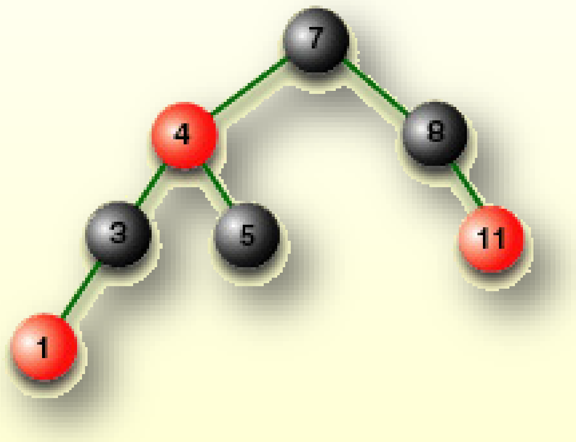
CS161:
Design and Analysis of Algorithms



Lecture 6
Leonidas Guibas

Outline

- Review of last lecture: Order statistics and (randomized/deterministic) selection

- Heaps and HeapSort
  - The heap data structure
  - The HeapSort algorithm
  - Priority queues

Slides modified from
- http://www.cs.virginia.edu/~luebke/cs332/

Order Statistics

- *i^{th} order statistic:* i^{th} smallest element of a set of n elements.
- Minimum: 1^{st} order statistic.
- Maximum: n^{th} order statistic.
- Median: $(n/2)^{th}$ order statistic -- "half-way point" of the set.

Randomized D&C Selection

RAND-SELECT(A, p, q, i) $\quad\triangleright$ ith smallest of $A[p\,..\,q]$

if $p = q$ & $i > 1$ **then** error!

$r \leftarrow$ RAND-PARTITION(A, p, q)

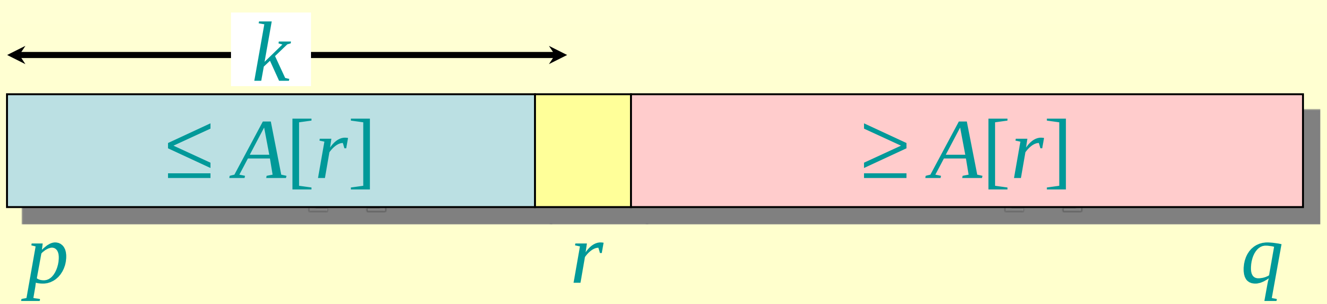
$k \leftarrow r - p + 1$ $\quad\triangleright$ $k = \text{rank}(A[r])$

if $i = k$ **then return** $A[r]$

if $i < k$

then return RAND-SELECT$(A, p, r - 1, i)$

else return RAND-SELECT$(A, r + 1, q, i - k)$



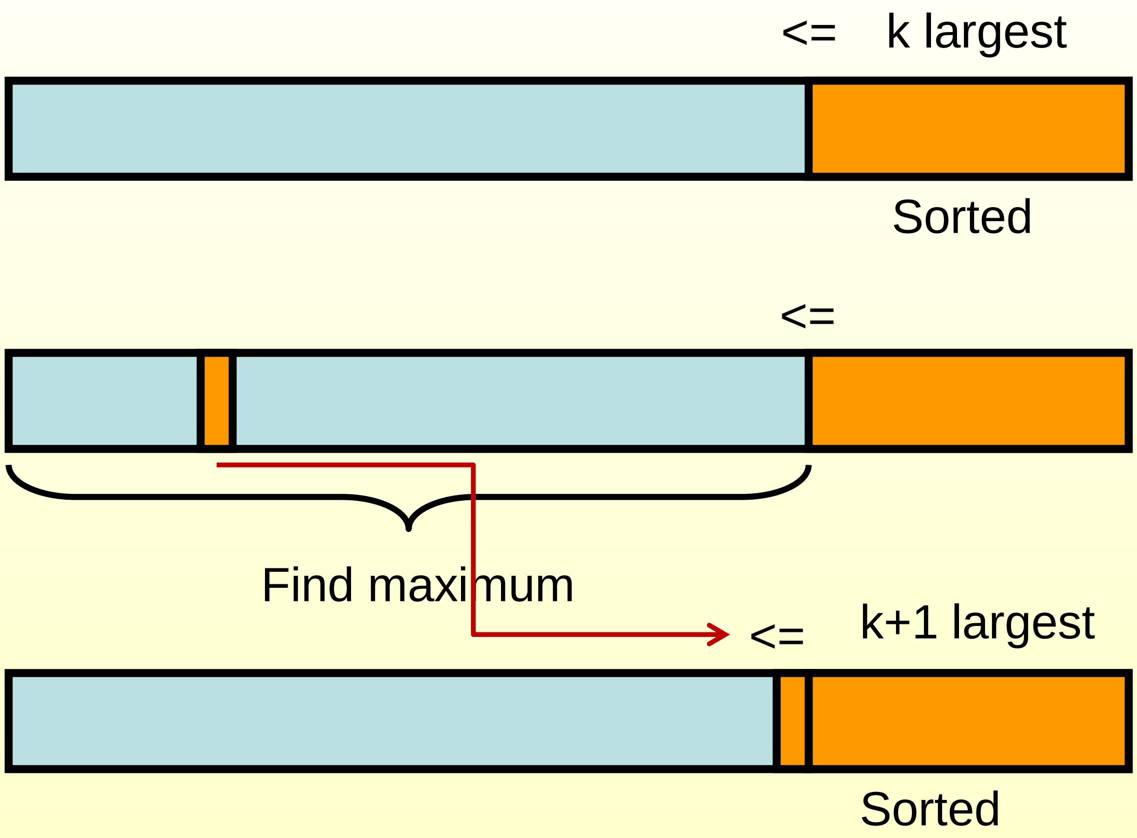
Worst-Case Linear-Time Selection

SELECT(i, n)

1. Divide the n elements into groups of 5. Find the median of each 5-element group by brute force.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
3. Partition around the pivot x. Let $k = \text{rank}(x)$.
4. **if** $i = k$ **then return** x
   elseif $i < k$
   then recursively SELECT the i-th smallest element in the lower part
   else recursively SELECT the $(i–k)$-th smallest element in the upper part

Same as RAND-SELECT

SelectionSort



SelectionSort

```
SelectionSort(A[1..n])
  for (i = n; i > 0; i--)
      index = max_element(A[1..i])
      swap(A[i], A[index]);
  end
```

What's the time complexity?

If max_element takes Θ(n),
selection sort takes $\sum_{i=1}^{n} i = \Theta(n^2)$

HeapSort

- Another $\Theta(n \log n)$ sorting algorithm
- In practice, QuickSort wins
- However, the heap data structure and its variants are very useful for many other algorithms (beyond sorting)

Heap

- A heap is a data structure that allows us to quickly retrieve the largest (or smallest) element from a set

- It takes time $\Theta(n)$ to build the heap

- If we need to retrieve largest element, second largest, third largest…, in the long run the time taken for building heaps will be rewarded

Idea of HeapSort

HeapSort(A[1..n])

Build a “heap” from A

For $i = n$ down to 1

Retrieve largest element from heap

Put element at end of A

Reduce heap size by one

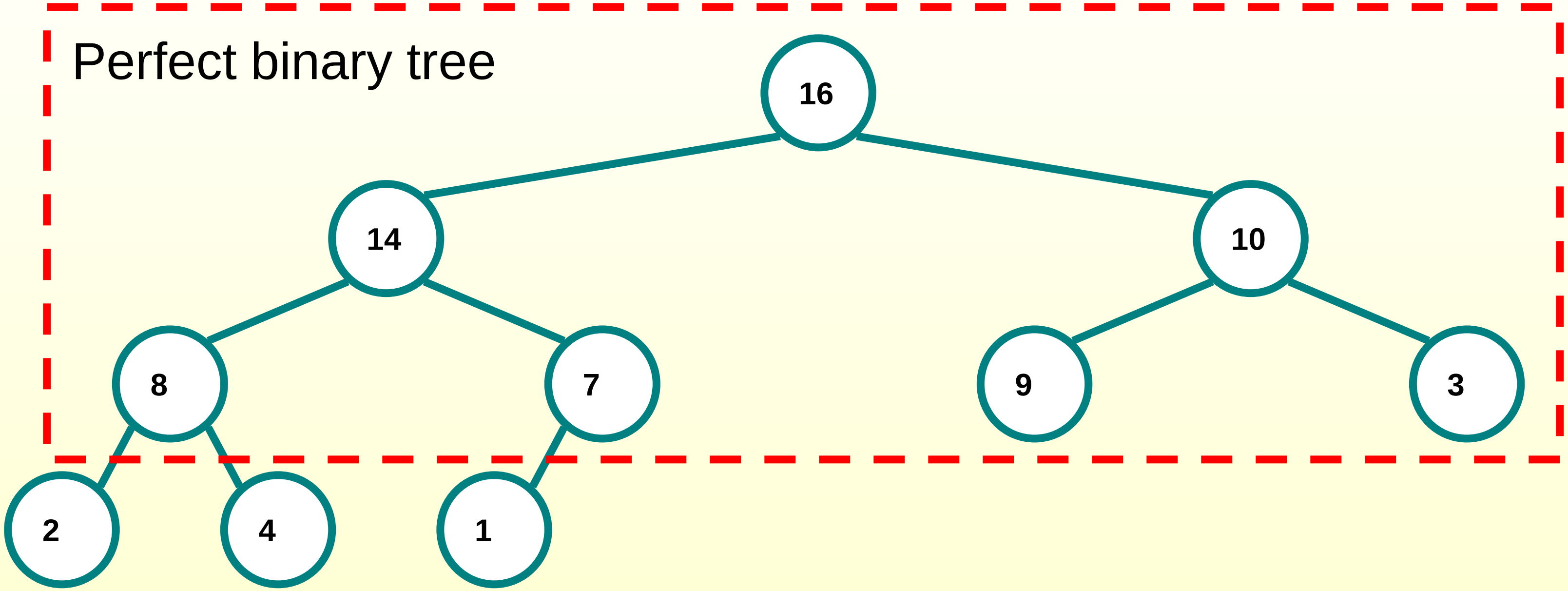
end

Key:

1. Build a heap in linear time
2. Retrieve largest element (and make it ready for next retrieval) in O(log n) time

Heaps

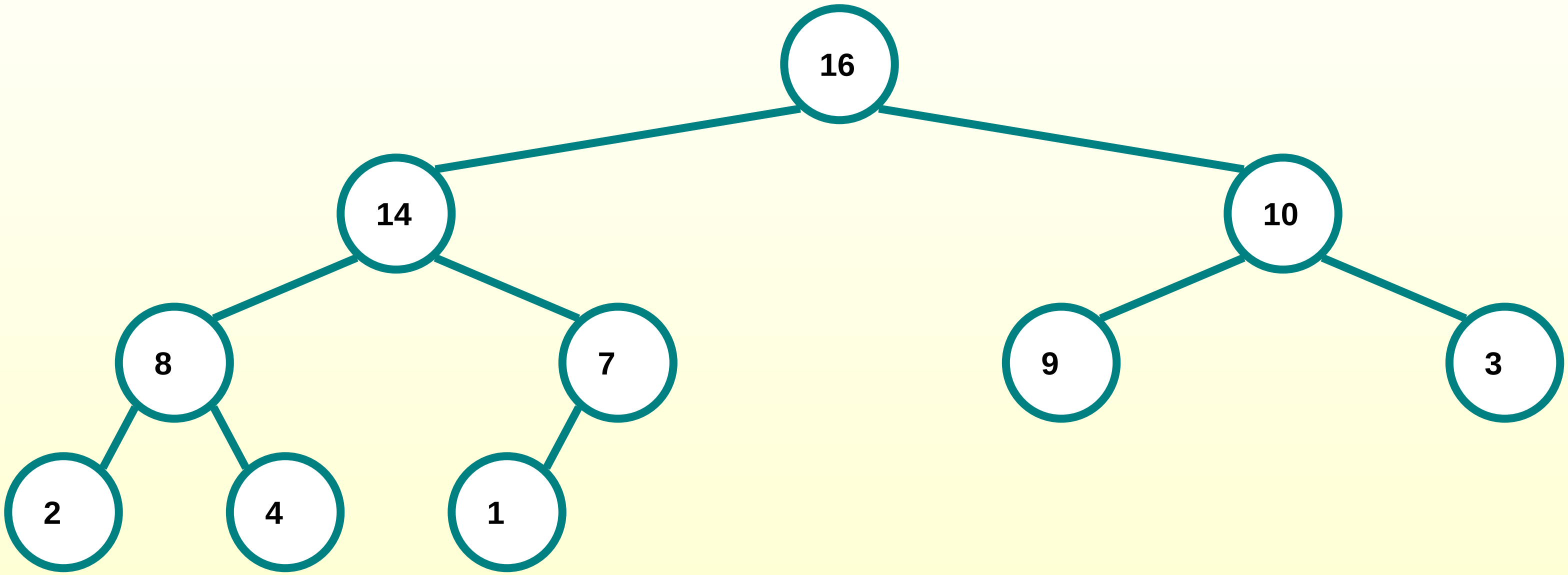
- A *heap* can be seen as a complete binary tree:



*A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled and, in that level, all nodes are as far to the left as possible.*

Key Heap Property

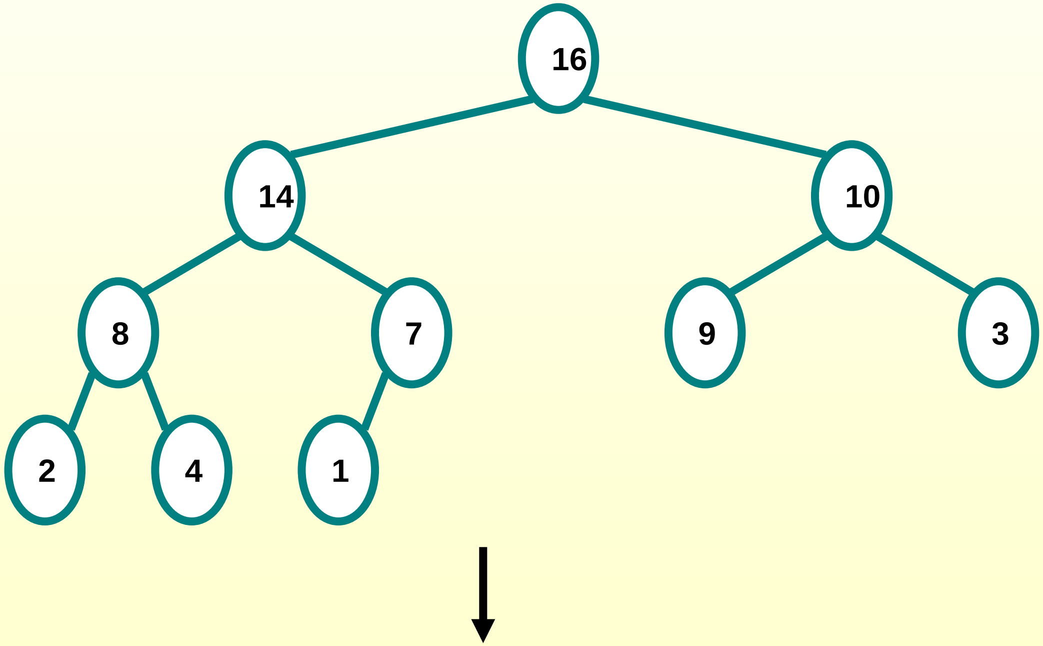
- A *heap* can be seen as a complete binary tree



- A tree in which every node holds a key larger than or equal to those of its children -- heap condition

Heaps

- In practice, heaps are usually realized / implemented as arrays:



16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

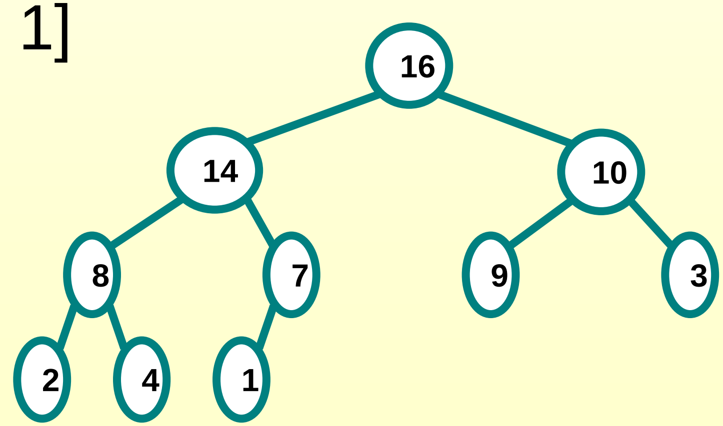
Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2] (note: integer divide, or floor)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

=



Referencing Heap Elements

- So…

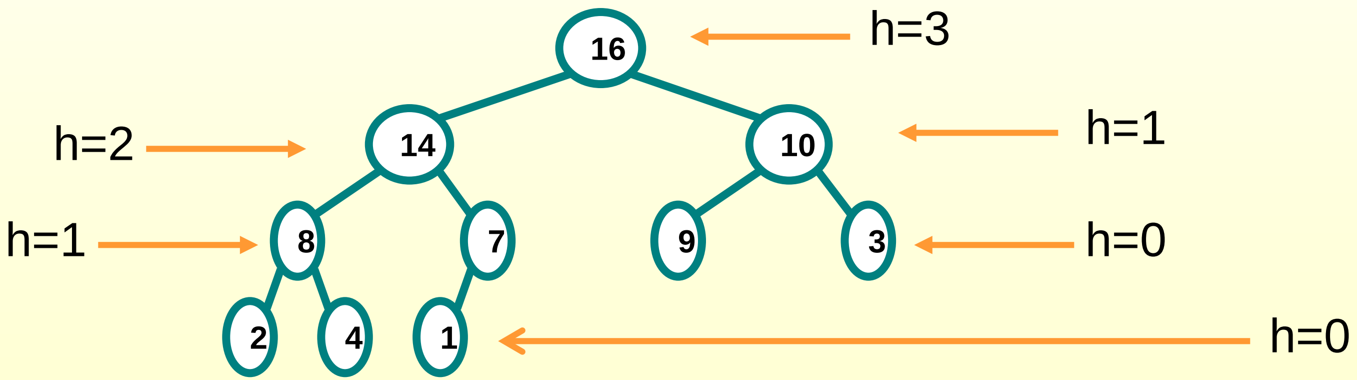
```
Parent(i)
  {return ⌊i/2⌋;}

Left(i)
  {return 2*i;}

right(i)
  {return 2*i + 1;}
```

Heap Height

- Definitions:
  - The *height of a node* in the tree = the number of edges on the longest downward path to a leaf
  - The *height of a tree* = the height of its root



- *What is the height of an n-element heap? Why?*
- $\lfloor \log_2(n) \rfloor$. Basic heap operations take at most time proportional to the height of the heap

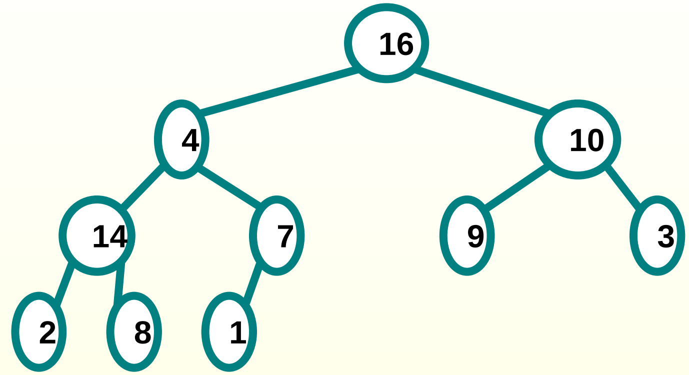
The Heap Property

- Heaps satisfy the *heap property*:
  $A[Parent(i)] \geq A[i]$ for all nodes $i > 1$
  - In other words, the value of a node is at most the value of its parent
  - The value of a node should be greater than or equal to both its left and right children
    - and, inductively, to that all of its descendants
  - *Where is the largest element in a heap stored?*

Are They Heaps?

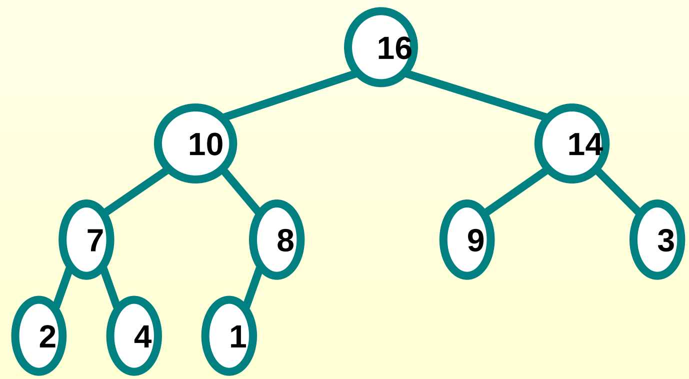
16	*4*	*10*	*14*	*7*	*9*	*3*	*2*	*8*	*1*

=



16	*10*	*14*	*7*	*8*	*9*	*3*	*2*	*4*	*1*

=



Violation to heap property: a node has value less than one of its children
How to find that?
How to resolve that?

Heap Operations: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node i in the heap with children l and r
  - Given: two subtrees rooted at l and r, assumed to be heaps
  - Problem: The subtree rooted at i may violate the heap property
  - Action: let the value of the parent node "sift down" so subtree at i satisfies the heap property
    - Fix up the relationship between i, l, and r recursively

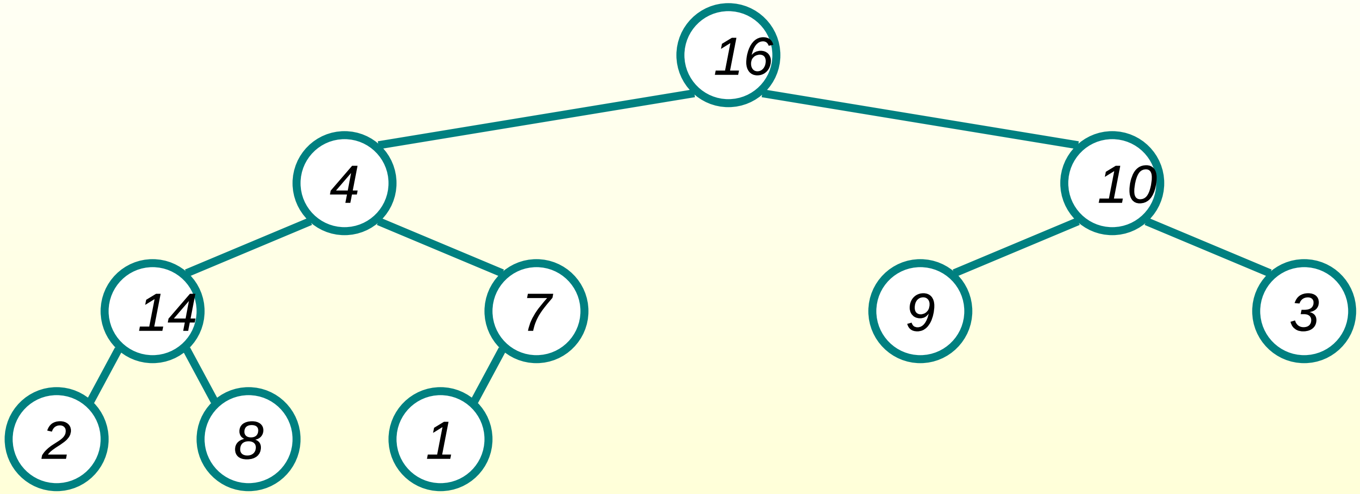
Heap Operations: Heapify()

```
Heapify(A, i)
{ // precondition: subtrees rooted at l and r are heaps
  l = Left(i); r = Right(i);
  if (l <= heap_size(A) && A[l] > A[i])
      largest = l;
  else
      largest = i;
  if (r <= heap_size(A) && A[r] > A[largest])
      largest = r;
  if (largest != i) {
      Swap(A, i, largest);
      Heapify(A, largest);
  }
} // postcondition: subtree rooted at i is a heap
```

Among A[l], A[i], A[r], which one is largest?

If violation, fix it.

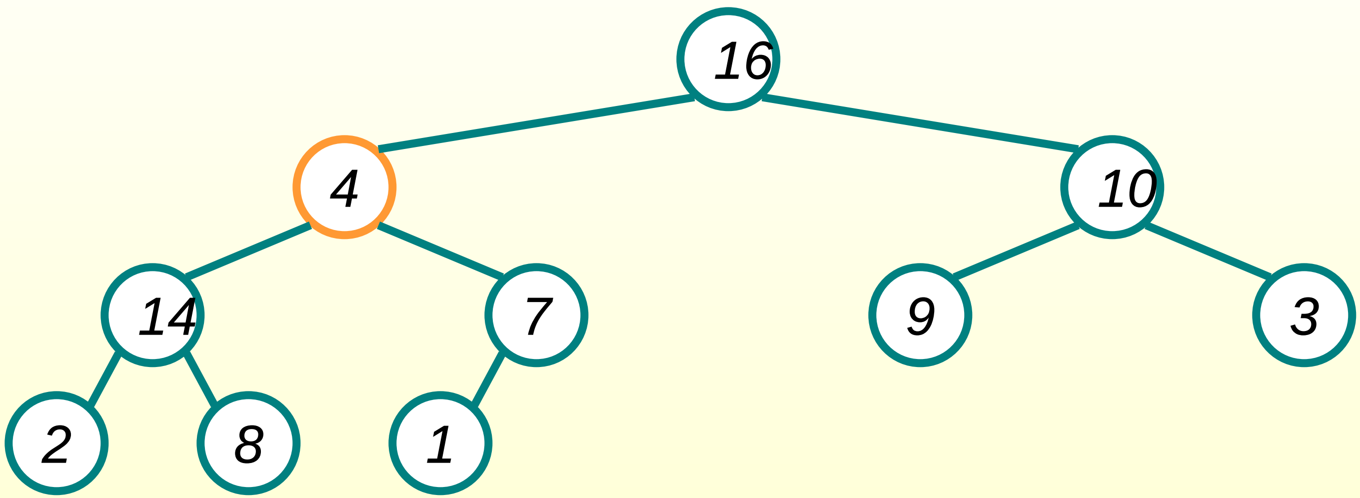
Heapify() Example



A =

16	4	10	14	7	9	3	2	8	1

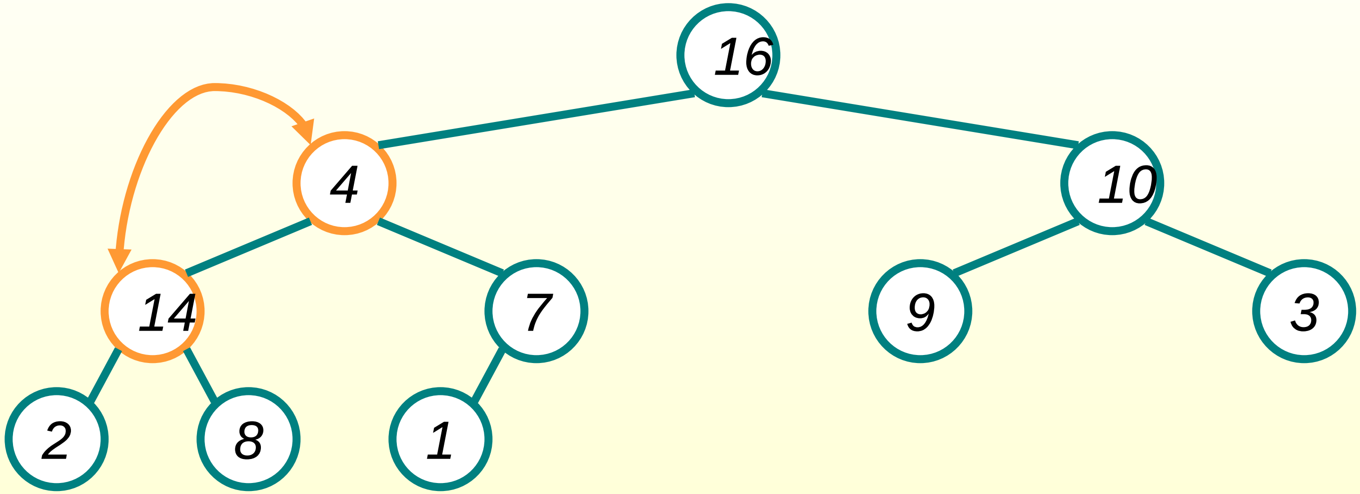
Heapify() Example



A =

16	*4*	*10*	*14*	*7*	*9*	*3*	*2*	*8*	*1*

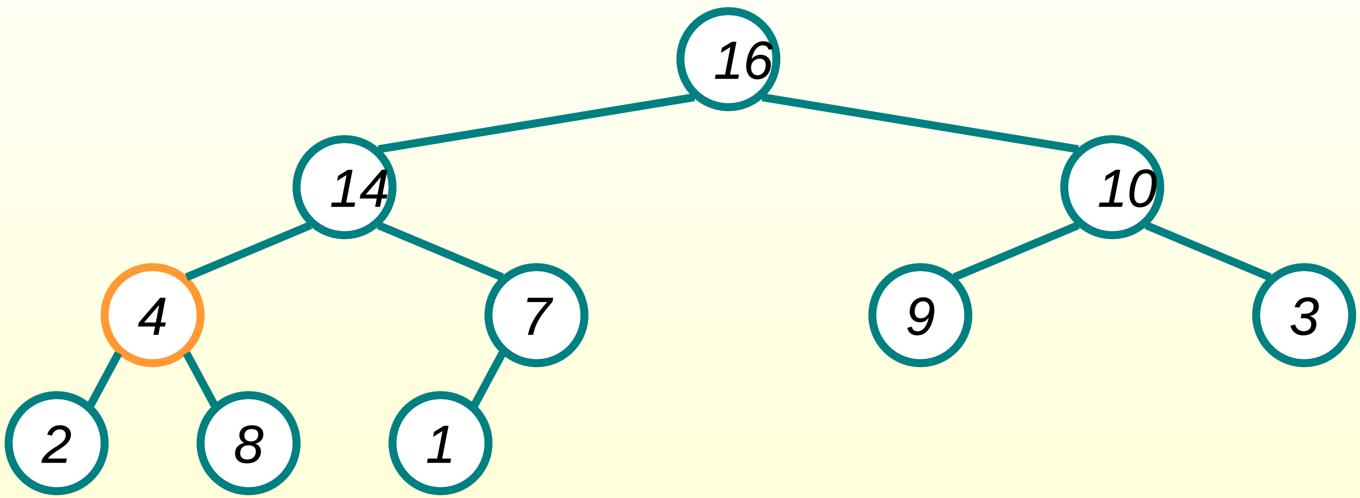
Heapify() Example



A =

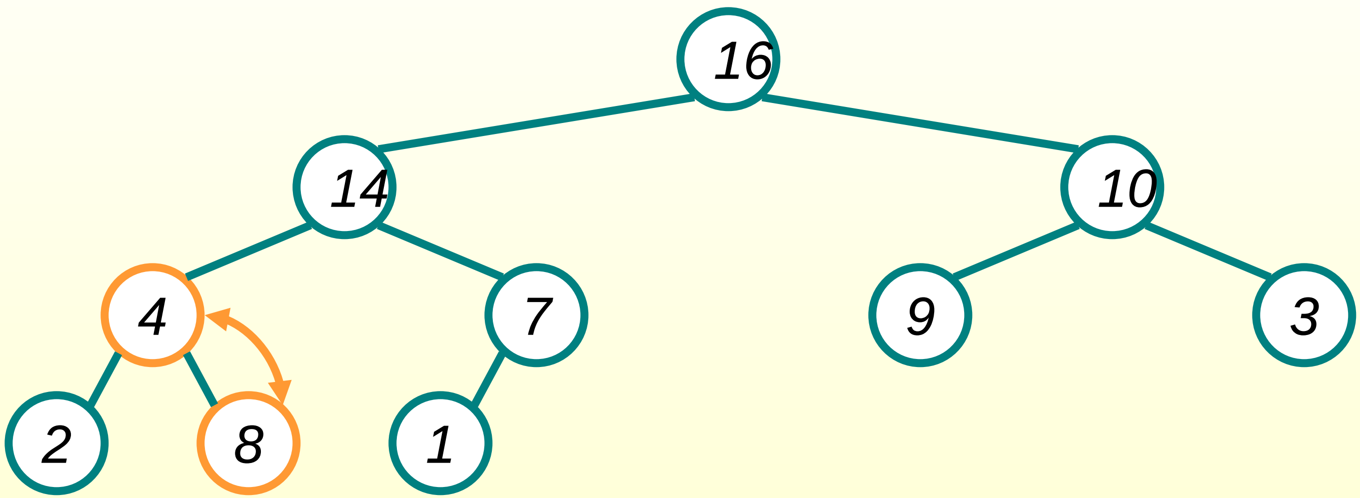
16	4	10	14	7	9	3	2	8	1

Heapify() Example



A =									
16	14	10	4	7	9	3	2	8	1

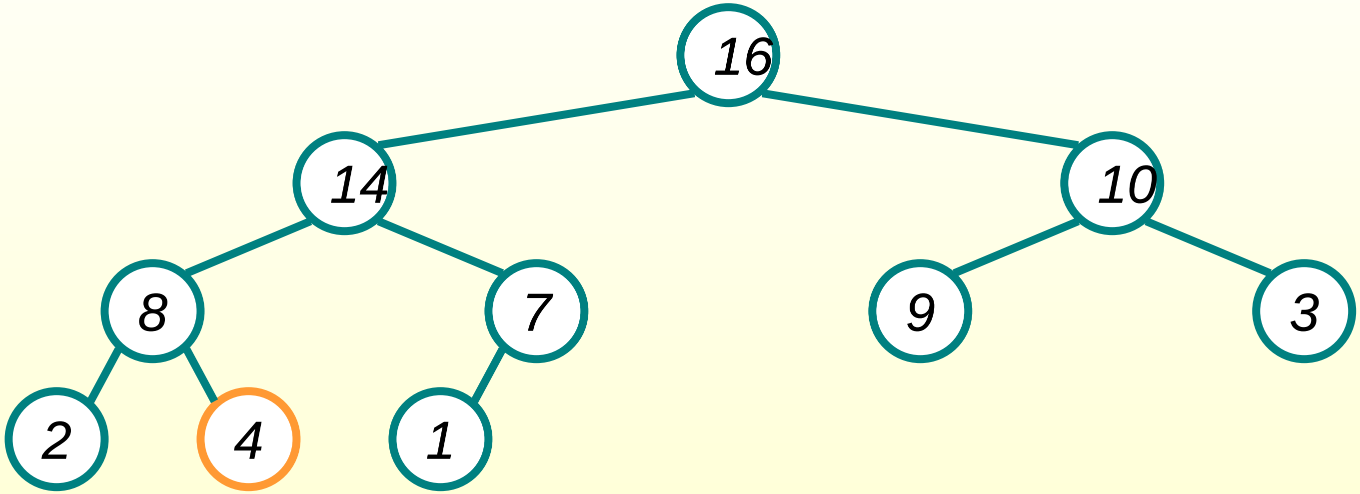
Heapify() Example



$A =$

16	14	10	4	7	9	3	2	8	1

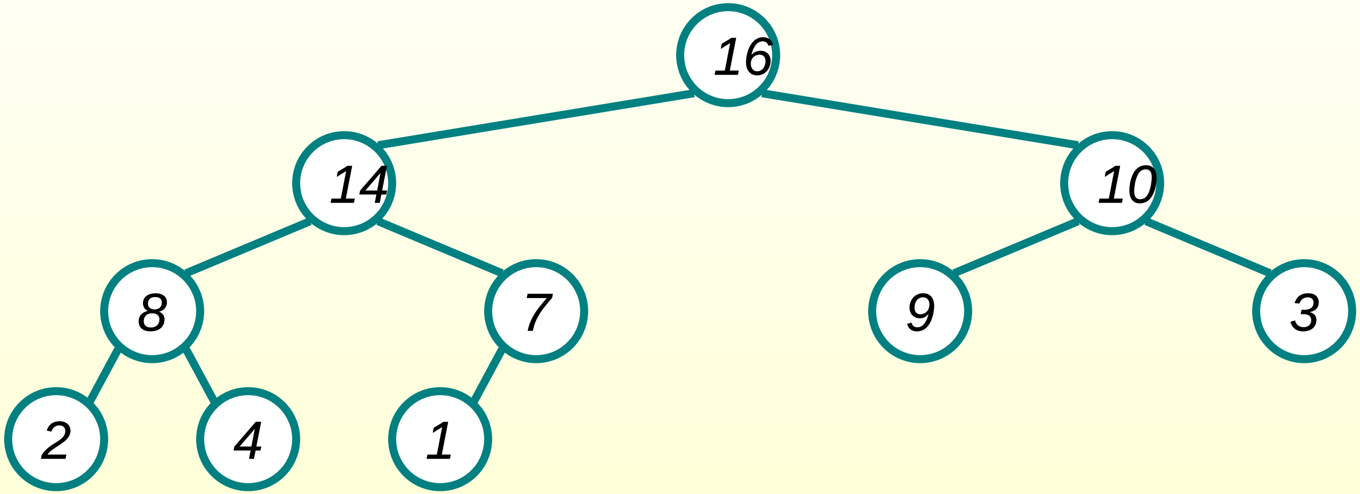
Heapify() Example



A =

16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	*16*	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

Analyzing Heapify(): Informal

- *Aside from the recursive call, what is the running time of* ***`Heapify()`****?*
- *How many times can* ***`Heapify()`*** *recursively call itself?*
- *What is the worst-case running time of* ***`Heapify()`*** *on a heap of size n?*

Analyzing Heapify(): Formal

- Fixing up relationships between *i*, *l*, and *r* takes $\Theta(1)$ time
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
- Answer: $2n/3$ (worst case: bottom row 1/2 full)
- So time taken by **`Heapify()`** is given by

$$T(n) \leq T(2n/3) + \Theta(1)$$

Analyzing Heapify(): Formal

- So we have

  $T(n) \le T(2n/3) + \Theta(1)$

- By case 2 of the Master Theorem,

  $T(n) = O(\lg n)$

- Thus, **`Heapify()`** takes logarithmic time

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

- Fact: for array of length *n*, all elements in range $A[\lfloor n/2 \rfloor + 1 .. n]$ are heaps (*Why?*)

Heap size	# leaves	# internal nodes
1	1	0
2	1	1
3	2	1
4	2	2
5	3	2

0 <= # leaves - # internal nodes <= 1

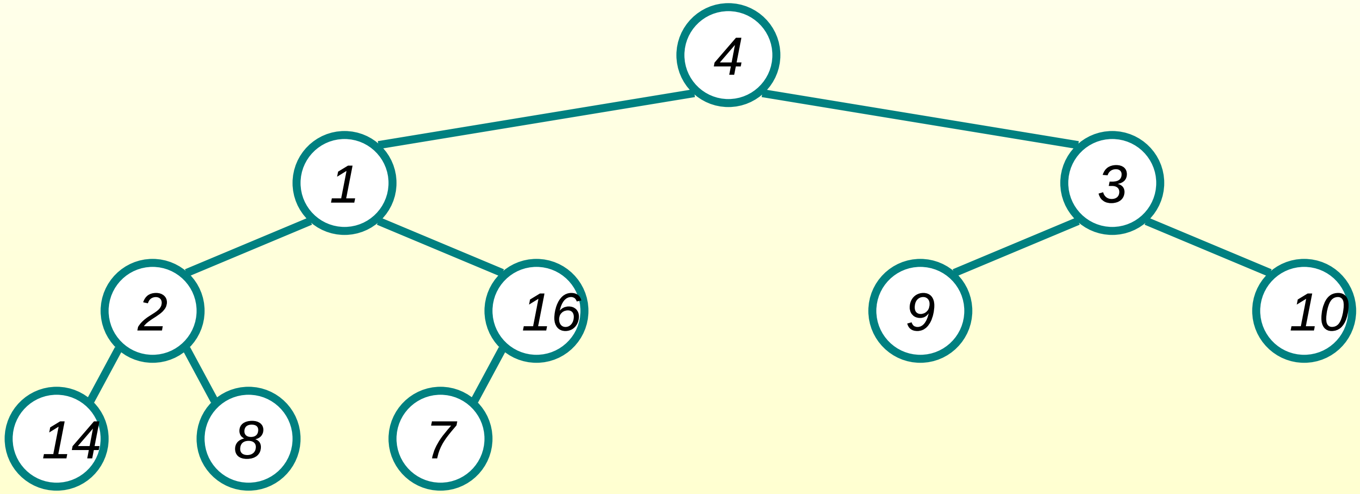
# of internal nodes = $\lfloor n/2 \rfloor$

BuildHeap()

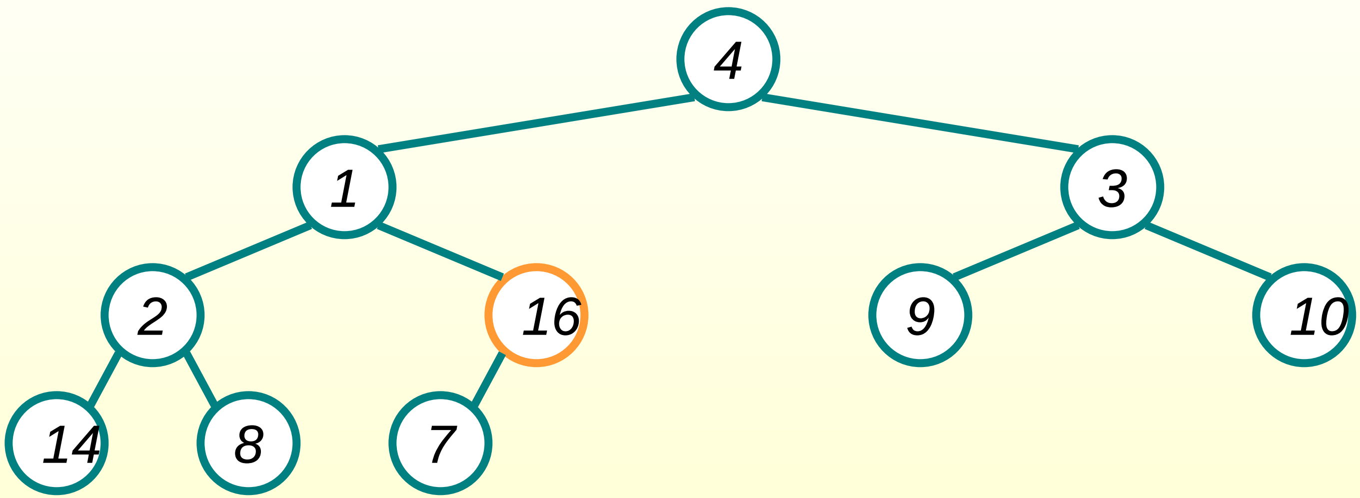
```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
      Heapify(A, i);
}
```

BuildHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}

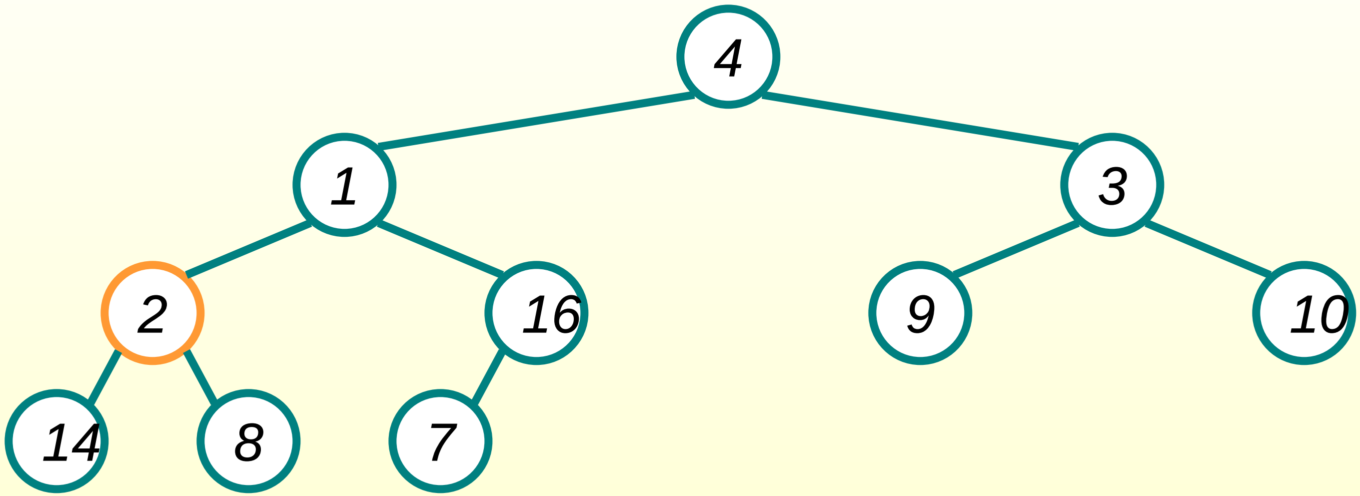


A =									
4	*1*	*3*	*2*	*16*	*9*	*10*	*14*	*8*	*7*

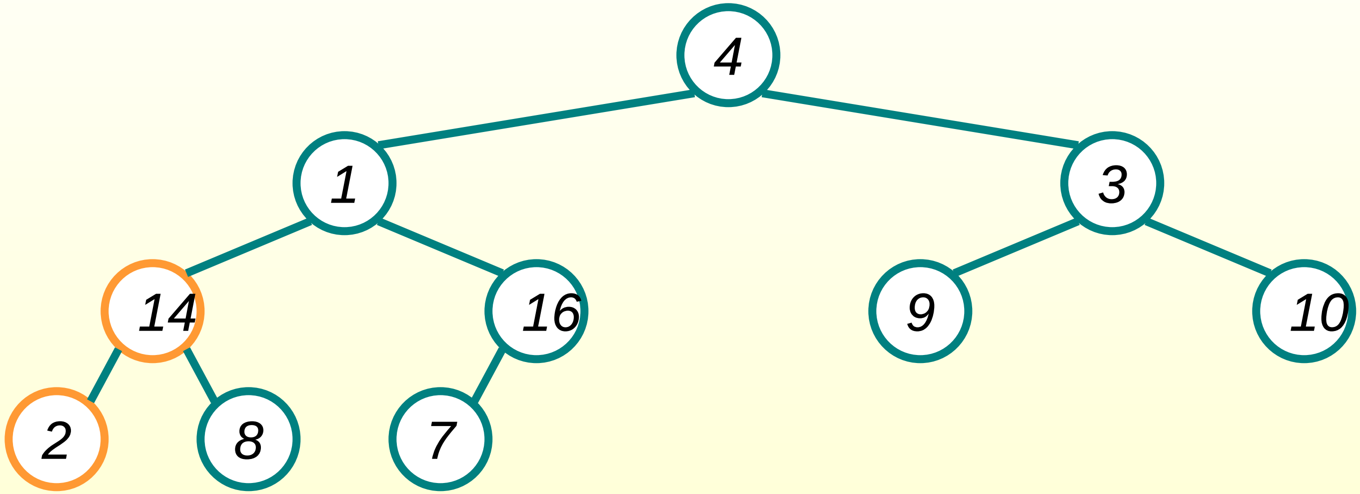


$A =$

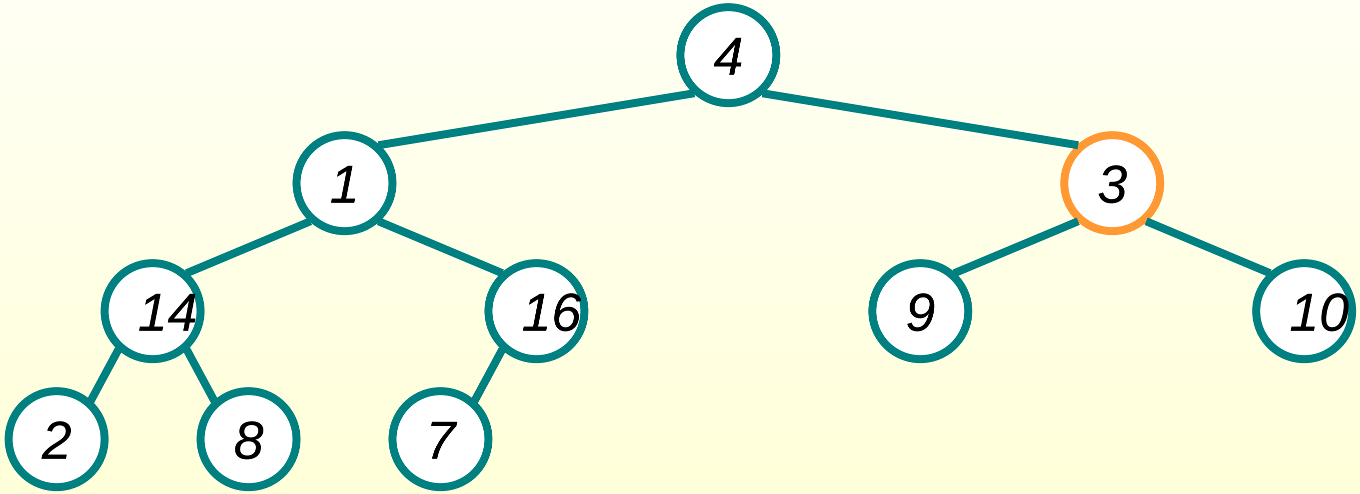
4	1	3	2	16	9	10	14	8	7



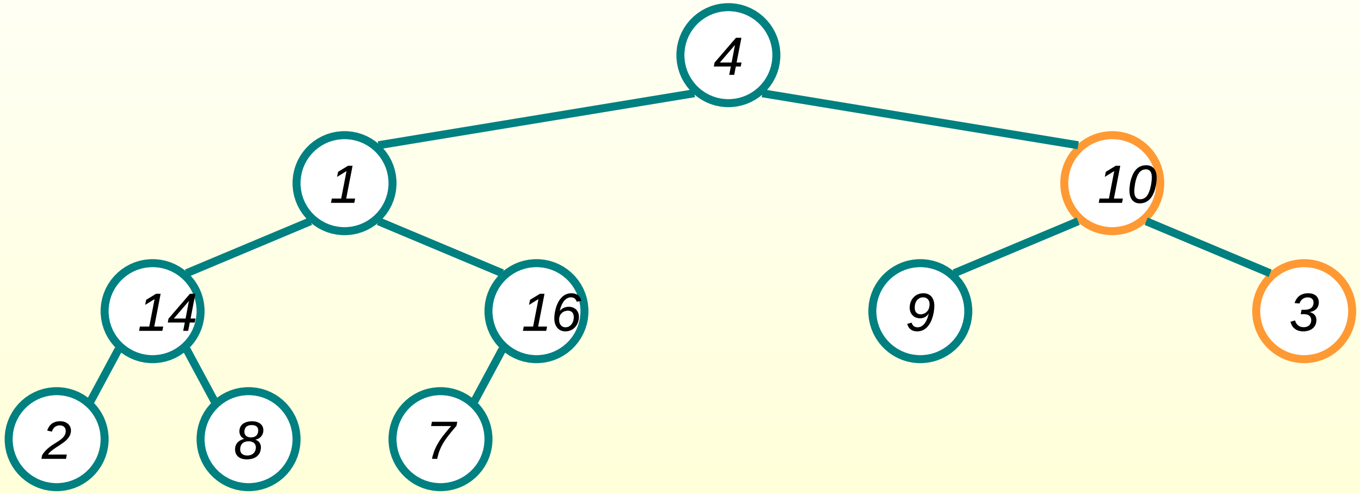
A =									
4	1	3	2	16	9	10	14	8	7



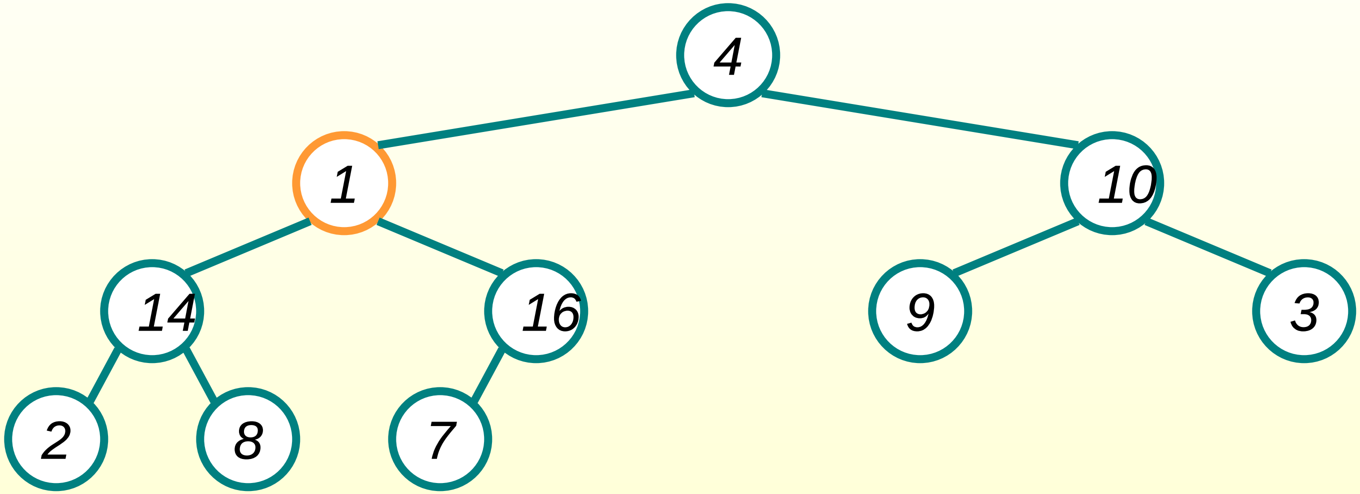
$A =$	4	1	3	14	16	9	10	2	8	7



A =									
4	1	3	14	16	9	10	2	8	7

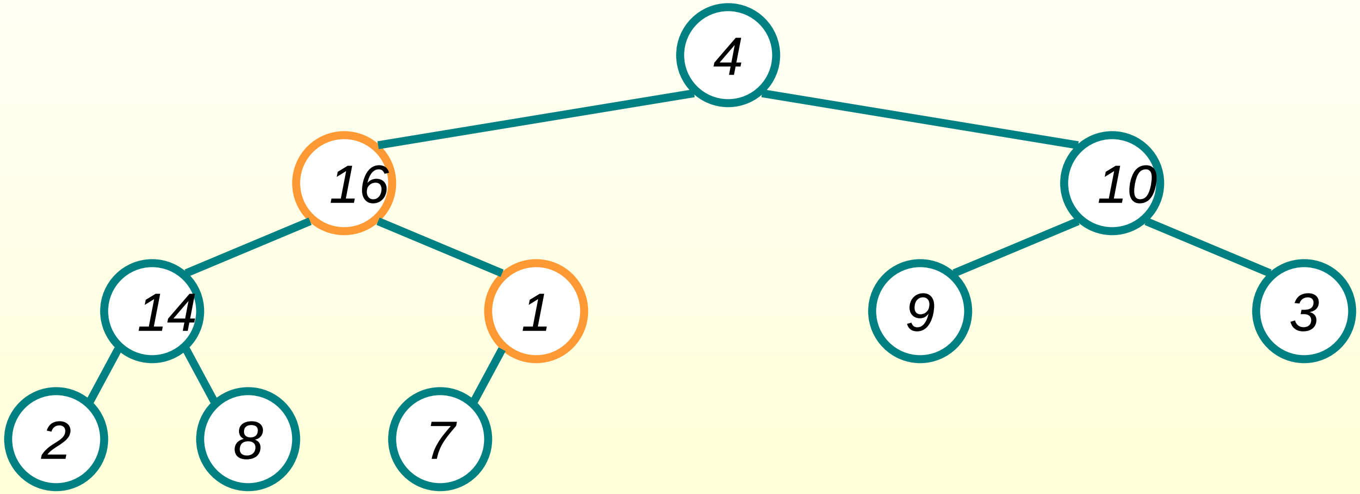


$A =$	4	1	10	14	16	9	3	2	8	7

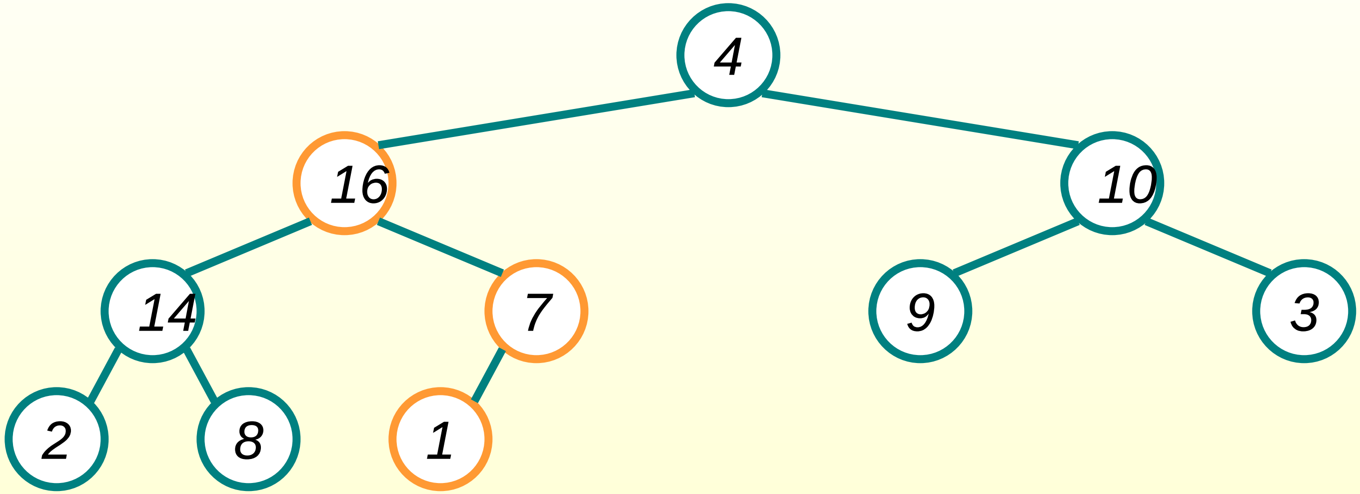


A =

4	1	10	14	16	9	3	2	8	7

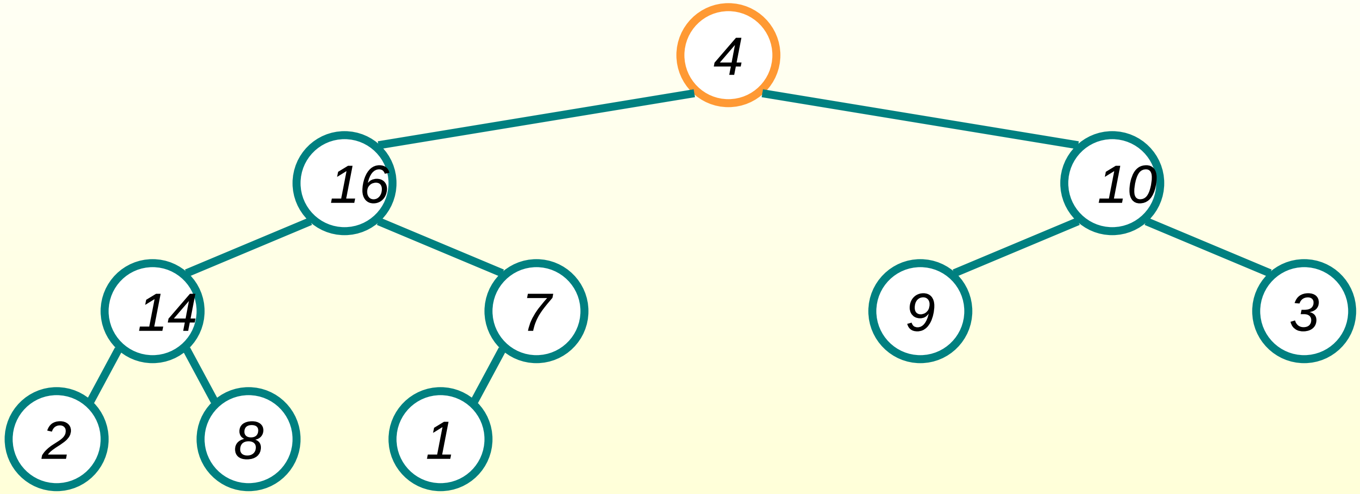


A = | 4 | 16 | 10 | 14 | 1 | 9 | 3 | 2 | 8 | 7 |

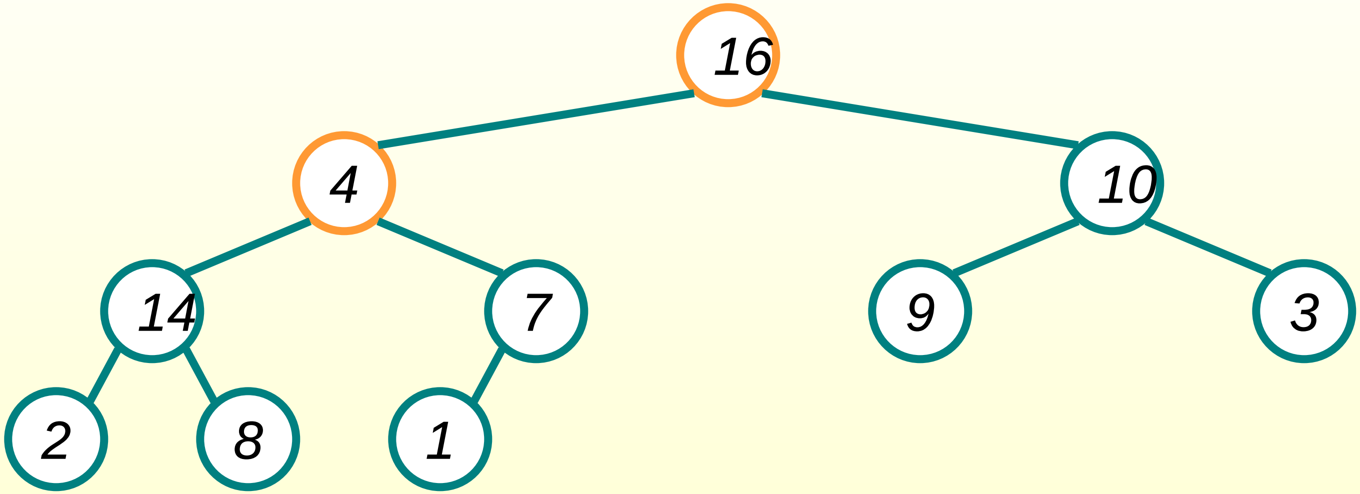


A =

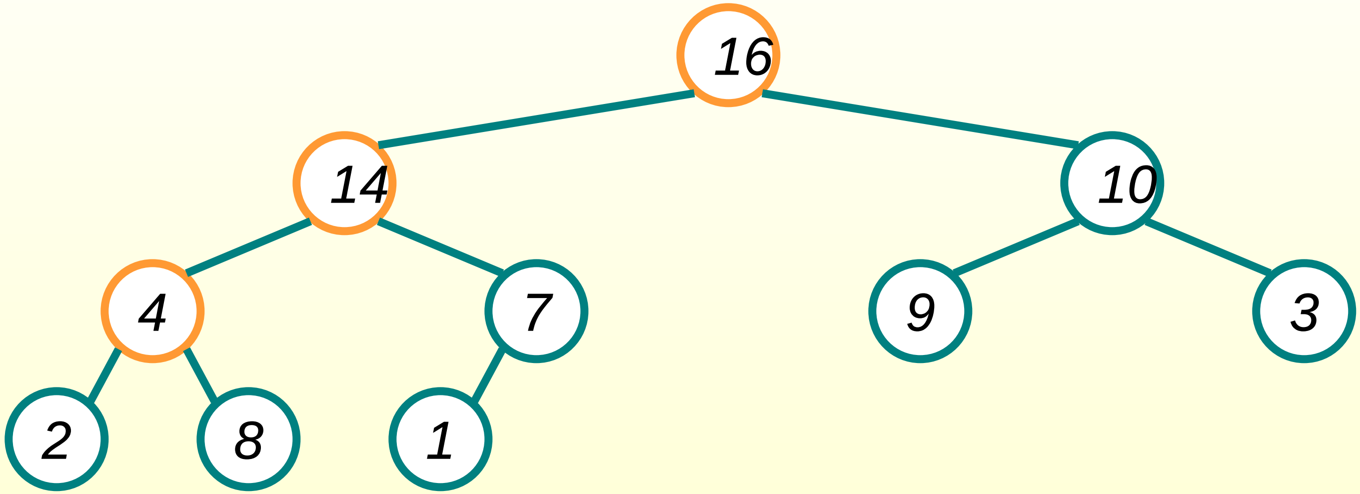
4	16	10	14	7	9	3	2	8	1



A =	4	16	10	14	7	9	3	2	8	1

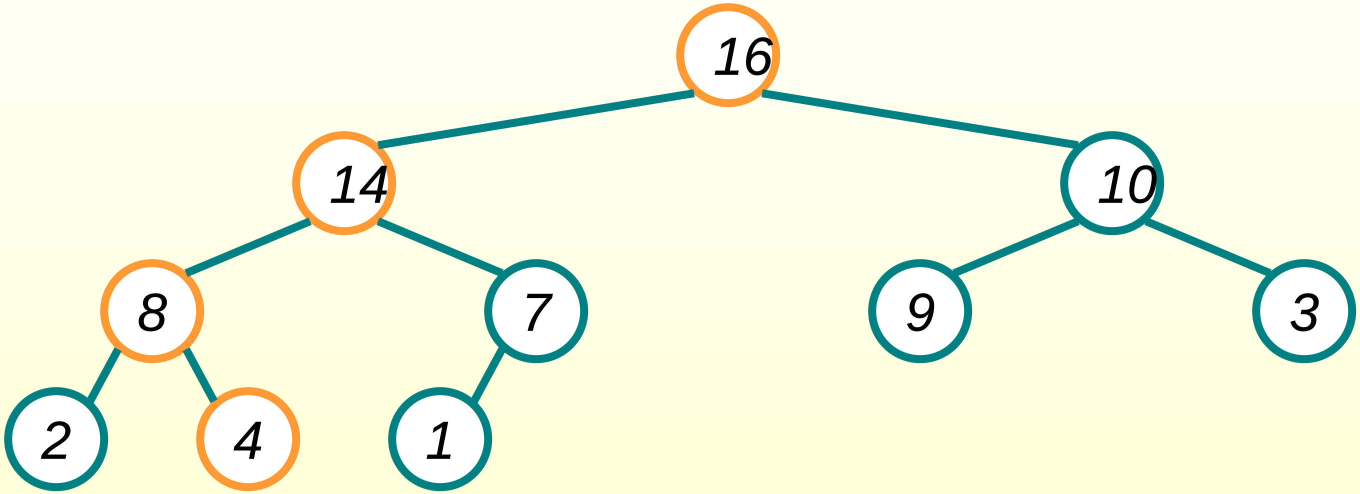


A = | 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |



A =

16	14	10	4	7	9	3	2	8	1



A = | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

Analyzing BuildHeap()

- Each call to **`Heapify()`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is O(*n*)
  - *How can this be? Is there a flaw in the above reasoning?*

Analyzing BuildHeap(): Tight

- To **`Heapify()`** a subtree takes O(*h*) time where *h* is the height of the subtree
  - *h* = O(lg *m*), m = # nodes in subtree
  - The height of most subtrees is small
- Fact: an *n*-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height *h* (*why?*)

$$T(n) \le \sum_{h=1}^{\log_2 n} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \le \sum_{h=1}^{\log_2 n} \frac{nh}{2^h} = n \sum_{h=1}^{\log_2 n} \frac{h}{2^h} \le 2n$$

- Therefore T(n) = O(n)

- Fact: an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h (*why?*)
- $\lceil n/2 \rceil$ leaf nodes (h = 0): $f(0) = \lceil n/2 \rceil$
- $f(1) \le (\lceil n/2 \rceil + 1)/2 = \lceil n/4 \rceil$
- The above fact can be proved using induction
- Assume $f(h) \le \lceil n/2^{h+1} \rceil$
- $f(h+1) \le (f(h)+1)/2 \le \lceil n/2^{h+2} \rceil$

$$T(n) \leq \sum_{h=1}^{\log_2 n} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \leq \sum_{h=1}^{\log_2 n} \frac{nh}{2^h} = n \sum_{h=1}^{\log_2 n} \frac{h}{2^h} \leq 2n$$

$$\sum_{h=1}^{\log_2 n} \frac{h}{2^h} \leq \sum_{h=1}^{\infty} \frac{h}{2^h} = 2$$ Appendix A.8

$$T(n) \leq 2n$$

Heapsort

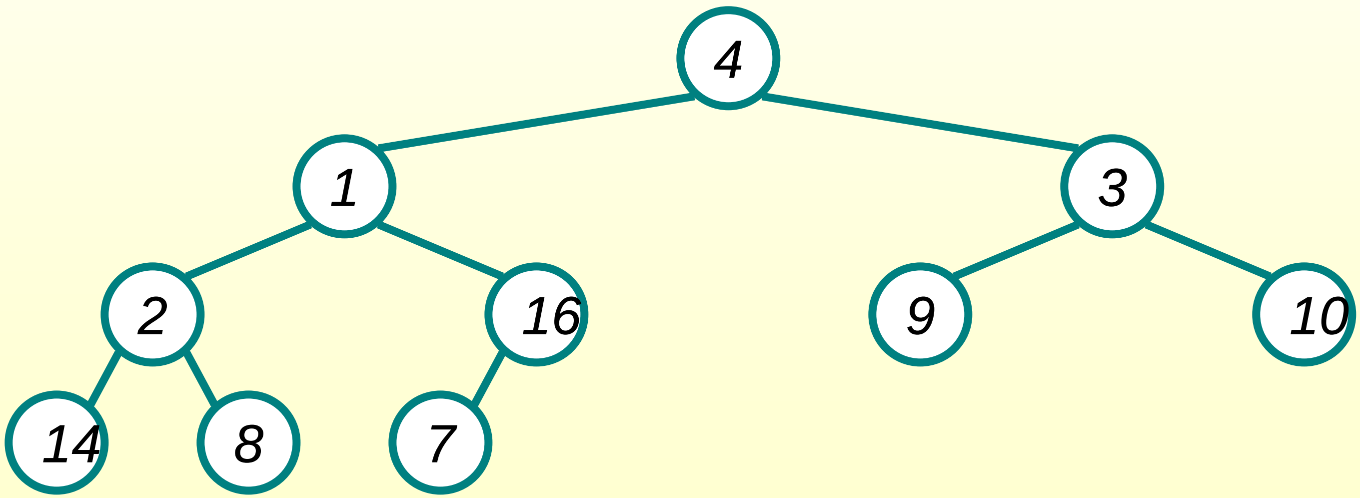
- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

```
Heapsort(A)
{
     BuildHeap(A);
     for (i = length(A) downto 2)
     {
          Swap(A[1], A[i]);
          heap_size(A) -= 1;
          Heapify(A, 1);
     }
}
```

Heapsort Example

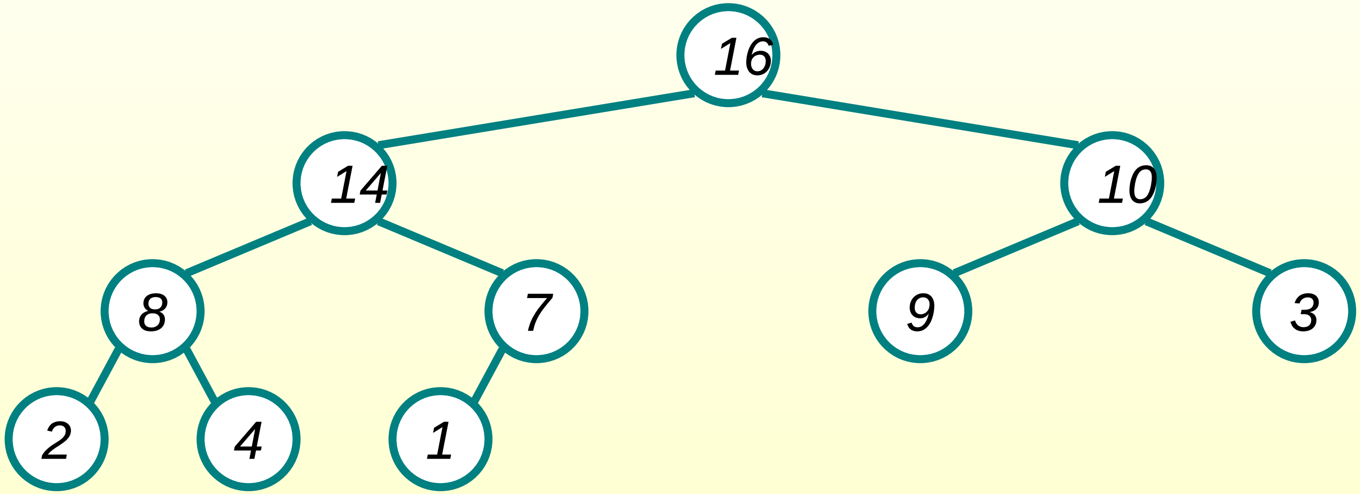
- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



A =										
	4	1	3	2	16	9	10	14	8	7

Heapsort Example

- First: build a heap

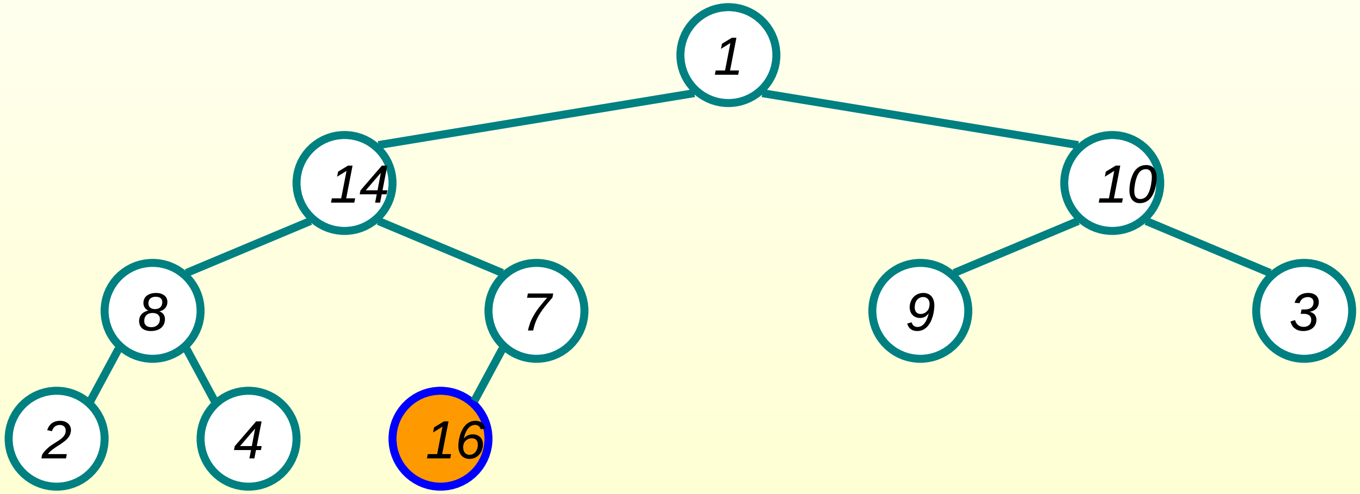


A =

16	14	10	8	7	9	3	2	4	1

Heapsort Example

- Swap last and first

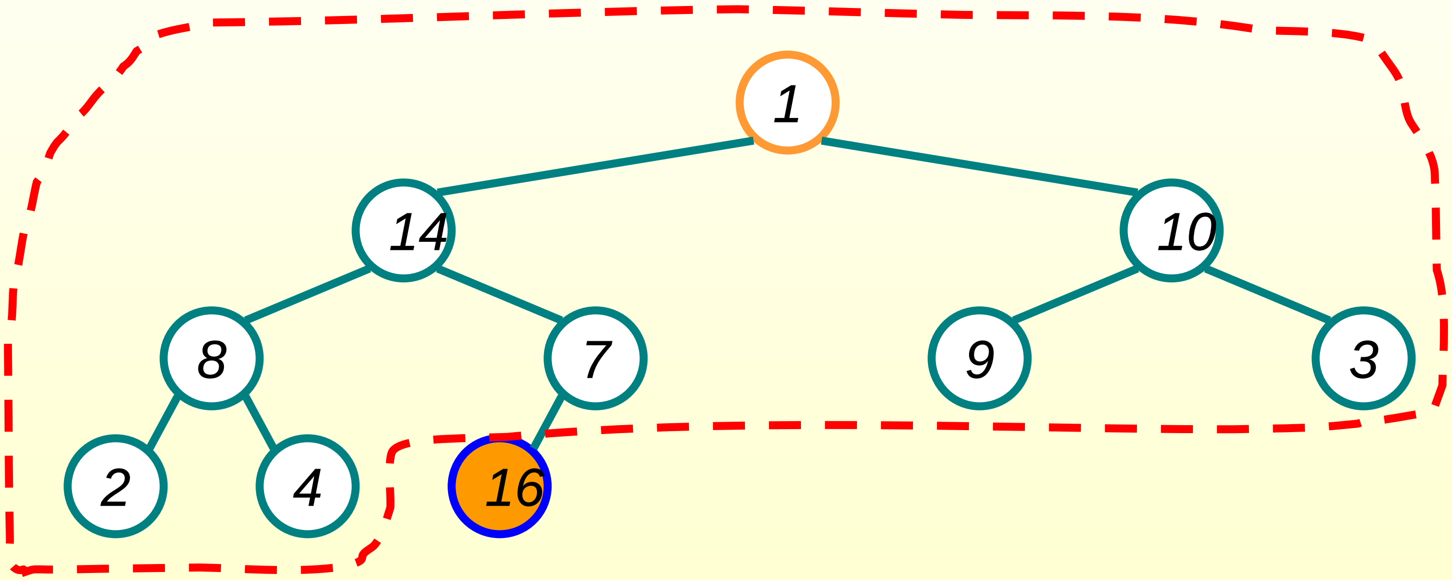


A =

1	14	10	8	7	9	3	2	4	16

Heapsort Example

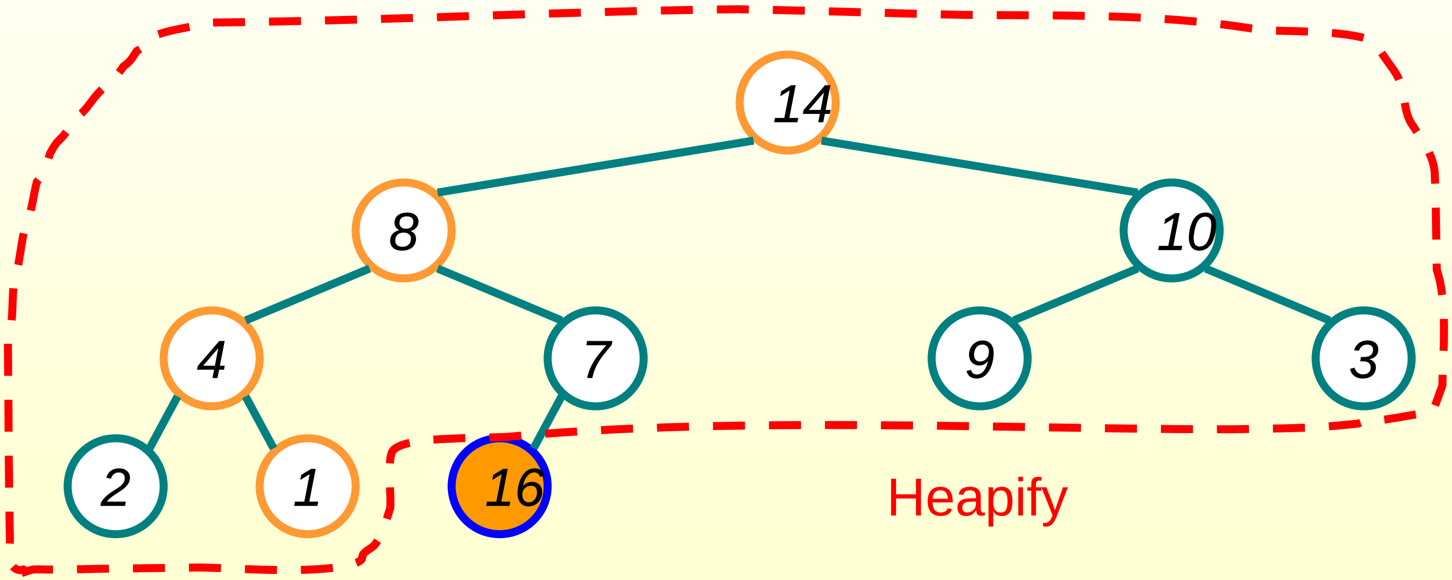
- Last element sorted



A =										
	1	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*16*

Heapsort Example

- Restore heap on remaining unsorted elements

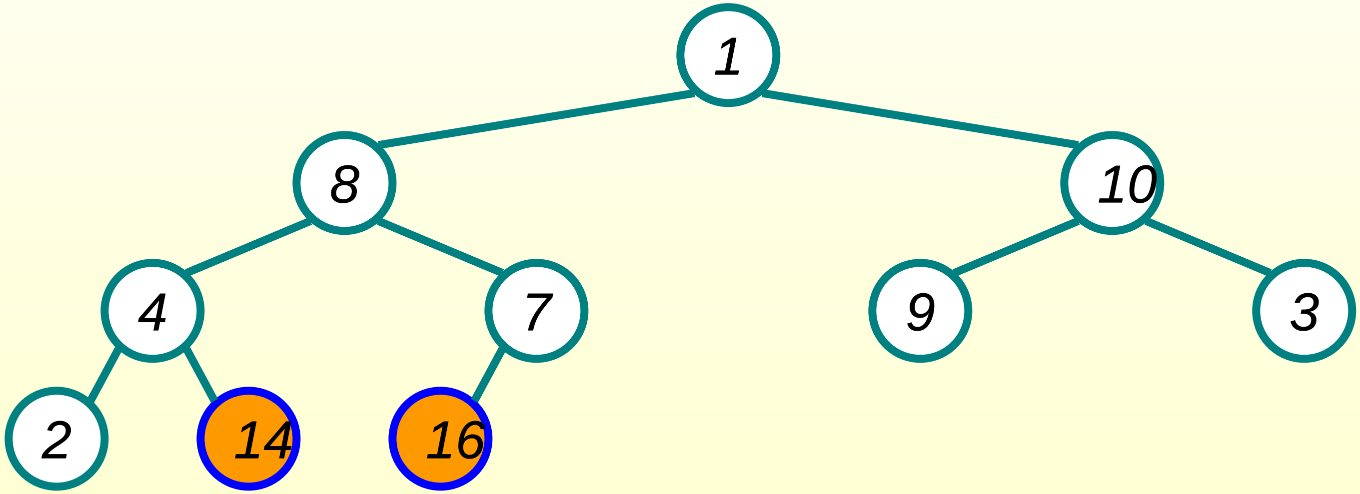


A =

14	8	10	4	7	9	3	2	1	16

Heapsort Example

- Repeat: swap new last and first

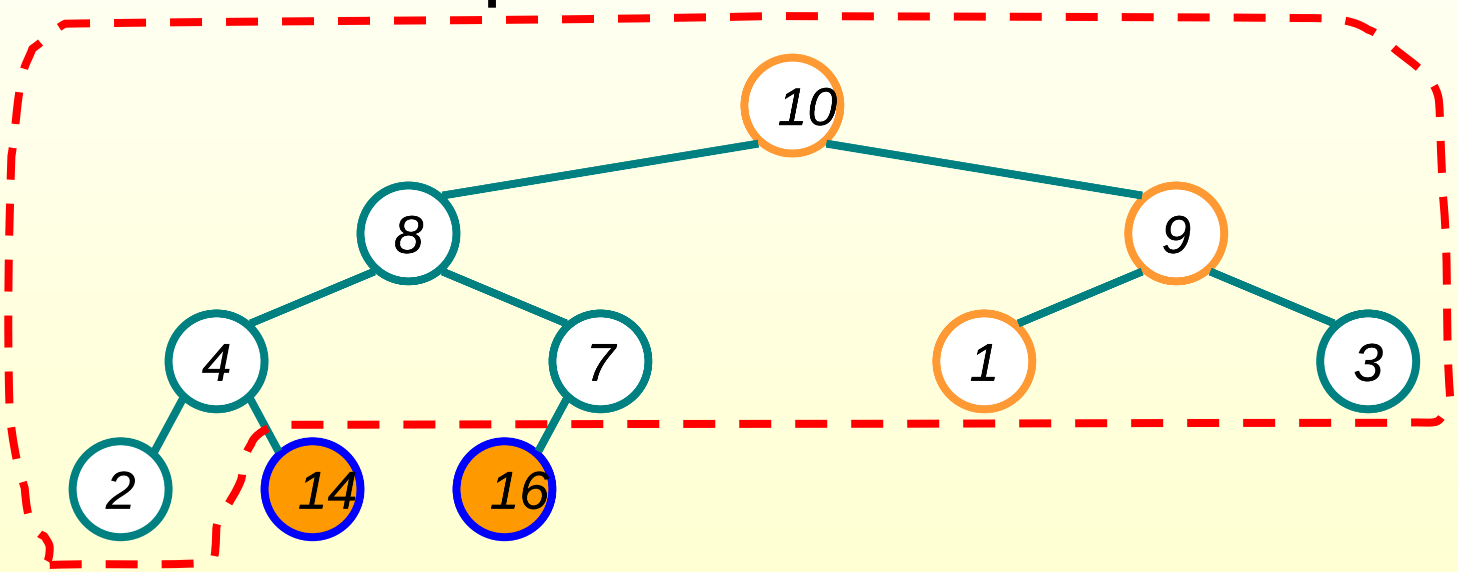


A =

1	8	10	4	7	9	3	2	14	16

Heapsort Example

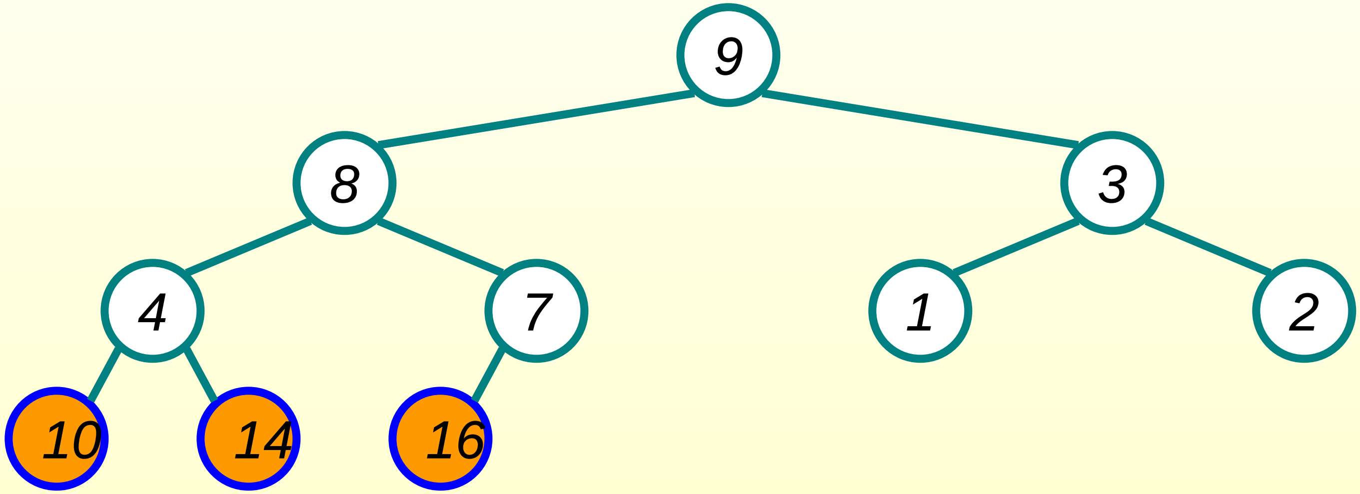
- Restore heap



A =	10	8	9	4	7	1	3	2	14	16

Heapsort Example

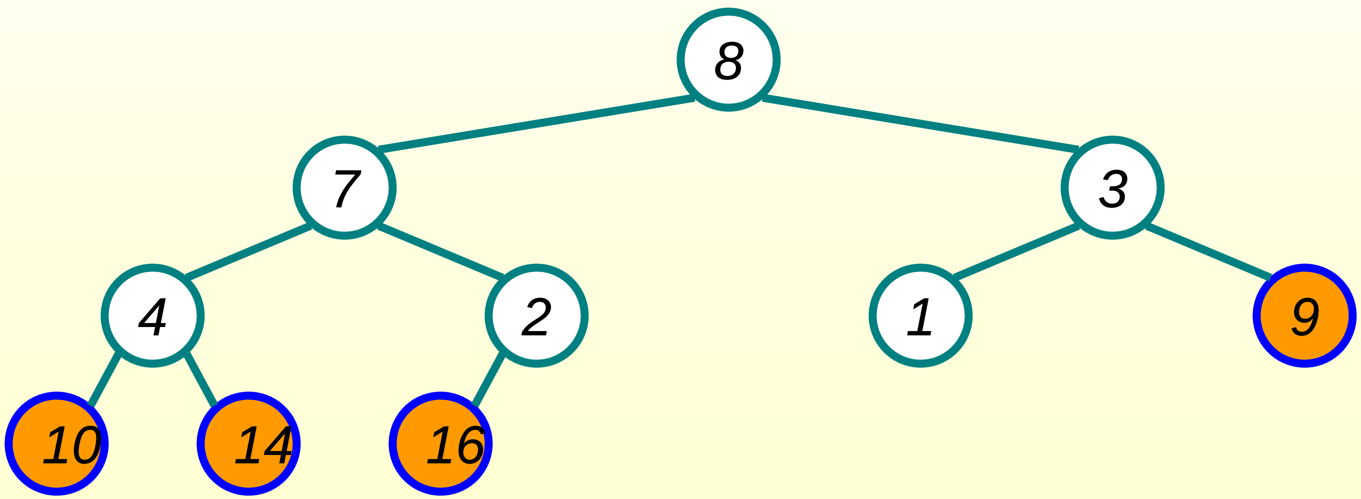
- Repeat



A =	9	8	3	4	7	1	2	10	14	16

Heapsort Example

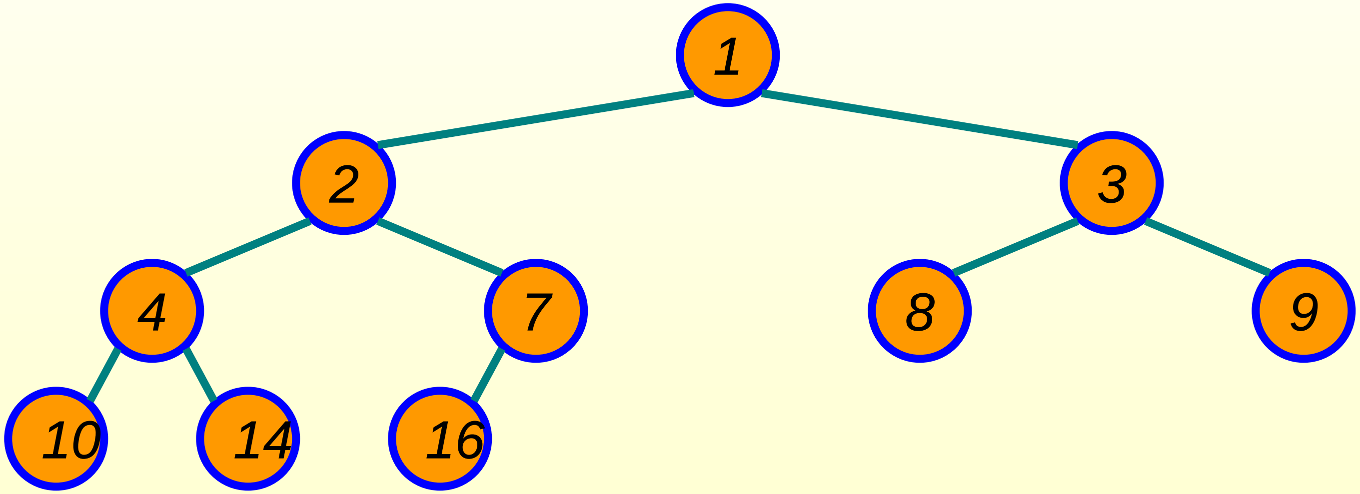
- Repeat



A =	*8*	*7*	*3*	*4*	*2*	*1*	*9*	*10*	*14*	*16*

Heapsort Example

- Repeat



A =	*1*	*2*	*3*	*4*	*7*	*8*	*9*	*10*	*14*	*16*

Analyzing Heapsort

- The call to **`BuildHeap()`** takes $O(n)$ time
- Each of the $n - 1$ calls to **`Heapify()`** takes $O(\lg n)$ time
- Thus the total time taken by **`HeapSort()`**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

Comparison

	Time complexity	Stable?	In-place?
MergeSort			
QuickSort			
HeapSort			

Comparison

	Time complexity	Stable?	In-place?
MergeSort	Θ (n log n)	Yes	No
QuickSort	Θ(n log n) expected. Θ(n^2) worst case	No	Yes
HeapSort	Θ (n log n)	No	Yes

Priority Queues

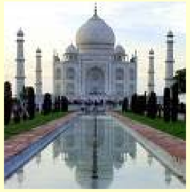
- HeapSort is a nice algorithm, but in practice QuickSort usually wins
- The heap data structure, however, is incredibly useful for implementing priority queues
  - A data structure for maintaining a set S of elements, each with an associated value or key
  - Supports the operations Insert(), Maximum(), ExtractMax(), ChangeKey()
- What might a priority queue be useful for?

Personal Travel Destination List

- You have a list of places that you want to visit, each with a preference score
- Always visit the place with highest score
- Remove a place after visiting it
- You frequently add more destinations
- You may change score for a place when you have more information
- What's the best data structure?







Priority Queue Operations

- Insert(S, x) inserts the element x into set S
- Maximum(S) returns the element of S with the maximum key
- ExtractMax(S) removes and returns the element of S with the maximum key
- ChangeKey(S, i, key) changes the key for element i to something else
- How could we implement these operations using a heap?

Implementing Priority Queues

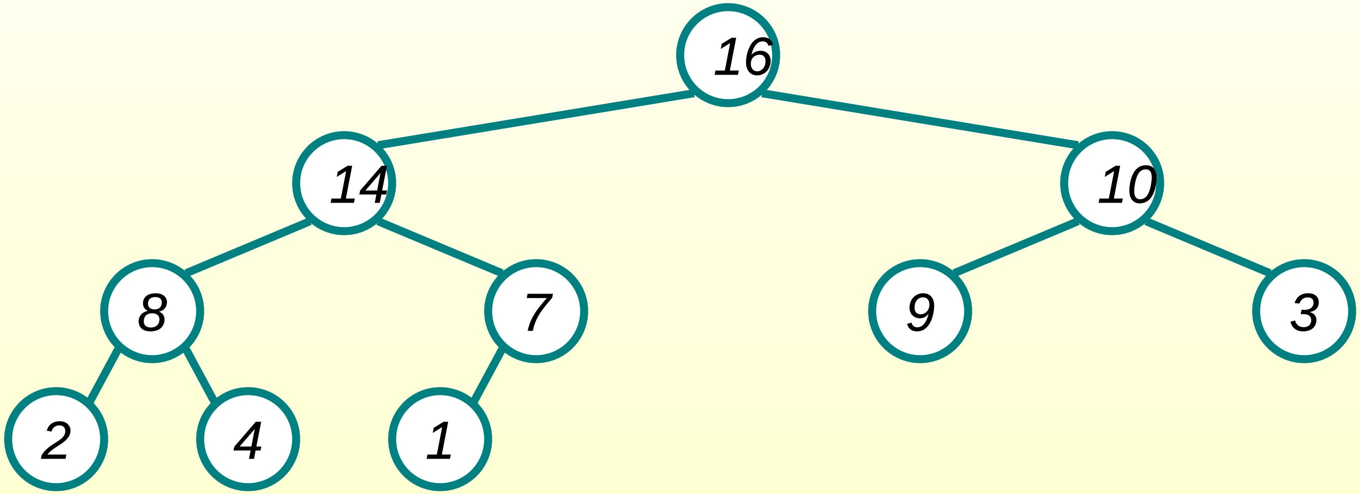
```
HeapMaximum(A)
{

    return A[1];
}
```

Implementing Priority Queues

```
HeapExtractMax(A)
{
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]]
    heap_size[A] --;
    Heapify(A, 1);
    return max;
}
```

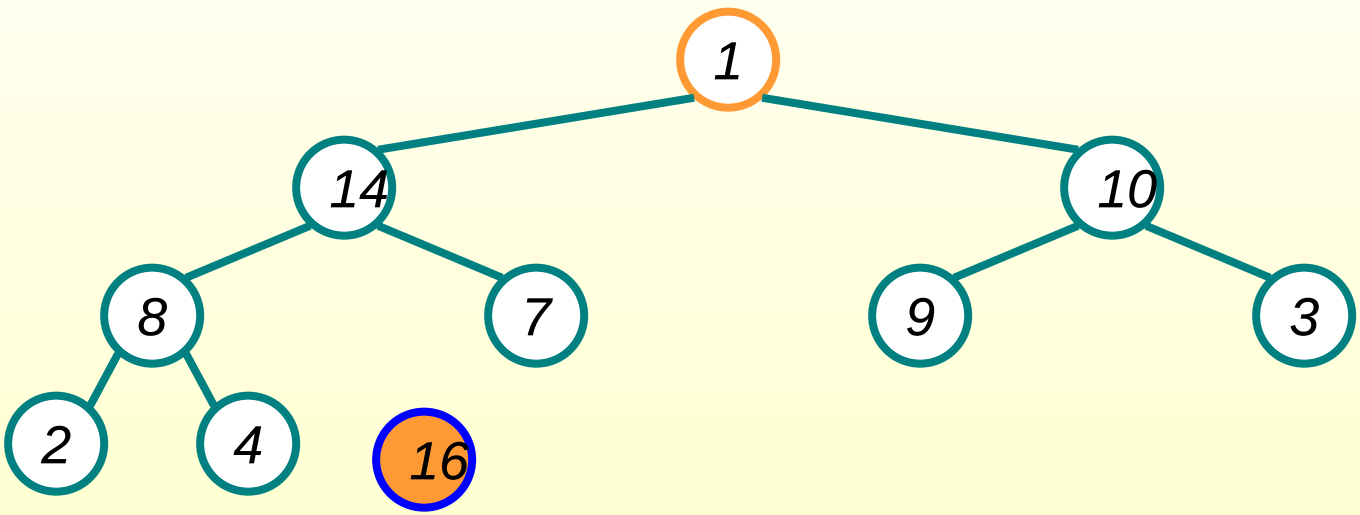
Heap ExtractMax Example



A =	16	14	10	8	7	9	3	2	4	1

Heap ExtractMax Example

Swap first and last, then remove last



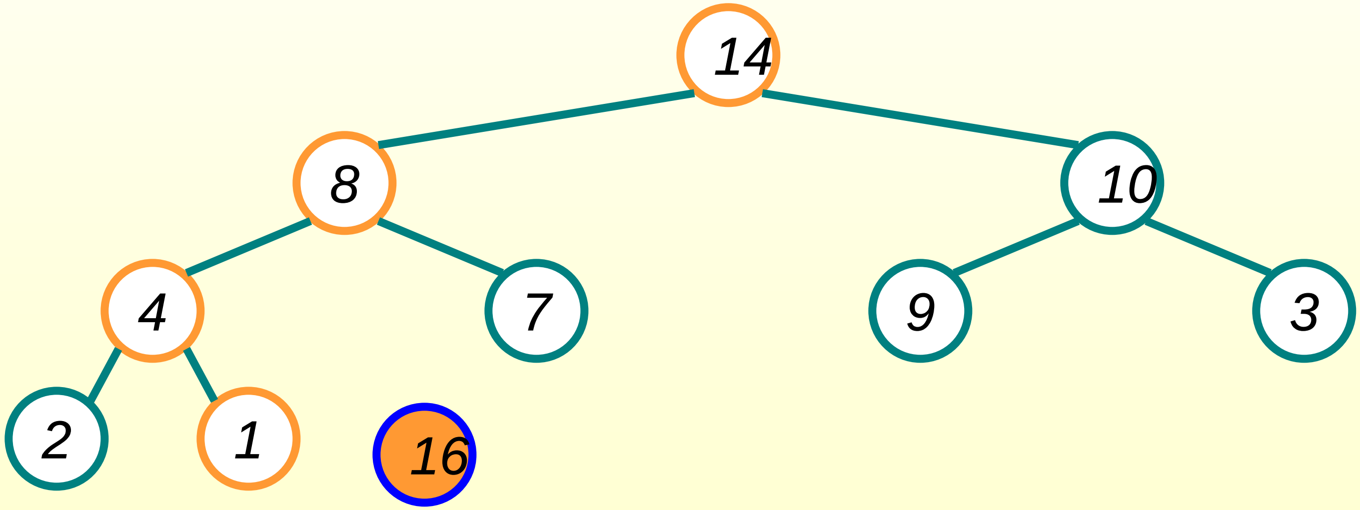
A =

1	14	10	8	7	9	3	2	4

16

Heap ExtractMax Example

Heapify



A =

14	8	10	4	7	9	3	2	1

16

Implementing Priority Queues

```
HeapChangeKey(A, i, key){
    if (key <= A[i]){ //
        A[i] = key;
        heapify(A, i);
    } else { // increase
        A[i] = key;
        while (i>1 &
  A[parent(i)]<A[i])
            swap(A[i], A[parent(i)];
    }
}
```

Sift down

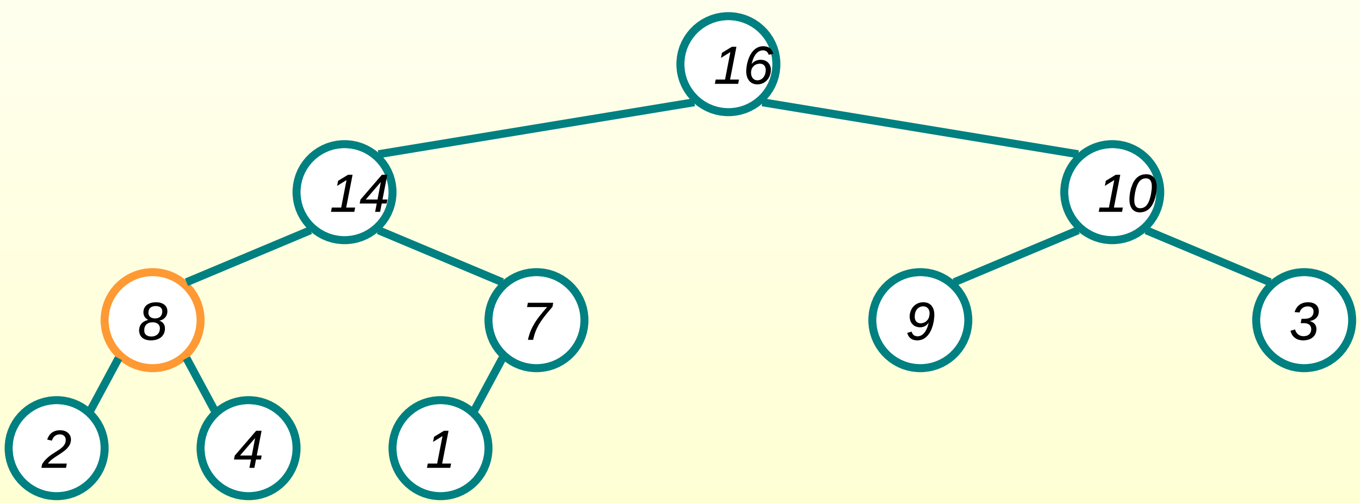
Bubble up

Heap ChangeKey Example

HeapChangeKey(A, 4, 15)

Change key value to 15

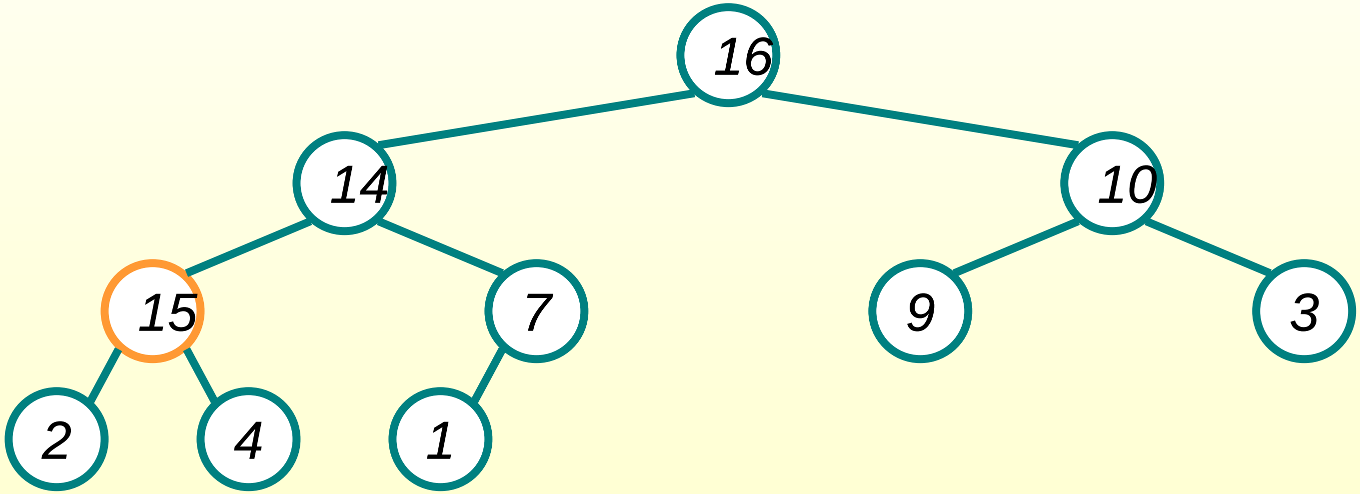
4th element



A =	*16*	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

Heap ChangeKey Example

HeapChangeKey(A, 4, 15)

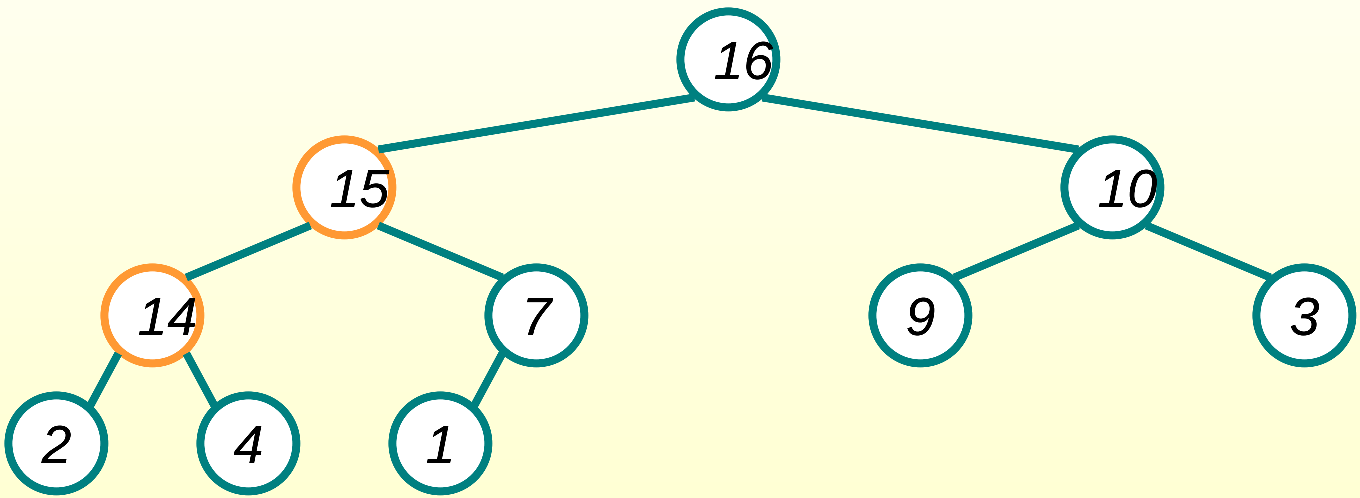


A =

16	14	10	15	7	9	3	2	4	1

HeapChangeKey Example

HeapChangeKey(A, 4, 15)



A =

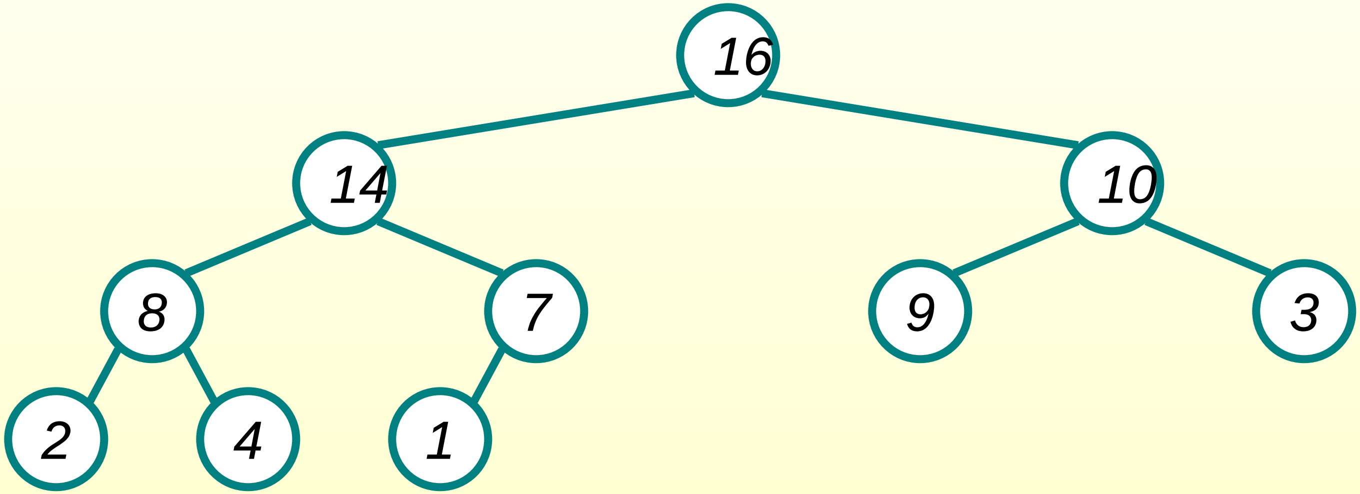
16	15	10	14	7	9	3	2	4	1

Implementing Priority Queues

```
HeapInsert(A, key) {
    heap_size[A] ++;
    i = heap_size[A];
    A[i] = -∞;
    HeapChangeKey(A, i, key);
}
```

Heap Insert Example

HeapInsert(A, 17)

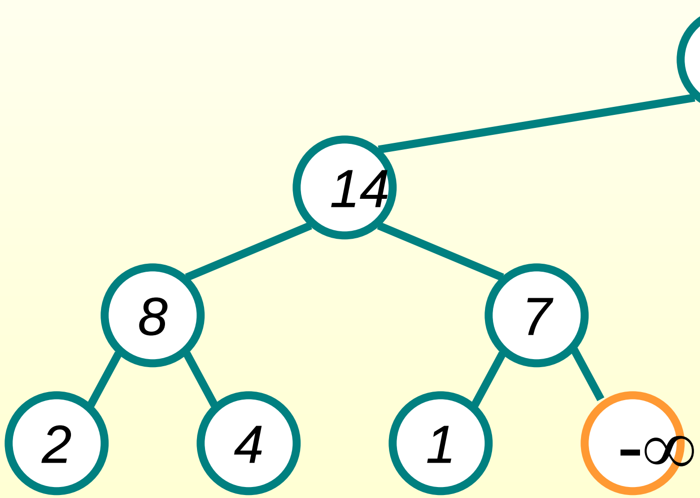


A =

16	14	10	8	7	9	3	2	4	1

Heap Insert Example

HeapInsert(A, 17)



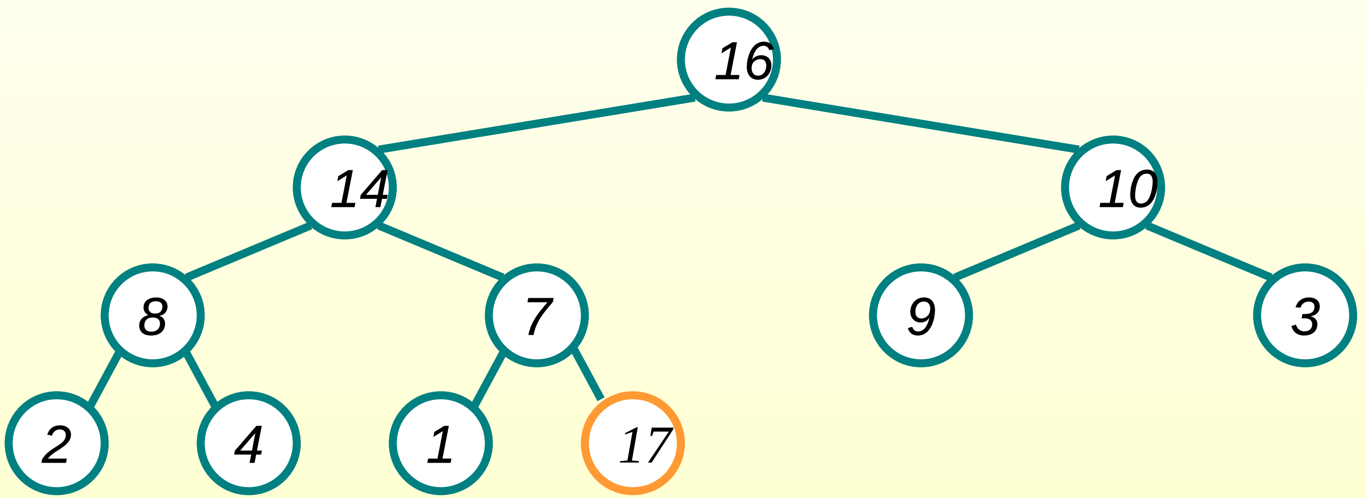
-∞ makes it a valid heap

A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*	**-∞**

Heap Insert Example

HeapInsert(A, 17)



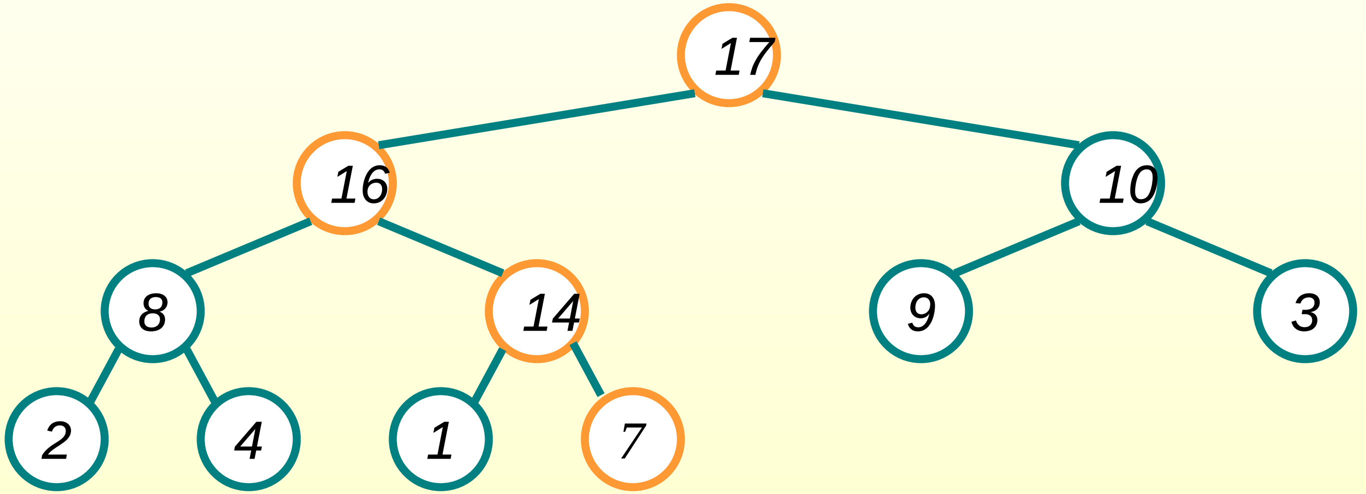
Now call HeapChangeKey

A =

16	14	10	8	7	9	3	2	4	1	17

Heap Insert Example

HeapInsert(A, 17)



A =

17	16	10	8	14	9	3	2	4	1	7

- Heapify: $\Theta(\log n)$
- BuildHeap: $\Theta(n)$
- HeapSort: $\Theta(n \log n)$

- HeapMaximum: $\Theta(1)$
- HeapExtractMax: $\Theta(\log n)$
- HeapChangeKey: $\Theta(\log n)$
- HeapInsert: $\Theta(\log n)$

Compare: Sorted Array / Linked List

- Sort: $\Theta(n \log n)$
- Afterwards:

- arrayMaximum: $\Theta(1)$
- arrayExtractMax: $\Theta(n)$ or $\Theta(1)$
- arrayChangeKey: $\Theta(n)$
- arrayInsert: $\Theta(n)$