CSE 373: Data Structures and Algorithms

Floyd's buildHeap, Sorting

Autumn 2018

Shrirang (Shri) Mare
shri@cs.washington.edu

Thanks to Kasey Champion, Ben Jones, Adam Blank, Michael Lee, Evan McCarty, Robbie Weber, Whitaker Brand, Zora Fung, Stuart Reges, Justin Hsia, Ruth Anderson, and many others for sample slides and materials ...

Today

- Heap review and Array implementation of Heap
- Floyd's buildHeap algorithm
- Intro to Sorting
- Insertion sort
- Heap sort

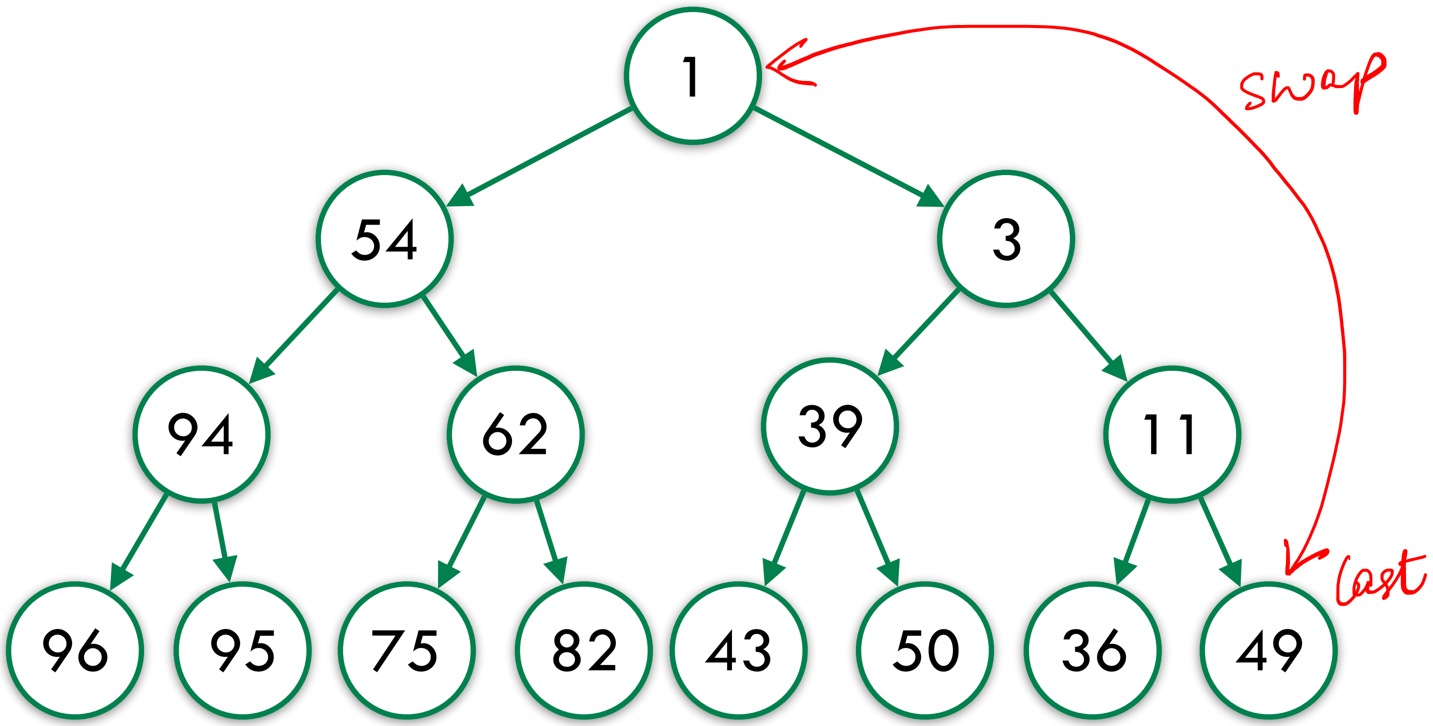
Heap review

Heap is a tree-based data structure that satisfies

- (a) structure property: it's a complete tree
- (b) heap property, which states:
  - for min-heap: $parent \leq children$
  - for max-heap: $parent \geq children$
- Operations (for min-heap):
  - removeMin()
  - peekMin()
  - insert()
- Applications: priority queue, sorting, ..

removeMin()

Calls:
removeMin()

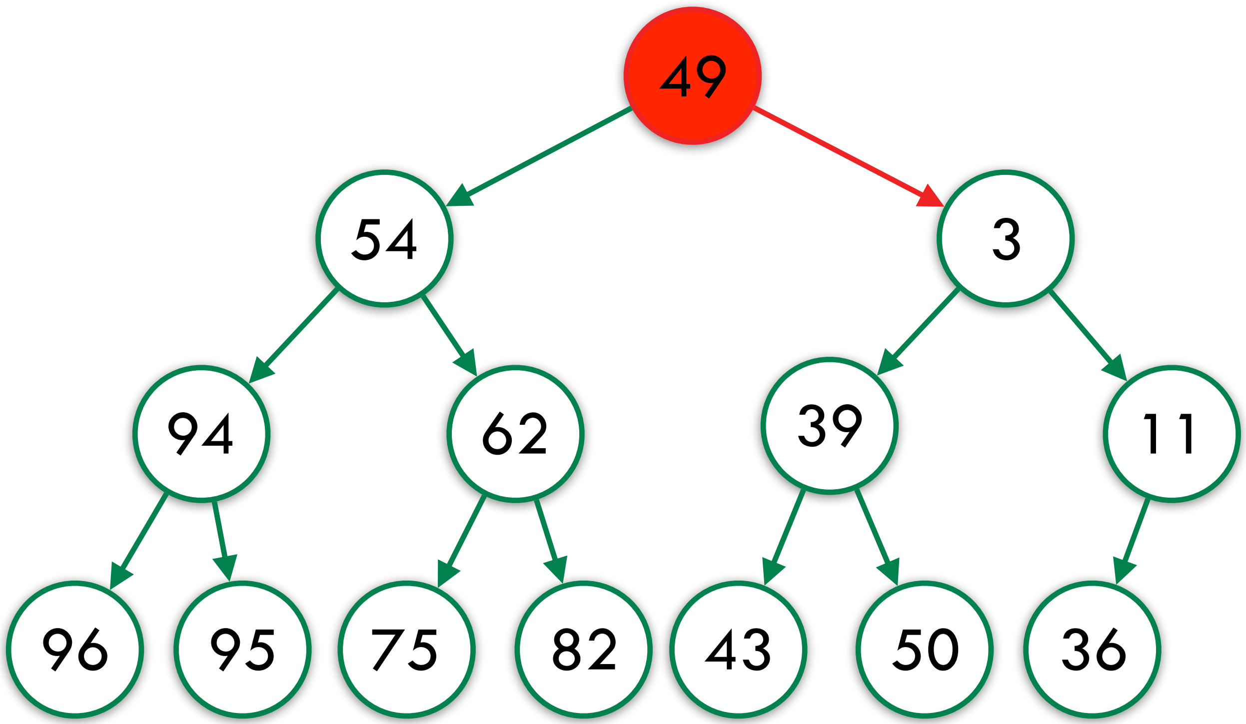


min-heap

function removeMin
  last = last node in the tree
  minvalue = *root*
  swap *root* with *last*
  percolateDown(*root*)
  return *minvalue*
end function

removeMin()

Calls:
removeMin() 1

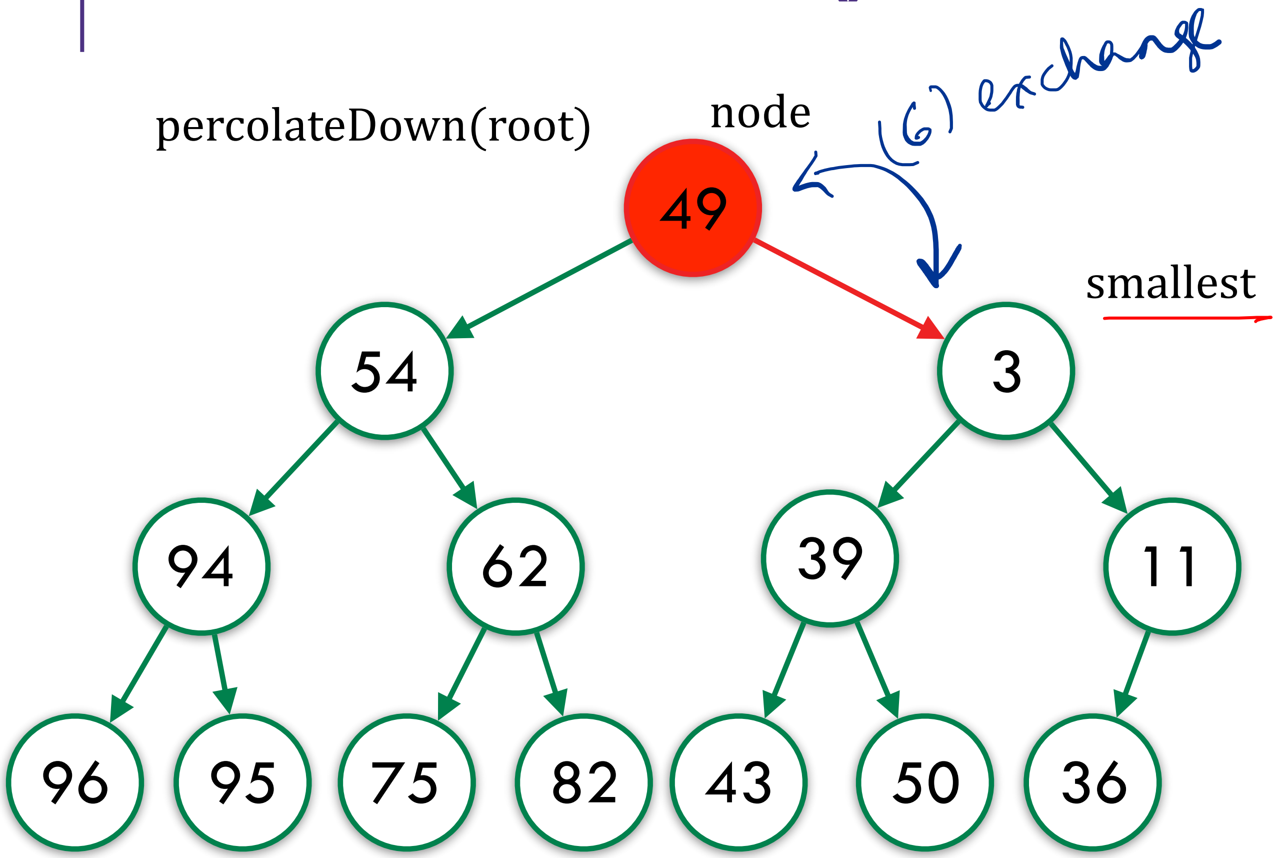


Broken heap

function removeMin
$last$ = last node in the tree
$minvalue = root$
swap $root$ with $last$
percolateDown($root$)
return $minvalue$
end function

removeMin()

Calls:
removeMin() 1

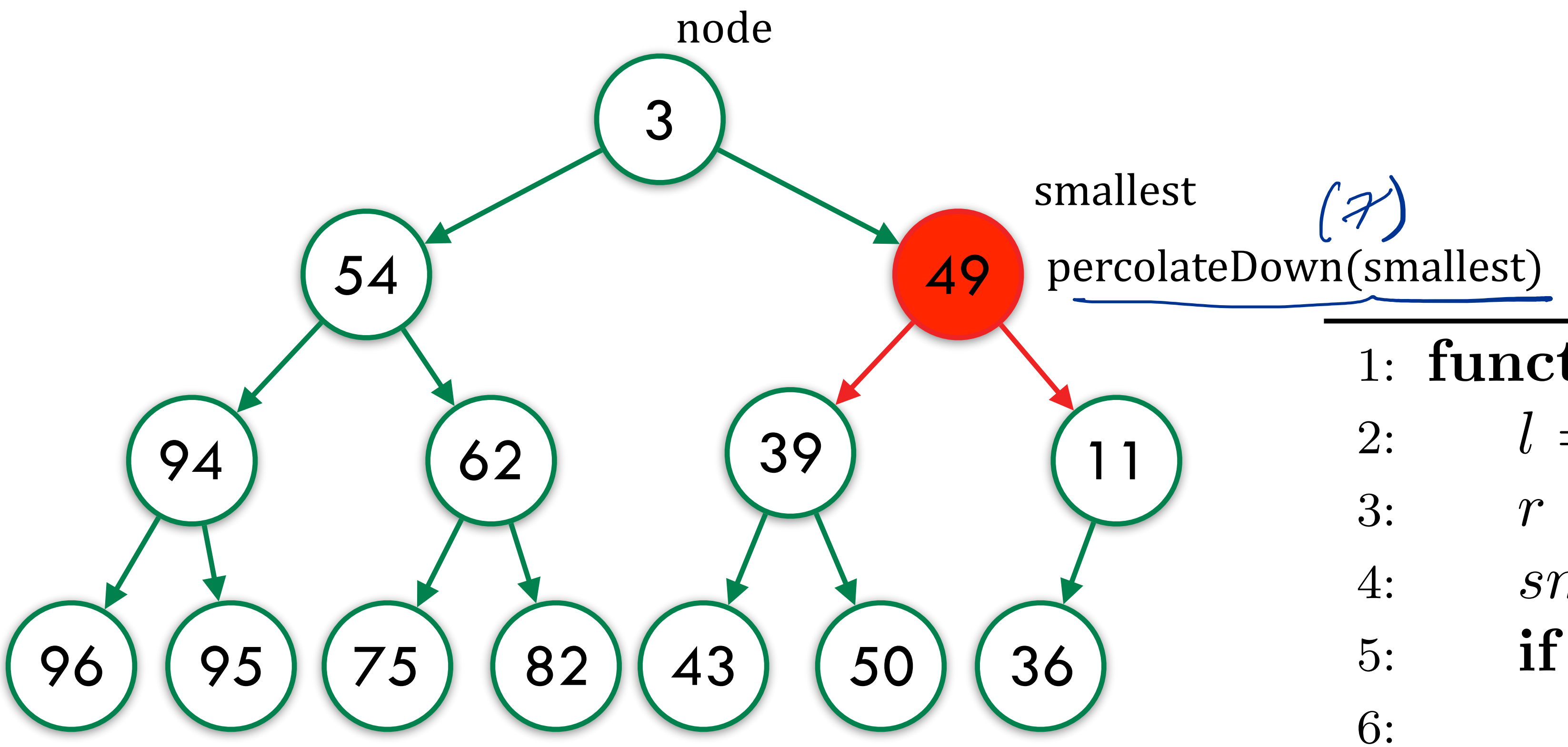


Broken heap

1: **function** percolateDown($node$)
2: l = left child of $node$
3: r = right child of $node$
4: $smallest$ = smallest in $\{l, r, node\}$
5: **if** $smallest \neq node$ **then**
6: exchange $node$ with $smallest$
7: percolate($smallest$)
8: **end if**
9: **end function**

removeMin()

Calls:
removeMin() 1

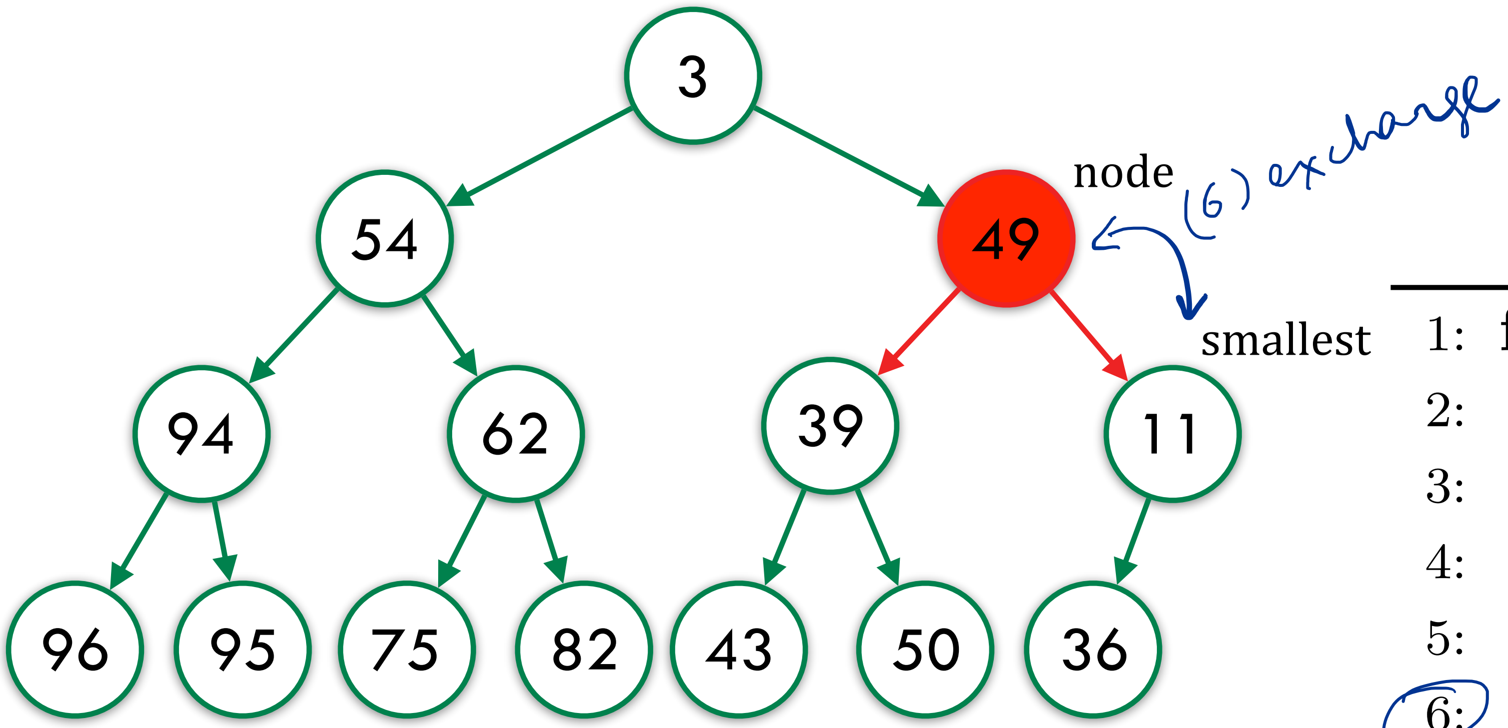


Broken heap

```
function percolateDown(node)
    l = left child of node
    r = right child of node
    smallest = smallest in {l, r, node}
    if smallest ≠ node then
        exchange node with smallest
        percolate(smallest)
    end if
end function
```

removeMin()

Calls:
removeMin() 1

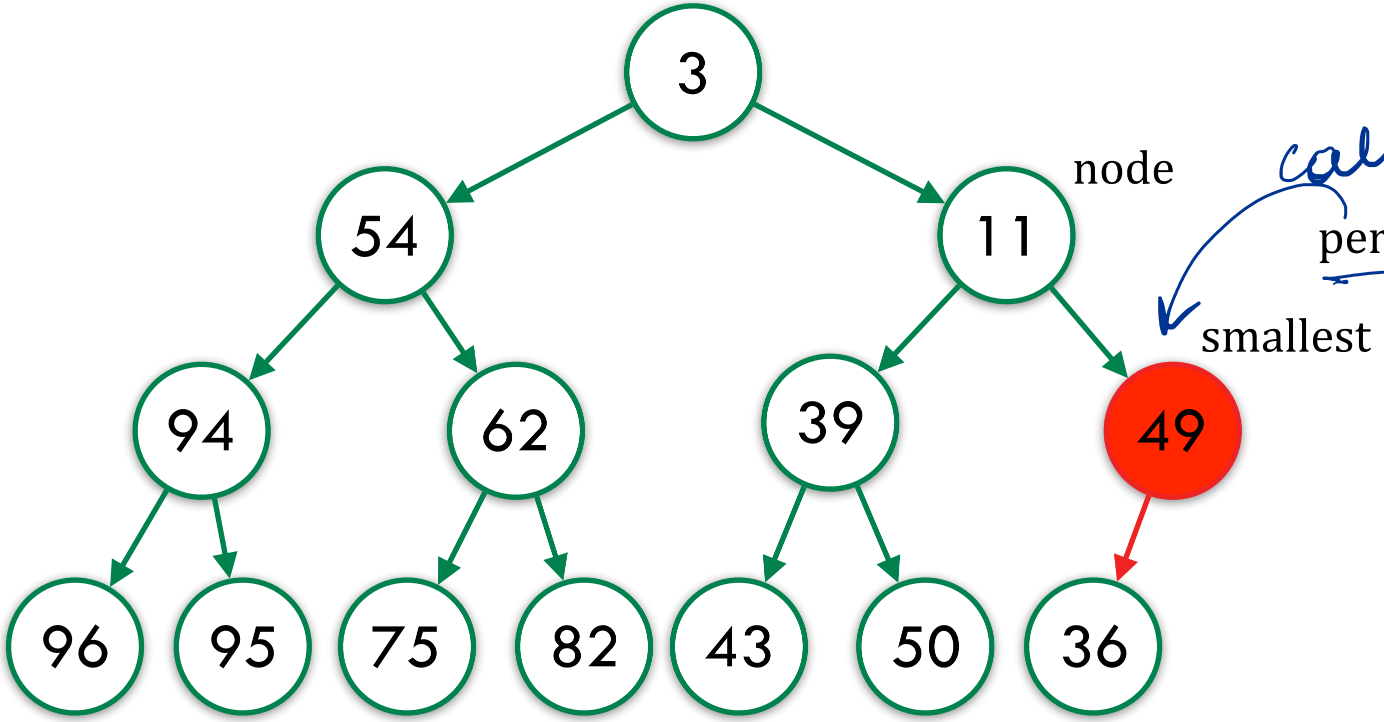


Broken heap

```
1: function percolateDown(node)
2:     l = left child of node
3:     r = right child of node
4:     smallest = smallest in {l, r, node}
5:     if smallest ≠ node then
6:         exchange node with smallest
7:         percolate(smallest)
8:     end if
9: end function
```

removeMin()

Calls:
removeMin() 1



Broken heap

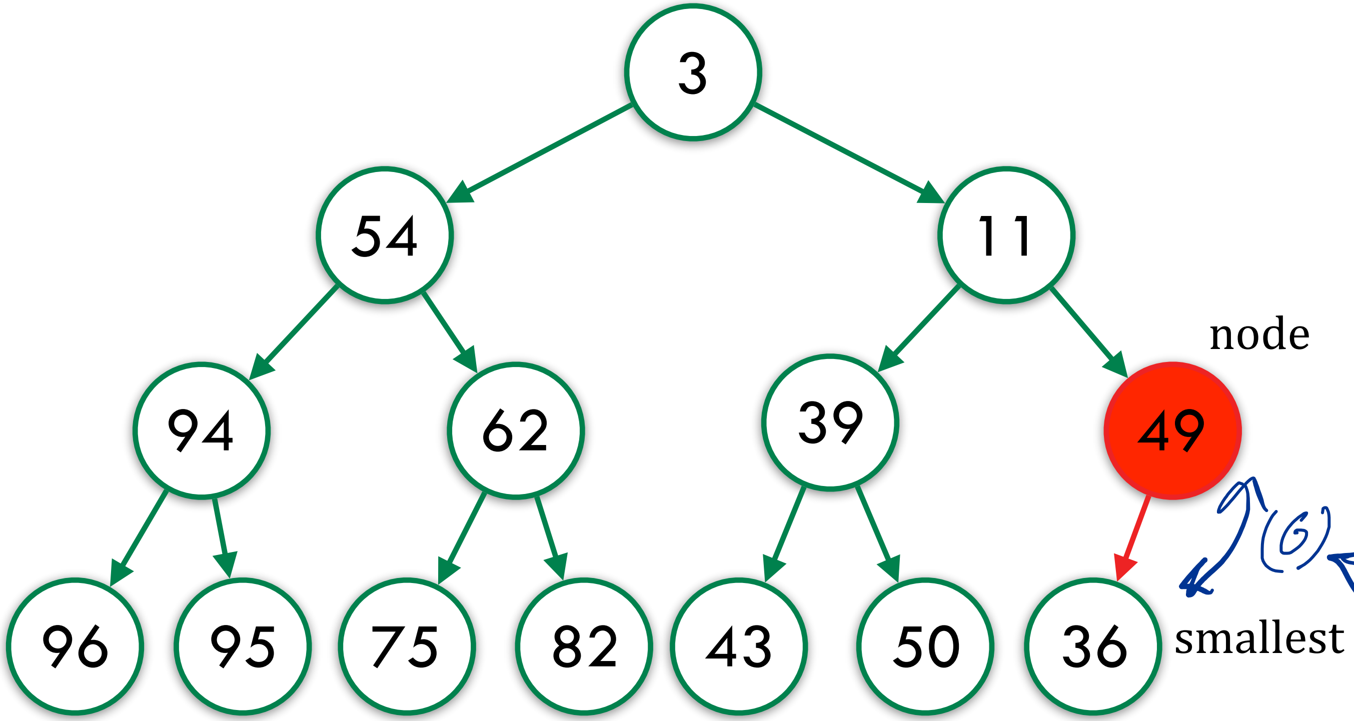
called on this node (7)

percolateDown(smallest)

```
1: function percolateDown(node)
2:     l = left child of node
3:     r = right child of node
4:     smallest = smallest in {l, r, node}
5:     if smallest ≠ node then
6:         exchange node with smallest
7:         percolate(smallest)
8:     end if
9: end function
```

removeMin()

Calls:
removeMin() 1

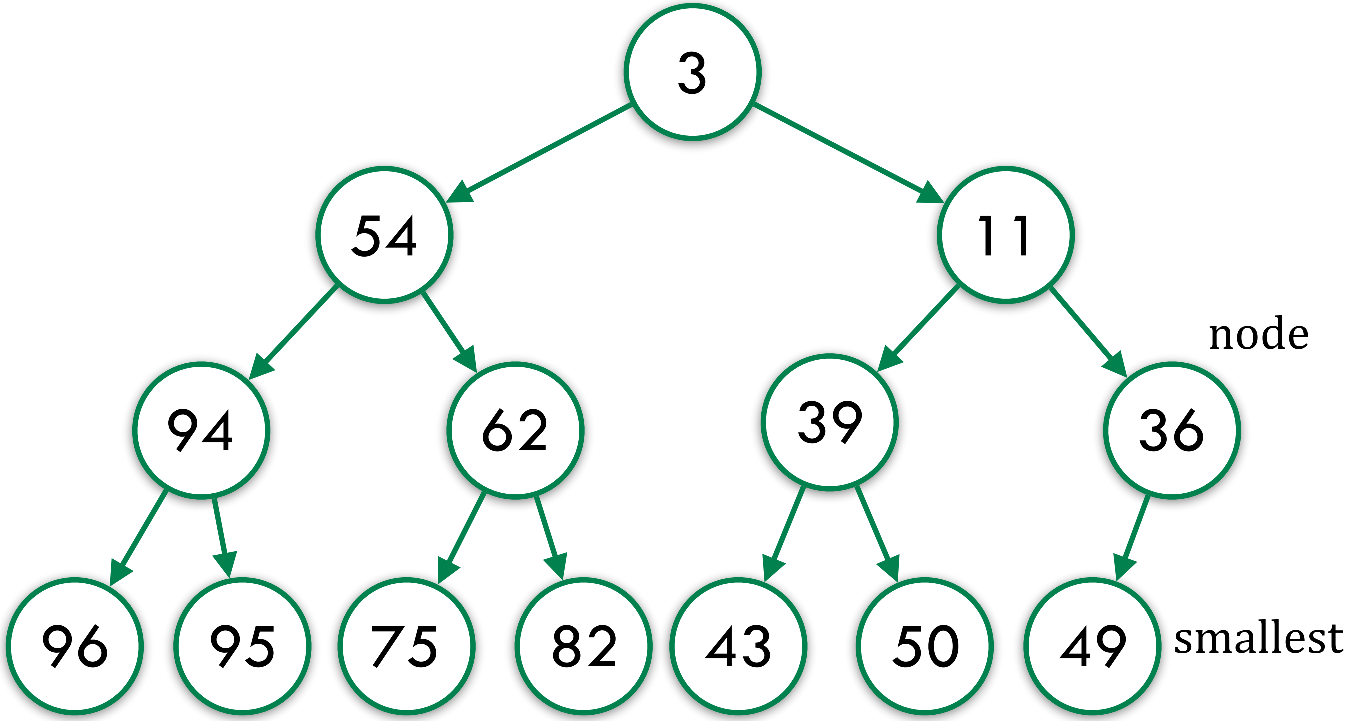


Broken heap

```
function percolateDown(node)
    l = left child of node
    r = right child of node
    smallest = smallest in {l, r, node}
    if smallest ≠ node then
        exchange node with smallest
        percolate(smallest)
    end if
end function
```

removeMin()

Calls:
removeMin() 1



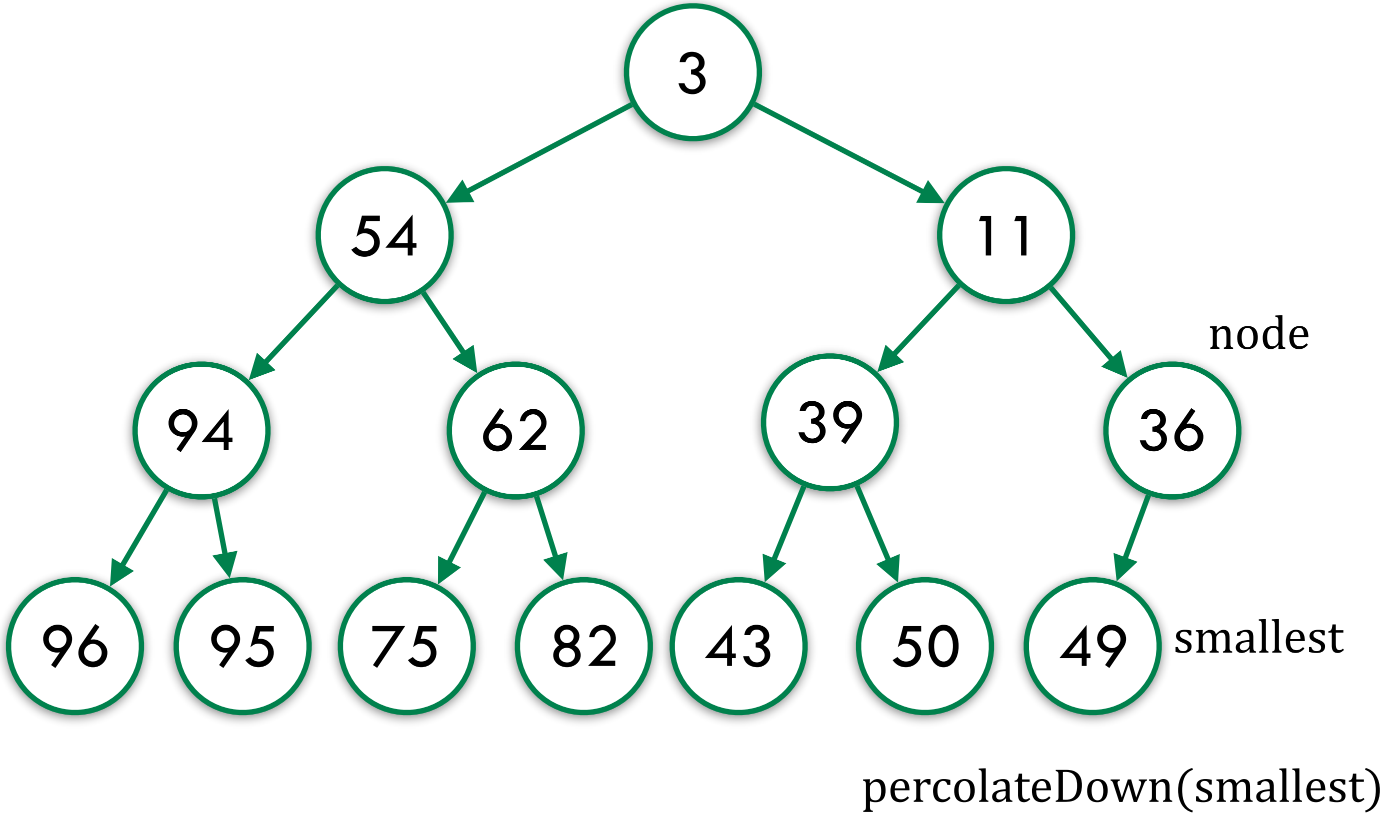
percolateDown(smallest)

final recursive call

min-heap

```
function percolateDown(node)
    l = left child of node
    r = right child of node
    smallest = smallest in {l, r, node}
    if smallest ≠ node then
        exchange node with smallest
        percolate(smallest)
    end if
end function
```

removeMin()



min-heap

---

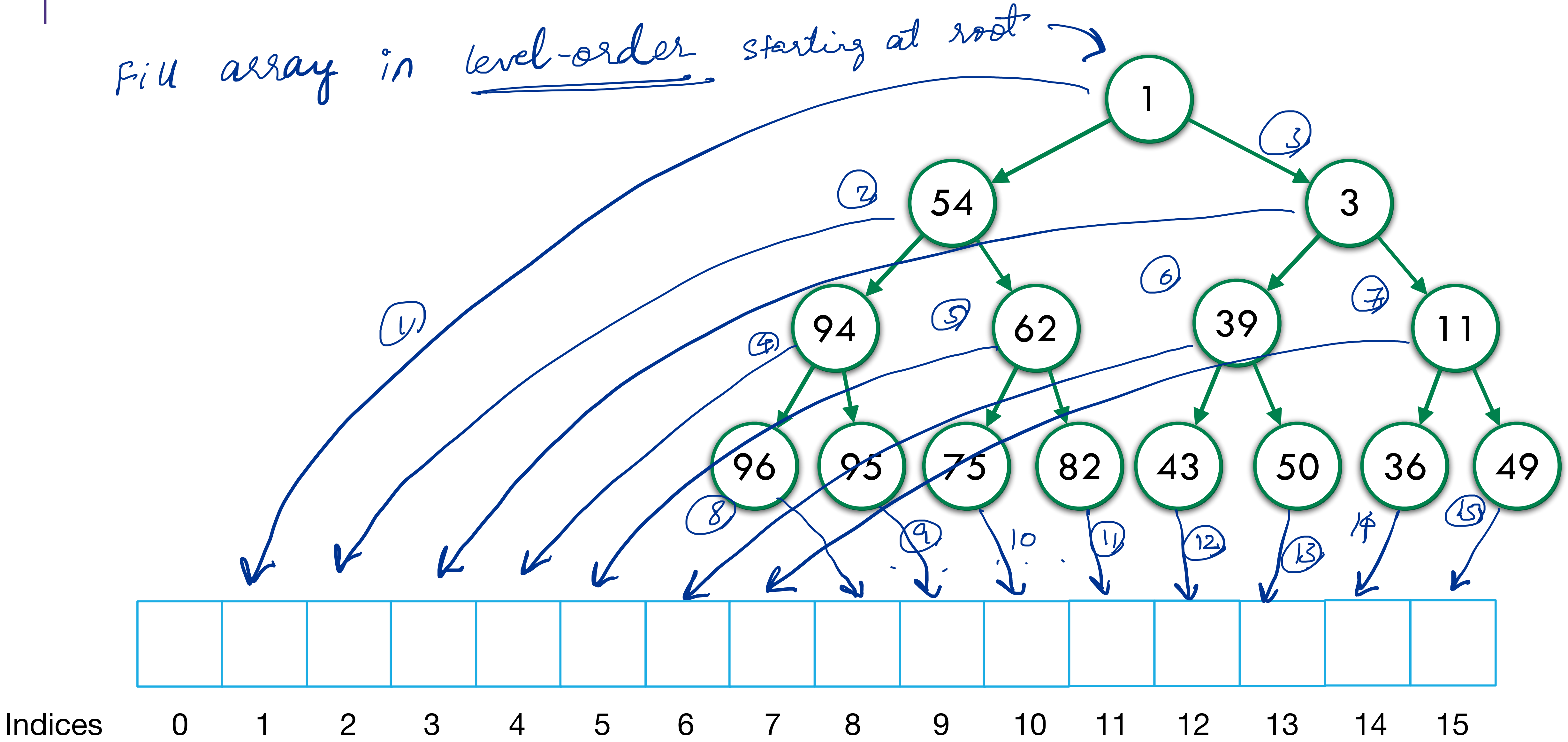
function removeMin
    last = last node in the tree
    minvalue = *root*
    swap *root* with *last*
    percolateDown(*root*)
    return *minvalue*
end function

Runtime of removeMin():

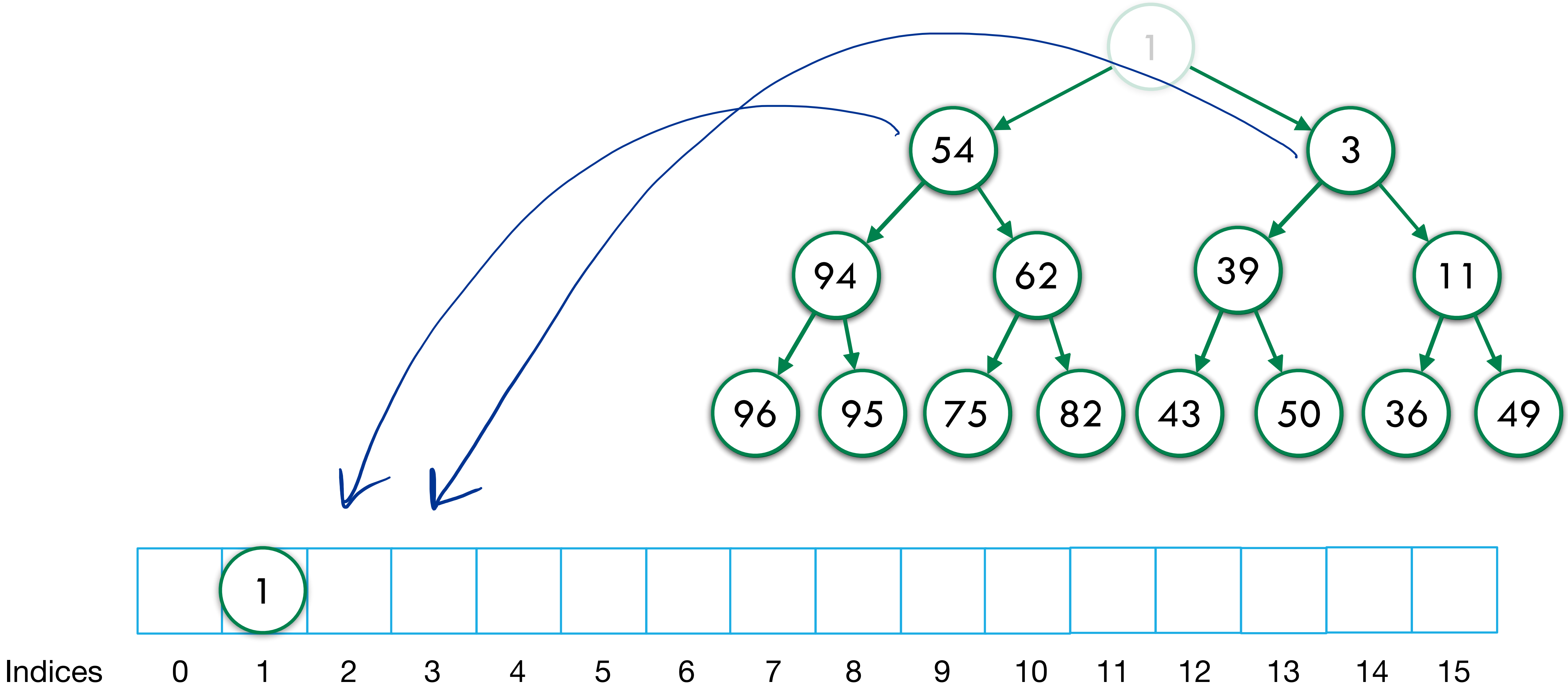
$$O(n) + c_1 + O(n \log n) = O(n)$$

How can we do better?

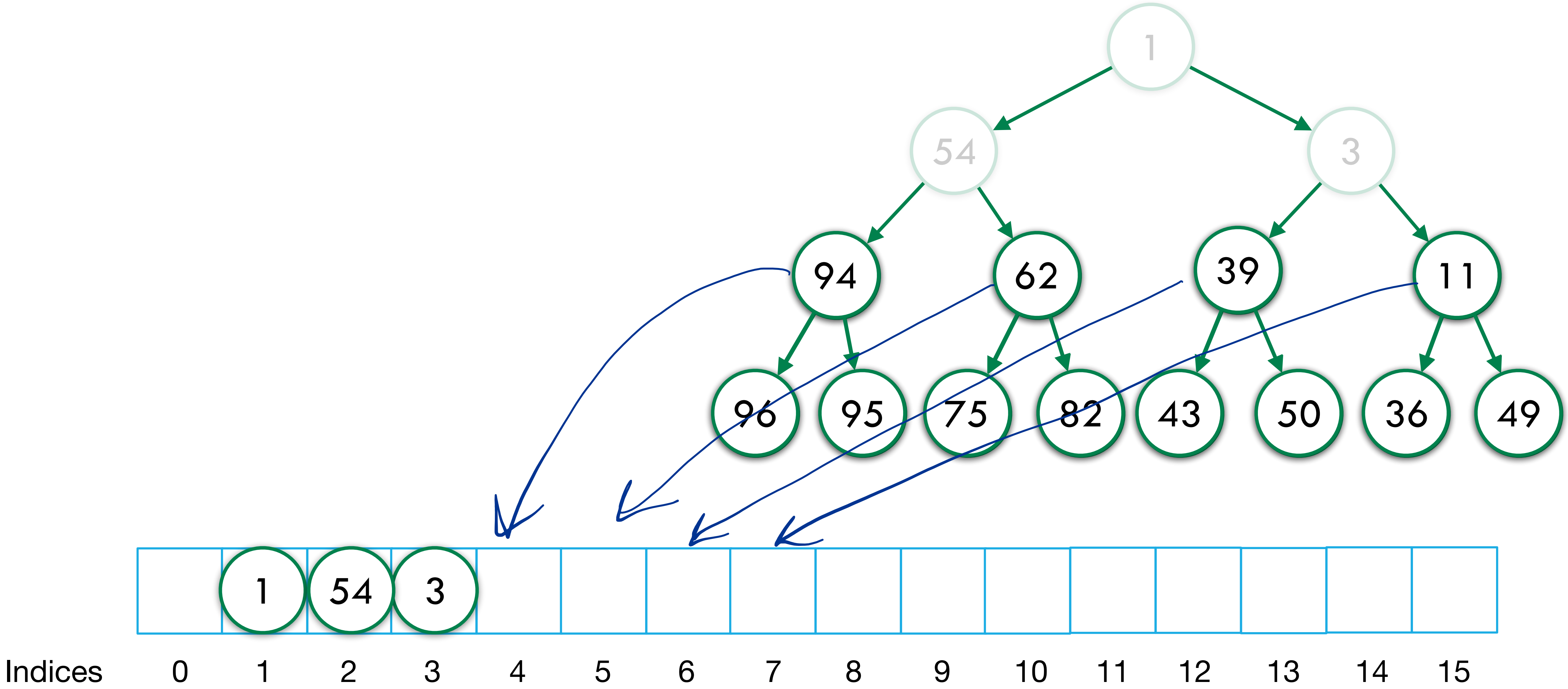
Binary heap: Array implementation



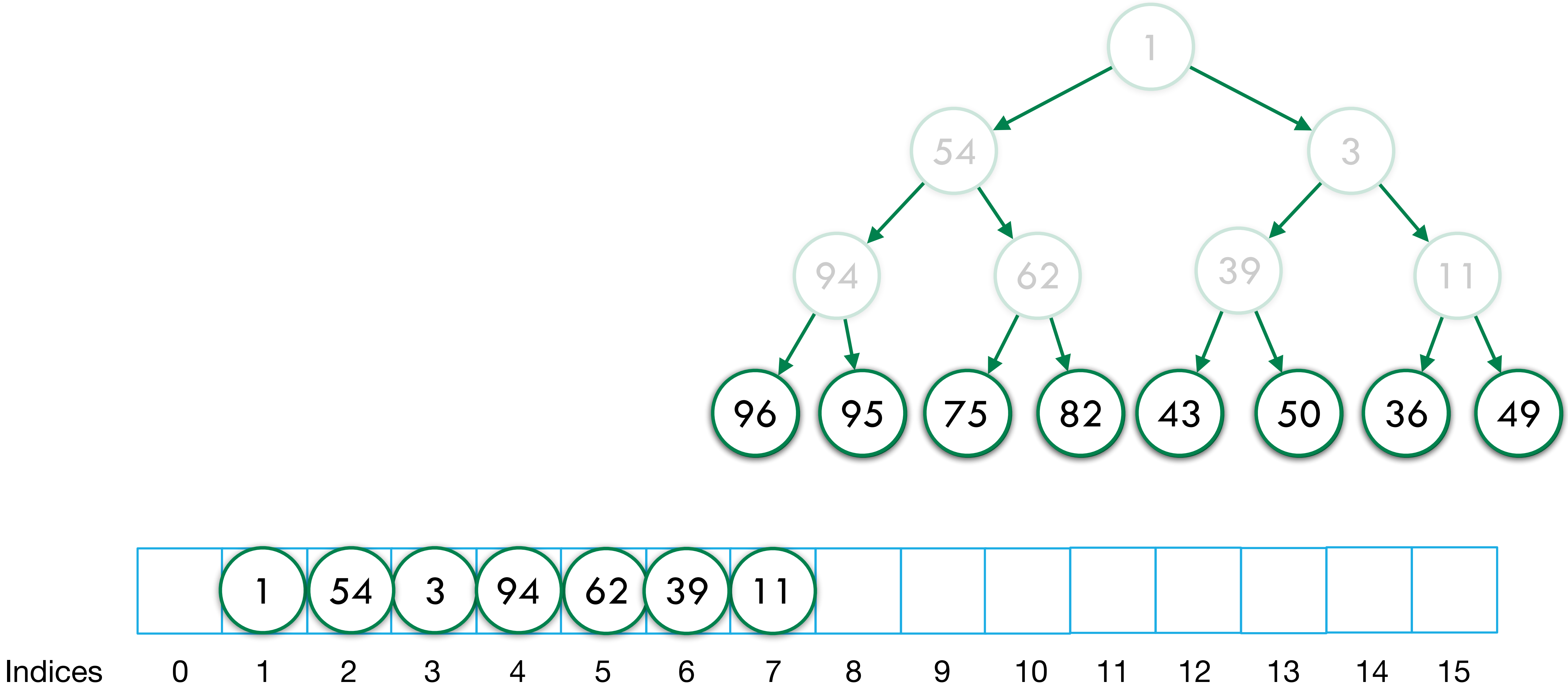
Binary heap: Array implementation



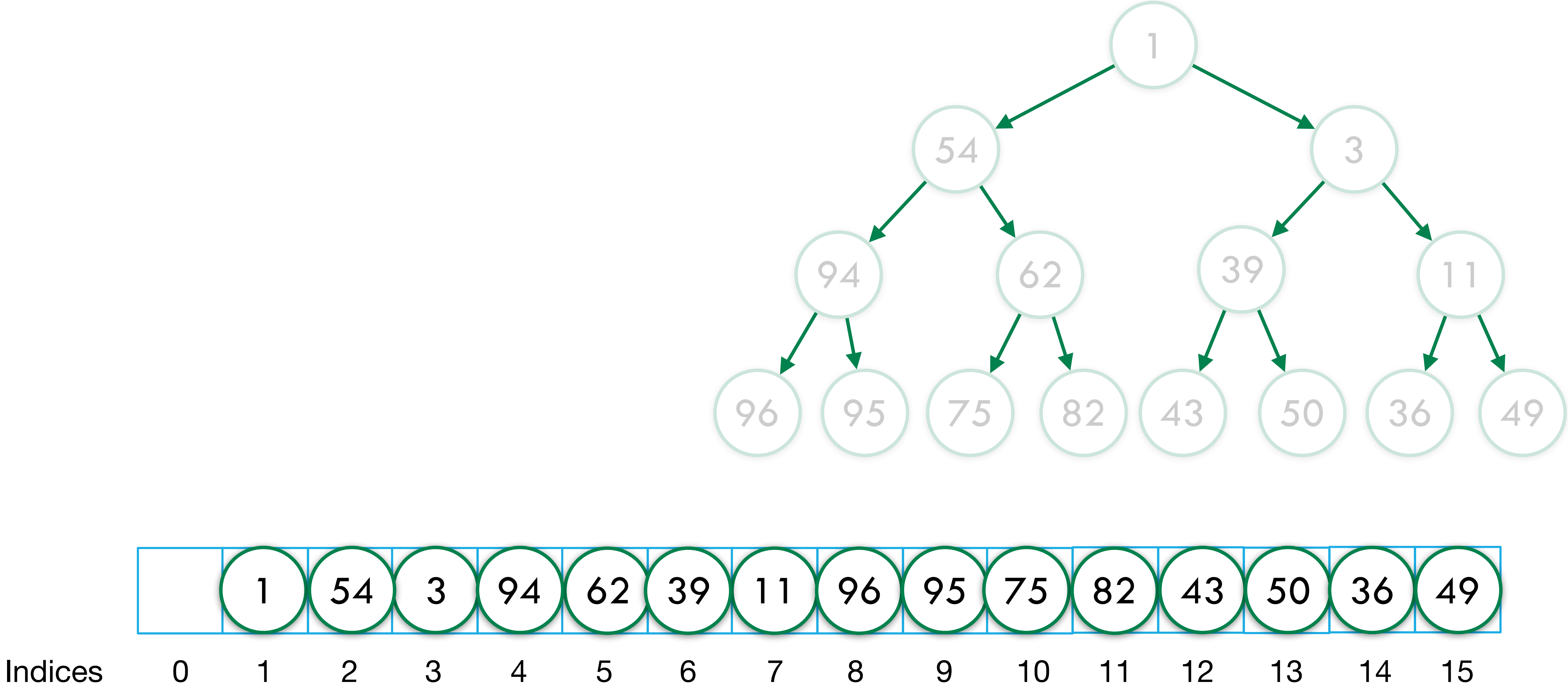
Binary heap: Array implementation



Binary heap: Array implementation



Binary heap: Array implementation

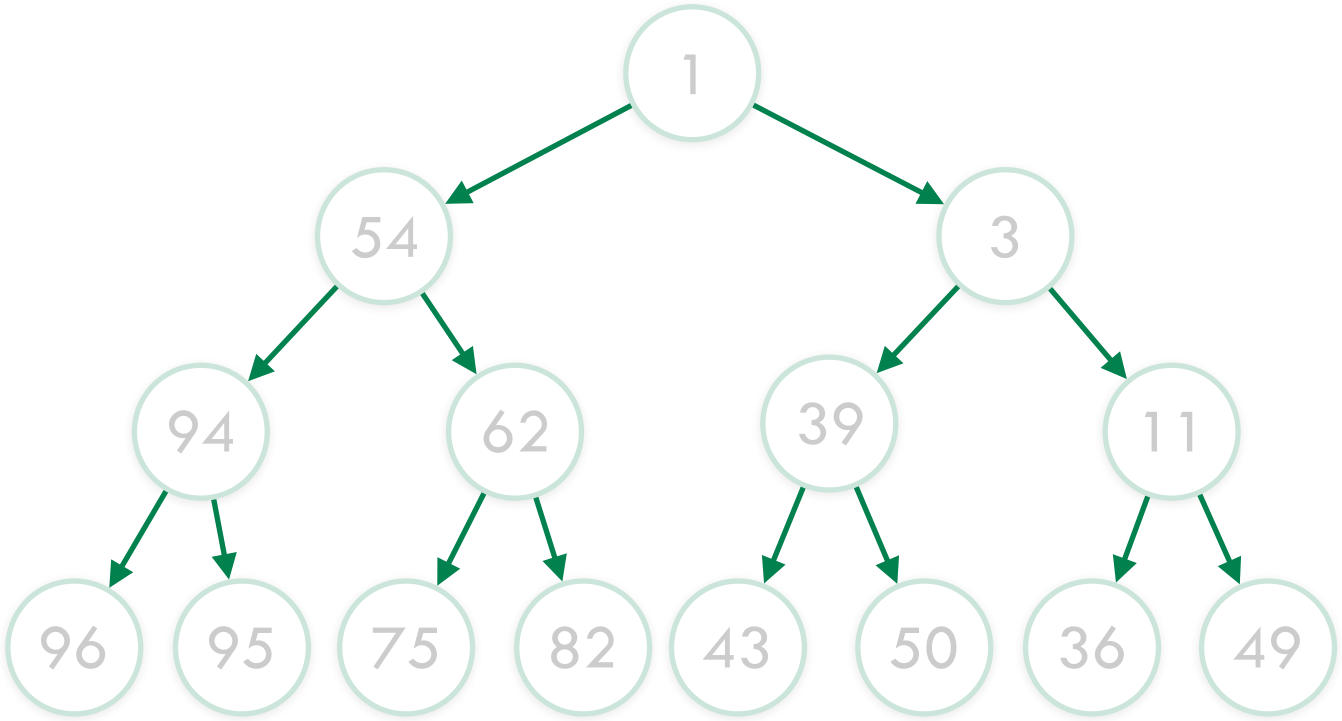


Binary heap: Array implementation

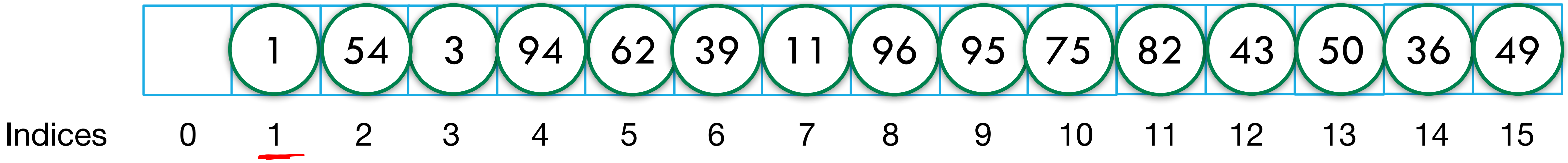
$$leftChild(i) = 2i$$

$$rightChild(i) = 2i + 1$$

$$parent(i) = \left\lfloor \frac{i}{2} \right\rfloor$$



With array starting at index 1

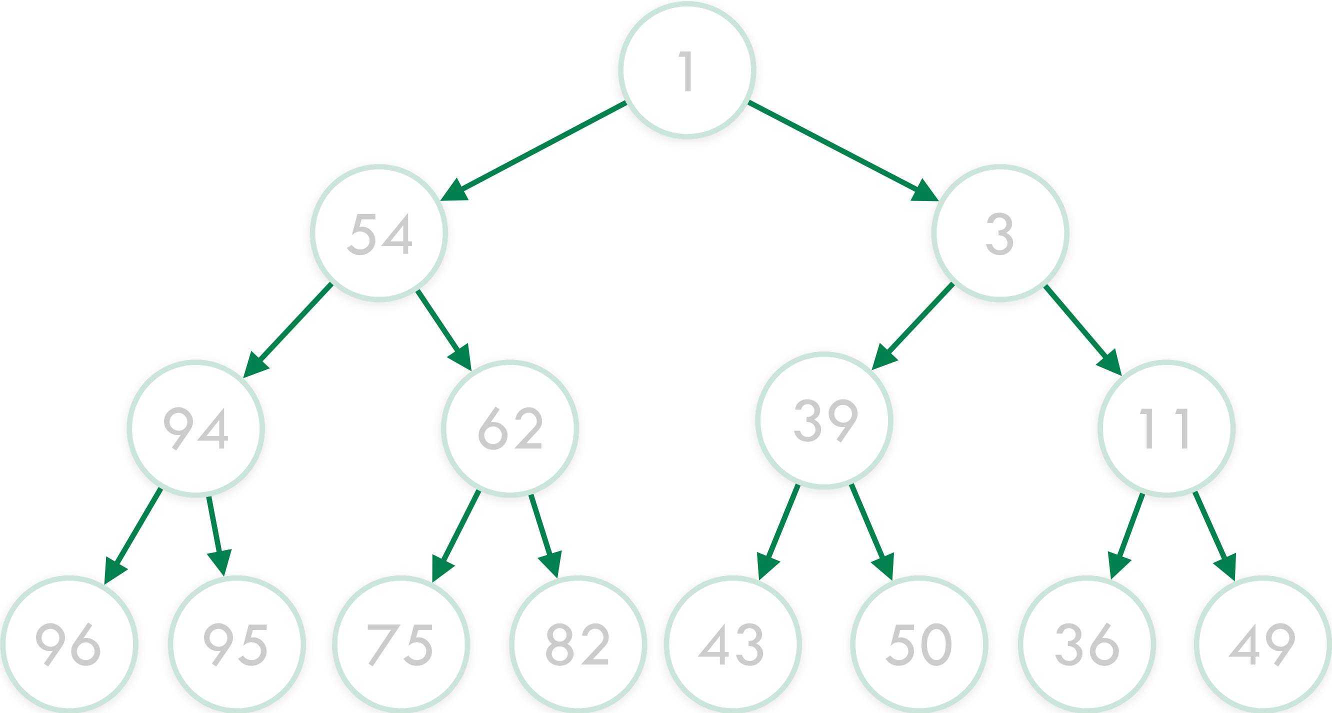


Binary heap: Array implementation

$$leftChild(i) = 2i + 1$$

$$rightChild(i) = 2i + 2$$

$$parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$$



With array starting at index 0

	1	54	3	94	62	39	11	96	95	75	82	43	50	36	49	
Indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sorting

Sorting

- - Problem: Arrange items in a collection in a specified order.
- - Lots of applications:
  - lookup / search
  - merging sequences
  - data processing
- Lots of sorting algorithms out there
- Why study sorting?

Types of sorting algorithms

1. Comparison Sorts
   - Order of elements determined by comparing them
   - Fastest comparison sort:
   - Elements should support compareTo()

$O(n \log n)$

2. Non-comparison Sorts
   - Order of elements determined by leveraging properties of input
   - Typical runtime:
   - Also called as Niche Sorts aka “linear sorts”

$O(n)$

In this class we’ll focus on comparison sorts

Insertion sort

0	1	2	3	4	5	6	7	8	9
2	3	6	7	5	1	4	10	2	8

https://visualgo.net/en/sorting

Insertion sort

0	1	2	3	4	5	6	7	8	9
2	3	6	7	5	1	4	10	2	8

- Runtime:
- Stable:
- In-place:

for i = 0 to n **do**
  $current = A[i]$
  $j = findNewIndex(current, i)$
  shift elements from j to $i - 1$ by 1
  $A[j] = current$
end for

Heap sort

- 1. Run Floyd’s buildHeap
- 2. removeMin() n times