Week 3: Wednesday

EECS 281

MAXIM ALEKSA

maximal.io

Roadmap

Wed 5/17	Ordered arrays and related algorithms	Section: Everything but sorts	
Thu 5/18	Elementary sorts		
Fri 5/19	Lab 2 and Lab 3 due		
Mon 5/22	Quicksort	Section: Sorting algorithms	Project 2 due
Tue 5/23	Mergesort		
Wed 5/24	Midterm Exam review	Section: Midterm Exam review	
Thu 5/25	Midterm Exam		
Fri 5/16	Lab 4 and Lab 5 due		

Agenda

Standard Template Library

Containers

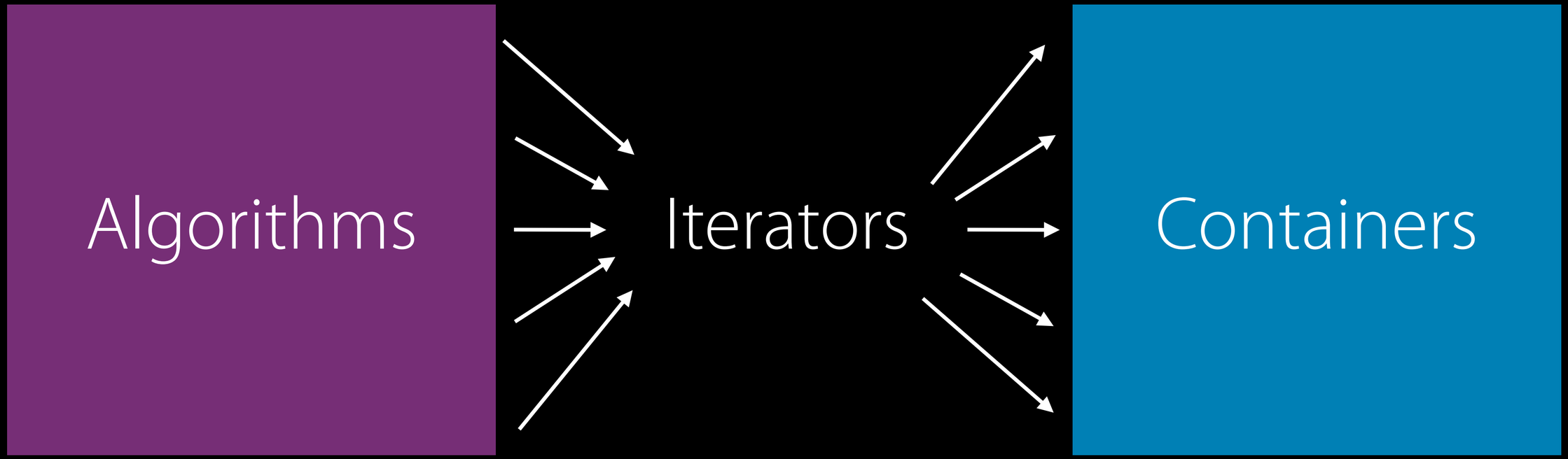
Algorithms

Strings

Heaps

Disjoint Sets

Standard Template Library



A + *C* implementations for *A* algorithms and *C* containers.

Containers

① Sequence Containers

Vector

Deque

List

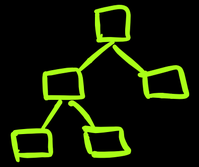
Forward List

Array

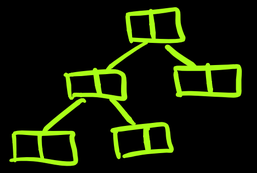
(String)

② Associative Containers

(Multi) set

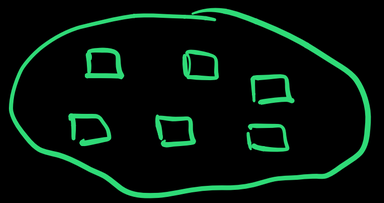


(Multi) map

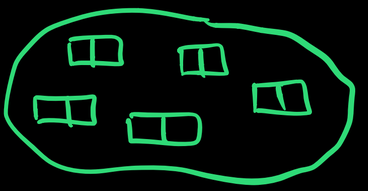


③ Unordered Associative Containers

Unordered (multi) set



Unordered (multi) map



④ Container Adapters

stack, queue, priority queue

Containers

An **ordered** container maintains elements in a specific order. The order is independent of the value.

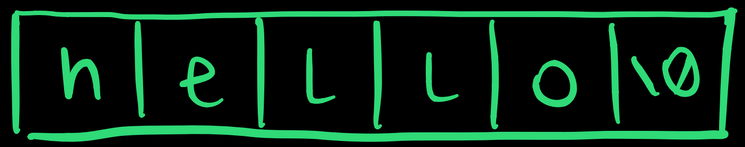
A **sorted** container maintains elements in a specific order that depends on the values of the elements.

In contrast, a collection without any order can maintain the elements in any order.

Strings

C++ string class

```
string s = "hello";
```



`s.size()`
`s.length()` → 5

"valid" to do `s[s.size()]` → '\0'

C-style string

```
const char * s = "hello";
```

reverse.cpp

Interview Question

Write a program that prompts the user for a message, and then outputs the message with all the characters in reversed order.

```
Give me a string: I solemnly swear that I am up to no good
doog on ot pu ma I taht raews ylnmelos I
```

Anagrams

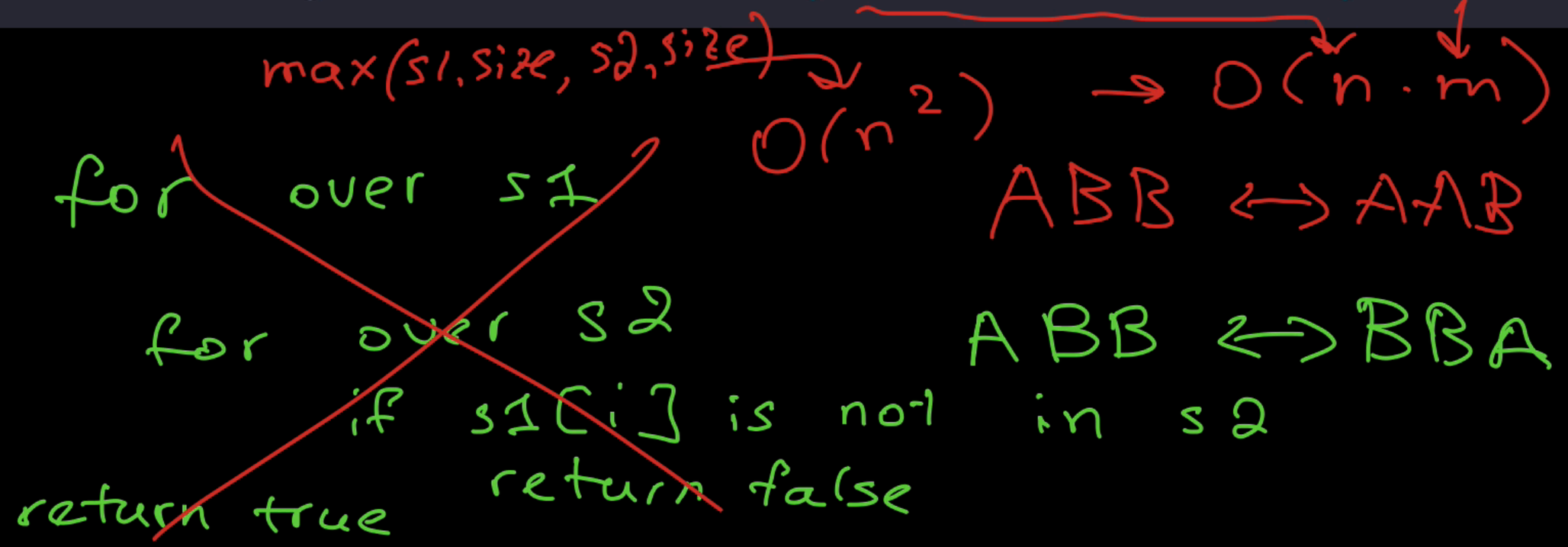
Implement **`areAnagrams`**, a function that checks if s1 and s2 are anagrams of each other.

```
bool areAnagrams(const string& s1, const string& s2);
```

Anagrams

Implement **`areAnagrams`**, a function that checks if s1 and s2 are anagrams of each other.

```
bool areAnagrams(const string& s1, const string& s2);
```



Anagrams

```
bool areAnagram(const string& s1, const string& s2) {
    string temp1 = s1;
    string temp2 = s2;
    sort (temp1.begin(), temp1.end());
    sort (temp2.begin(), temp2.end());
    return (temp1 == temp2);
}
```

Anagrams

```
bool areAnagram(const string& s1, const string& s2) {
    string temp1 = s1;
    string temp2 = s2;
    sort (temp1.begin(), temp1.end());
    sort (temp2.begin(), temp2.end());
    return (temp1 == temp2);
}
```

check size! Θ(1)

O(n)

O(m)

O(n log n)

O(m log m)

O(n+m)

O(n log n + m log m)

n = m

O(n log n)

ABB

BBA

ABB

Anagrams

```
bool areAnagram(const string& s1, const string& s2) {
    string temp1 = s1;
    string temp2 = s2;
    sort (temp1.begin(), temp1.end());
    sort (temp2.begin(), temp2.end());
    return (temp1 == temp2);
}
```

ABB

65+66+66 = . . .

66+66+65 = . . .

BBA

prime #s

65+65+67

A A C

Anagrams

```
bool areAnagram(const string& s1,
                const string& s2) {

    string temp1 = s1;

    string temp2 = s2;

    sort (temp1.begin(), temp1.end());

    sort (temp2.begin(), temp2.end());

    return (temp1 == temp2);

}
```

```
bool areAnagrams(const string& s1, const string& s2) {
  std::map<char, int> letter_inventory;
  for (size_t i=0; i<s1.size(); ++i) {
    if (letter_inventory.find(s1[i]) == letter_inventory.end()) {
      letter_inventory.insert(std::pair<s1[i], 1>);
    }
    else {
      (letter_inventory.find(s1[i]) -> second) ++;
    }
  }
  for (size_t i=0; i<s2.size(); ++i) {
    if (letter_inventory.find(s2[i]) == letter_inventory.end()) {
      return false;
    }
    else {
      (letter_inventory.find(s2[i]) -> second) --;
    }
  }
  std::map<char, int>::iterator itr = letter_inventory.begin();
  while (itr != letter_inventory.end()) {
    if (itr -> second != 0) {
      return false;
    }
  }
  return true;
}
```

Anagrams

```
bool areAnagram(const string& s1,
                const string& s2) {

    string temp1 = s1;

    string temp2 = s2;

    sort (temp1.begin(), temp1.end());

    sort (temp2.begin(), temp2.end());

    return (temp1 == temp2);

}
```

A: 1
B: 2
O(n+m)

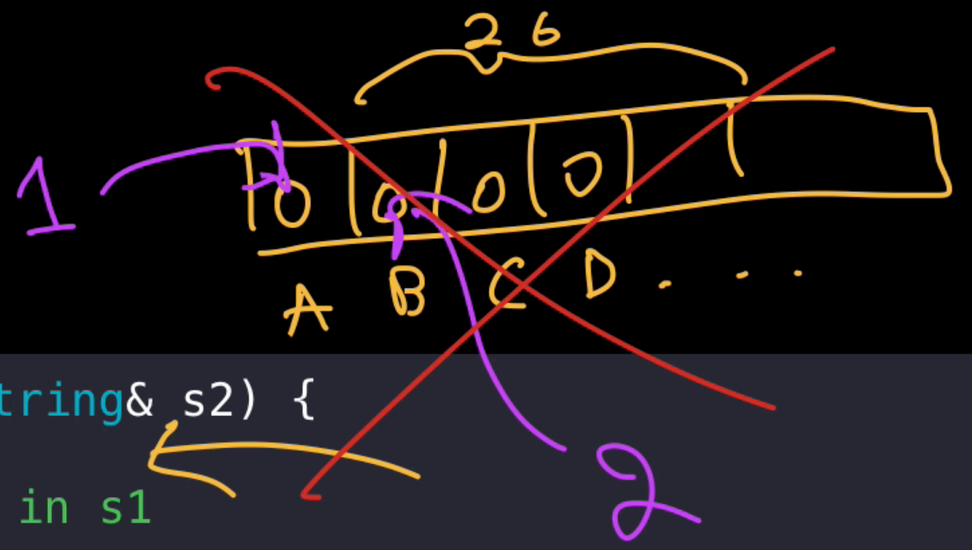
store values at unique keys in $\Theta(1)$

```
bool areAnagrams(const string& s1, const string& s2) {
  std::map<char, int> letter_inventory;
  for (size_t i=0; i<s1.size(); ++i) {
    if (letter_inventory.find(s1[i]) == letter_inventory.end()) {
      letter_inventory.insert(std::pair<s1[i], 1>);
    }
    else {
      (letter_inventory.find(s1[i]) -> second) ++;
    }
  }
  for (size_t i=0; i<s2.size(); ++i) {
    if (letter_inventory.find(s2[i]) == letter_inventory.end()) {
      return false;
    }
    else {
      (letter_inventory.find(s2[i]) -> second) --;
    }
  }
  std::map<char, int>::iterator itr = letter_inventory.begin();
  while (itr != letter_inventory.end()) {
    if (itr -> second != 0) {
      return false;
    }
  }
  return true;
}
```

Anagrams

```
bool areAnagrams(const string& s1, const string& s2) {
    int characterCounts[CHAR_MAX] = {};
    // count occurrences of each character in s1
    for (char c : s1) {
        characterCounts[c] += 1;
    }
    // subtract occurrences of each character in s2
    for (char c : s2) {
        characterCounts[c] -= 1;
    }
    // verify matching counts
    for (int i = 0; i < CHAR_MAX; i += 1) {
        if (characterCounts[i] != 0) {
            return false;
        }
    }
    return true;
}
```

Anagrams



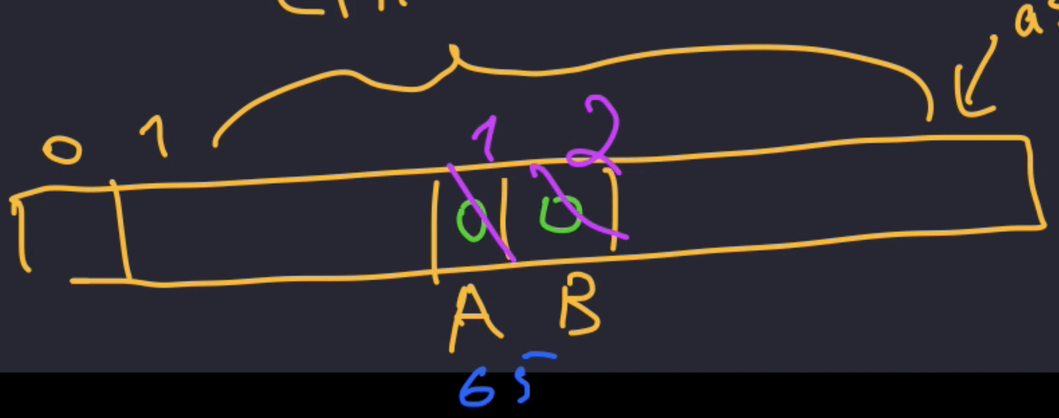

```
bool areAnagrams(const string& s1, const string& s2) {
    int characterCounts[CHAR_MAX] = {};
    // count occurrences of each character in s1
    for (char c : s1) {
        characterCounts[c] += 1;
    }
    // subtract occurrences of each character in s2
    for (char c : s2) {
        characterCounts[c] -= 1;
    }
    // verify matching counts
    for (int i = 0; i < CHAR_MAX; i += 1) {
        if (characterCounts[i] != 0) {
            return false;
        }
    }
    return true;
}
```

'A' = 65

$\Theta(n+m)$

chars range 0...255

CHAR_MAX = 256



Algorithms

```
#include <algorithm>
```

Algorithms

(Mostly loops)

```
min_element
sort
reverse
copy
find_if
for_each
```

…

#include <functional>

```
int result = multiplies<int>()(3, 4);

if (not_equal_to<int>()(result, 10)) {
    cout << result << endl;
}
```

#include <functional>

```
int result = multiplies<int>()(3, 4);

if (not_equal_to<int>()(result, 10)) {
    cout << result << endl;
}
```

3 x 4 ⇒ 12

12

12

Combining everything

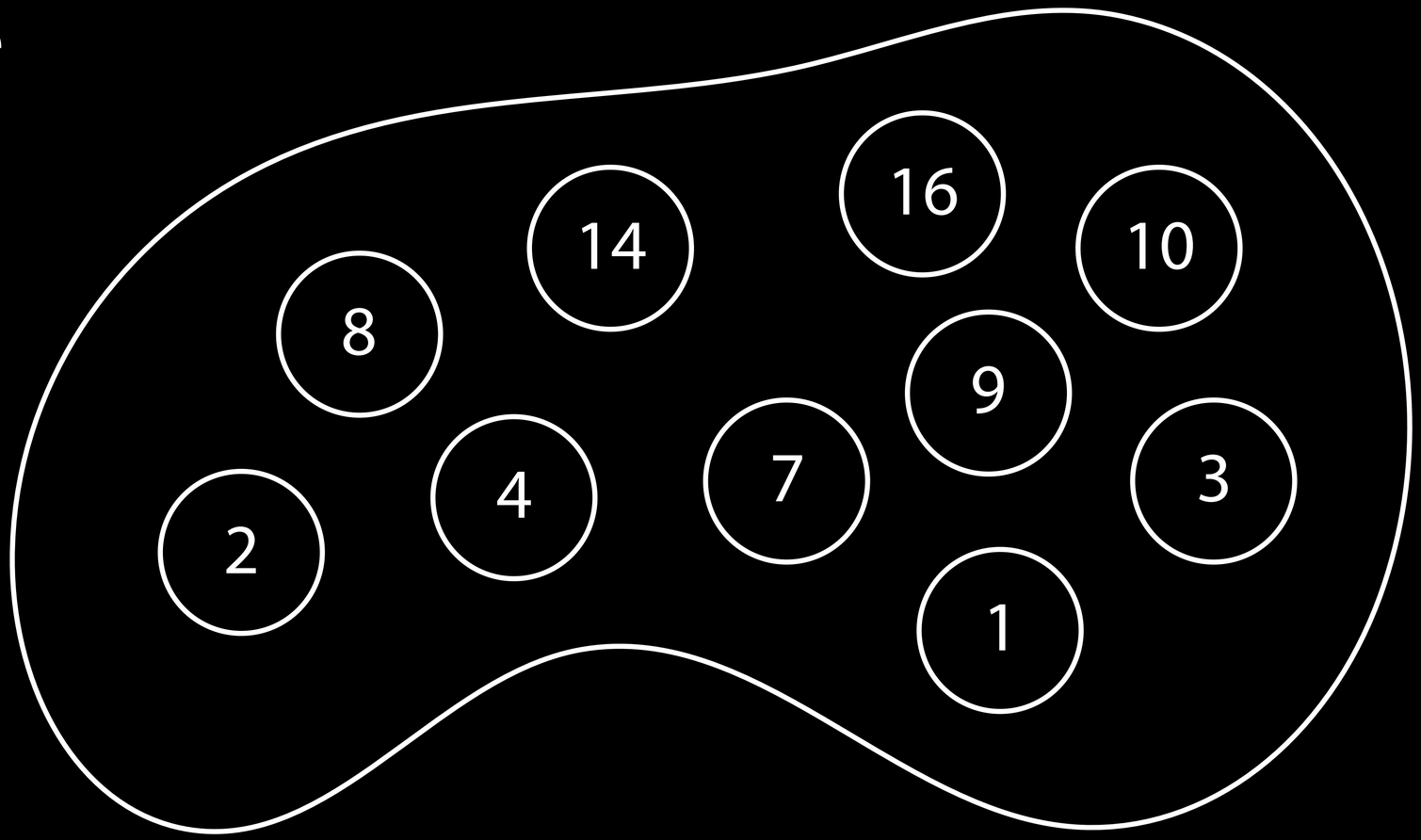
```
set<int> numberSet = {1, 2, 3, 4, 5};
vector<int> someVector;

// multiply numberSet's elements by 10 and
// save them in someVector
transform(numberSet.begin(), numberSet.end(),
          back_inserter(someVector),
          bind(multiplies<int>(),
          placeholders::_1, 10));
```

Priority Queue

Abstract data structure

Operations:

insert
maxElement
removeMaxElement



Priority Queue

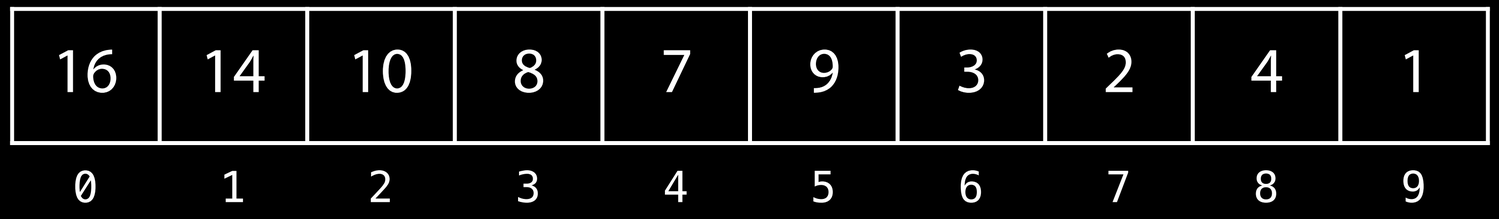
An abstract data structure implementing a set ***S*** of elements, each associated with a key, supporting these operations.

insert(*S*, x)	insert element ***x*** into set ***S***
max(*S*)	return element of ***S*** with largest key
removeMax(*S*)	return element of ***S*** with largest key and remove it from ***S***
increaseKey(*S*, *x*, *k*)	increase the value of element ***x***'s key to new value ***k*** (assumed to be as large as current value)

Heaps

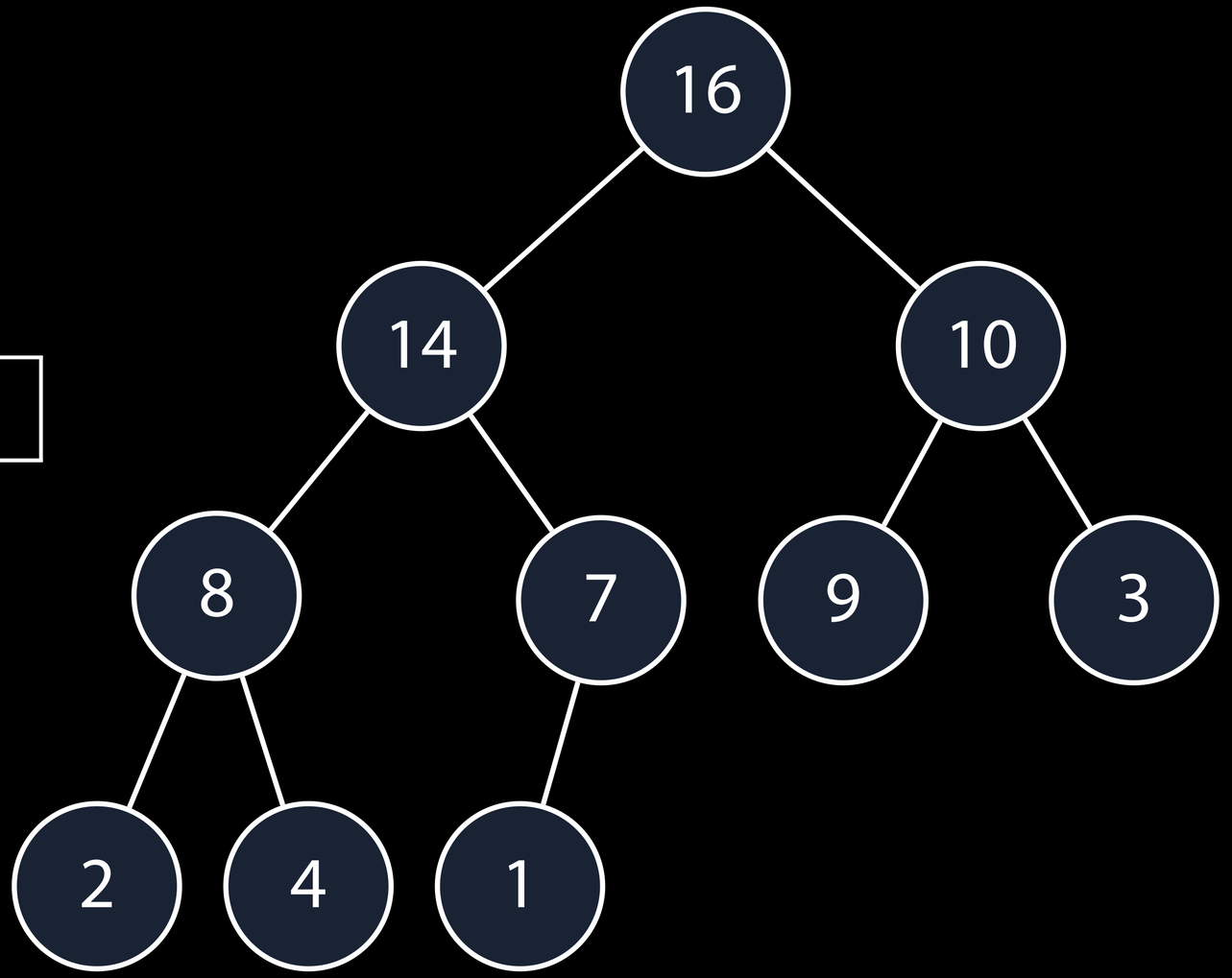
Heap

An implementation of a priority queue using an array that represents a nearly complete binary tree.



Heap

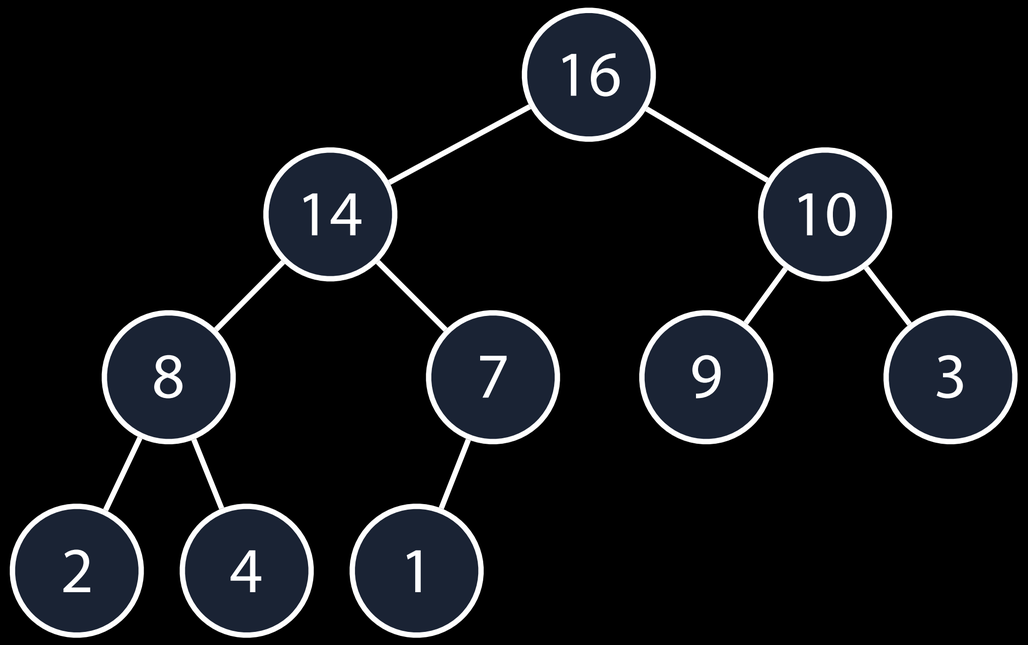
16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9



Heap property

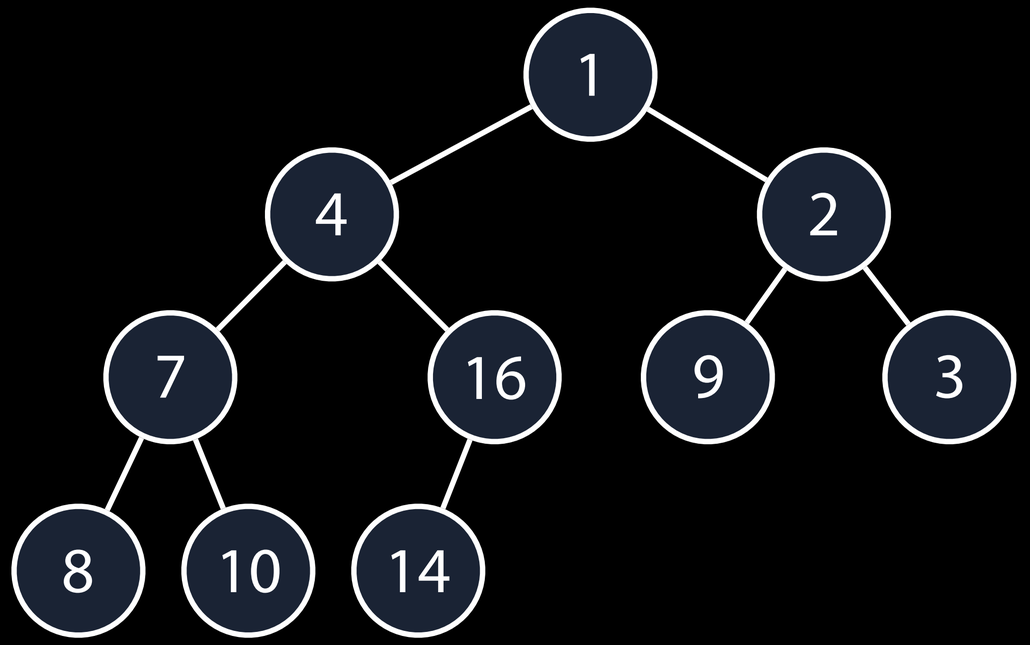
Max-heap

The key of a node is ≥ than the keys of its children



Min-heap

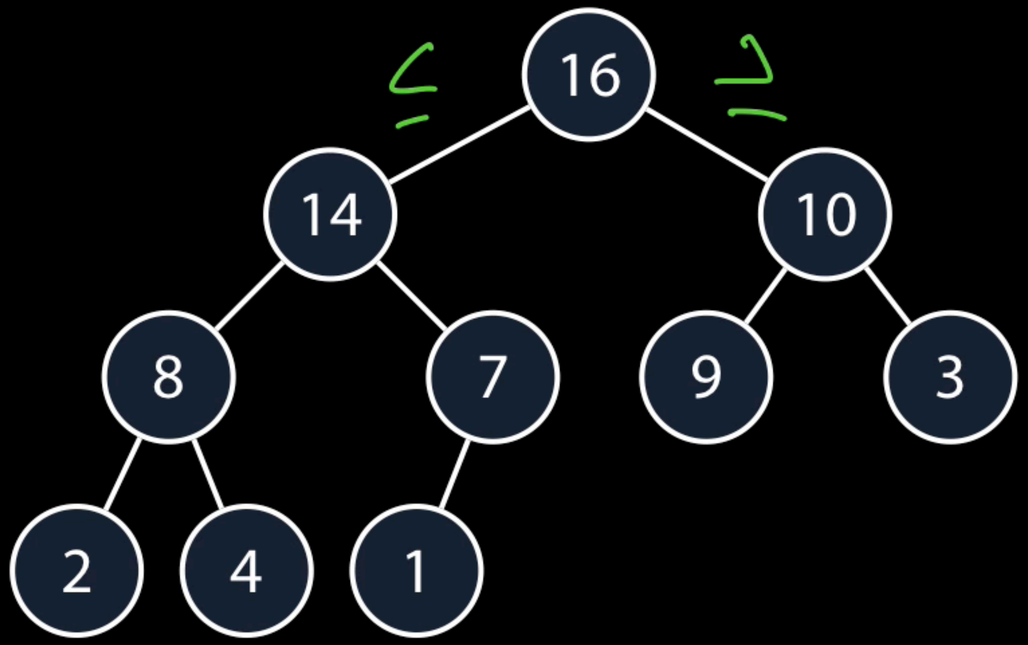
The key of a node is ≥ than the keys of its children



Heap property

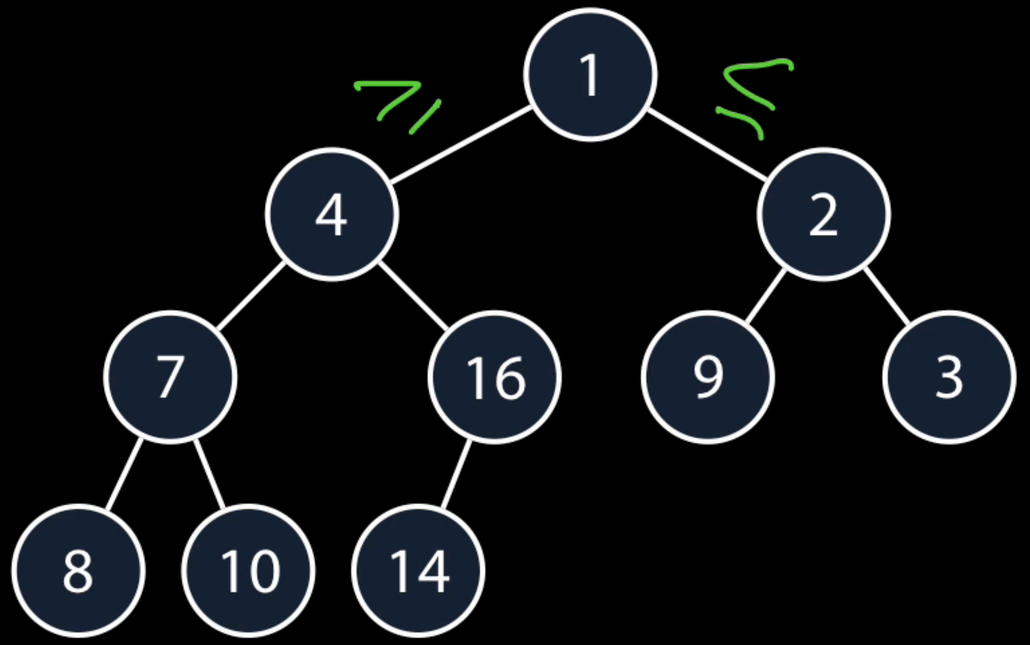
Max-heap

The key of a node is ≥ than the keys of its children



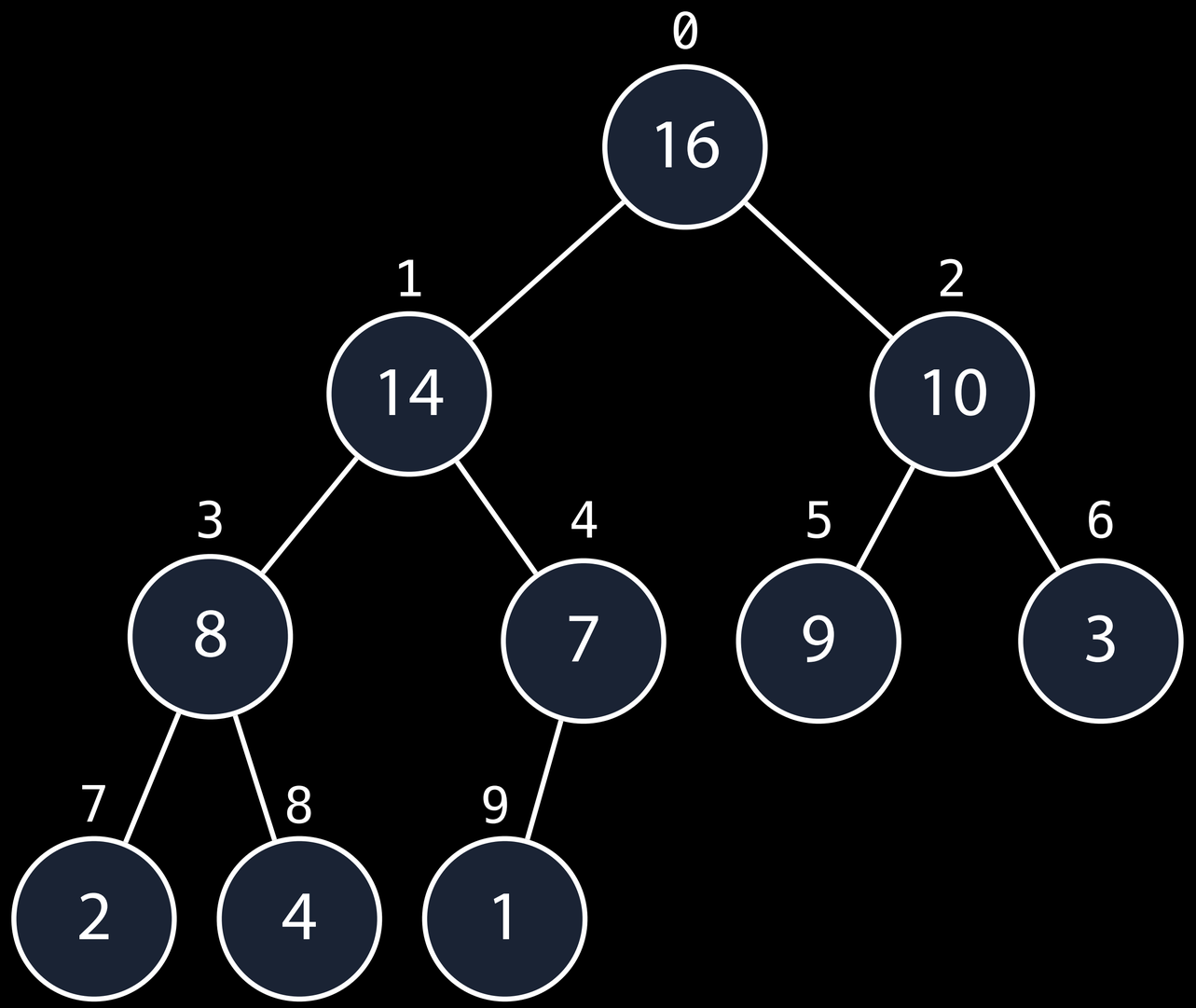
Min-heap

The key of a node is ≥ than the keys of its children



Heap as a tree

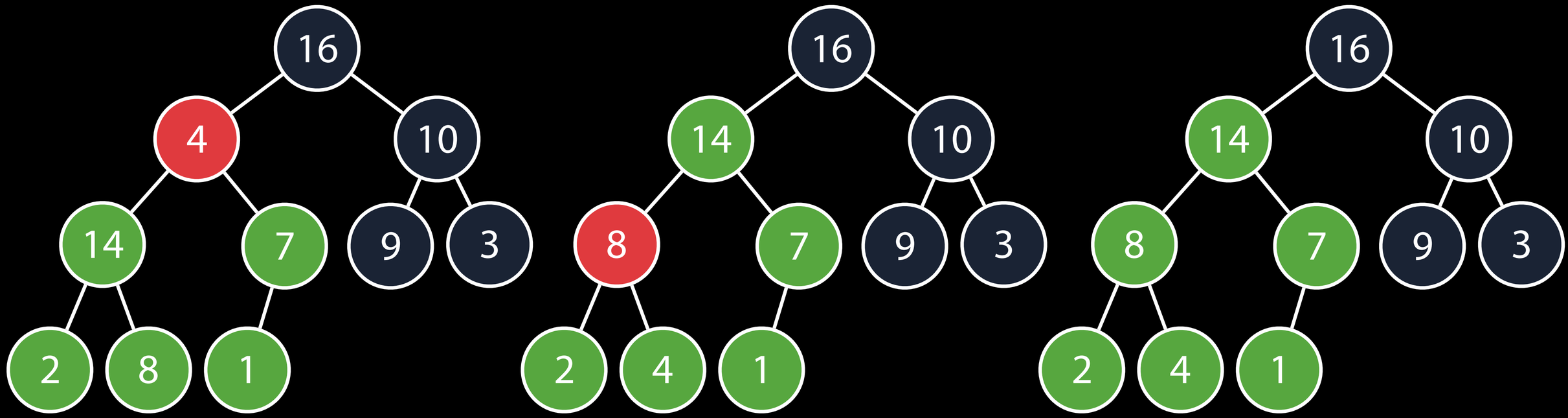
```
root(): A[0]

parent(i): (i – 1) / 2

left(i): 2i + 1

right(i): 2i + 2

height(): ⌈log(n + 1)⌉
```



Heap operations

buildMaxHeap	produce a max-heap from an unordered array
maxHeapify(*i*)	correct a single violation of the heap property in a subtree with root at ***i***
maxElement	return the largest element
removeMaxElement	remove the largest element
increaseKey(*i*, *k*)	update the key of an element to a larger value ***k***
insert(*k*)	add an element with key ***k***

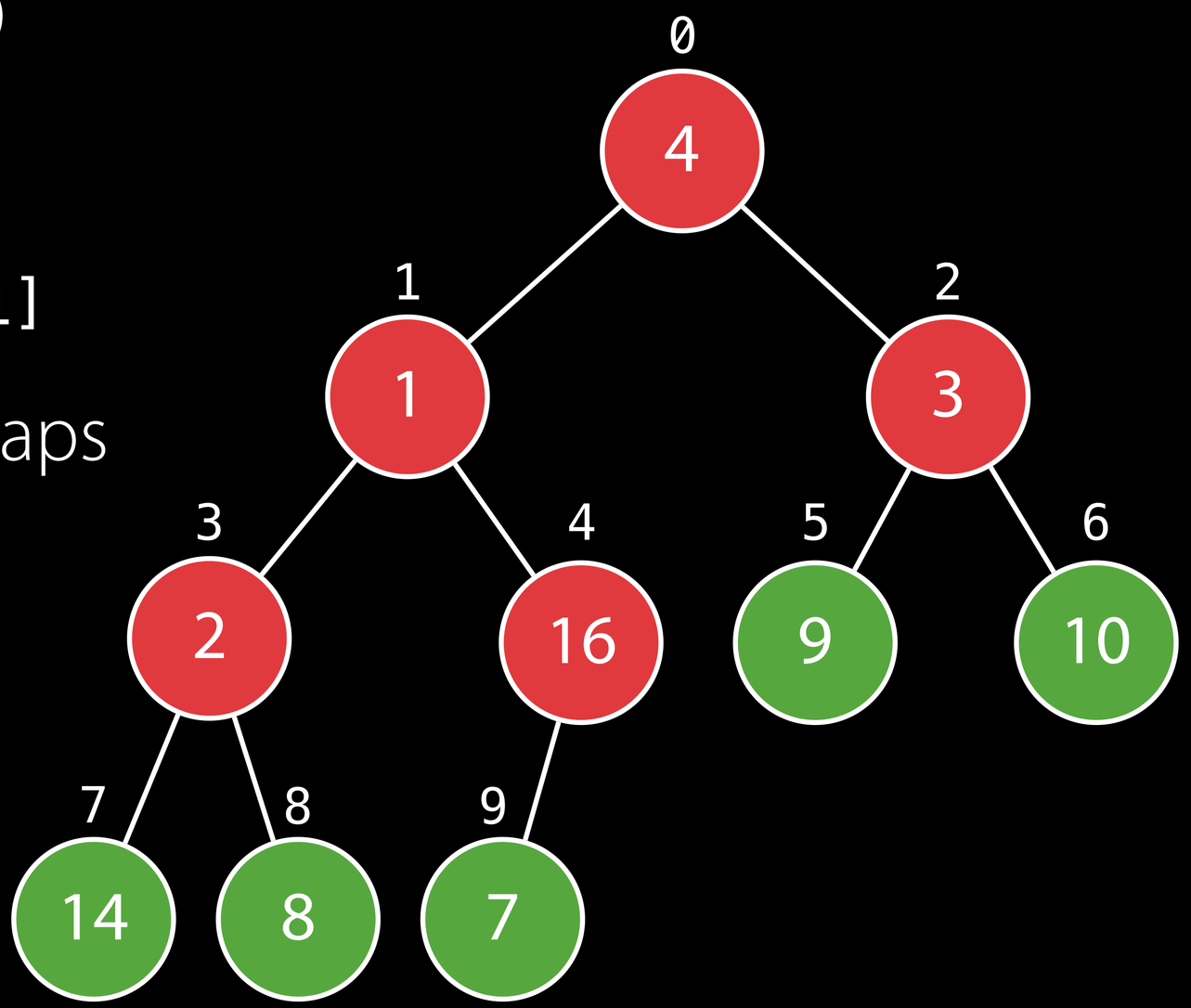
maxHeapify



buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n - 1]`

Leaves are always valid max-heaps



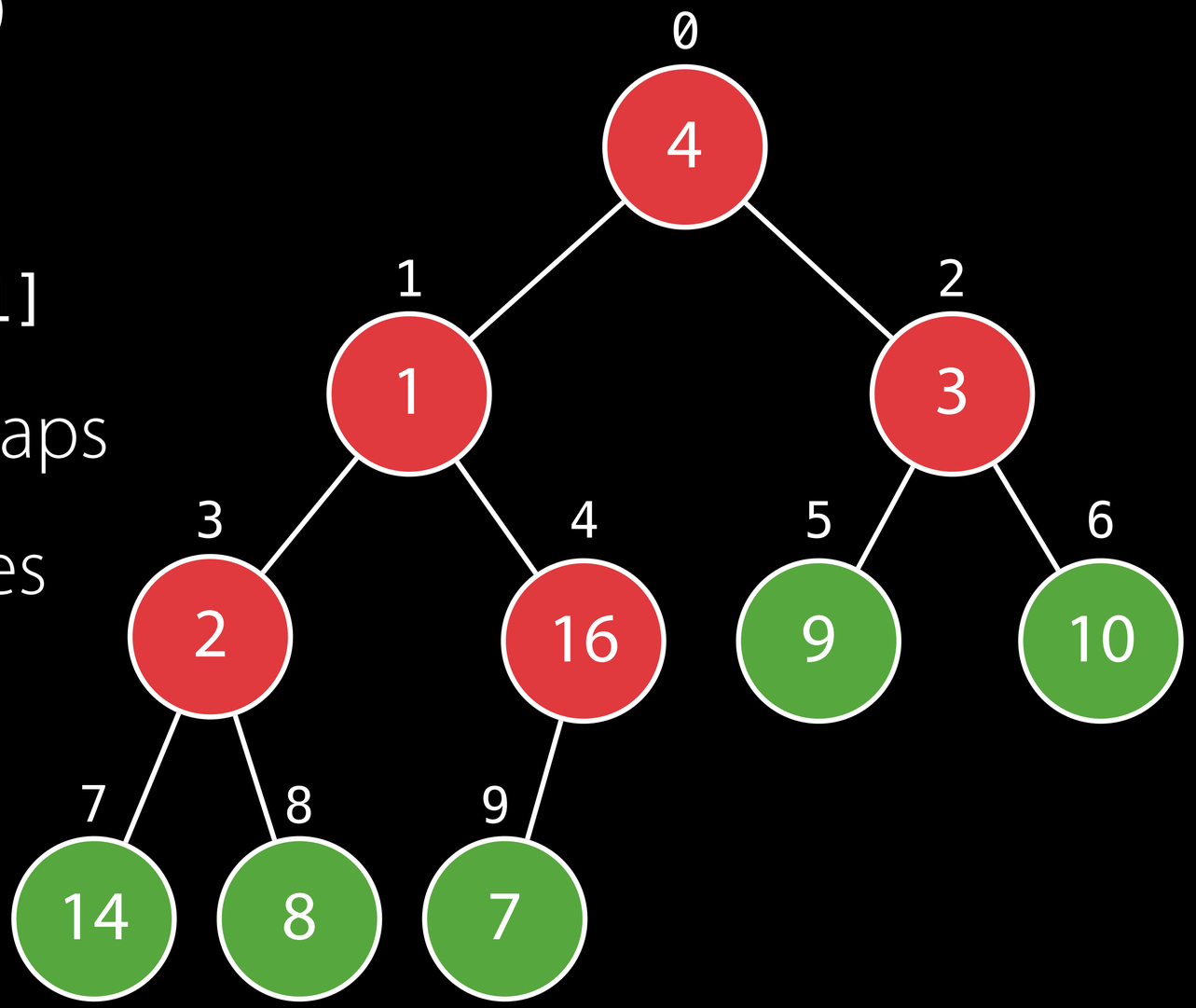
buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n - 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n - 1) / 2...0
    maxHeapify(i)
```



buildMaxHeap

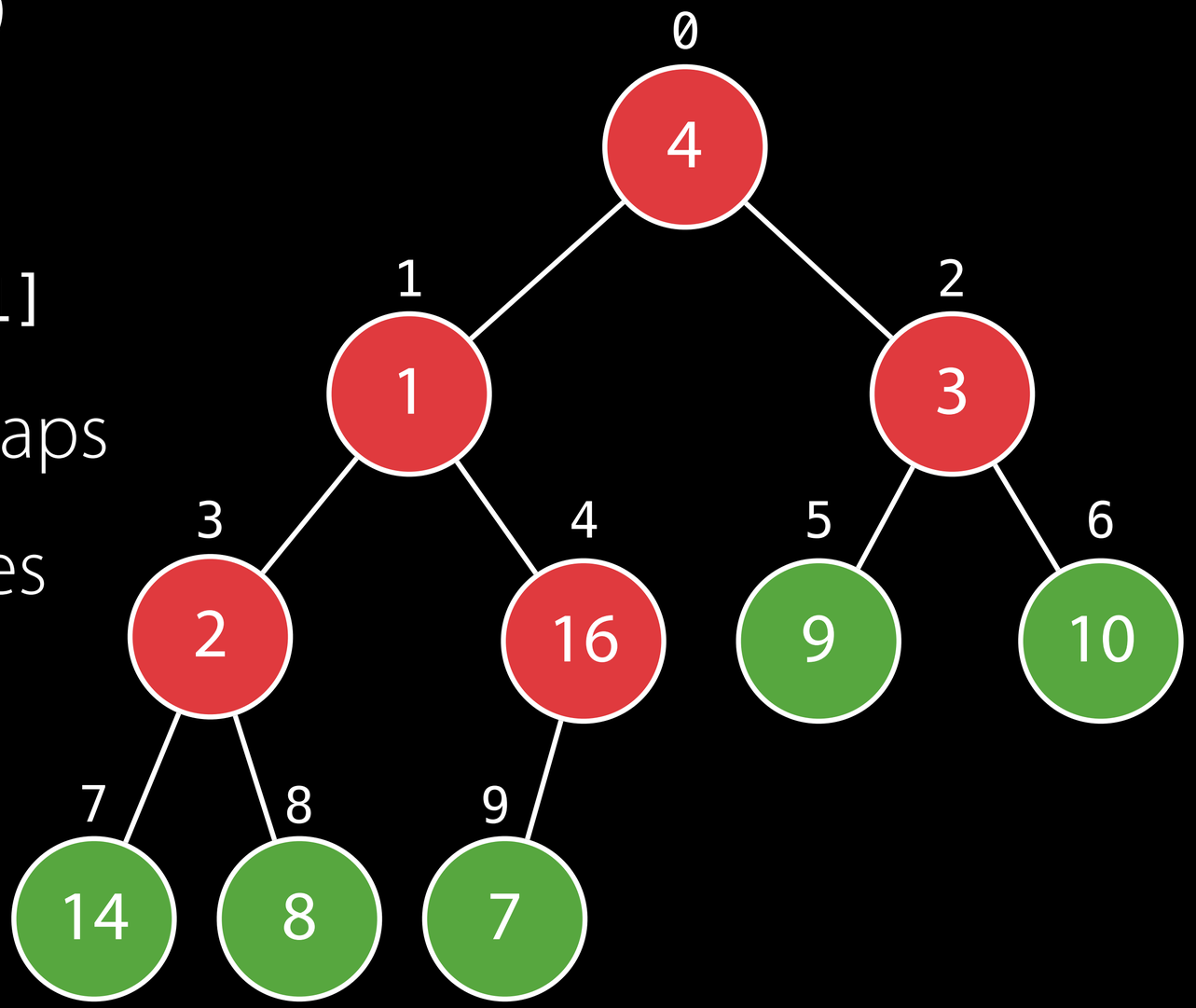
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(4)`



buildMaxHeap

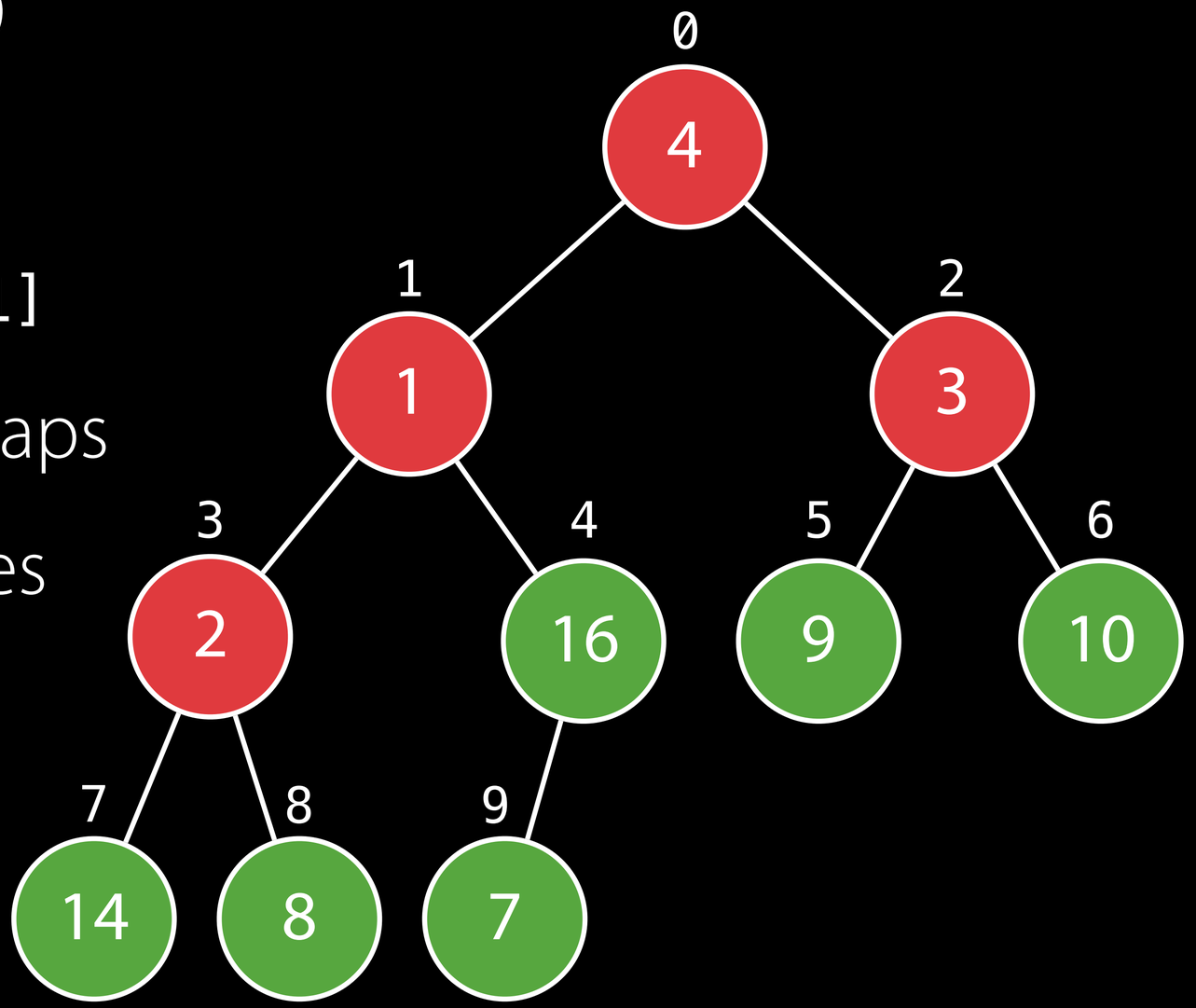
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(4)`



buildMaxHeap

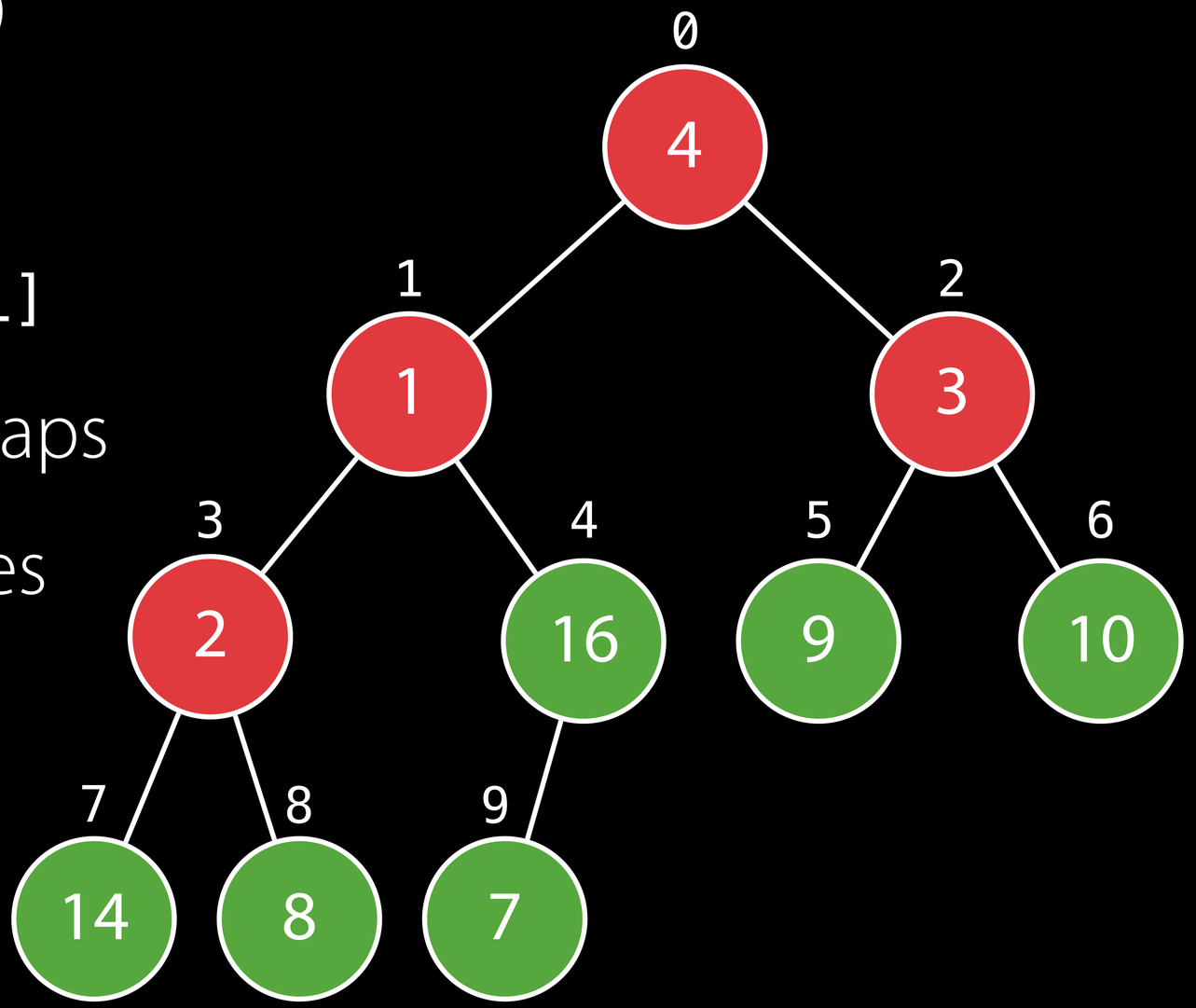
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(3)`



buildMaxHeap

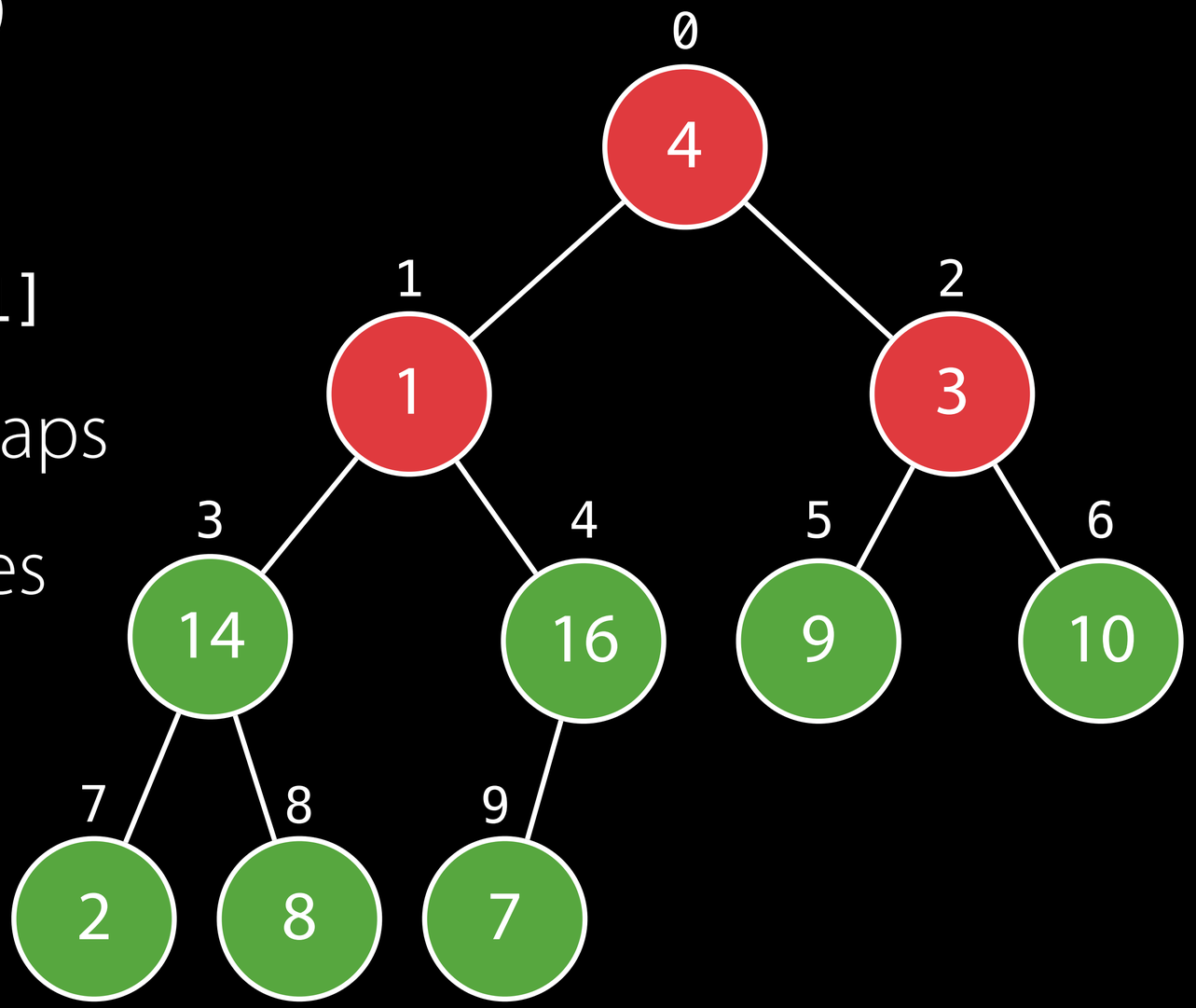
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(3)`



buildMaxHeap

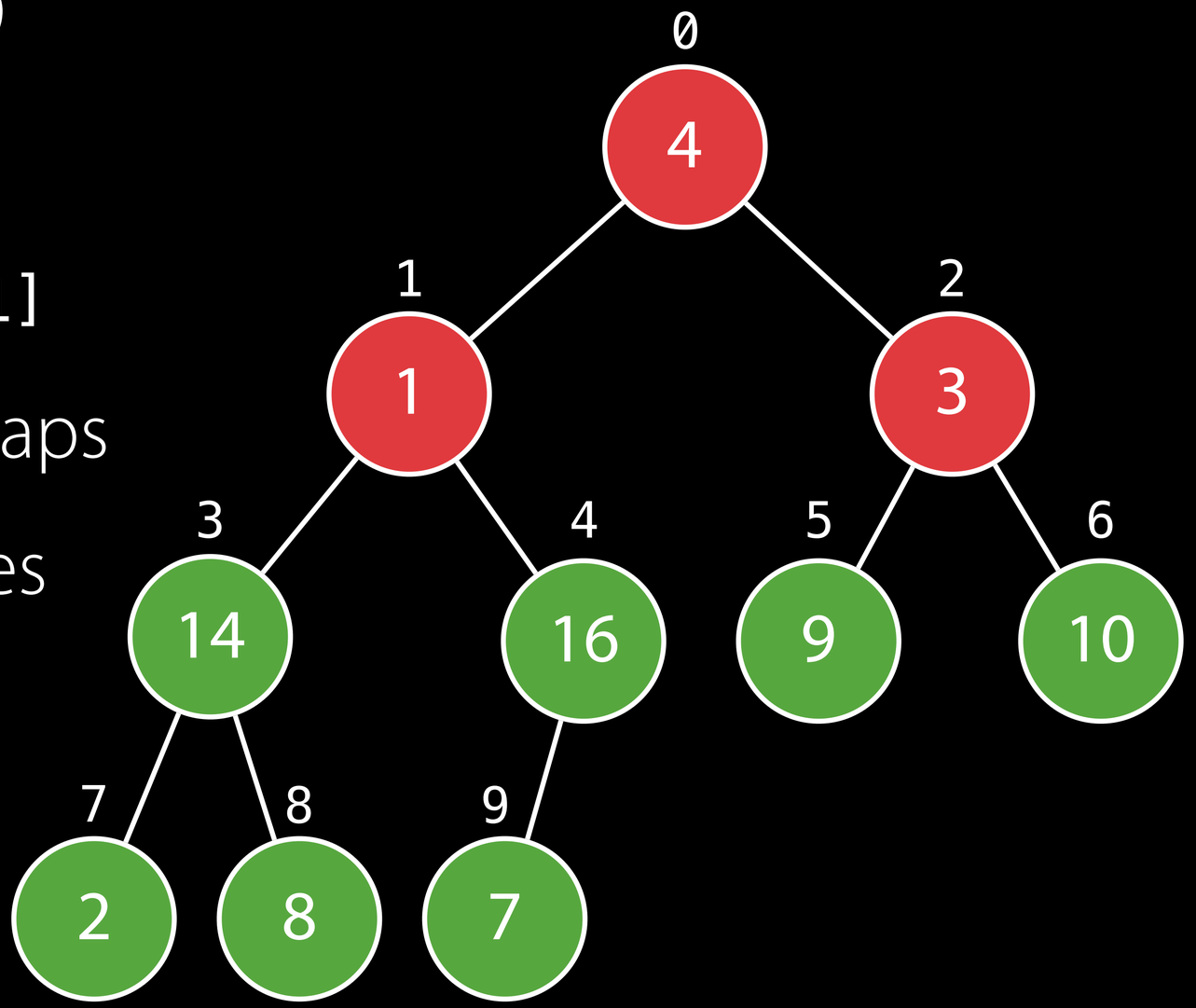
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(2)`



buildMaxHeap

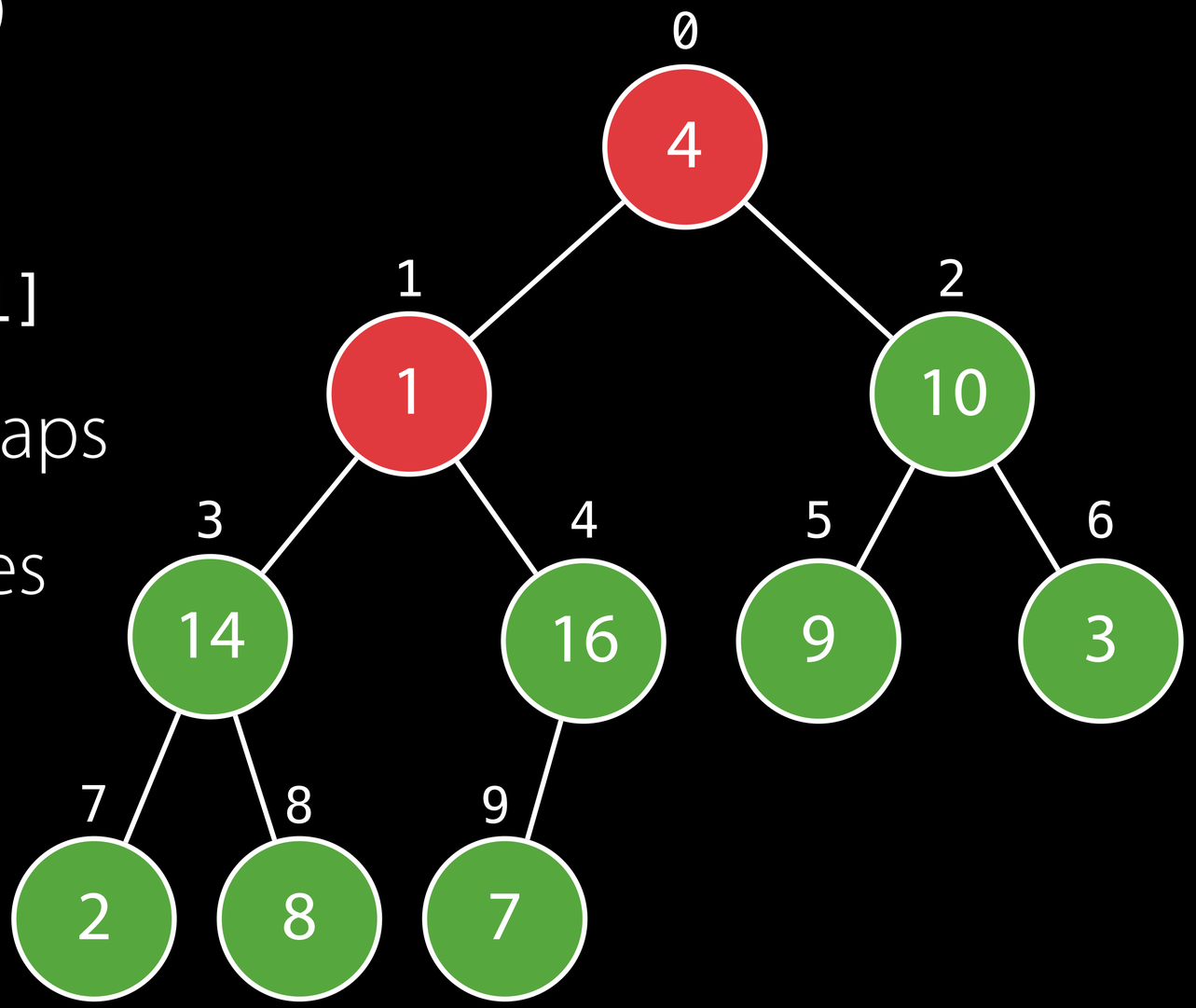
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(2)`



buildMaxHeap

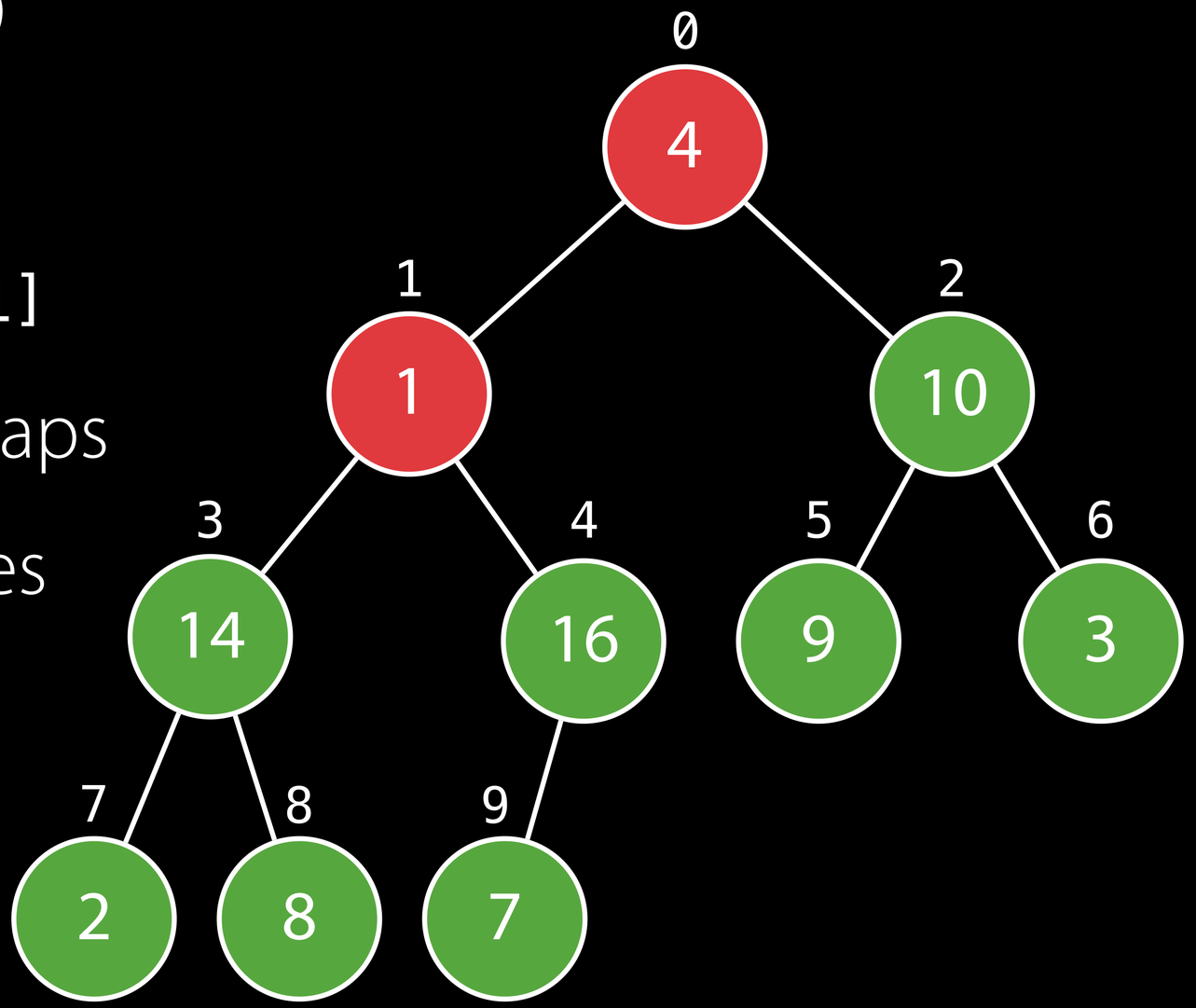
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(1)`



buildMaxHeap

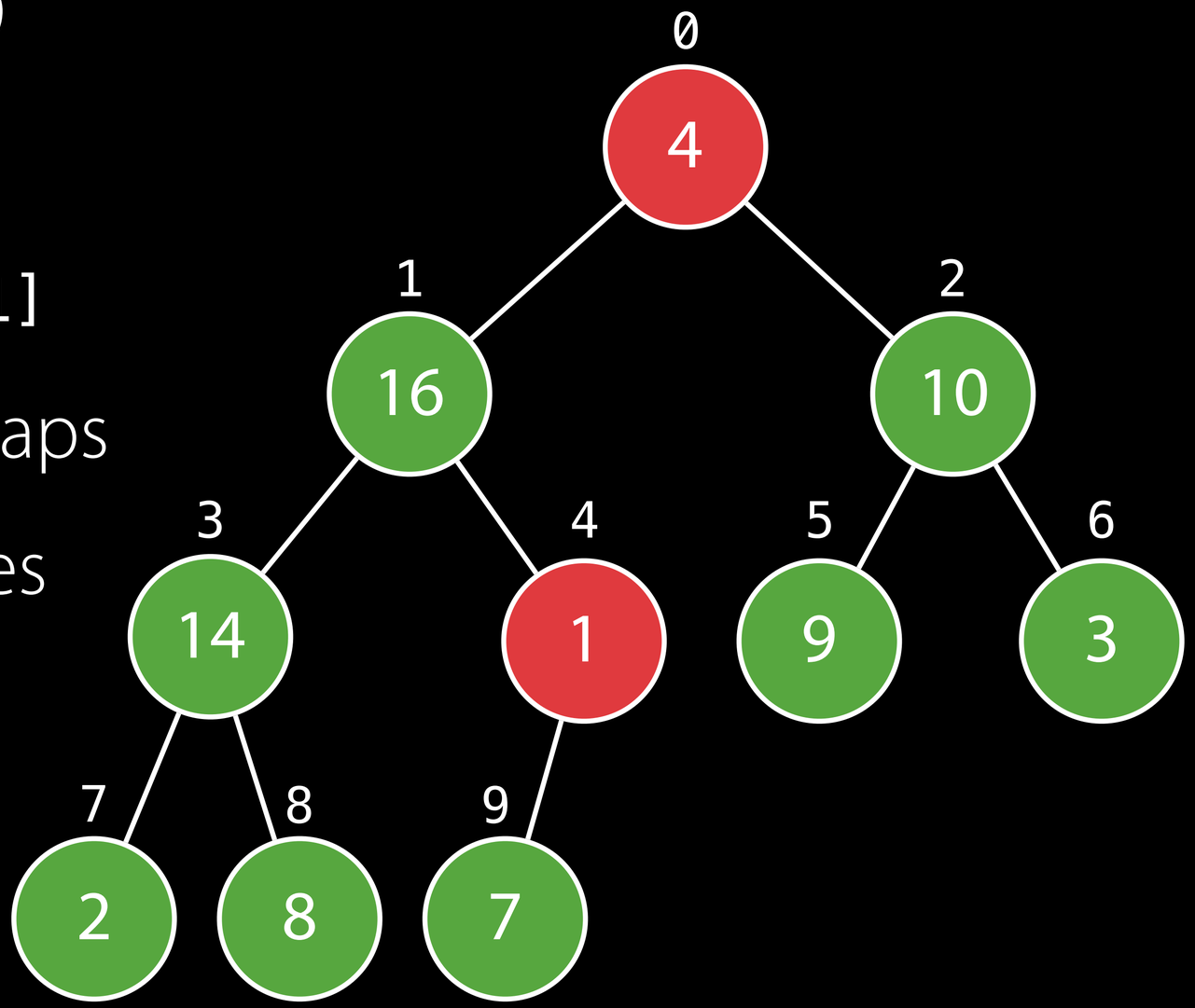
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(1)`



buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n – 1]`

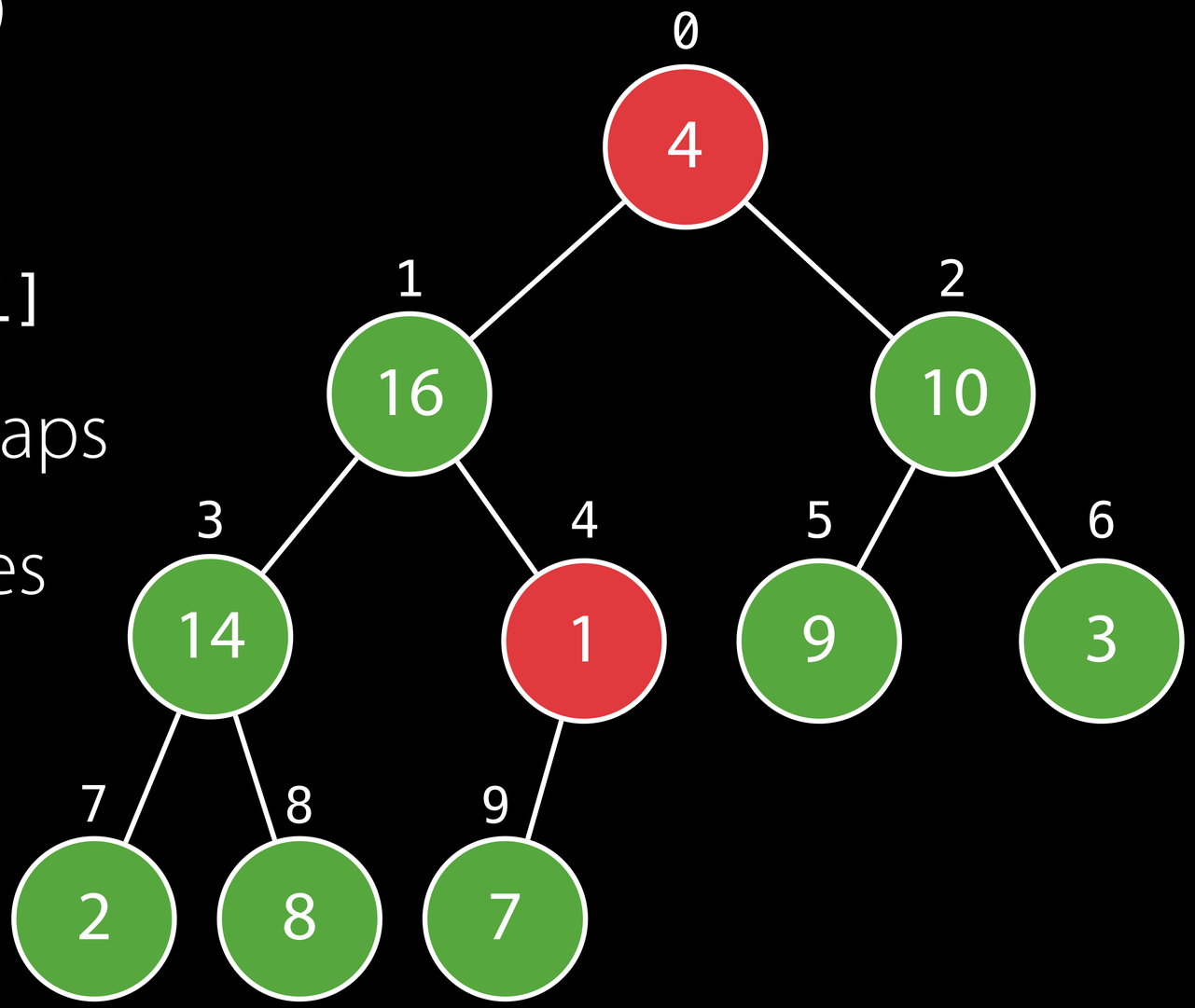
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(1)`

`maxHeapify(4)`



buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n – 1]`

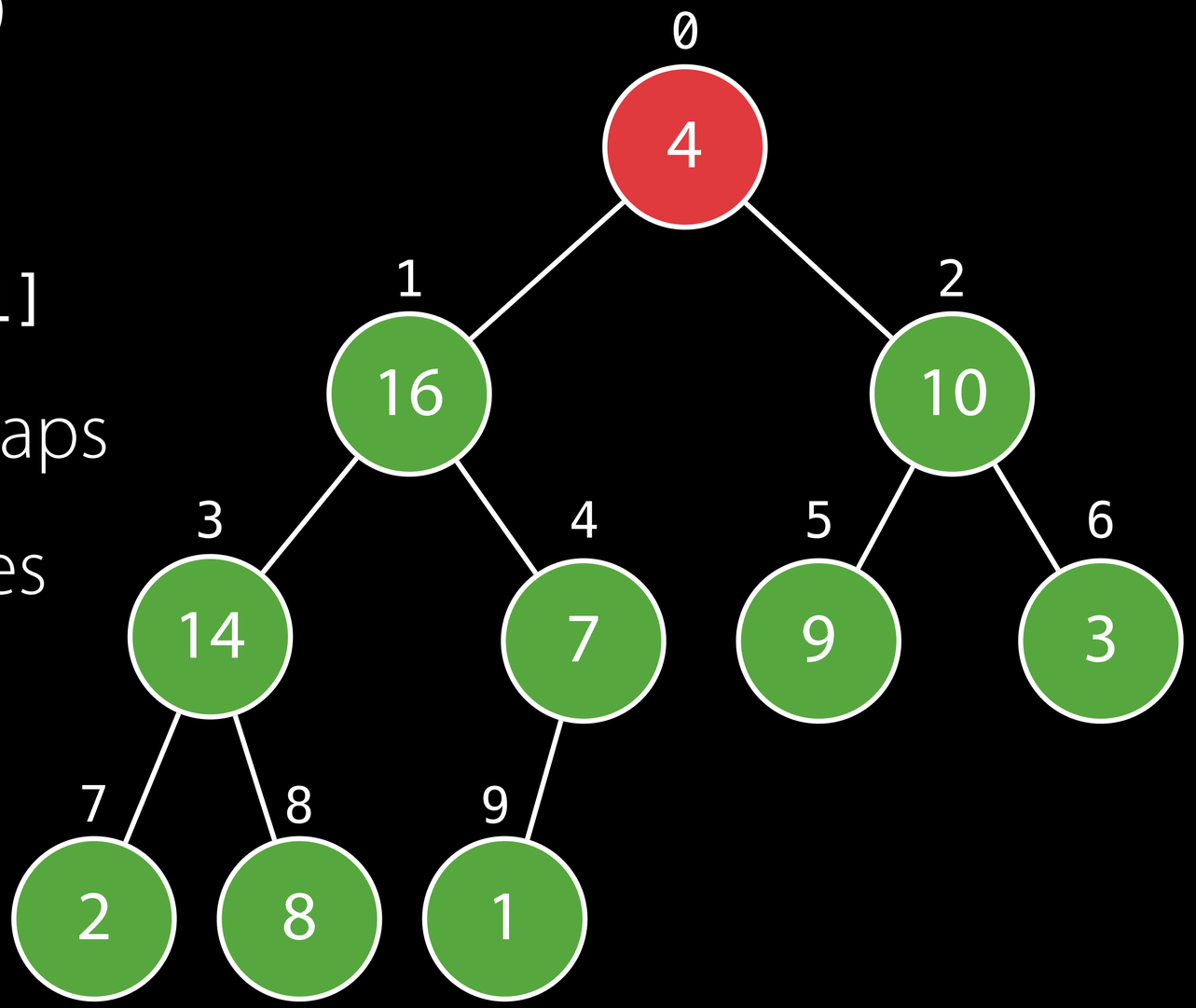
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(1)`

`maxHeapify(4)`



buildMaxHeap

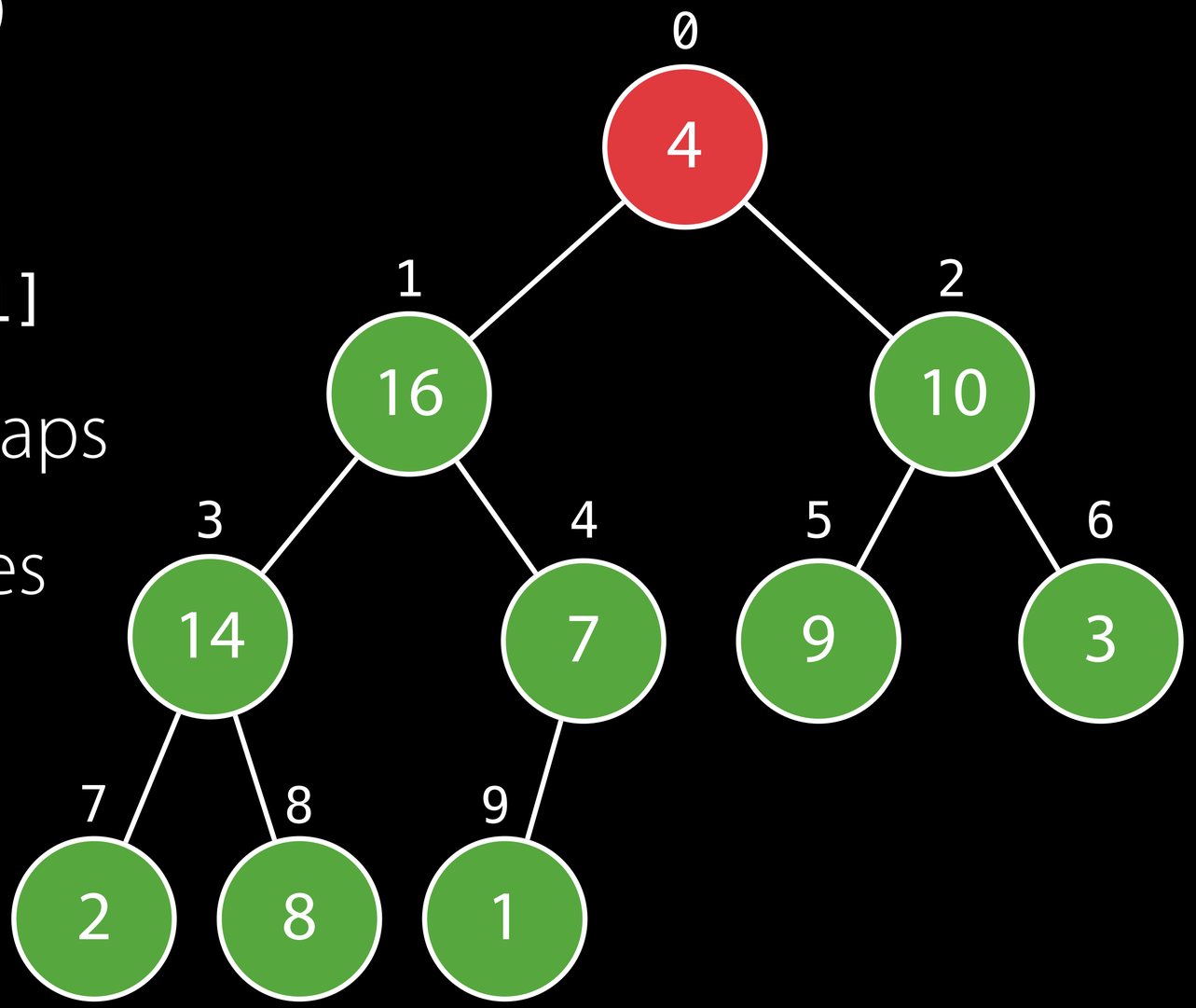
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(0)`



buildMaxHeap

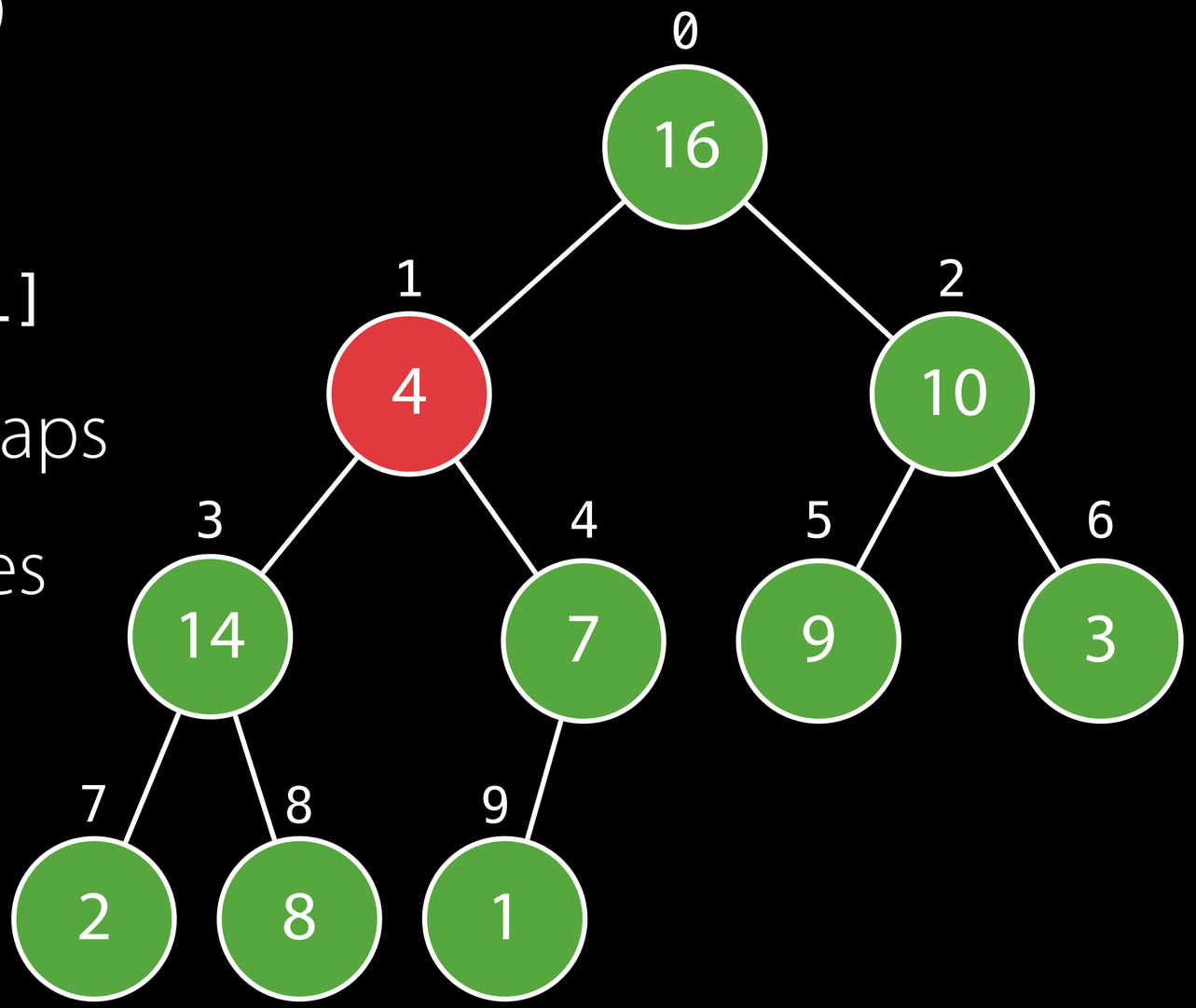
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(0)`



buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n - 1]`

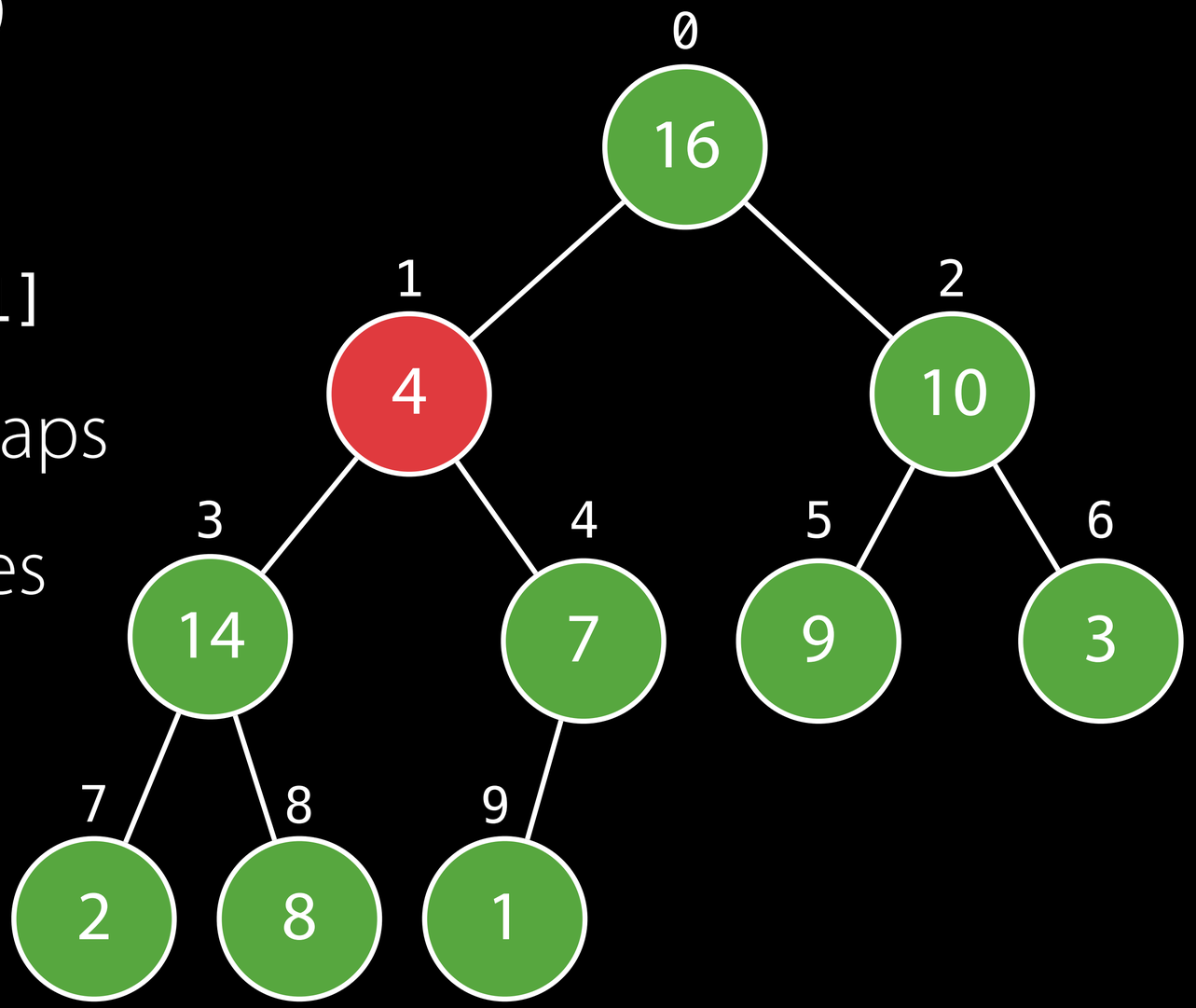
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n - 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(0)`

`maxHeapify(1)`



buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n - 1]`

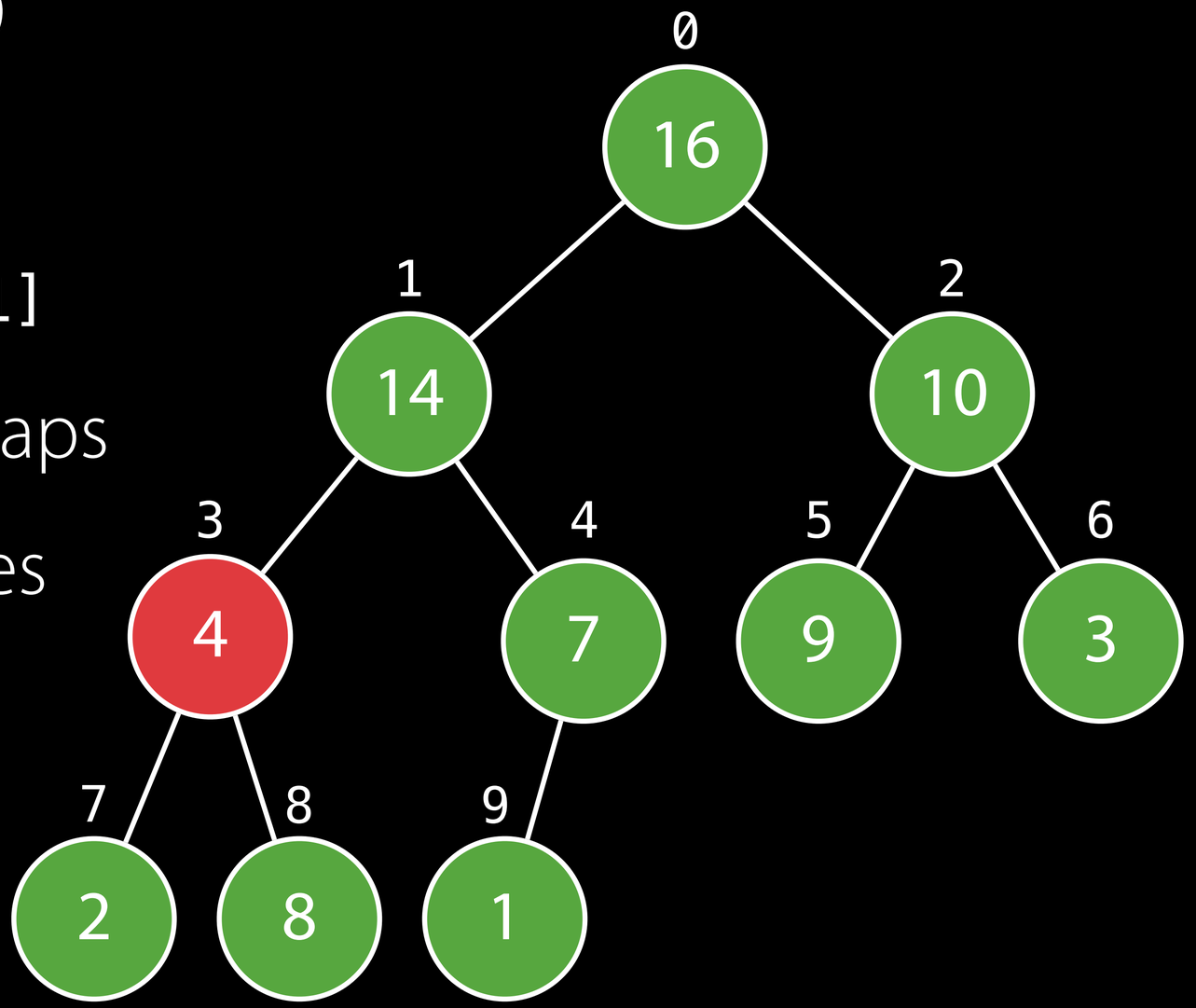
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n - 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(0)`

`maxHeapify(1)`



buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n – 1]`

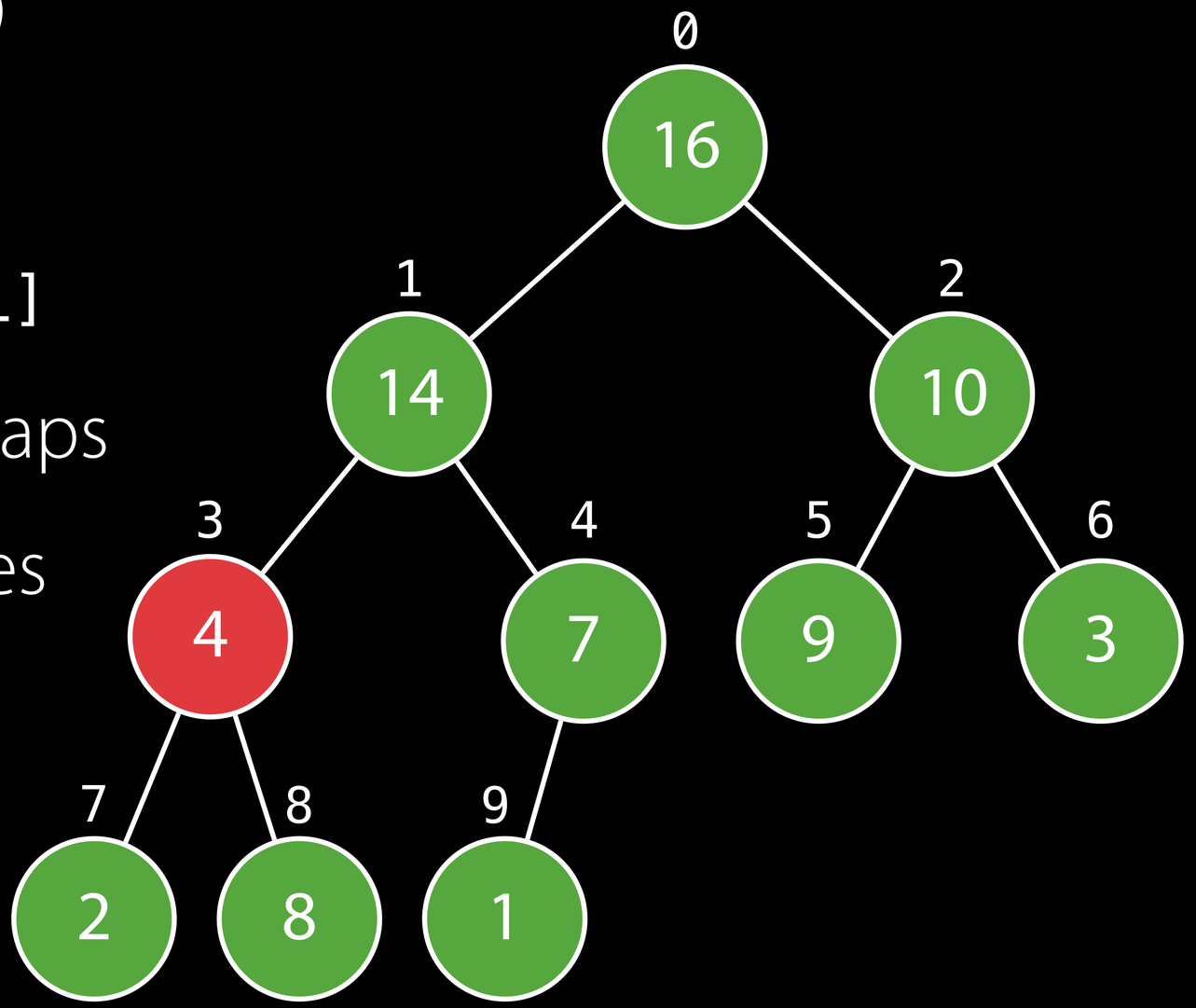
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(0)`

`maxHeapify(3)`



buildMaxHeap

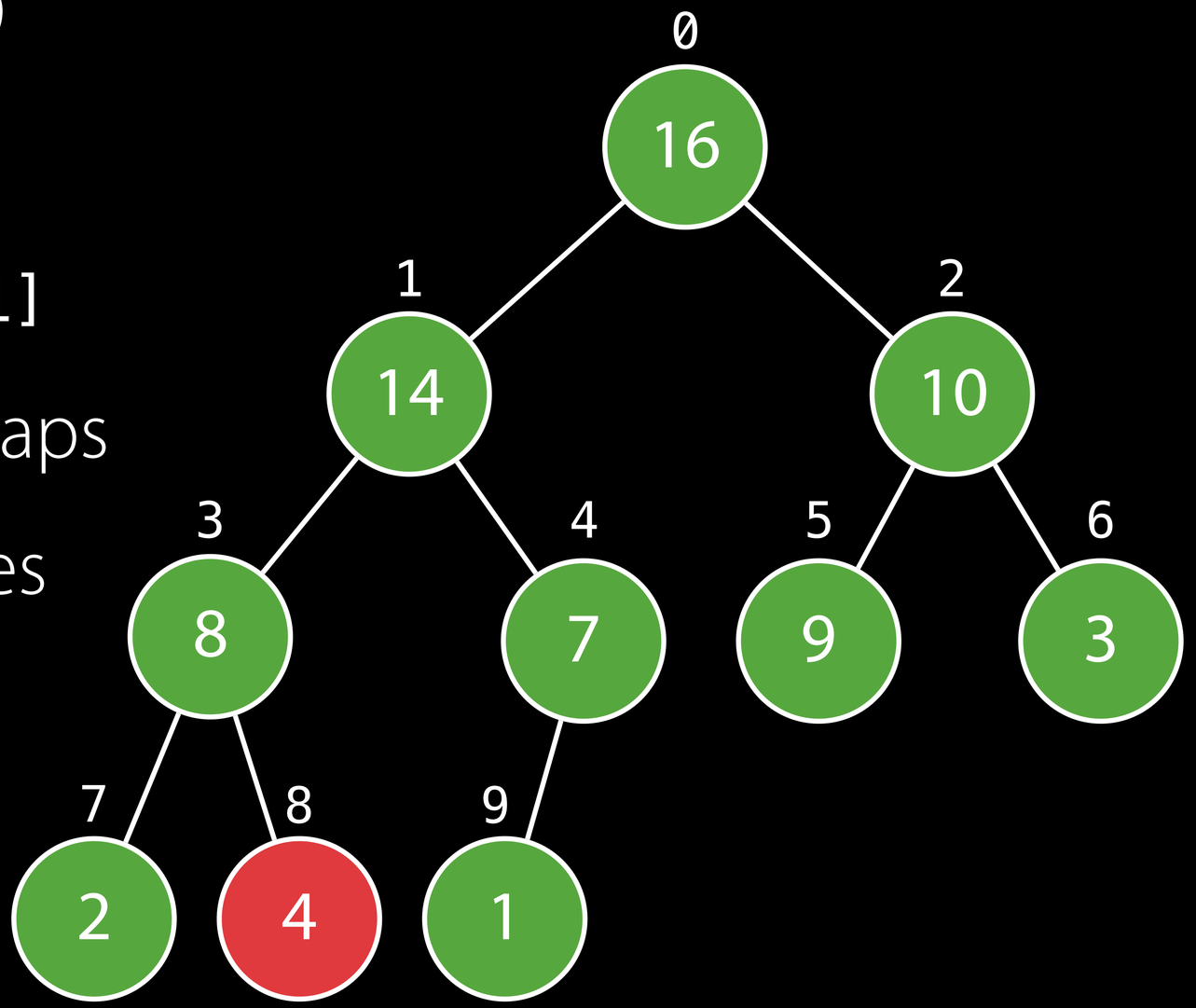
Produce a max-heap from an unordered array `A[0 ... n - 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n - 1) / 2...0
    maxHeapify(i)
```

- `maxHeapify(0)`
- `maxHeapify(3)`



buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n - 1]`

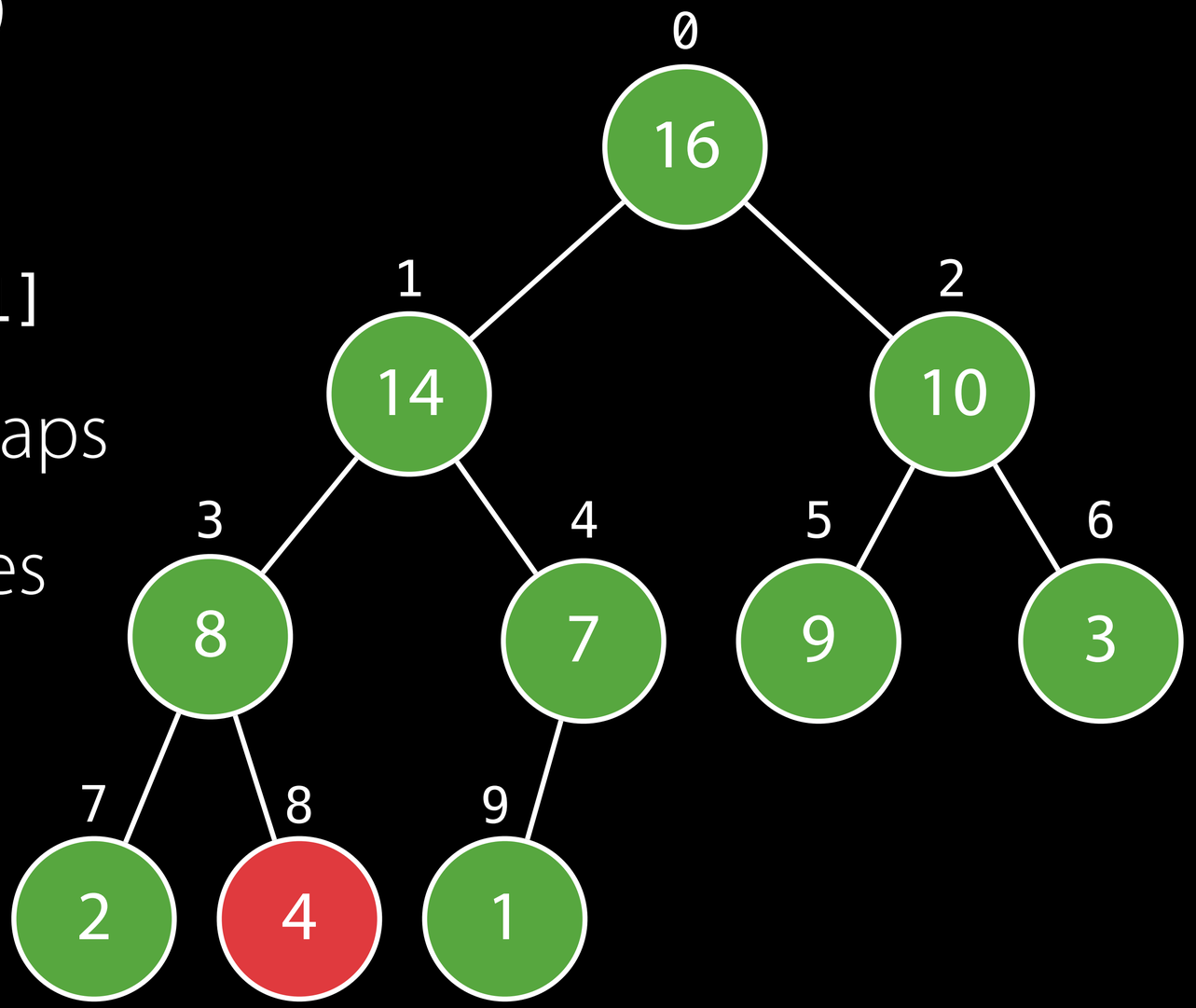
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n - 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(0)`

`maxHeapify(8)`



buildMaxHeap

Produce a max-heap from an unordered array `A[0 ... n – 1]`

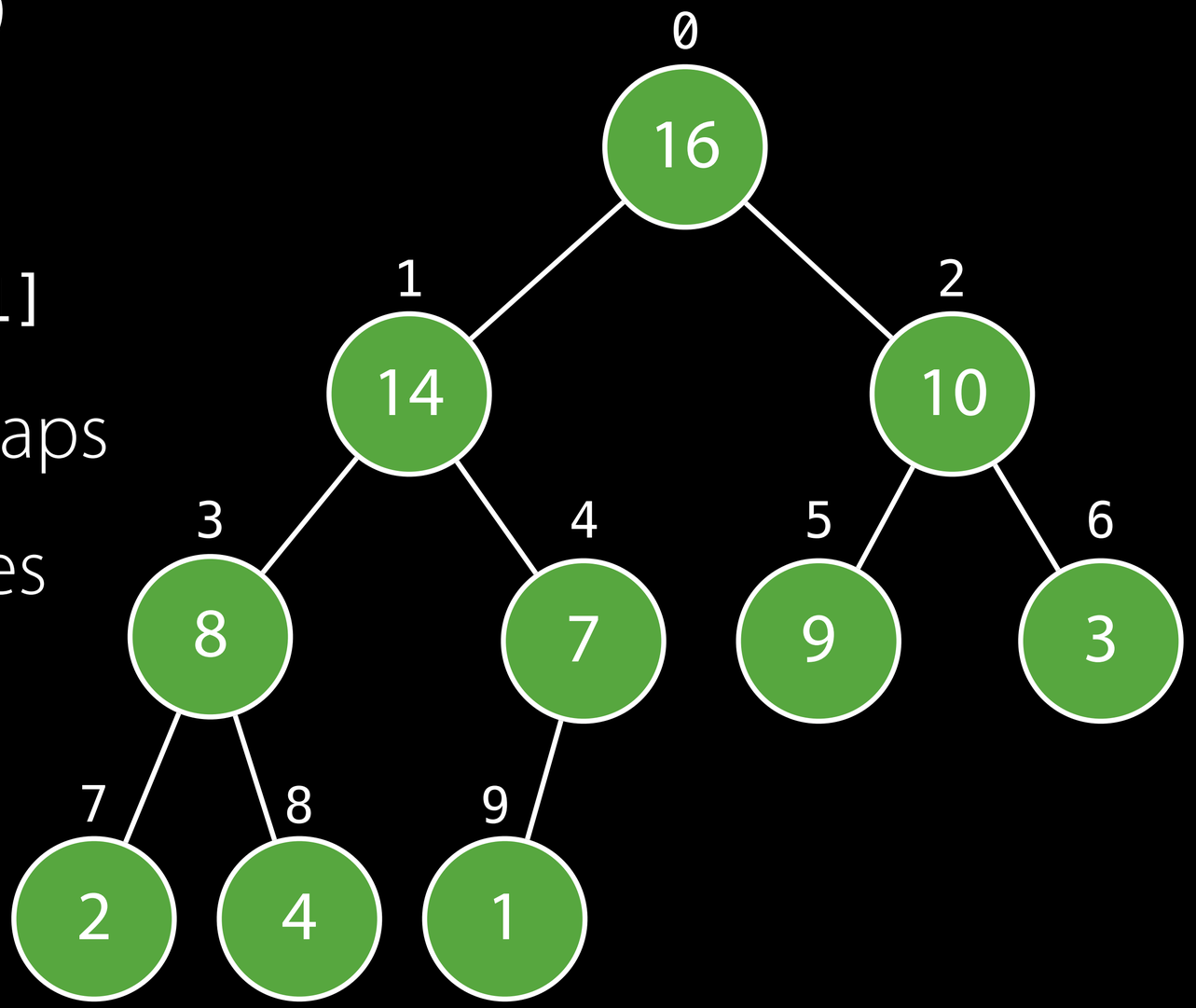
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

`maxHeapify(0)`

`maxHeapify(8)`



buildMaxHeap

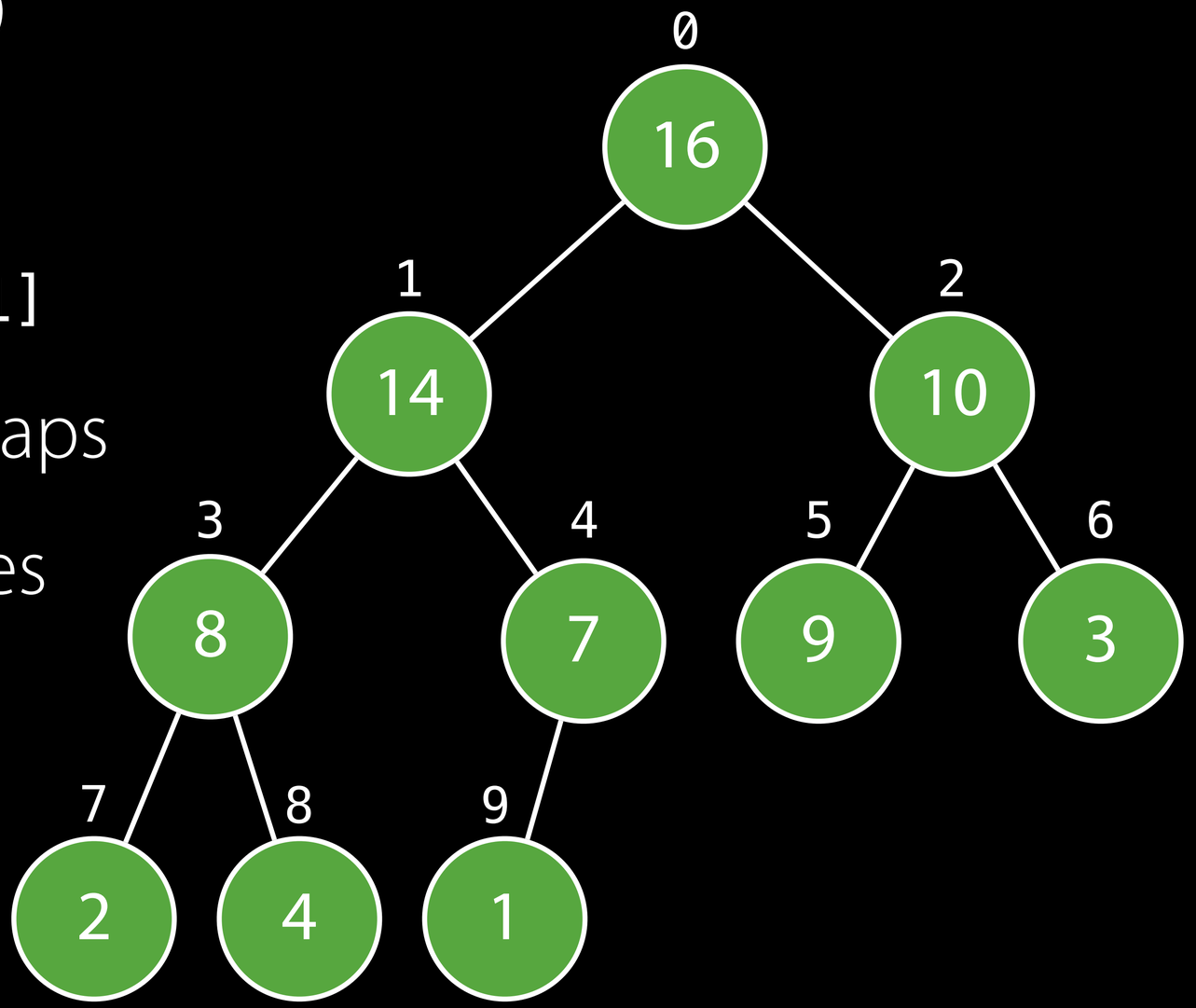
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

all nodes satisfy max-heap property

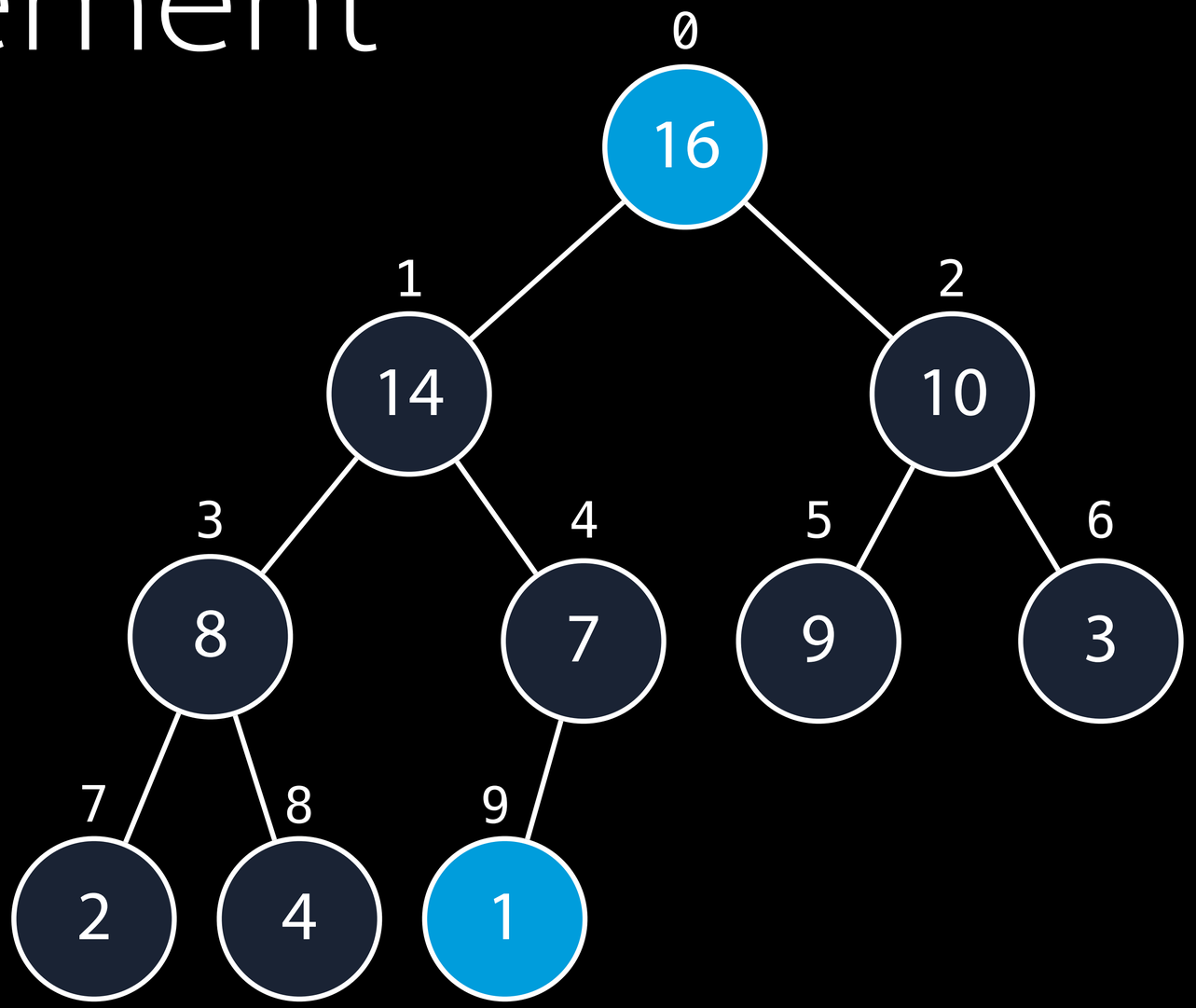


removeMaxElement

Swap root element with the last element

Decrement the size

Call `maxHeapify(0)` to correct the violation at root

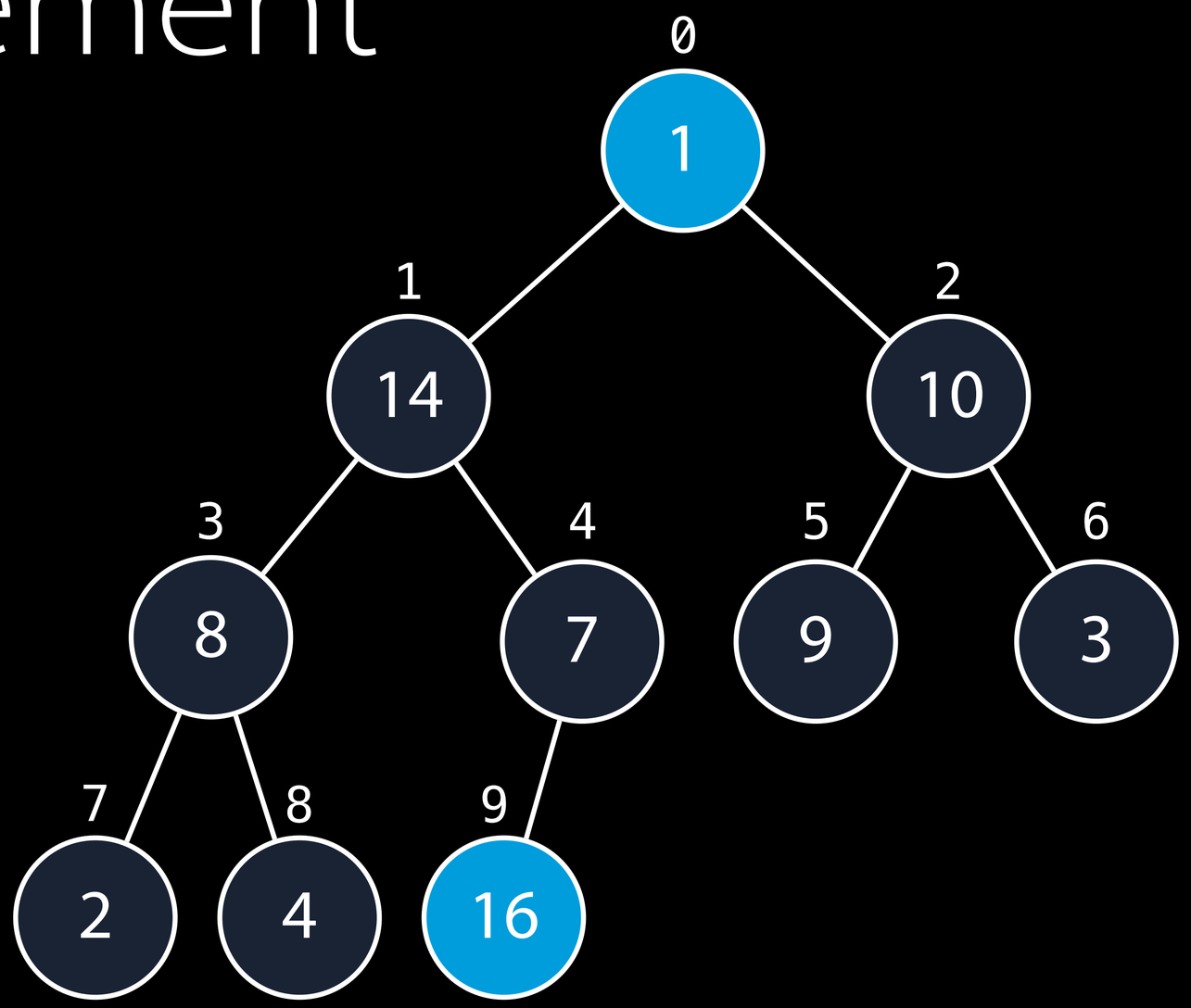


removeMaxElement

Swap root element with the last element

Decrement the size

Call `maxHeapify(0)` to correct the violation at root

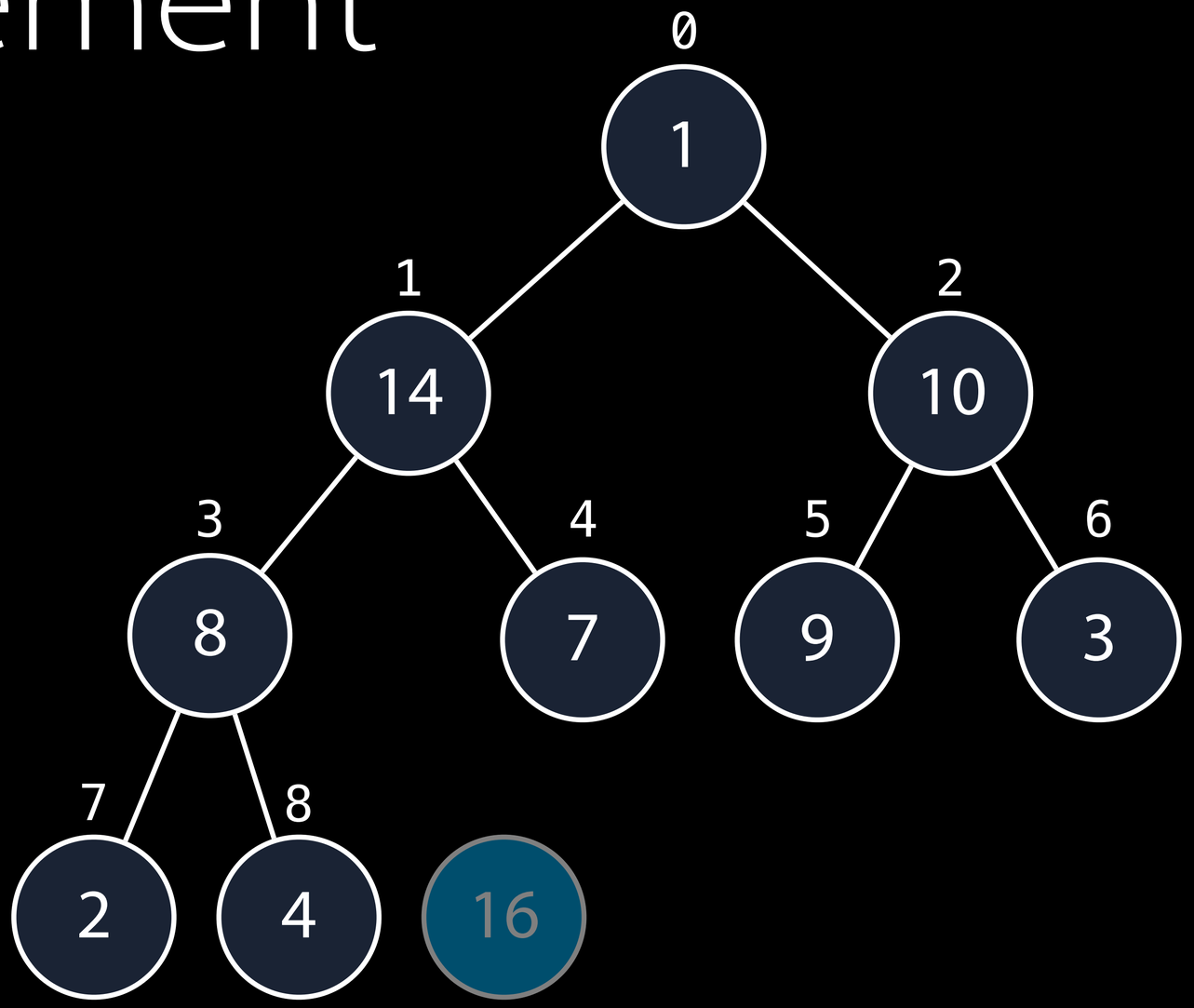


removeMaxElement

Swap root element with
the last element

Decrement the size

Call `maxHeapify(0)` to
correct the violation at root

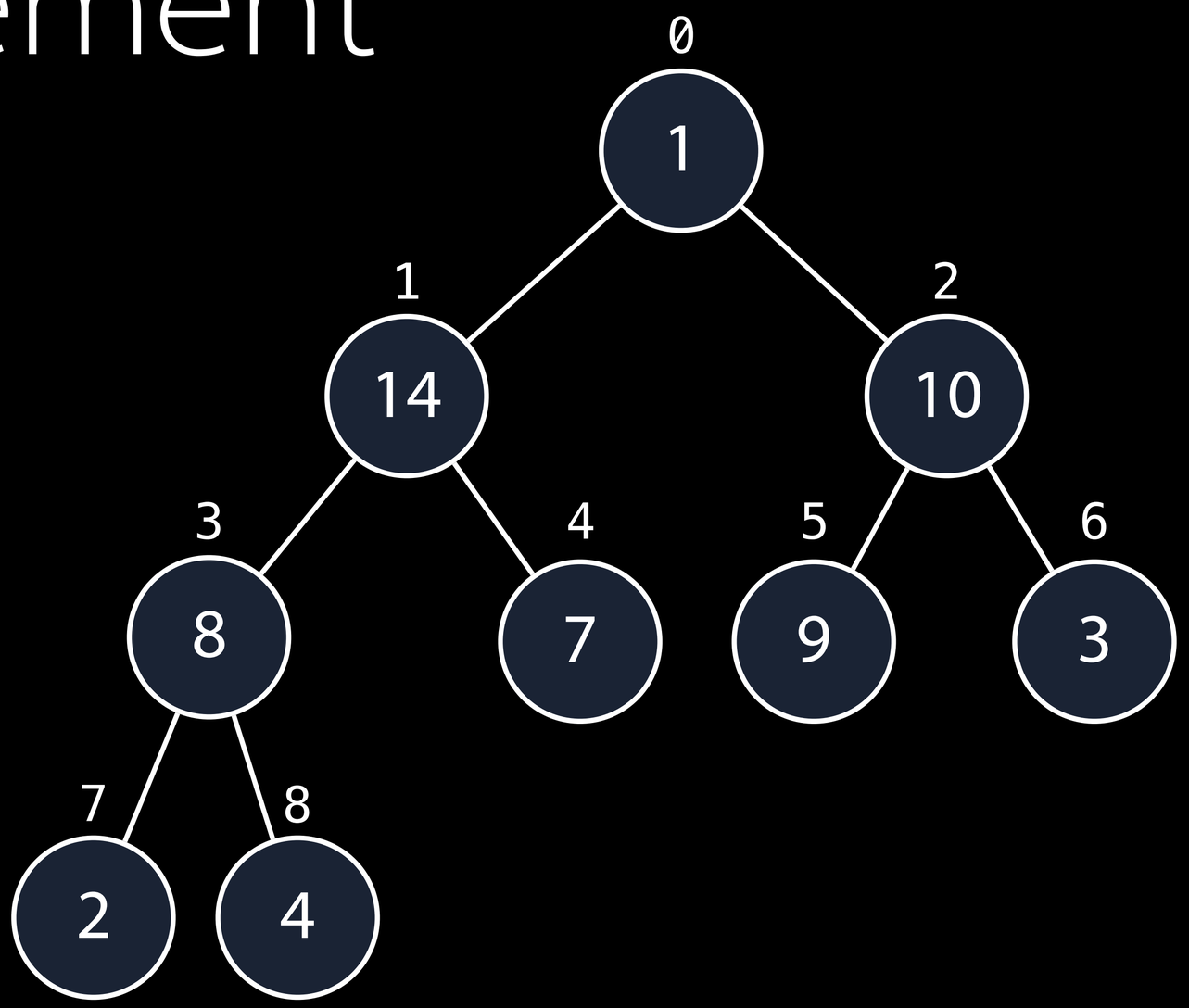


removeMaxElement

Swap root element with
the last element

Decrement the size

Call `maxHeapify(0)` to
correct the violation at root

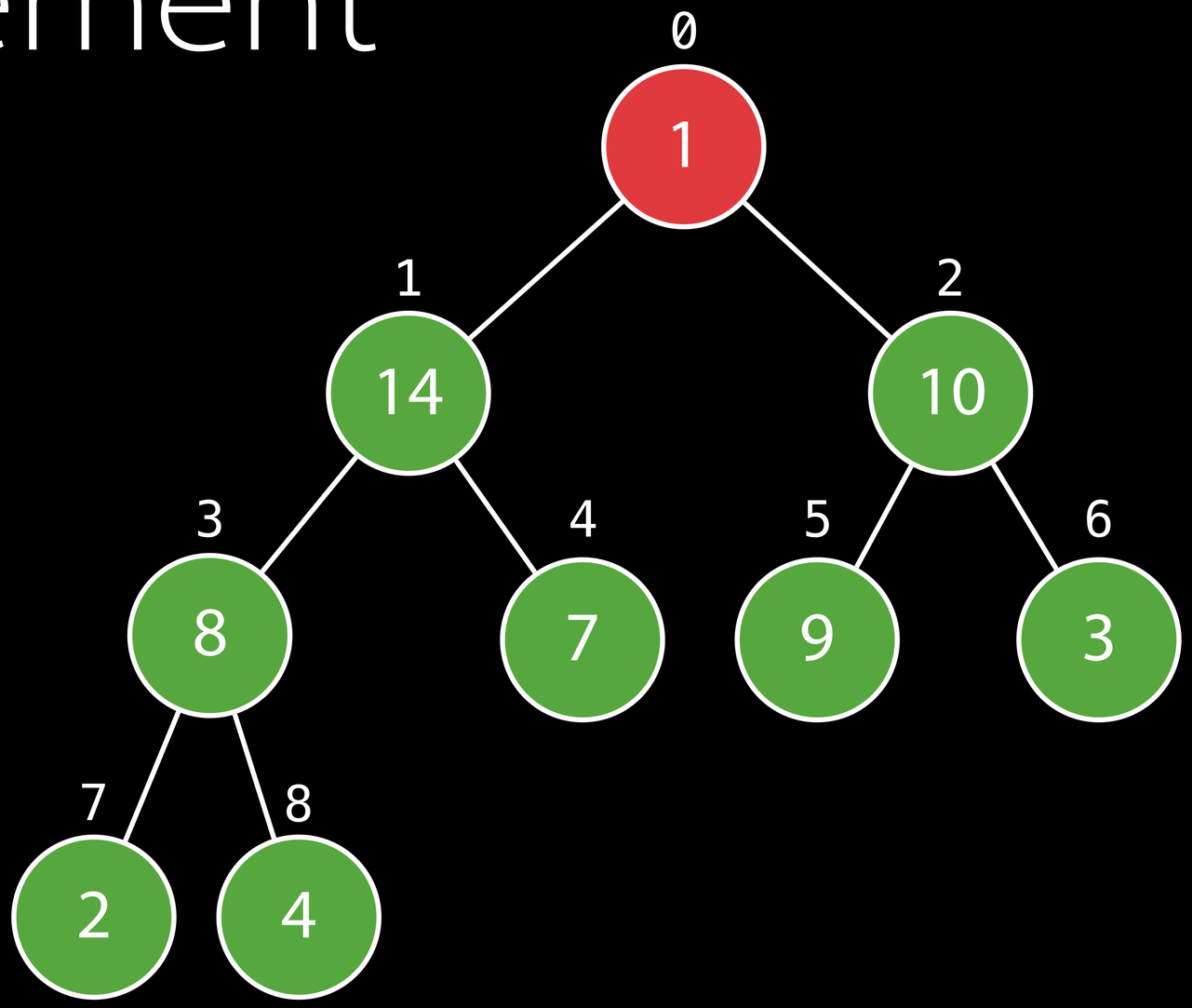


removeMaxElement

Swap root element with
the last element

Decrement the size

Call `maxHeapify(0)` to
correct the violation at root

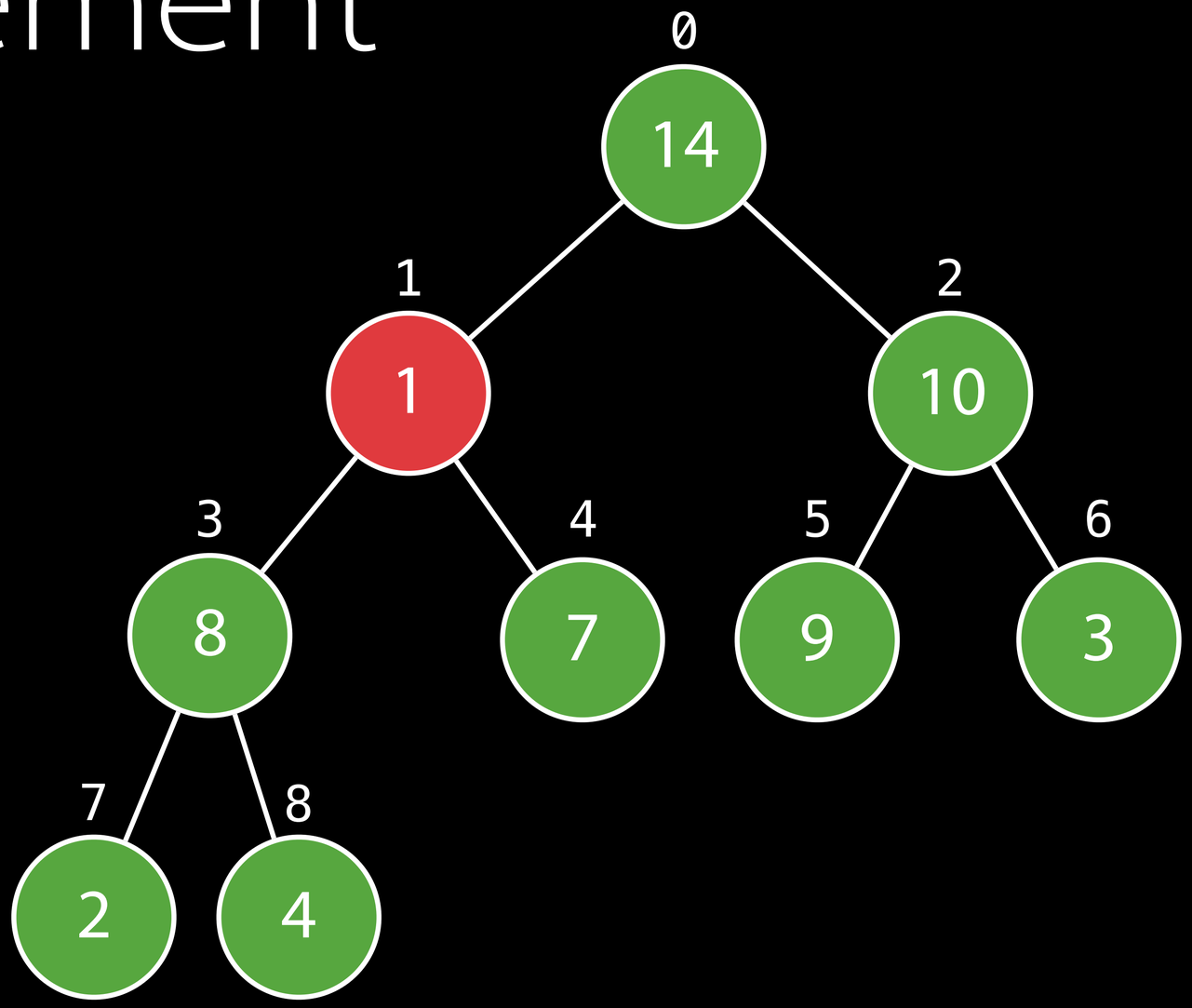


removeMaxElement

Swap root element with
the last element

Decrement the size

Call `maxHeapify(0)` to
correct the violation at root

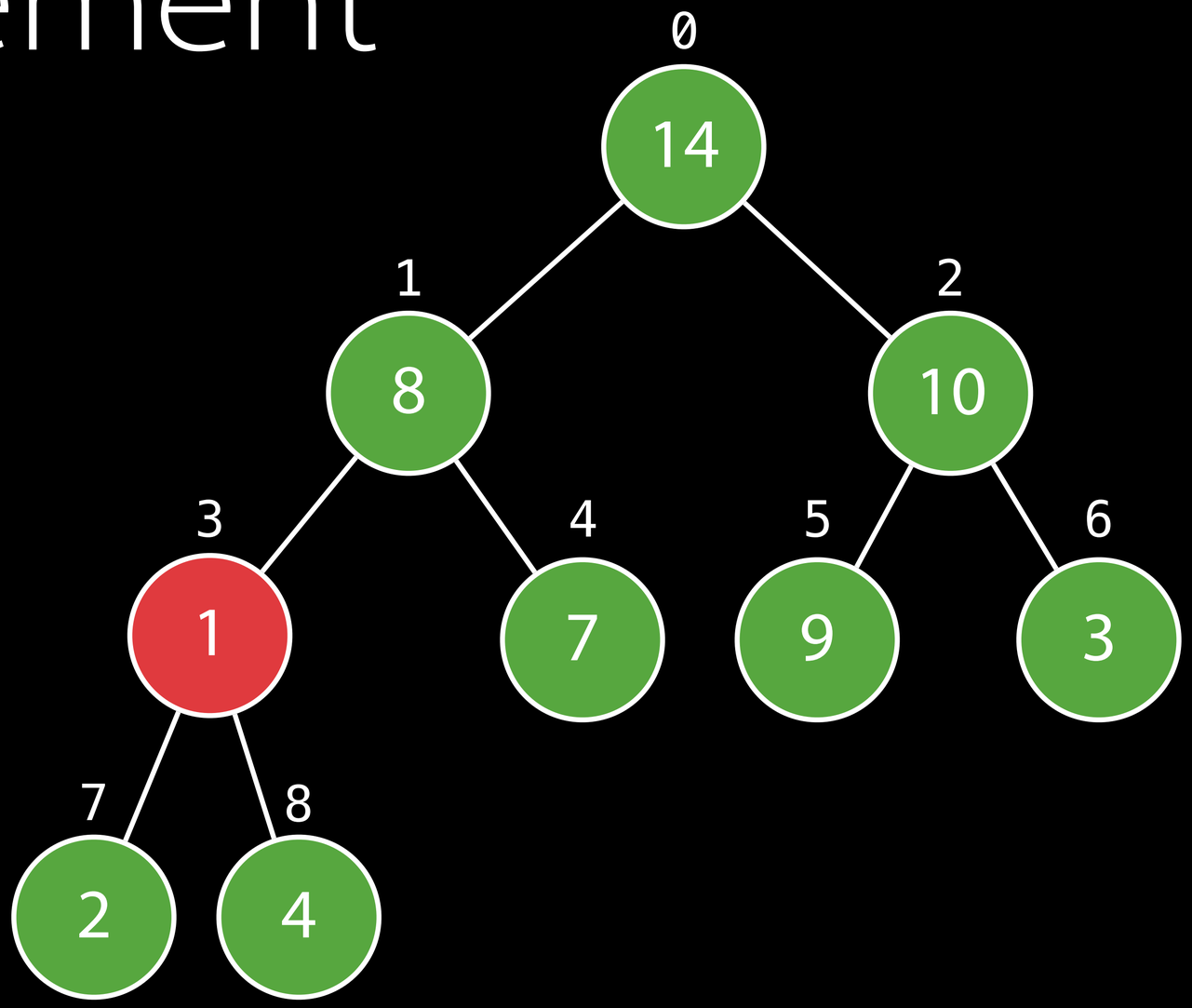


removeMaxElement

Swap root element with
the last element

Decrement the size

Call `maxHeapify(0)` to
correct the violation at root

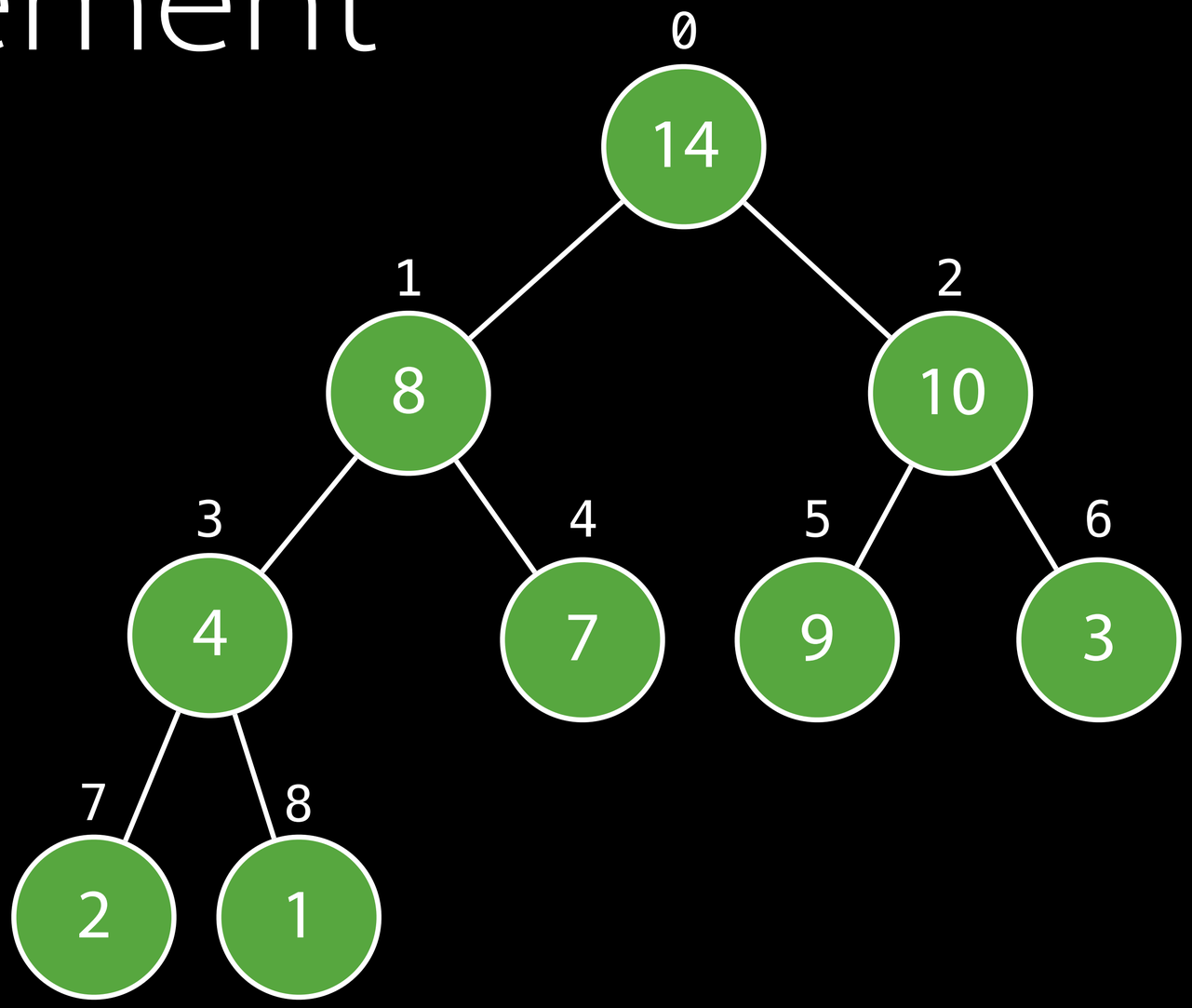


removeMaxElement

Swap root element with
the last element

Decrement the size

Call `maxHeapify(0)` to
correct the violation at root

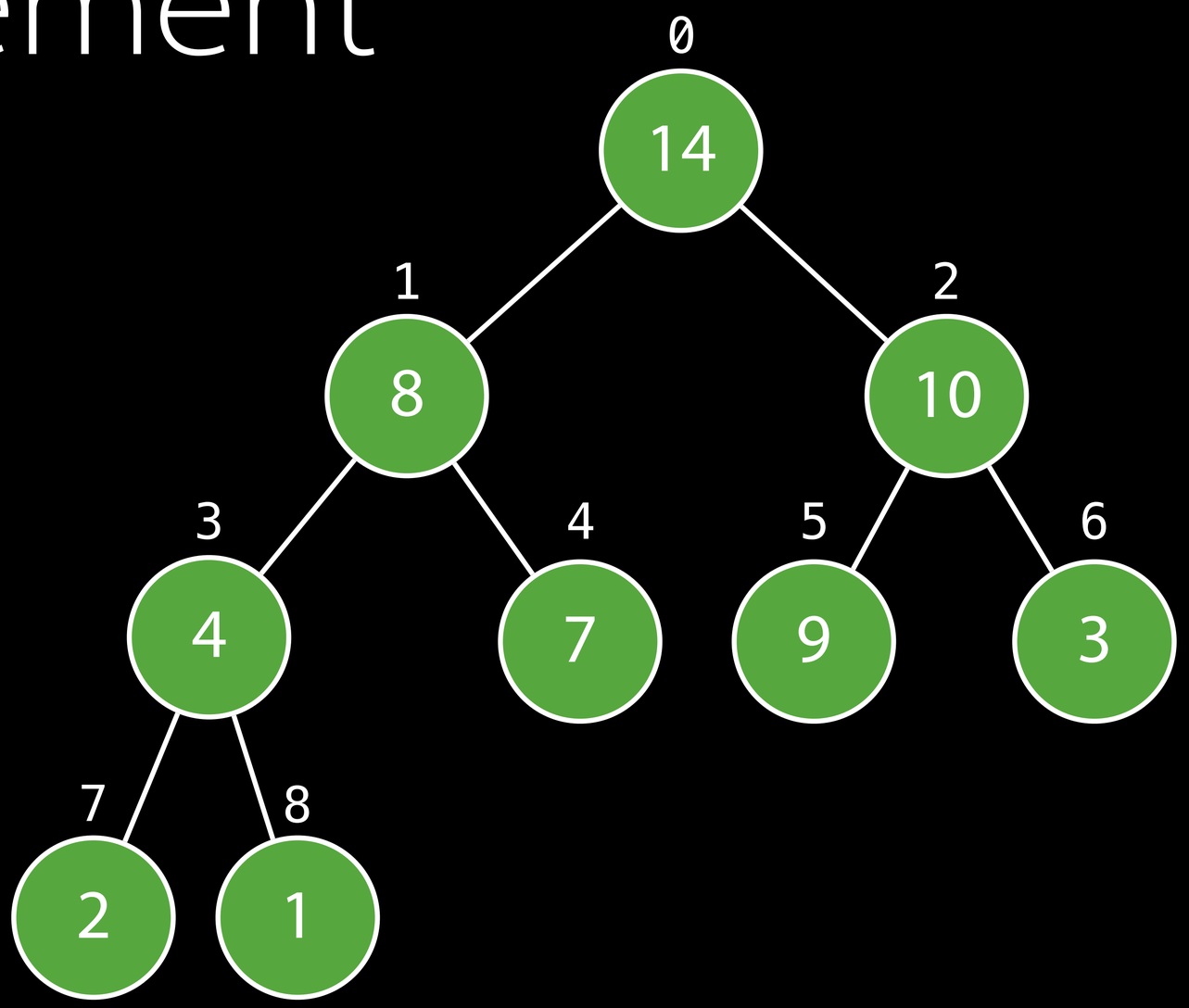


removeMaxElement

Swap root element with the last element

Decrement the size

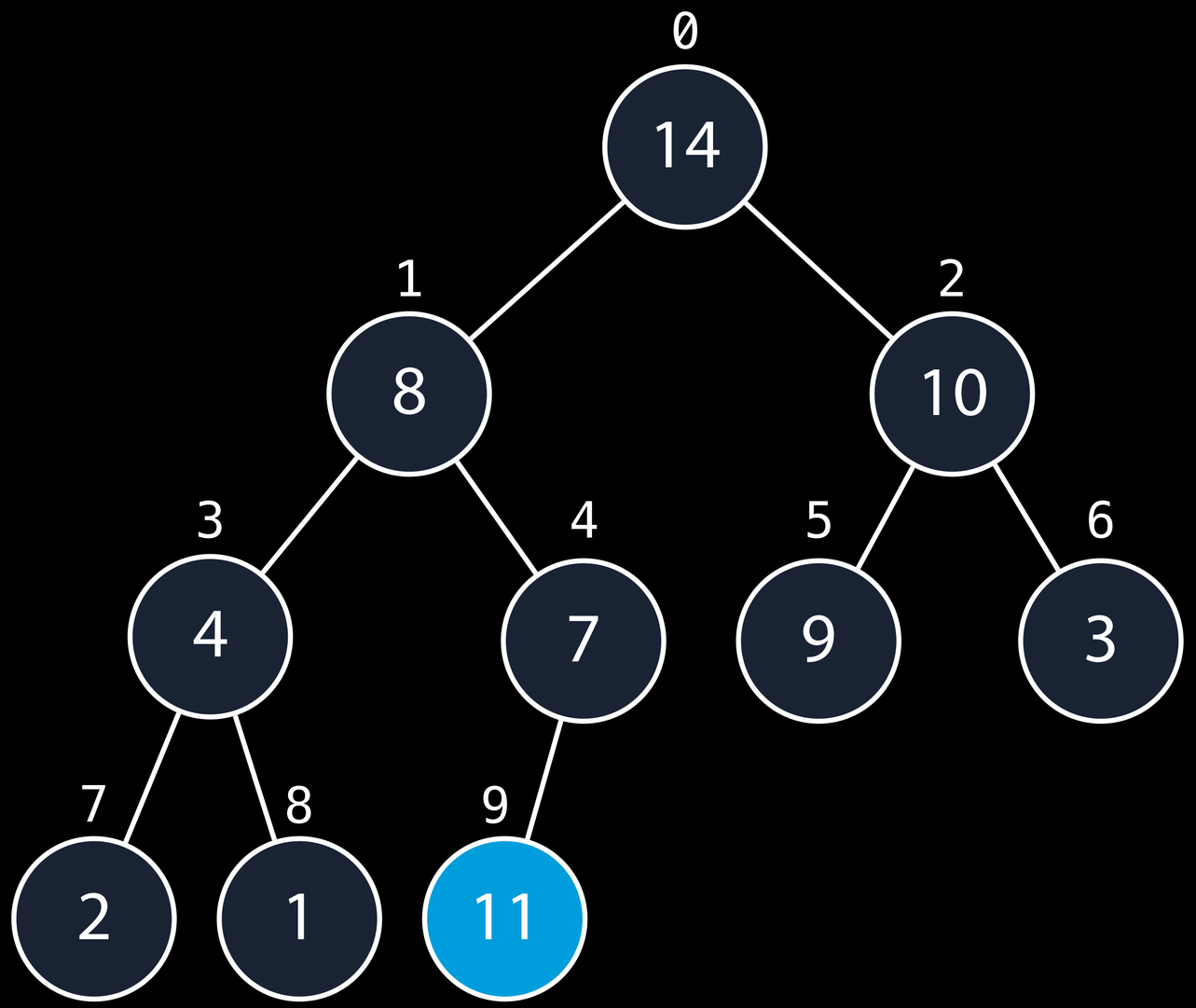
Call `maxHeapify(0)` to correct the violation at root



insert

Add an element to the end of array

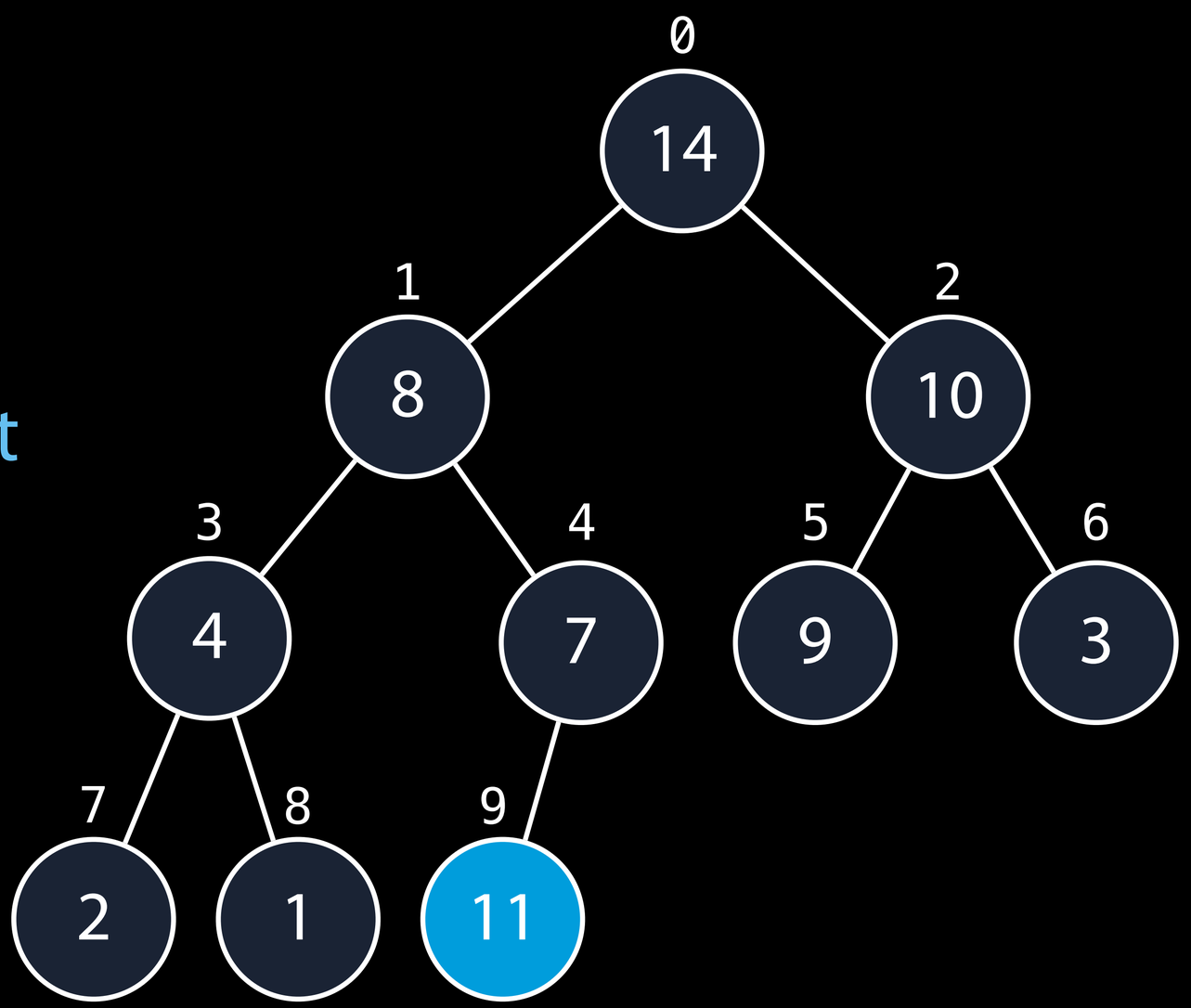
Swap with parent until correct position



insert

Add an element to the end of array

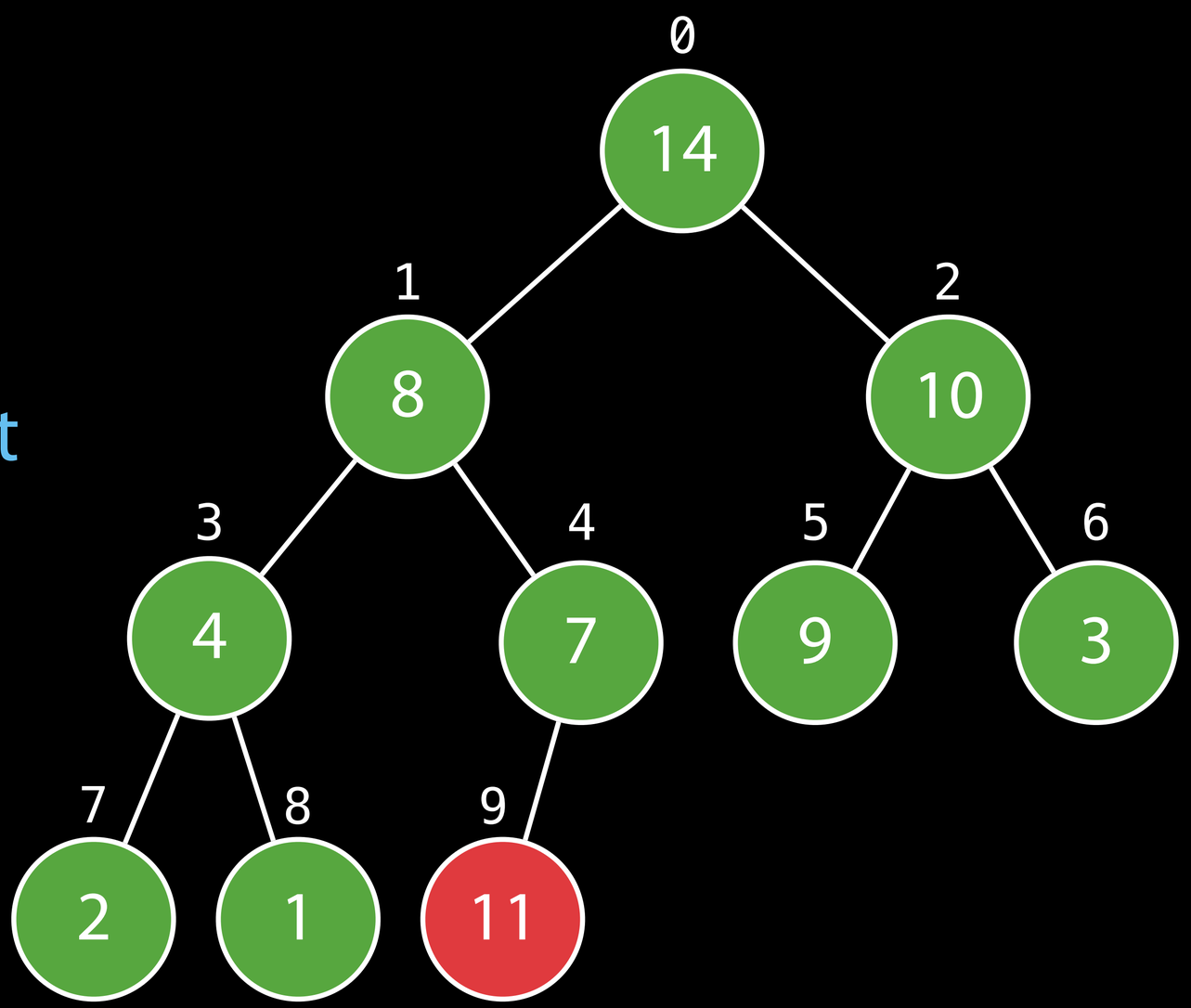
Swap with parent until correct position



insert

Add an element to the end of array

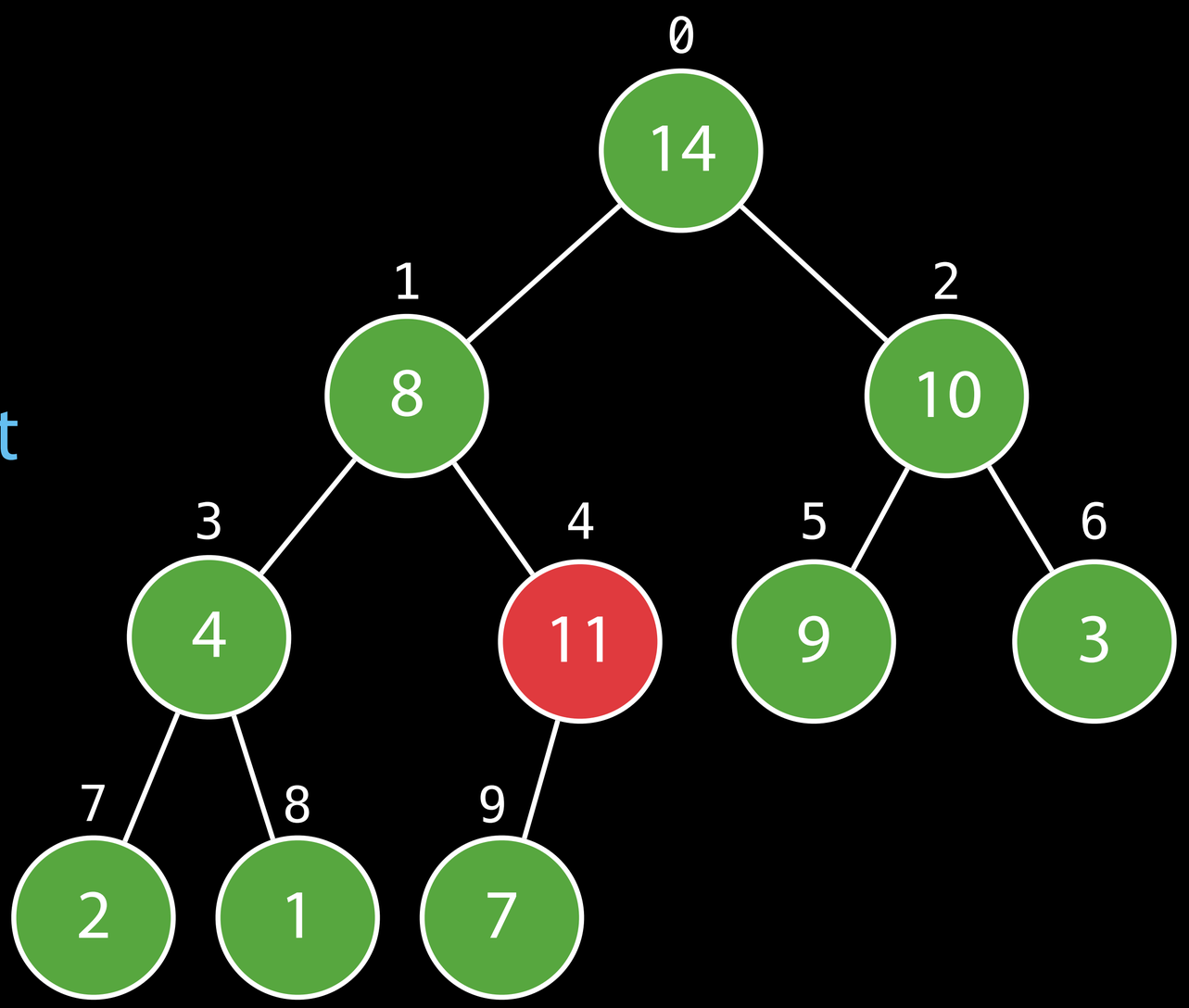
Swap with parent until correct position



insert

Add an element to the end of array

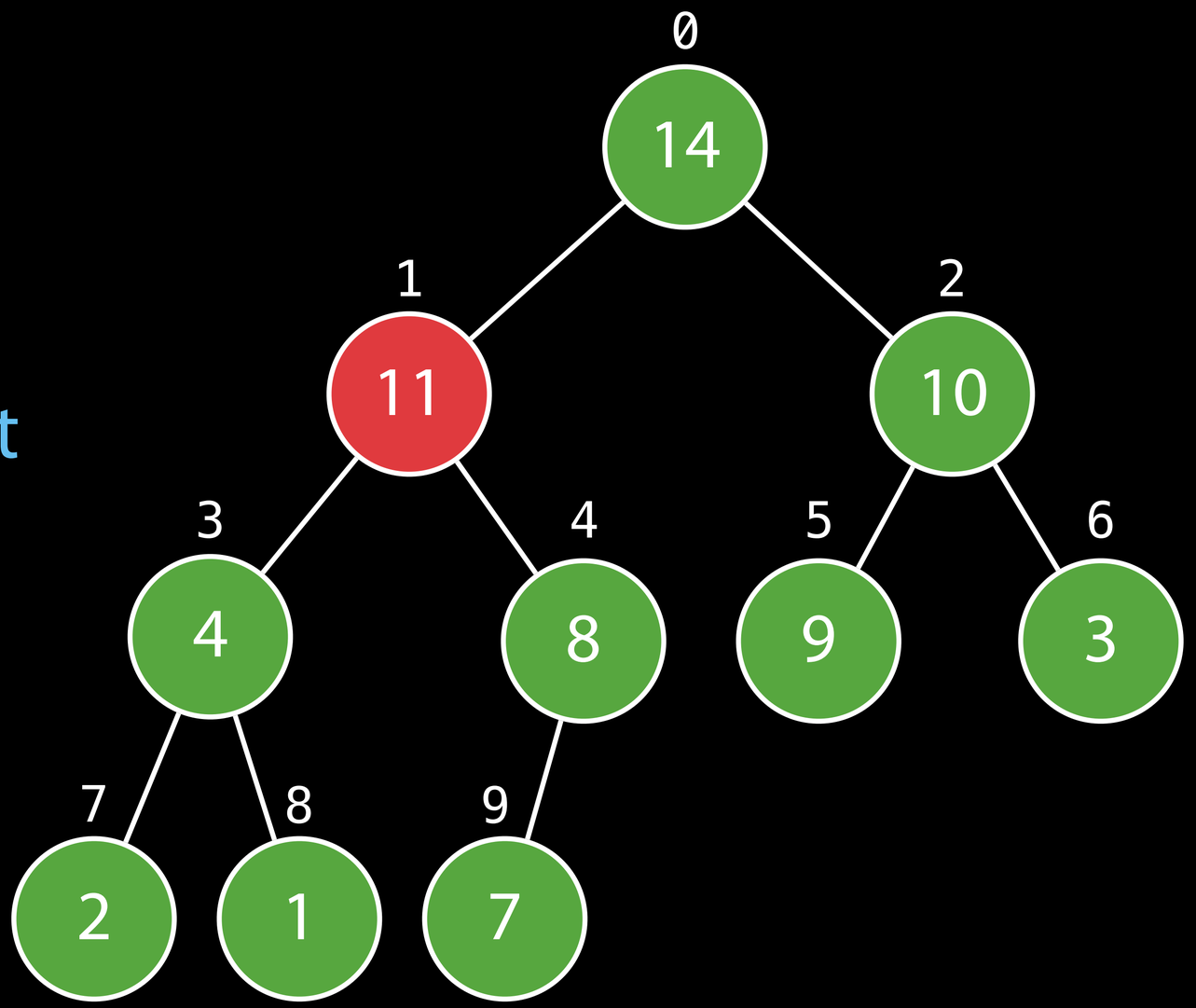
Swap with parent until correct position



insert

Add an element to the end of array

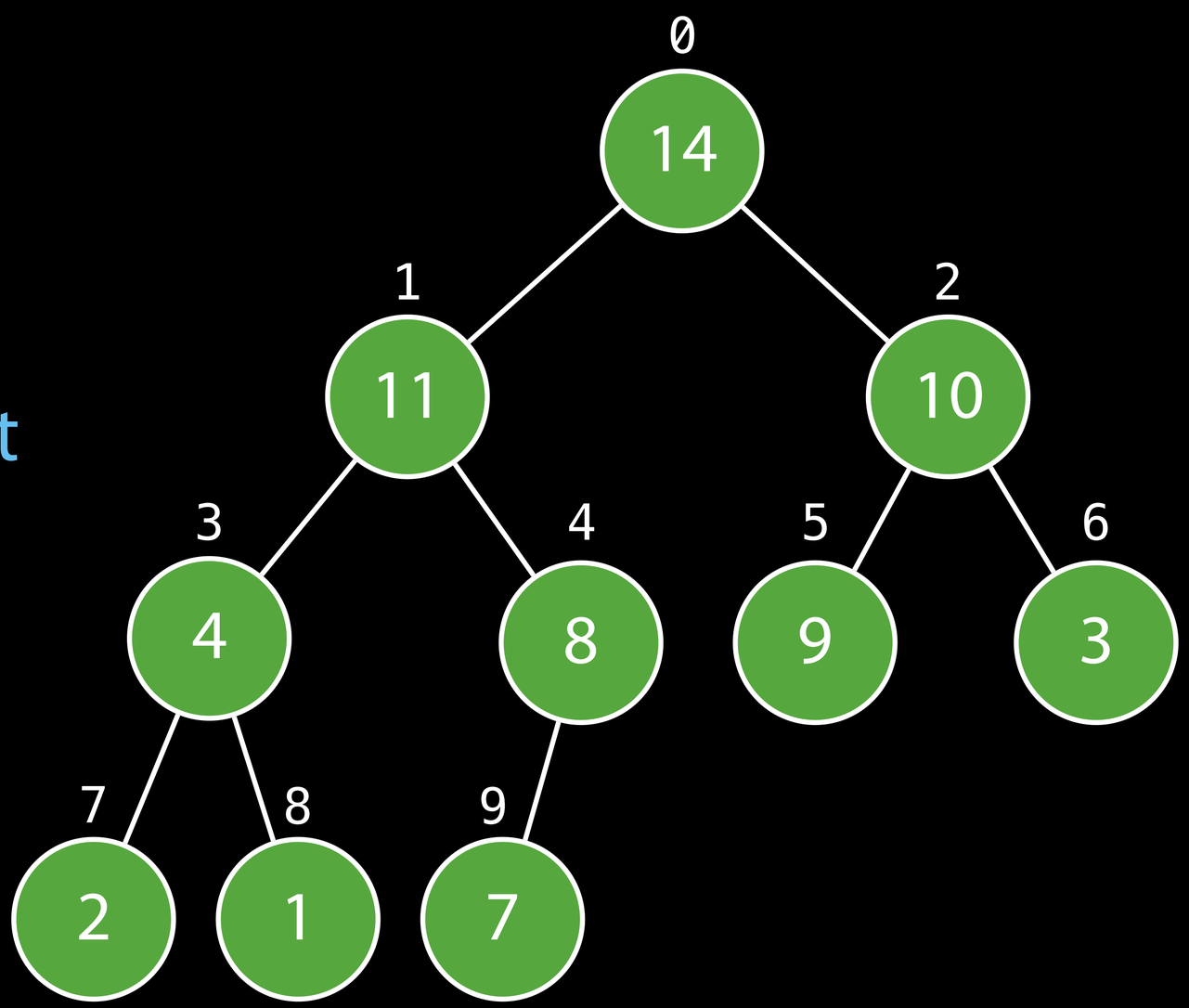
Swap with parent until correct position



insert

Add an element to the end of array

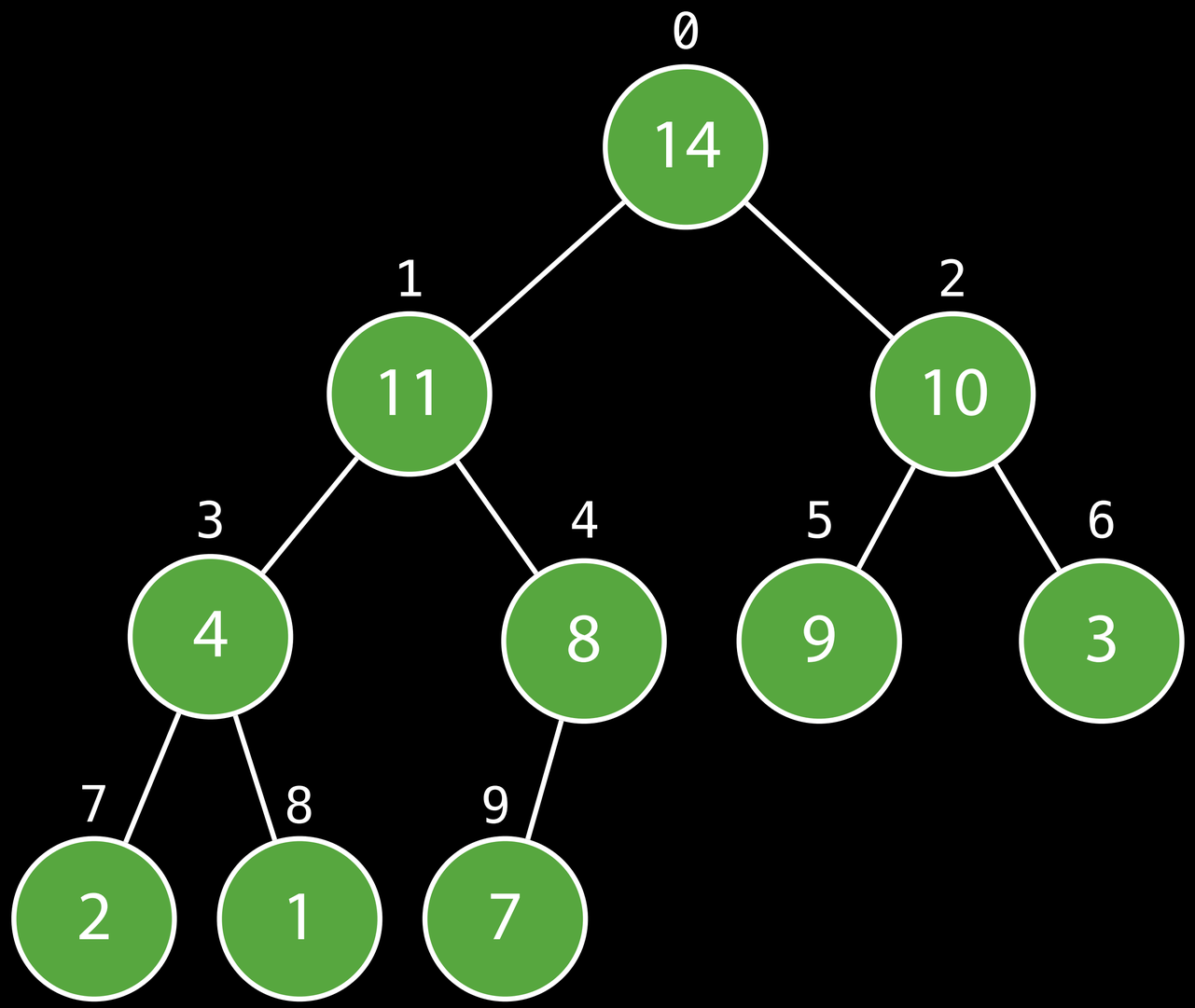
Swap with parent until correct position



insert

Add an element to the end of array

Swap with parent until correct position



Heap practice

Suppose that seven values are pushed into an empty maximum priority queue binary heap in the following order:

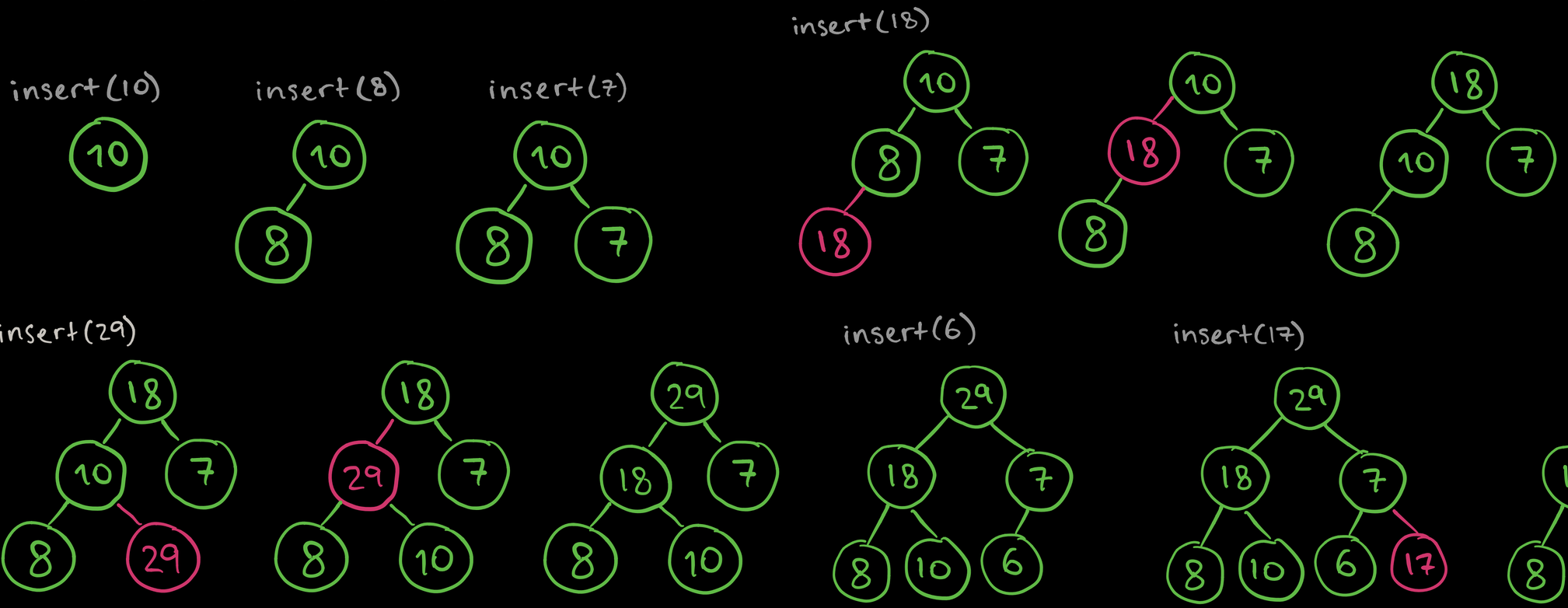
10, 8, 7, 18, 29, 6, 17

What would the array representation of the heap look like after all seven values are inserted?

Draw the tree representing the heap after each insertion.

Heap practice

10, 8, 7, 18, 29, 6, 17

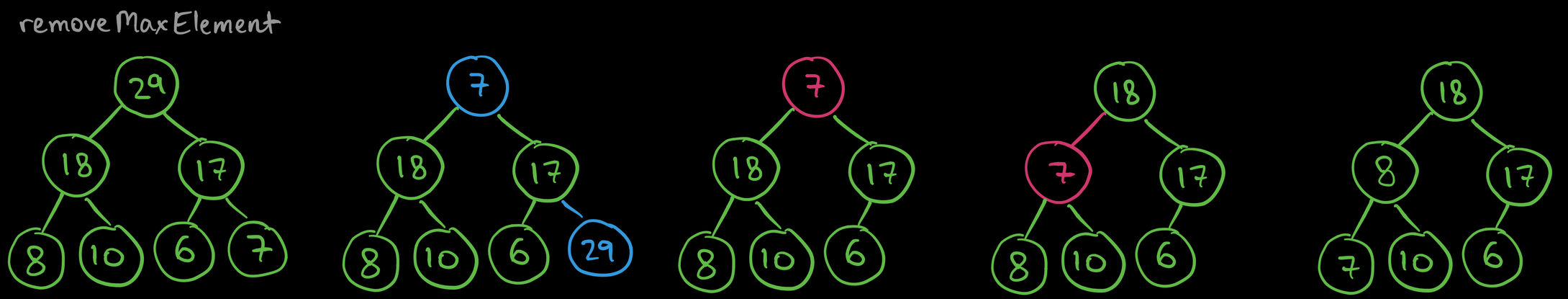



```
[29, 18, 17, 8, 10, 6, 7]
```

Heap practice

What is the array representation of the heap after calling `removeMaxElement` twice?

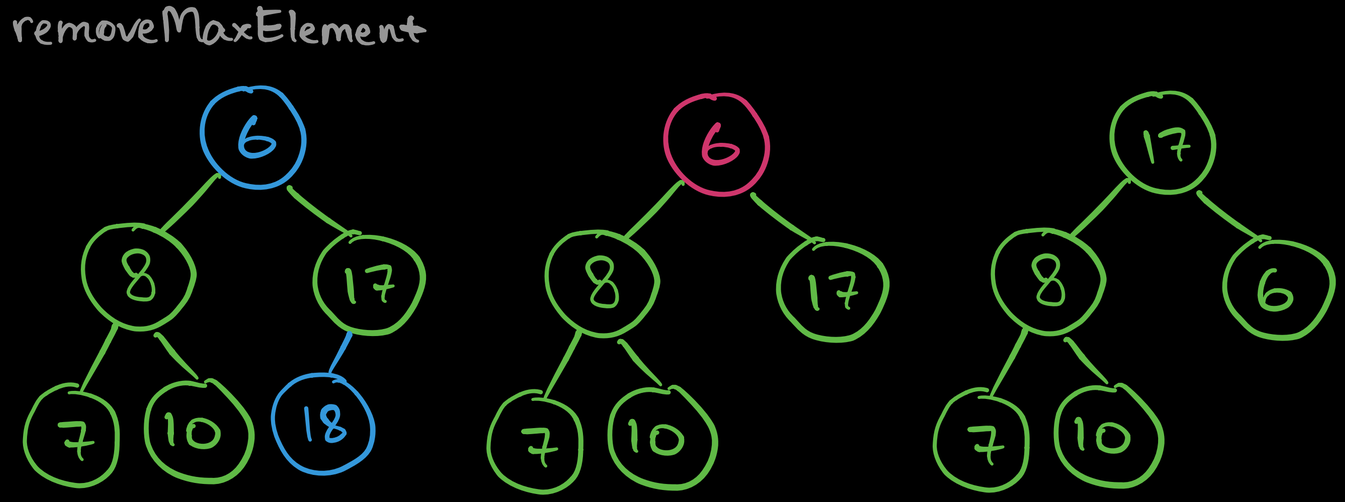
Draw the tree representing the heap.



Heap practice

What is the array representation of the heap after calling `removeMaxElement` twice?

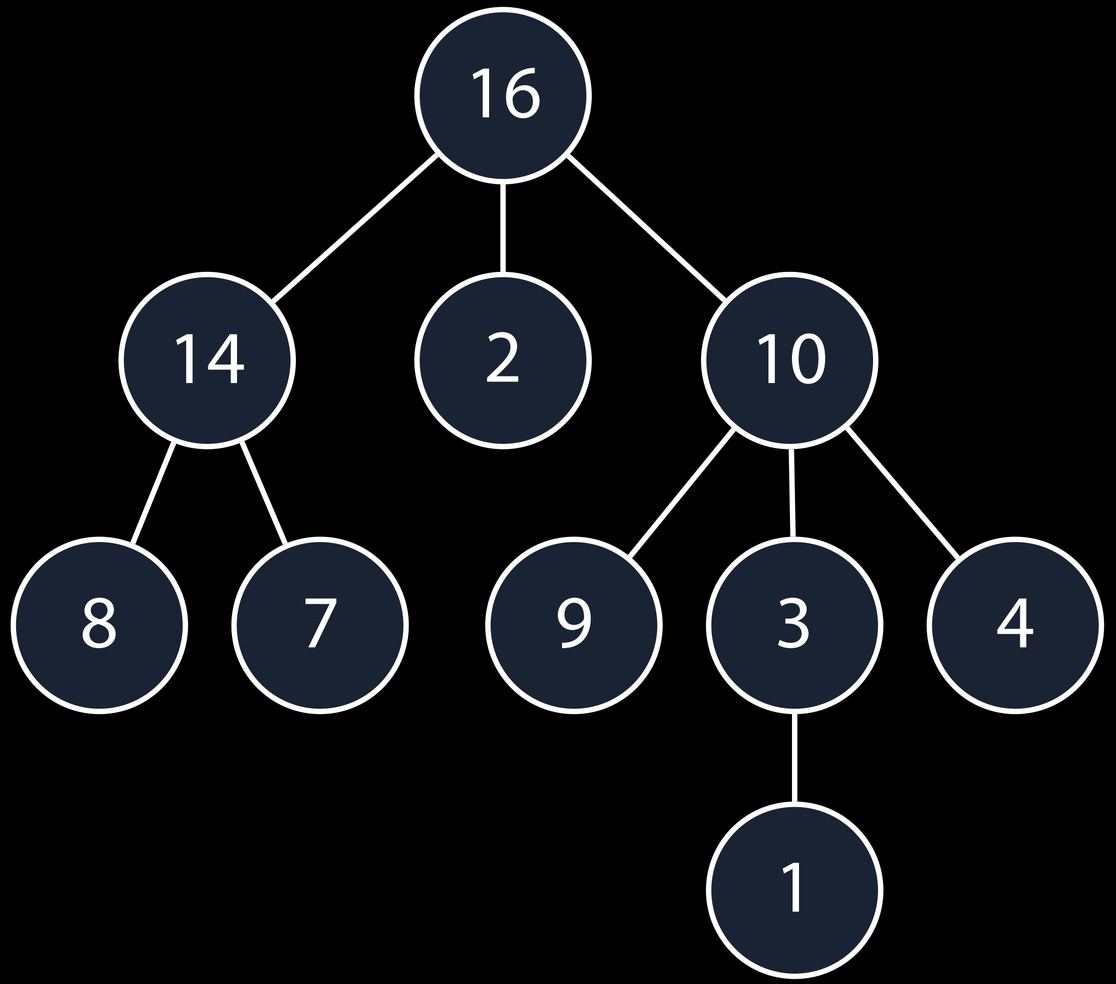
Draw the tree representing the heap.




```
[17, 8, 6, 7, 10]
```

Pairing Heap

A max-pairing heap is a tree where each node can have any number of children



Pairing Heap

For examples, see the annotated handout and
`datastructures.maximal.io/heaps/pairing-heap`

Disjoint sets

Find Sum

Interview Question

Given an integer **`sum`** and a sorted array **`numbers`** of **`N`** distinct integers, implement a function to find if there exist indices **`i`** and **`j`** such that **`numbers[i] + numbers[j] == x`**.

```
bool findSum(const vector<int> &numbers, int sum) {
    for (int i = 0; i < numbers.size(); i += 1) {
        for (int j = 0; j < numbers.size(); j += 1) {
            if (numbers[i] + numbers[j] == sum) {
                return true;
            }
        }
    }
    return false;
}
```

Find Sum

Interview Question

Given an integer **`sum`** and a sorted array **`numbers`** of **`N`** distinct integers, implement a function to find if there exist indices **`i`** and **`j`** such that **`numbers[i] + numbers[j] == x`**.

```
bool betterFindSum(const vector<int> &numbers, int sum);
```

Implement a better, more efficient version of the algorithm.

Project 2

due Monday 5/22

Q&A