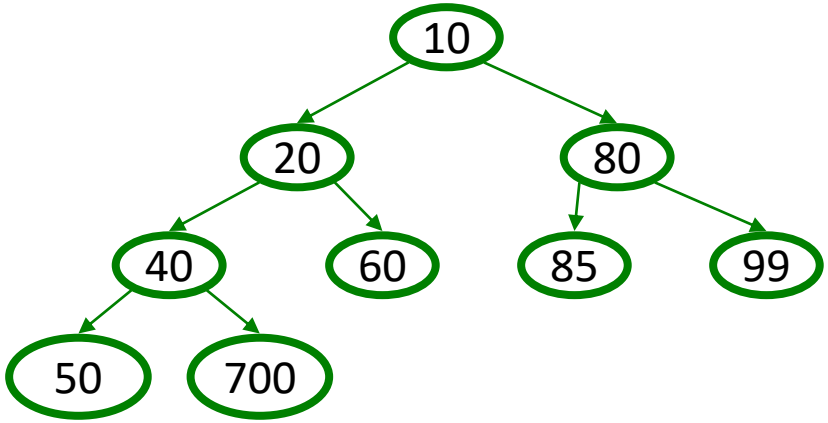
3. Finding the last node in the tree/array takes O(1) time.

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)

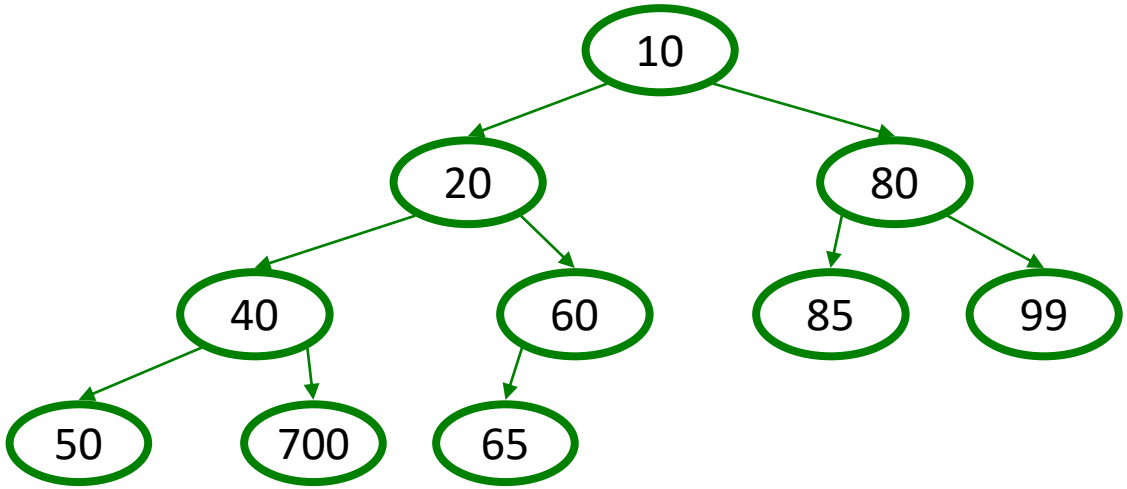


not a heap



Heap Operations

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



Heap – Insert(val)

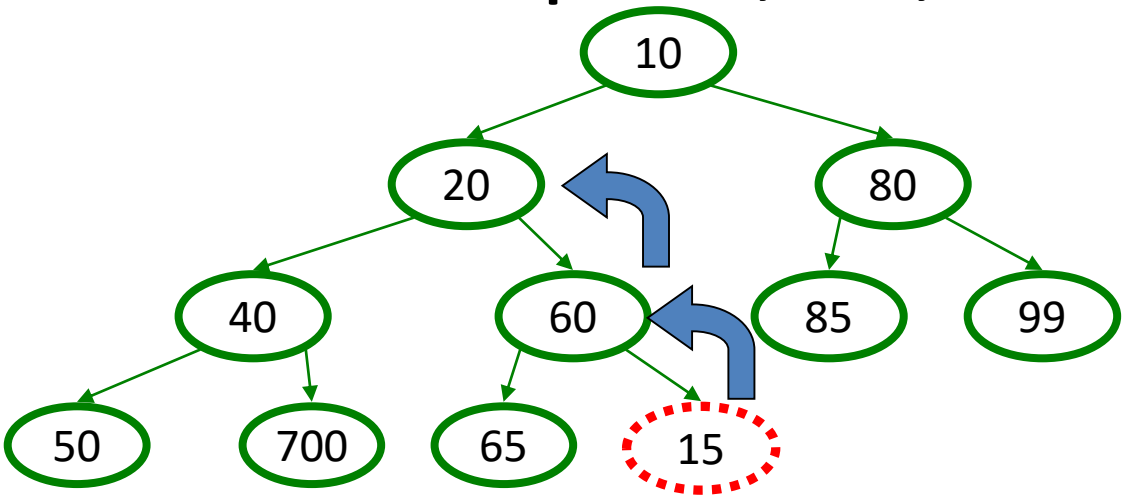
Basic Idea:

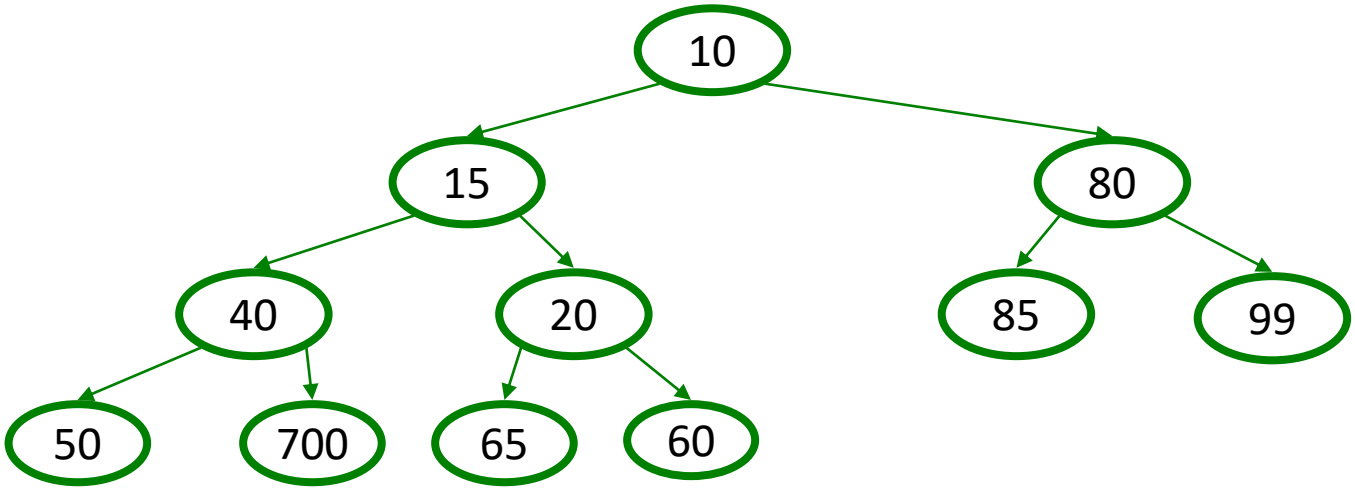
1. Put val at “next” leaf position
2. Bubble up by repeatedly exchanging node until no longer needed

QUESTION:
How long does step 1 take?
Max number of exchanges is ? O(log N) – must percolate up to root.

On average, analysis shows that you tend to only need to move up 1.67 levels, so get O(1) on average.

Insert: bubble up: 15, 90, 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

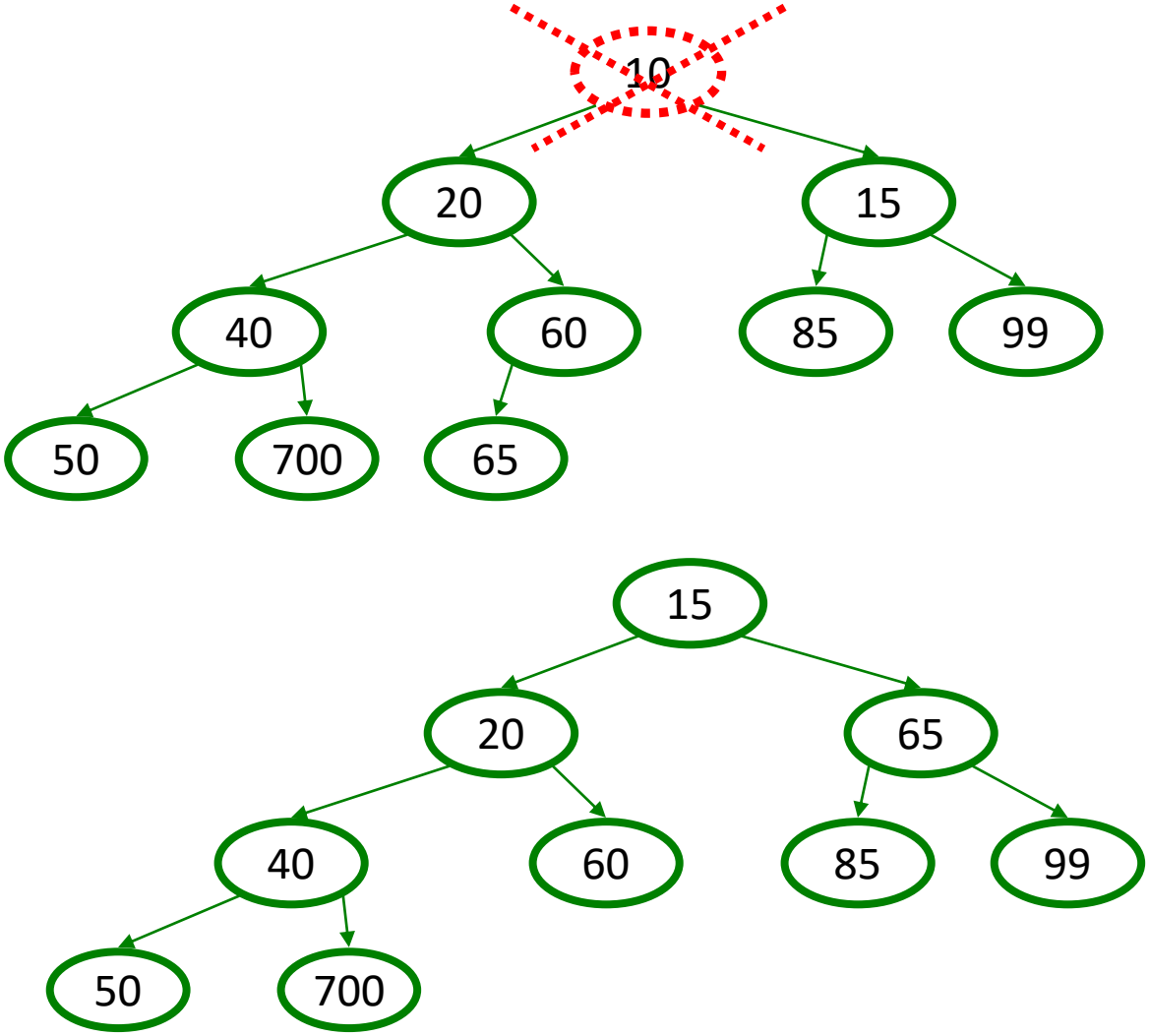
Heap – Deletemin

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.

QUESTION: Max number of exchanges is ?

DeleteMin: percolate down



DeleteMin Code (Optimized)

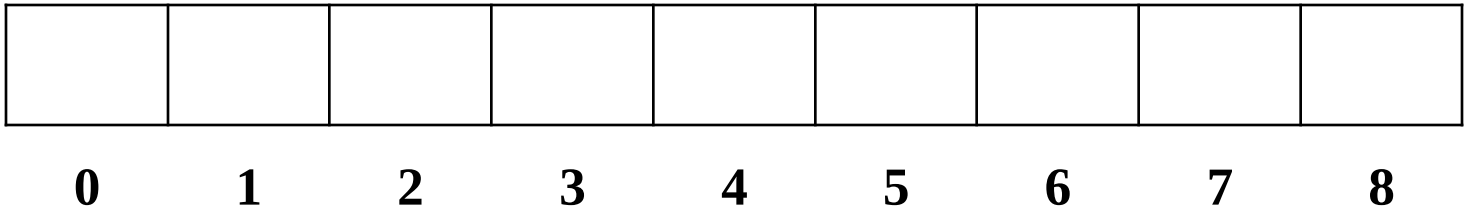
```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[1];
  size--;
  newPos =
    sinkDown(1,
         Heap[size+1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

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;
}
```

Insert: 16, 32, 4, 69, 105, 43, 2



Data Structures

Binary Heaps

Building a Heap

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

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

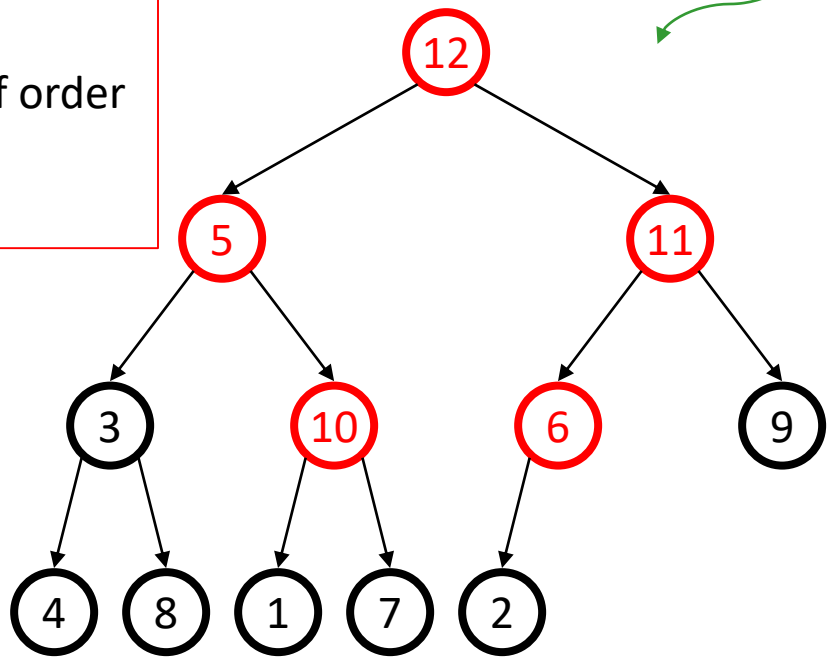
- How do we work on heaps?
  - Fix the structure
  - Fix the order

BuildHeap: Floyd's Method buttom 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?

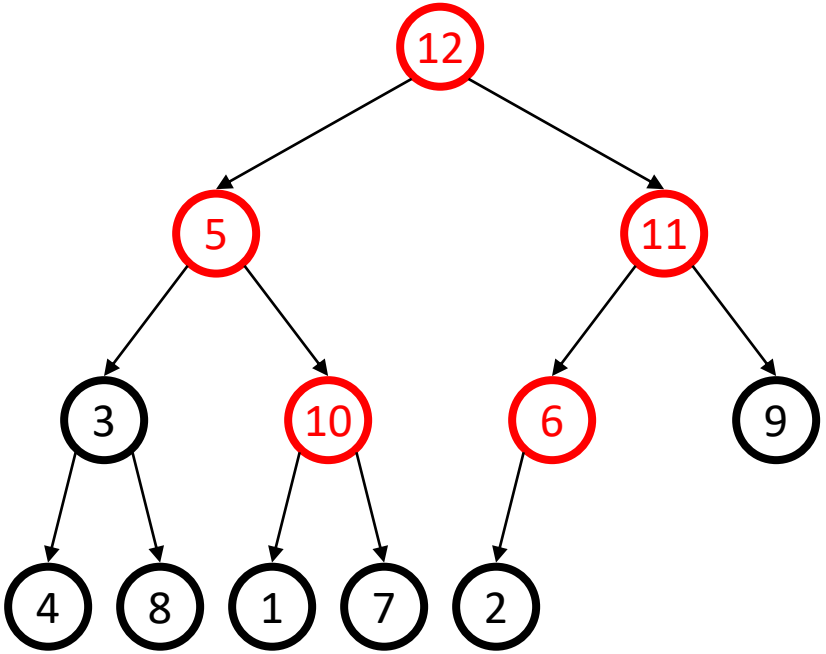


Buildheap pseudocode

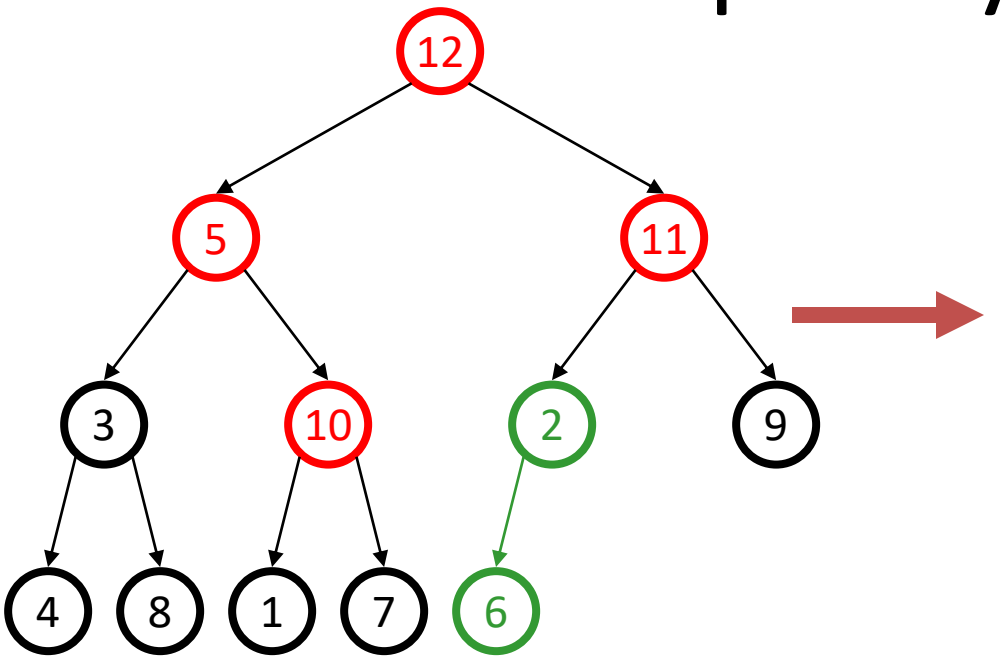
```
private void buildHeap() {
  for ( int i = currentSize/2; i > 0; i-- )
     sinkDown( i );
}
```

runtime:

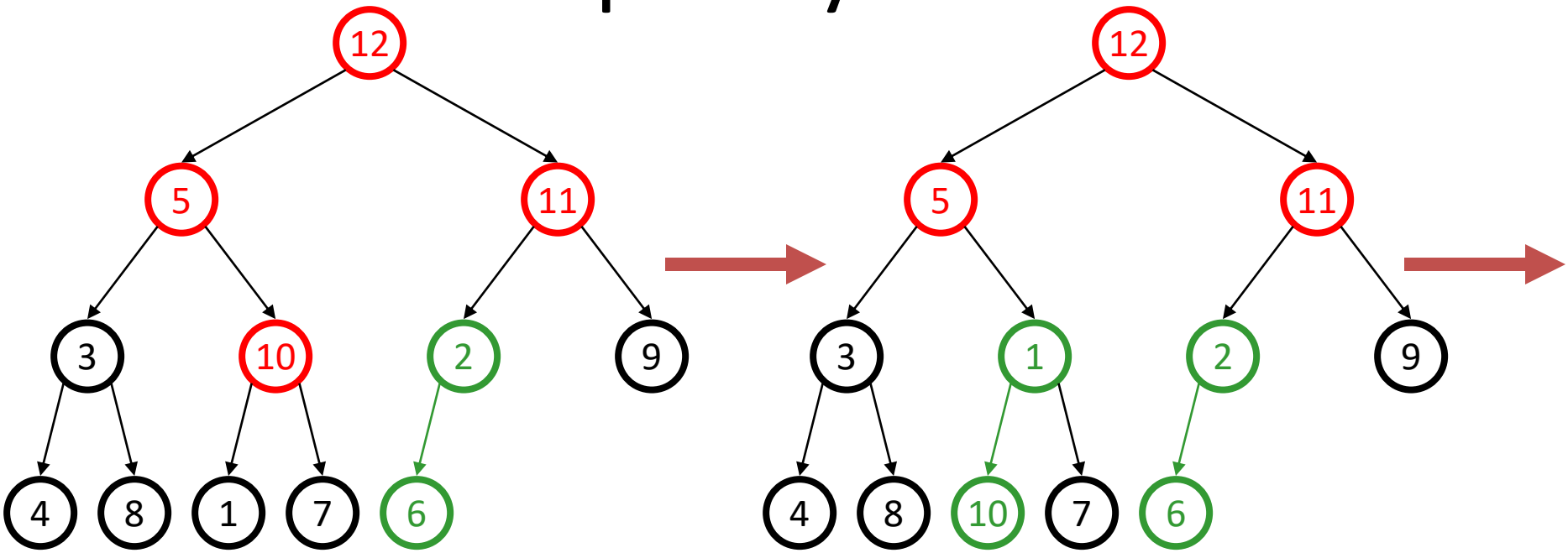
BuildHeap: Floyd's Method



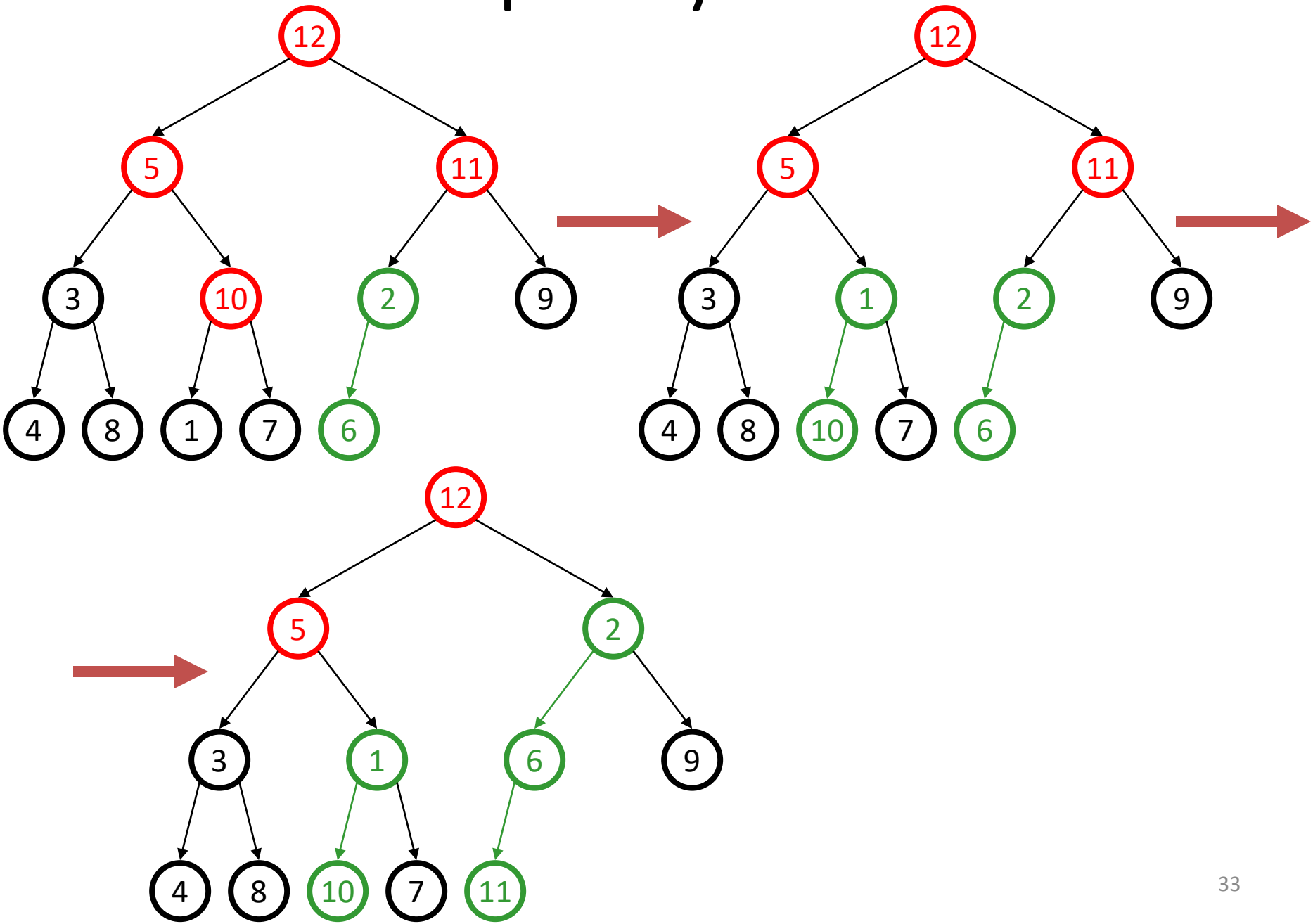
BuildHeap: Floyd's Method



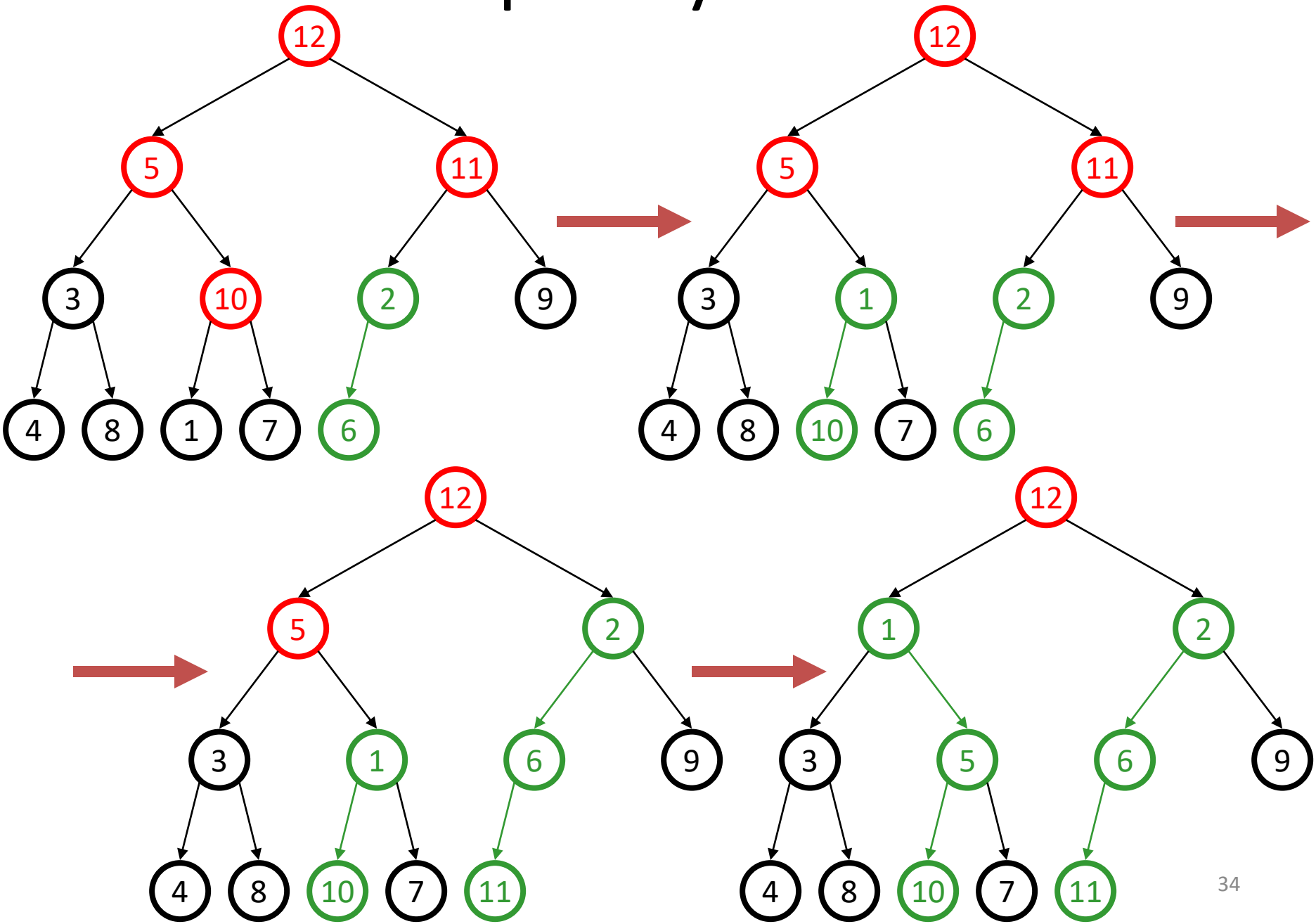
BuildHeap: Floyd's Method



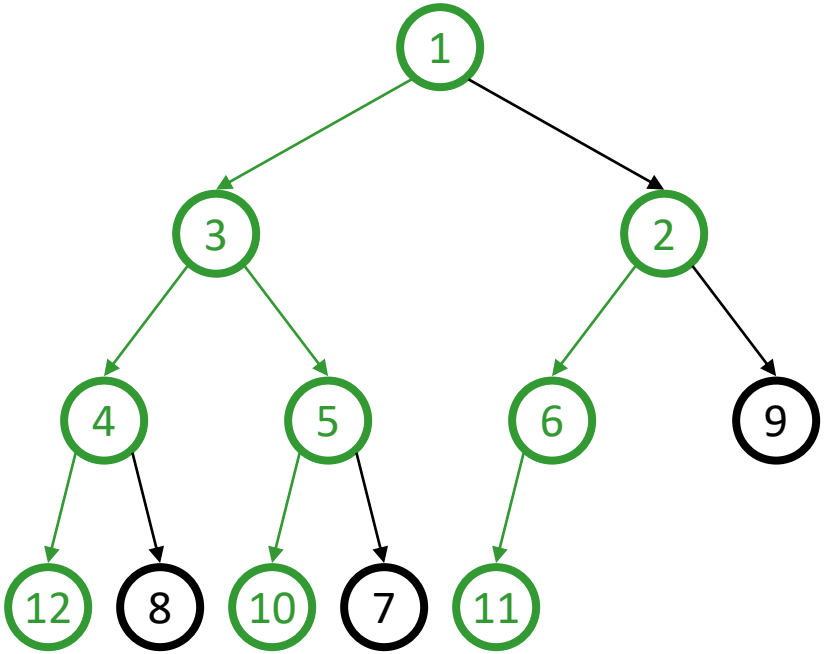
BuildHeap: Floyd's Method



BuildHeap: Floyd's Method



Finally…



runtime:

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

Sorting with a Priority Queue

```
void PriorityQueueSort(int a[], int size){
   PriorityQueue<int> pq(size);
   for(int i = 0; i < size; ++i)
     pq.insert(a[i]);
   for(int i = size-1; i >= 0; --i)
     a[i] = pq.remove();
}
```

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

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.

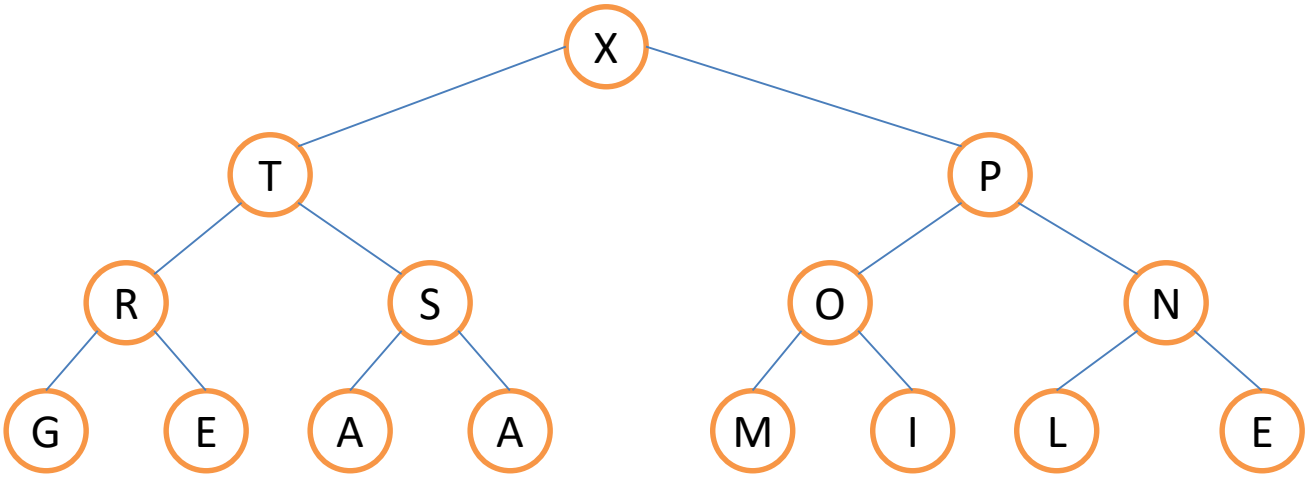
- Two step algorithm:
  1. Construct a heap within the passed array.
  2. While heap is not empty (size != 1)
     a. “extract” the largest item
     b. Heapify the remaining heap.

Example:
Step 1: Constructing the Heap

- Given the following unsorted array:

Example:
Step 1: Constructing the Heap

- Heap in tree representation after Step 1:



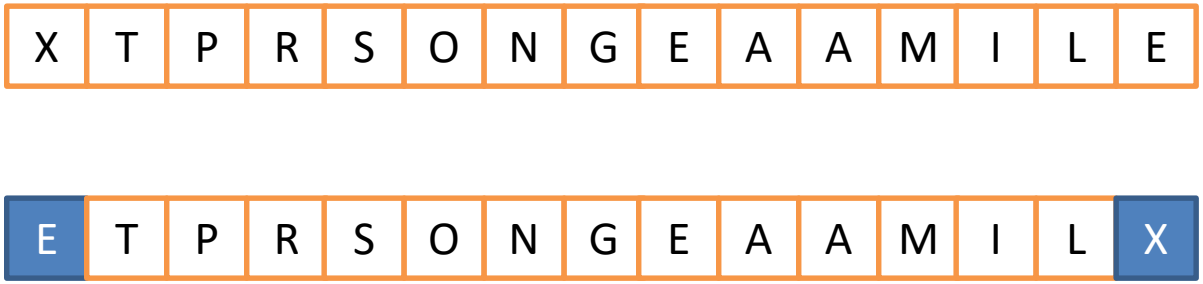
- Array now looks like this:

X	T	P	R	S	O	N	G	E	A	A	M	I	L	E

Example:
Step 2: Destroying the heap

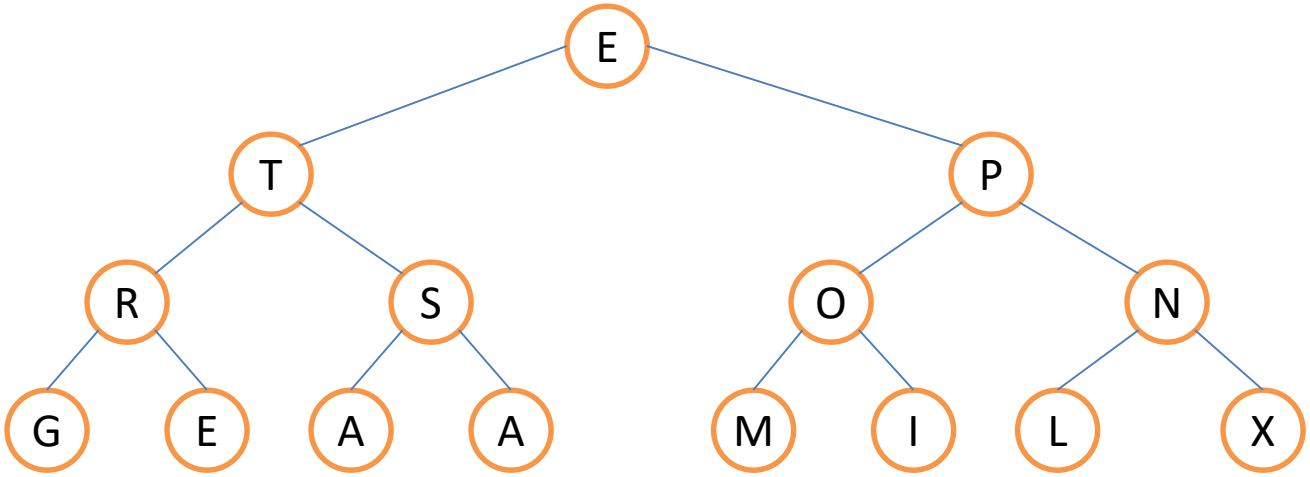
- At this point, we know that the largest element will be the first element in the array (in this case X).

- Because we want the array to be sorted in ascending order (smallest to greatest), we simply exchange the first element with the last element in the array



Example:
Step 2: Destroying the heap

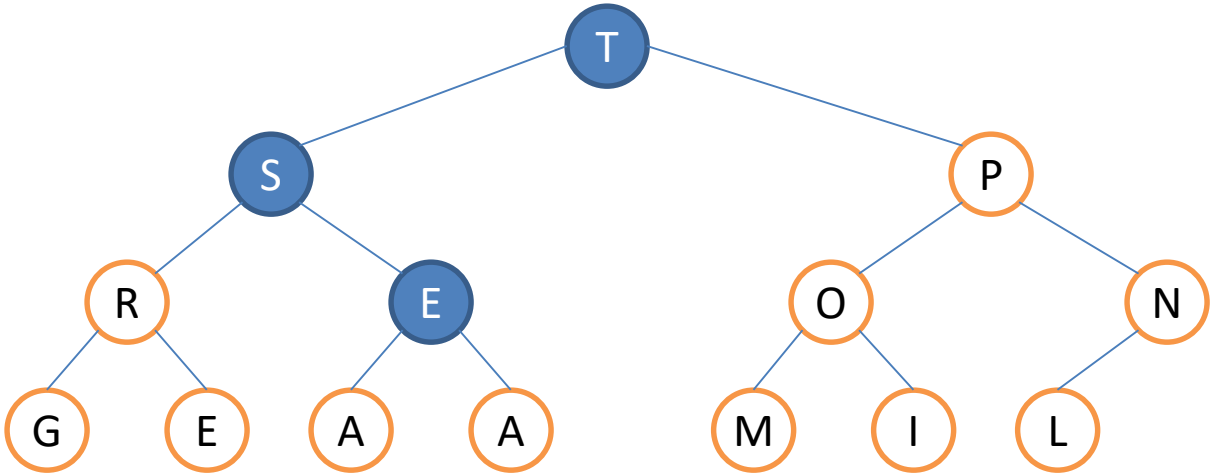
- The tree representation of the heap now looks like this:



- What we want at this point is to find the next largest element, so we can place it at its correct location:
- Simply heapify the resulting heap!
- Ignore the last element. Note that ignoring the last element doesn't violate the completeness of the tree.

Example:
Step 2: Destroying the heap

- Heapifying the heap (without X) using the top to bottom method:



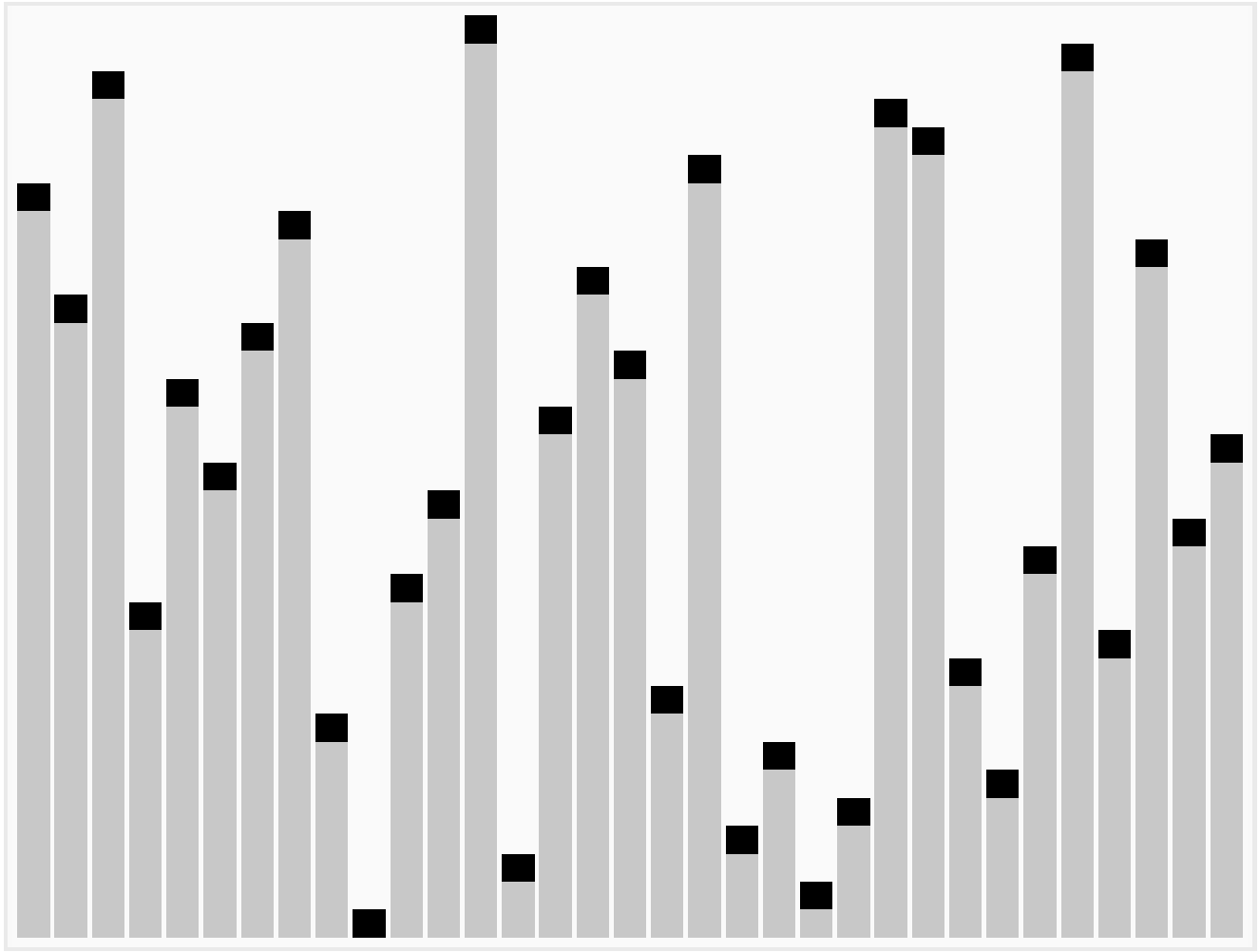
- Array now looks like this:

T	S	P	R	E	O	N	G	E	A	A	M	I	L	X

Example:
Step 2: Destroying the heap

- The resulting array after the heap has been "destroyed":

HeapSort



HeapSort

- HeapSort doesn’t need additional space.
- Complexity?