Week 2: Wednesday

EECS 281

MAXIM ALEKSA

maximal.io

Agenda

Templates

Standard Template Library

Containers

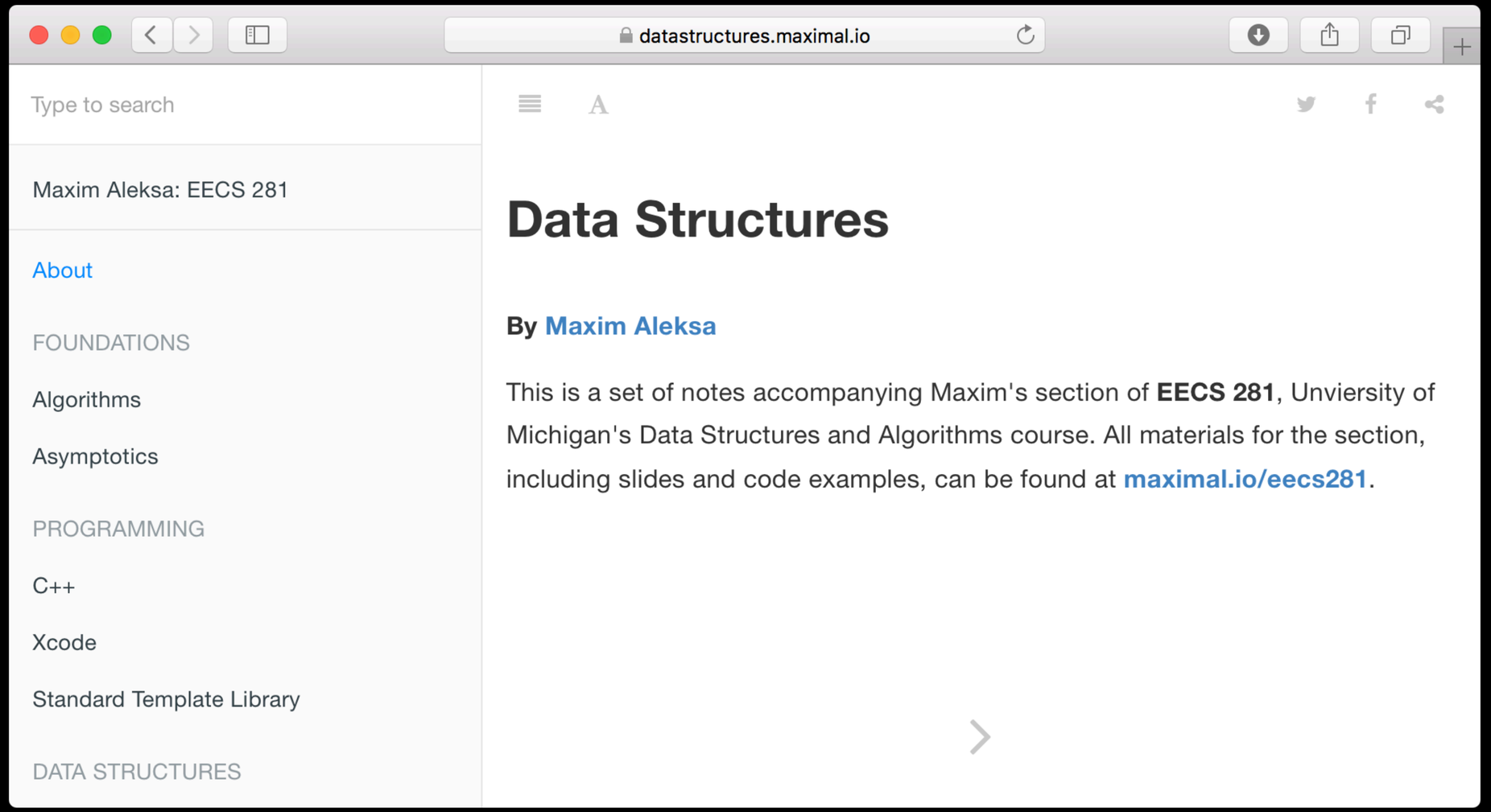
Array

Vector

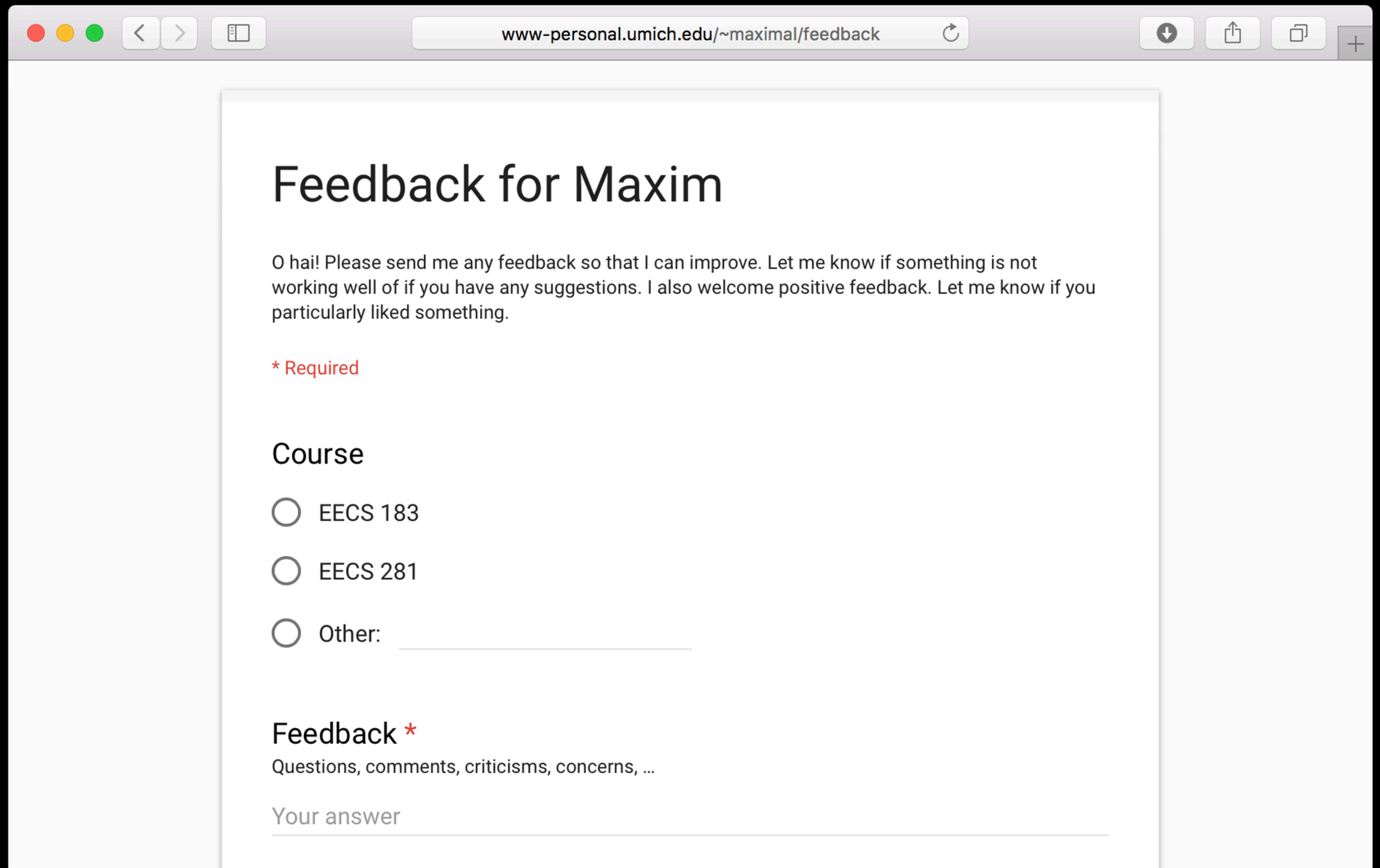
Deque

(Linked) list

datastructures.maximal.io

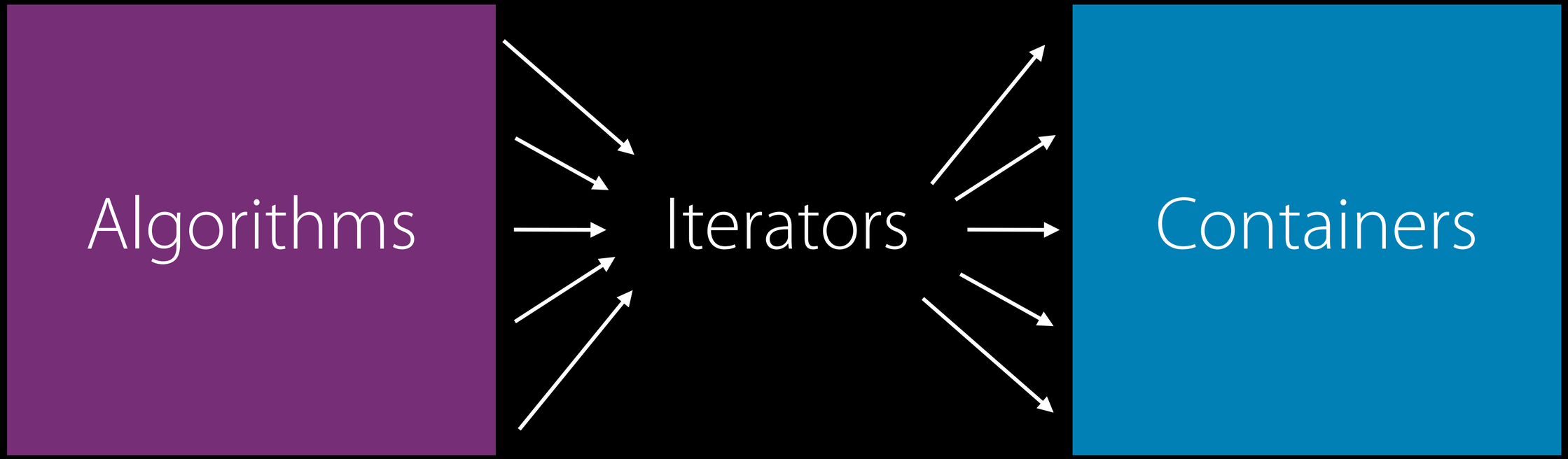


maximal.io/feedback



Standard Template Library

Standard Template Library



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

Iterators

Iterators

Random access **iterators:** random access

Bidirectional **iterators:** forward and backward moving

Forward **iterators:** forward moving

Input **iterators:** read only, forward moving

Output **iterators:** write only, forward moving

Iterator adapters

Insert iterators

Stream iterators

Reverse iterators

Move iterators

Insert iterators

```
vector<int> numbers = {183, 203, 281, 370, 376};
vector<int> otherNumbers = {203, 280};
vector<int>::iterator it = find(numbers.begin(),
                                numbers.end(), 281);
insert_iterator<vector<int>> insertIt(numbers, it);
copy(otherNumbers.begin(), otherNumbers.end(), insertIt);
```

Containers

Containers

Sequence containers:

`vector, deque, list, forward list`

Associative containers:

`(multi)set, (multi)map`

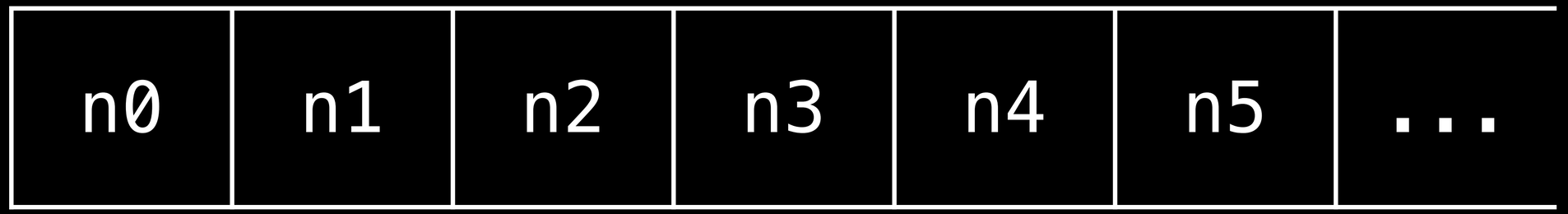
Unordered associative containers:

`unordered_(multi)set, unordered_(multi)map`

Container adapters:

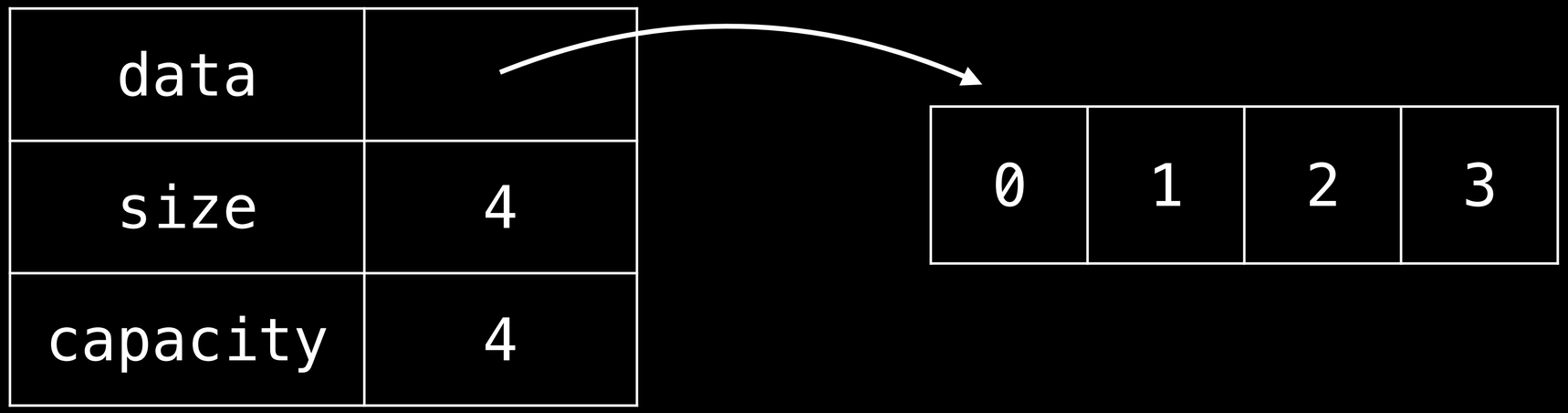
`queue, stack, priority queue`

Vector

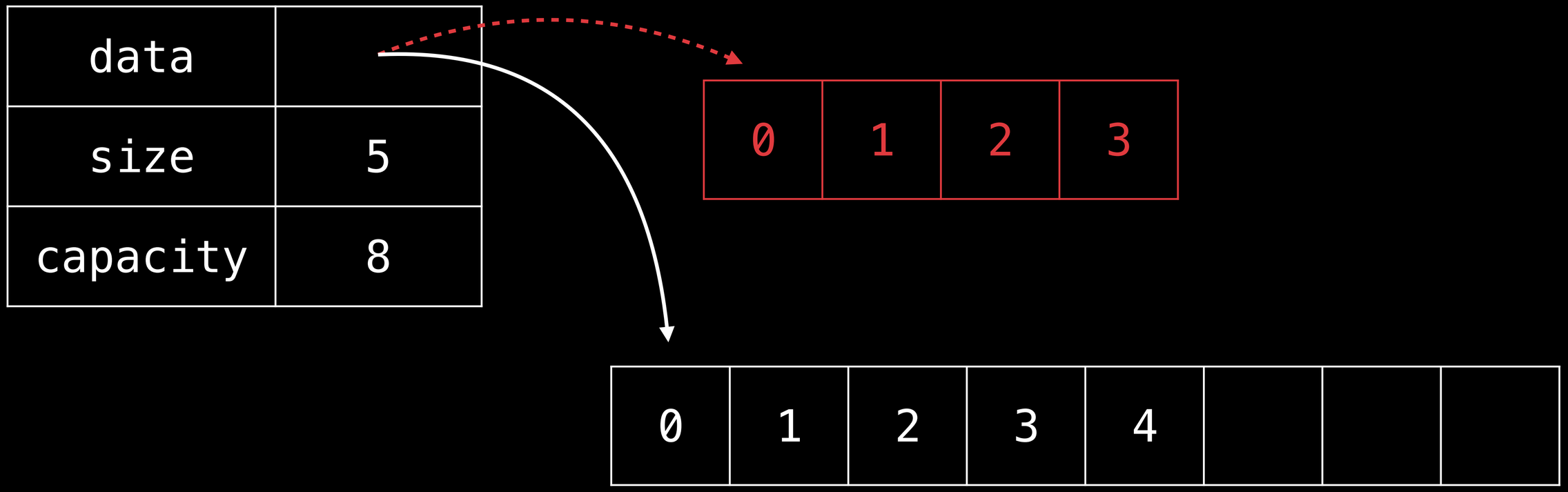


vector::push_back()

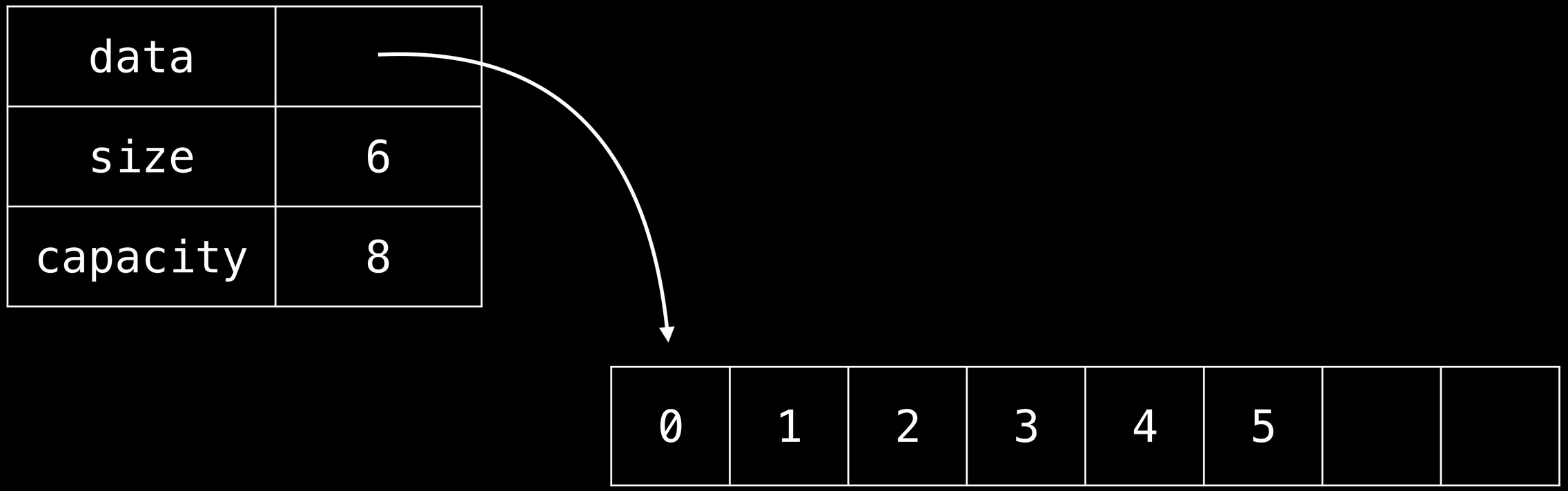
vector::push_back()



vector::push_back()



vector::push_back()



vector::push_back()

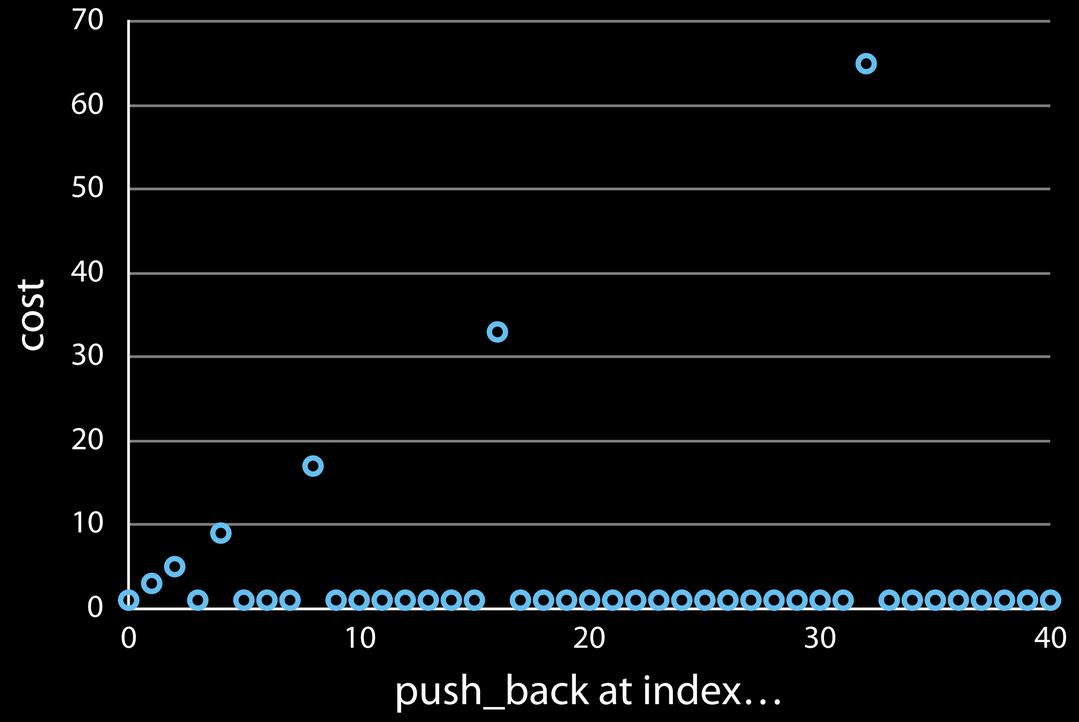
push_back at i…	0	1	2	3	4	5	6	7	8
size									
capacity									
write cost									
resize cost									
total cost									
cumulative cost									
average cost									

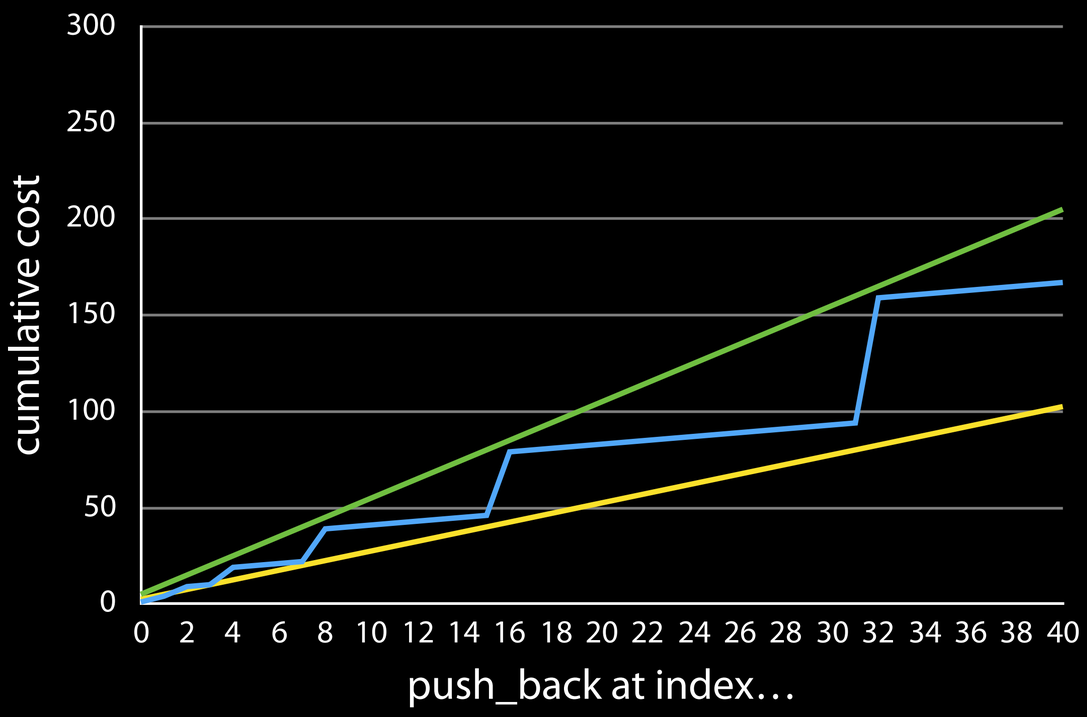
vector::push_back()

push_back at i…	0	1	2	3	4	5	6	7	8
size	1	2	3	4	5	6	7	8	9
capacity	1	2	4	4	8	8	8	8	16
write cost	1	1	1	1	1	1	1	1	1
resize cost	0	2	4	0	8	0	0	0	16
total cost	1	3	5	1	9	1	1	1	17
cumulative cost	1	4	9	10	19	20	21	22	39
average cost	1.00	2.00	3.00	2.50	3.80	3.33	3.00	2.75	4.33

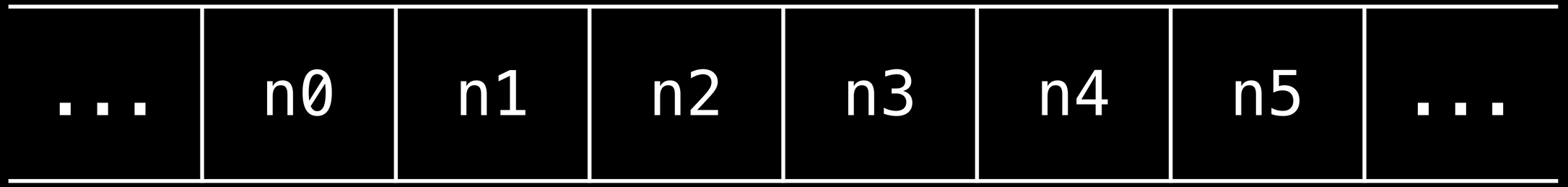
Amortized complexity

Average the time required to perform a sequence of operations over all the operations performed.



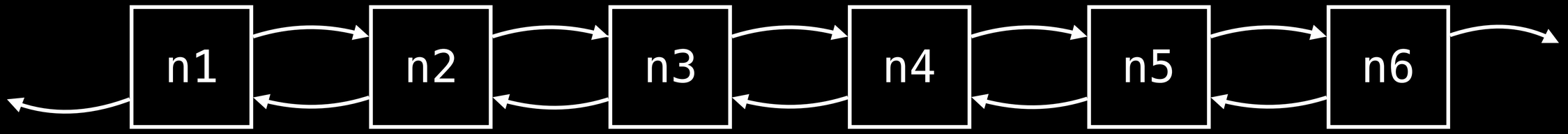


Deque

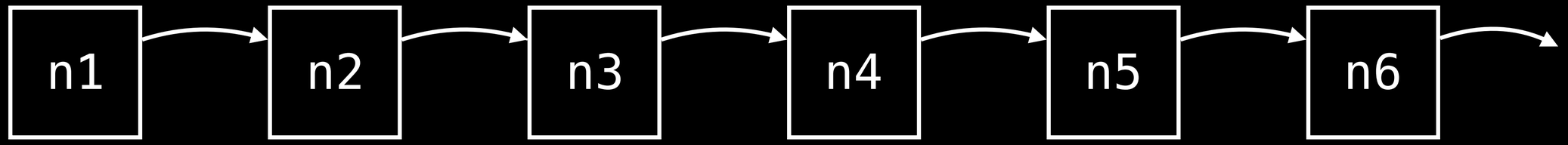


Array

List



Forward List



Sequence containers summary

Type				
Vector				
Deque				
Array				
List				
Forward List				

Iterator invalidation

`vector`
iterators, pointers and references are invalidated after insertion if new container size is greater than the previous capacity.

`deque`
iterators, pointers and references are invalidated after insertion (unless inserting at the front/end, then references are unaffected).

`list`
iterators, pointers and references are unaffected.

```
break;
```

Queues and stacks

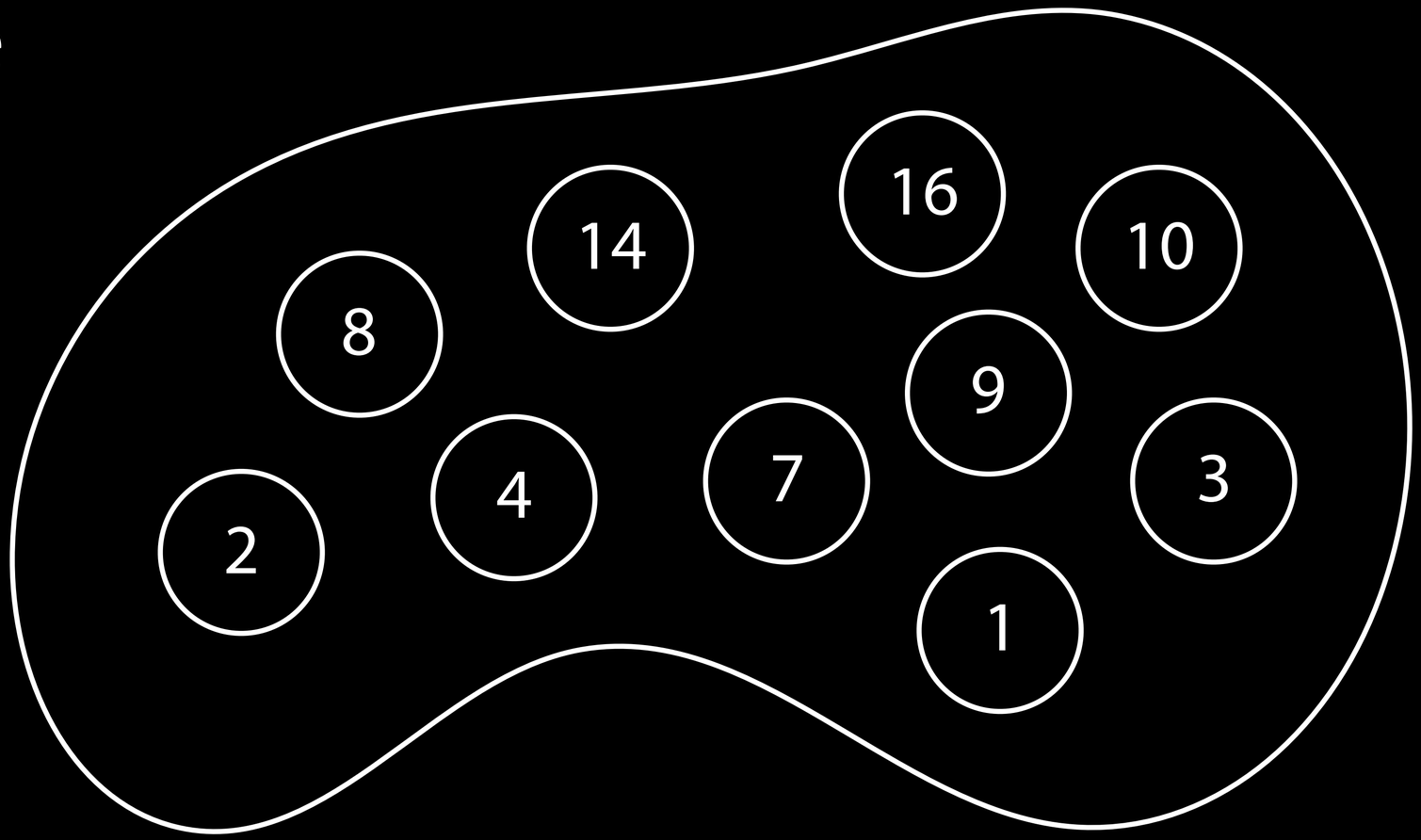
Priority Queue

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.

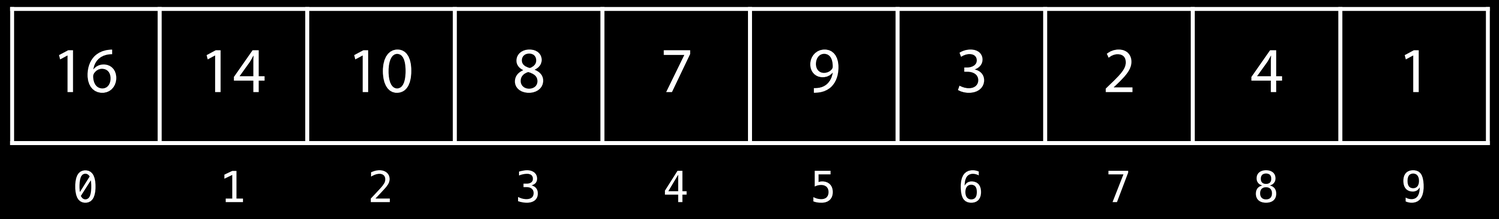
insert(*S*, x)	insert element ***x*** into set ***S***
max(*S*)	return element of ***S*** with largest key
removeMax(*S*)	return element of ***S*** with largest key and remove it from ***S***
increaseKey(*S*, *x*, *k*)	increase the value of element ***x***'s key to new value ***k*** (assumed to be as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

Array as a Priority Queue

Heaps

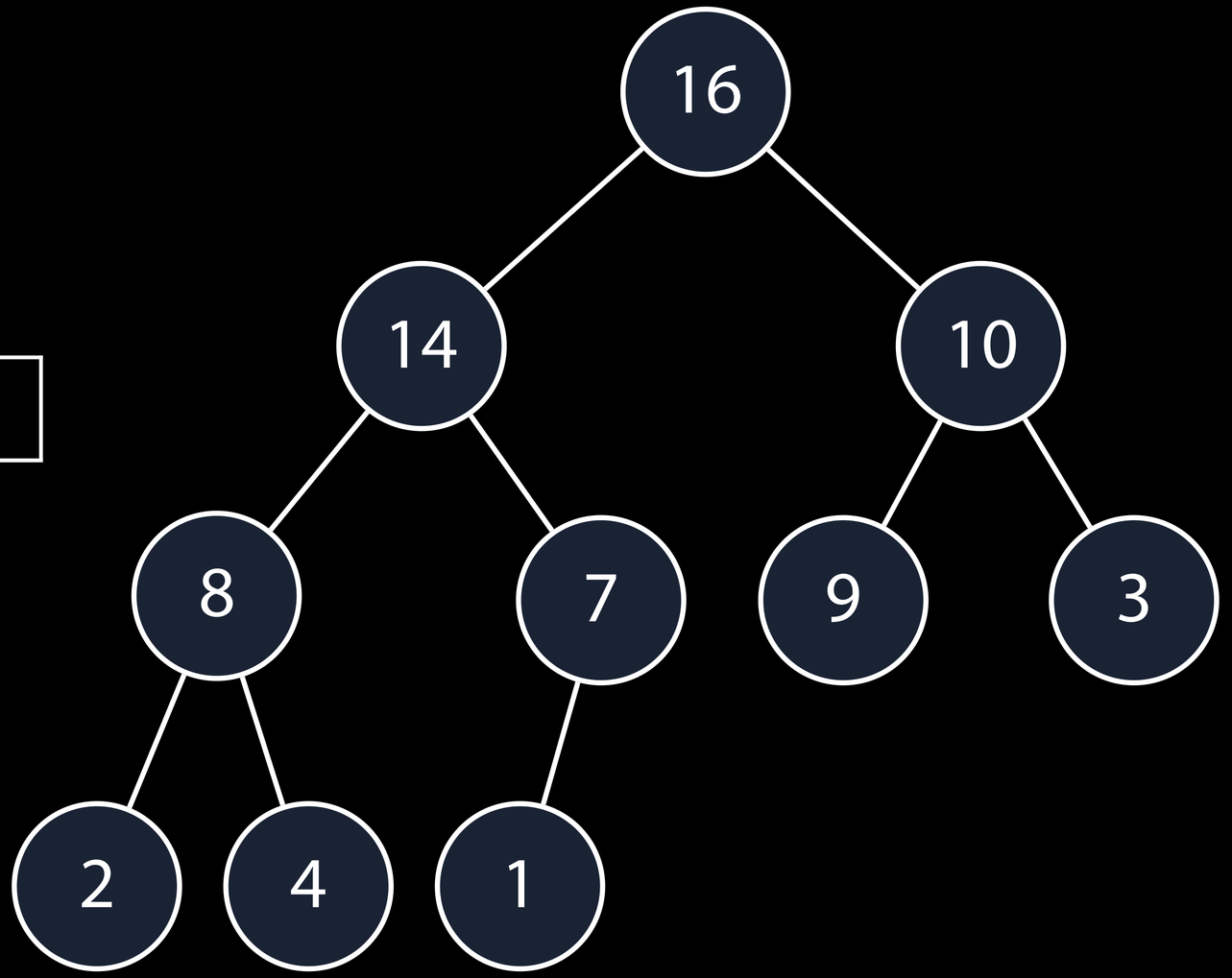
Heap

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



Heap

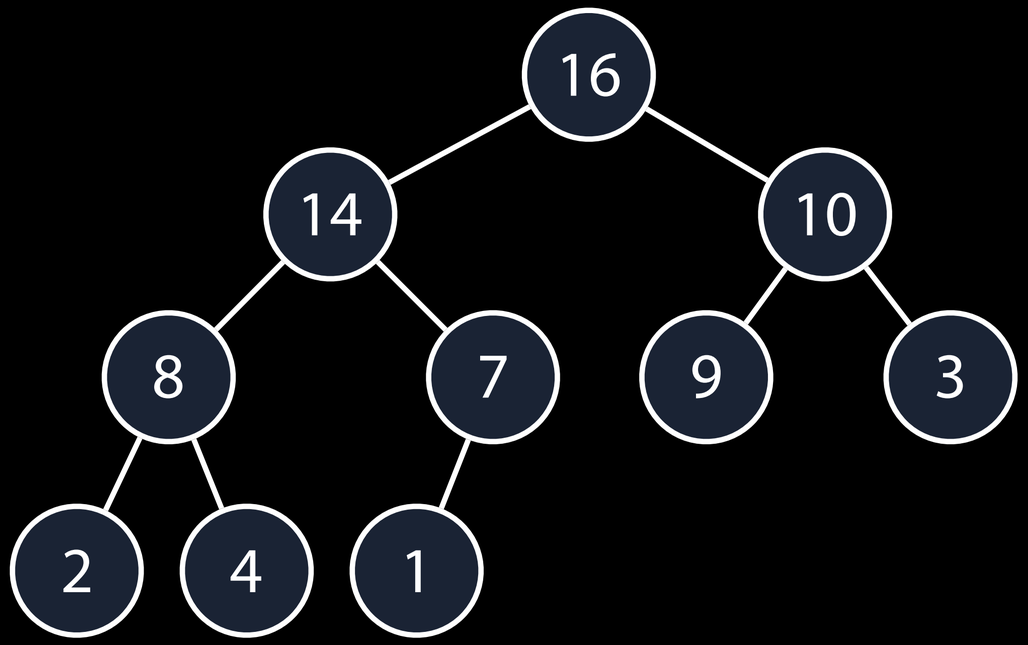
16	14	10	8	7	9	3	2	4	1
0	1	2	3	4	5	6	7	8	9



Heap property

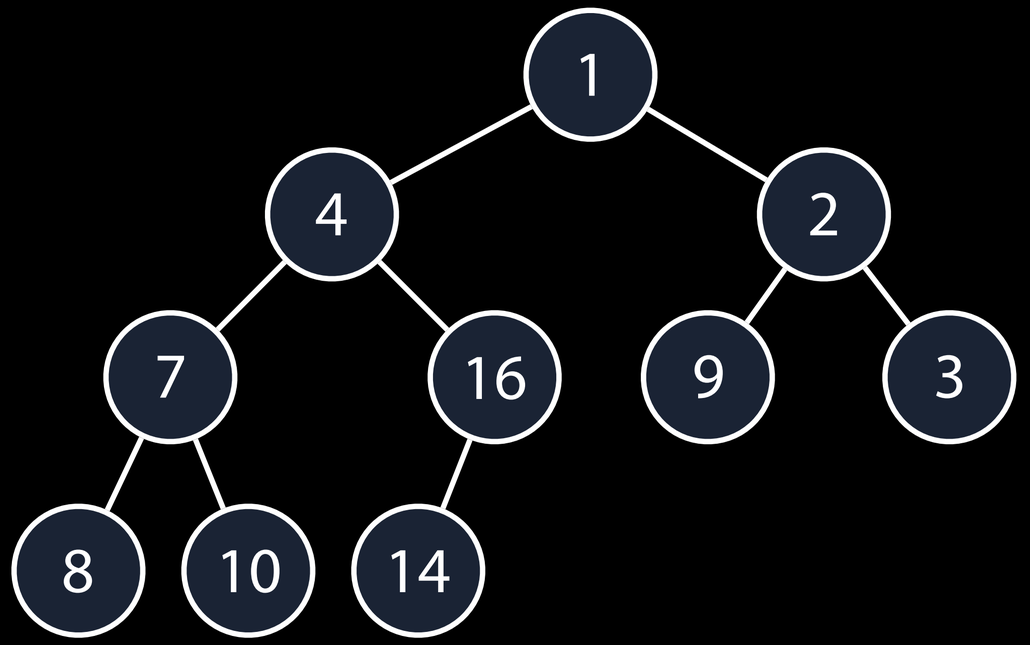
Max-heap

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



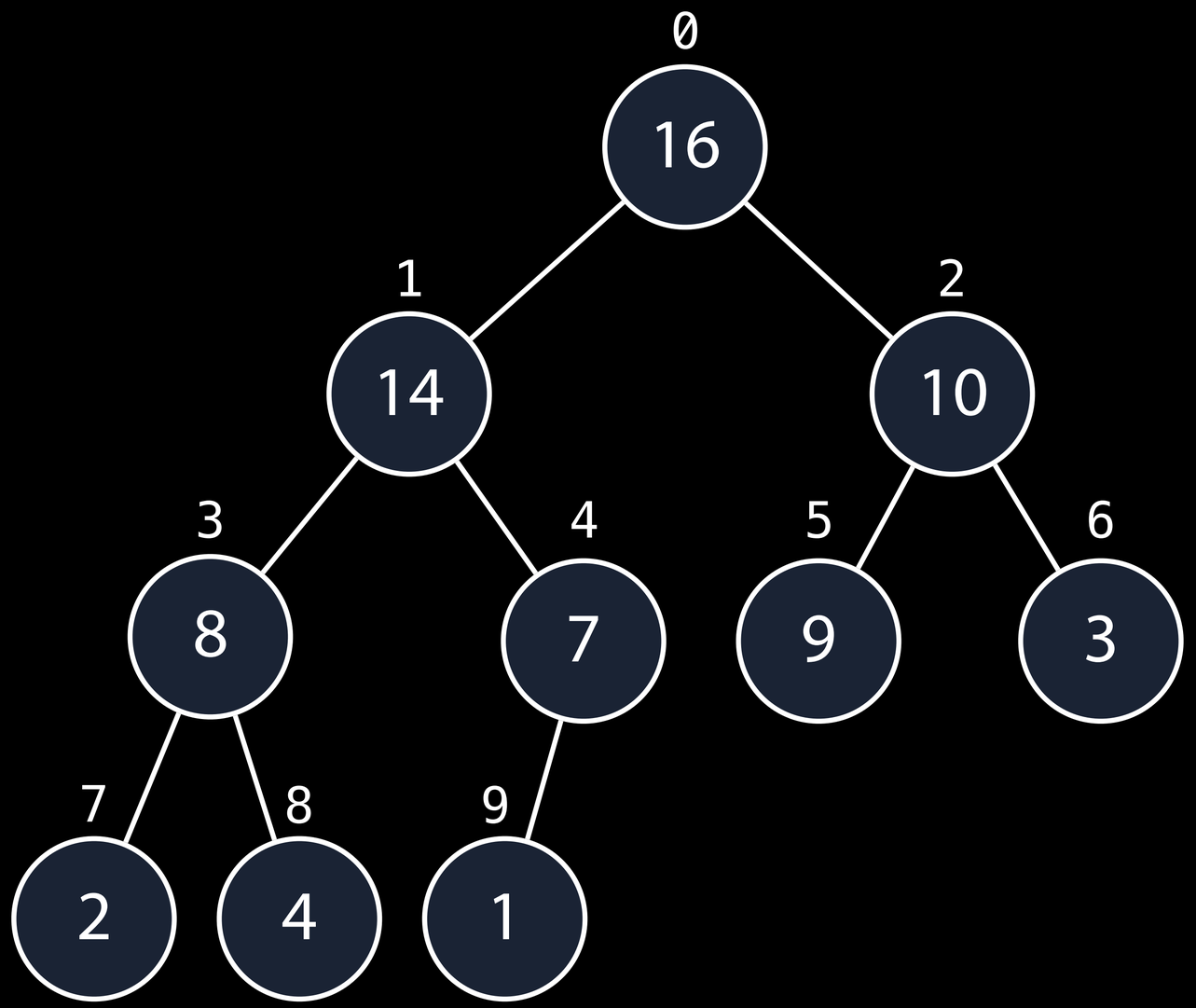
Min-heap

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



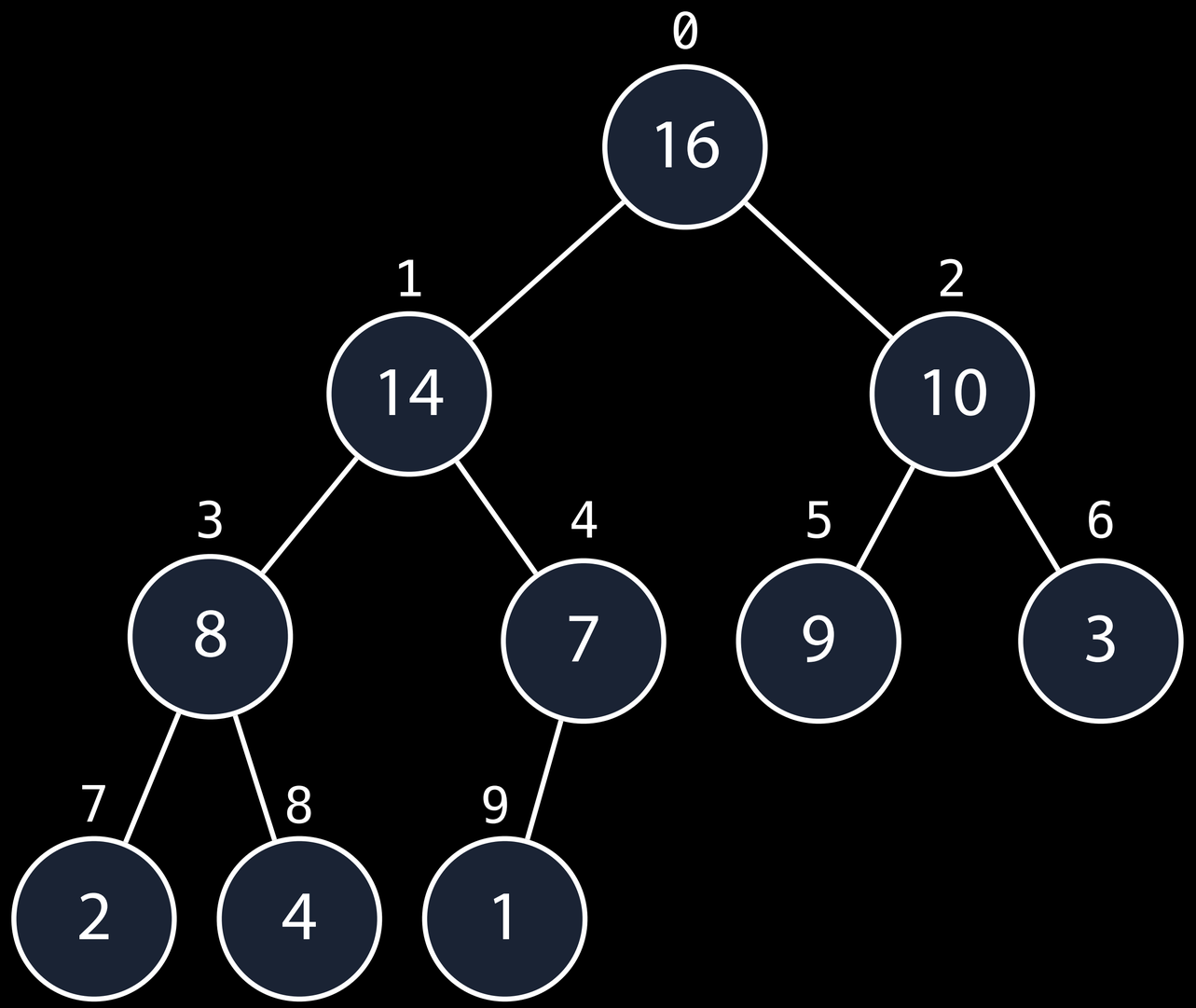
Heap as a tree

```
root()

parent(i)

left(i)

right(i)

height()
```

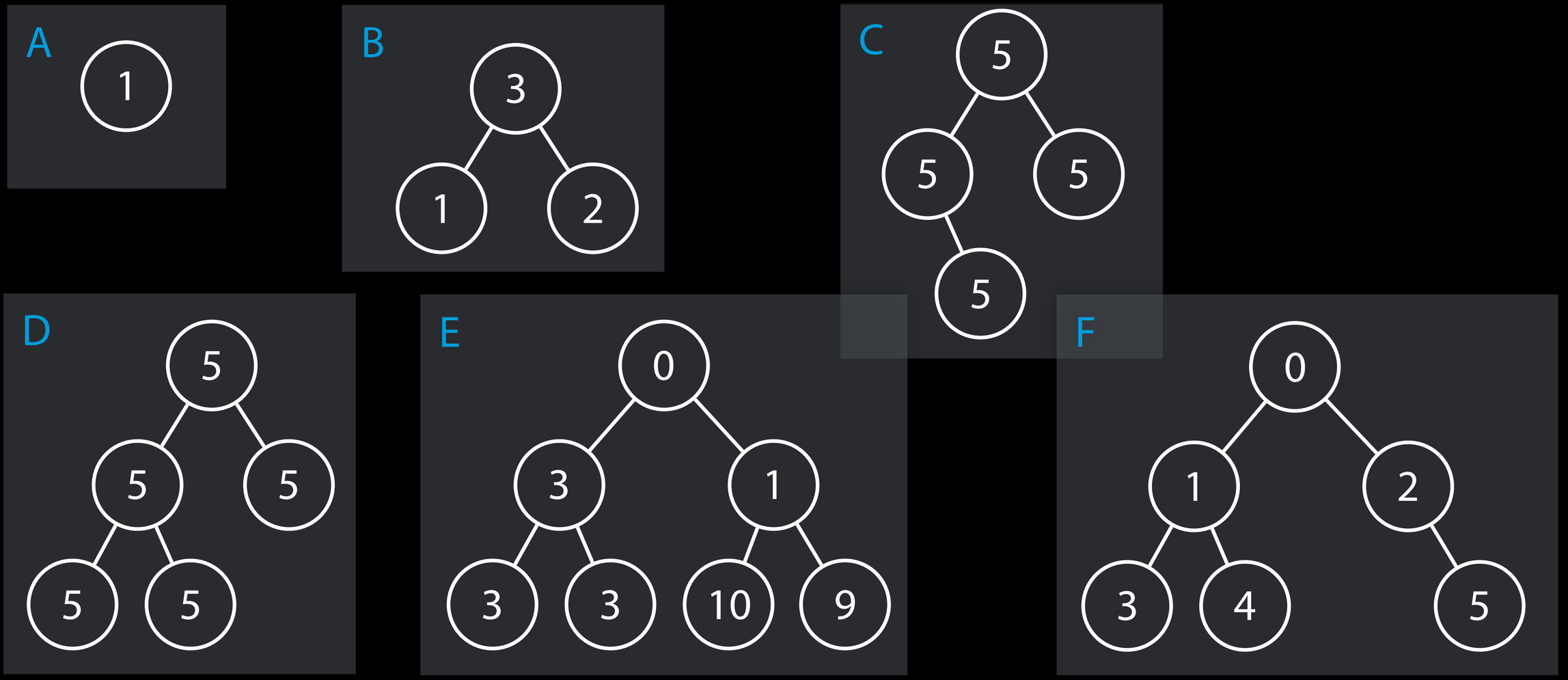


Heap as a tree

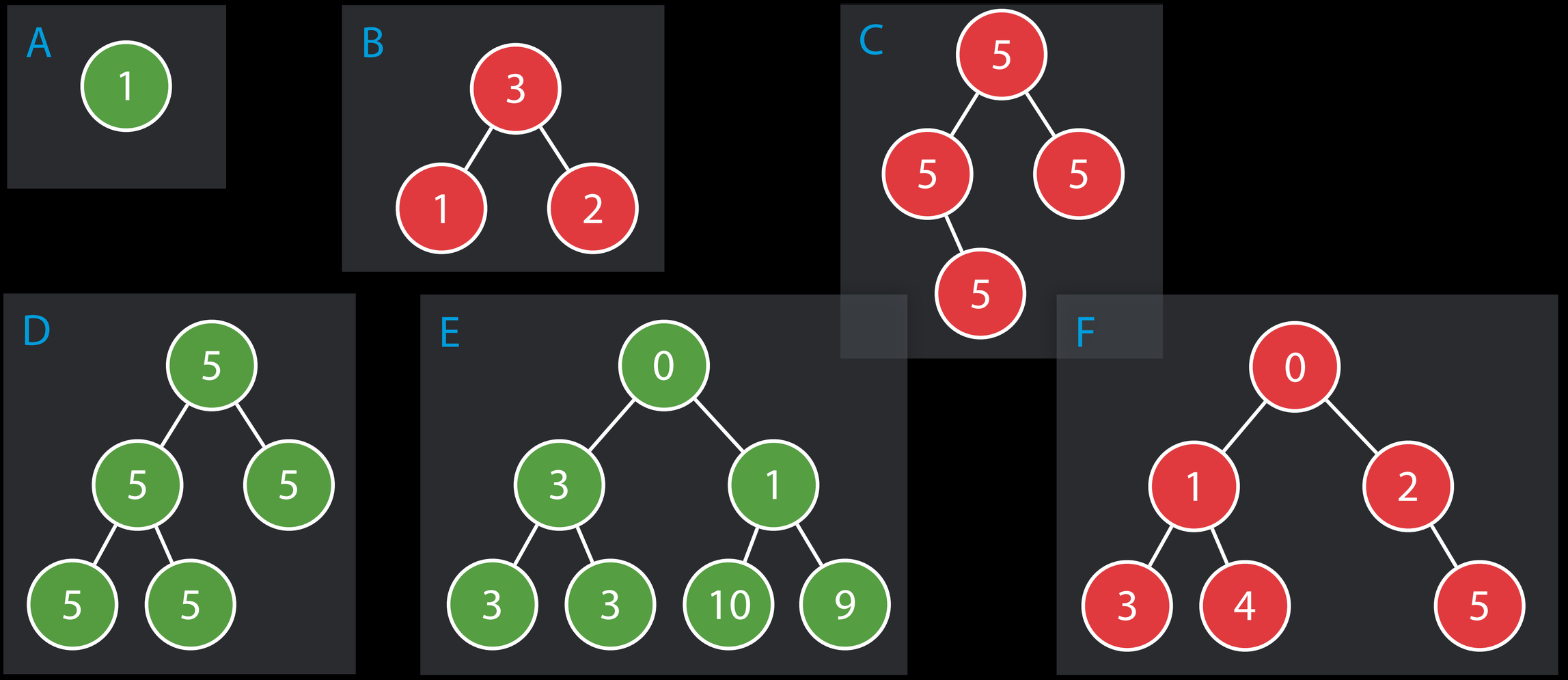
```
root(): A[0]

parent(i): (i – 1) / 2

left(i): 2i + 1

right(i): 2i + 2

height(): ⌈log(n + 1)⌉
```



Which are valid min-heaps?



Which are valid min-heaps?



Heap operations

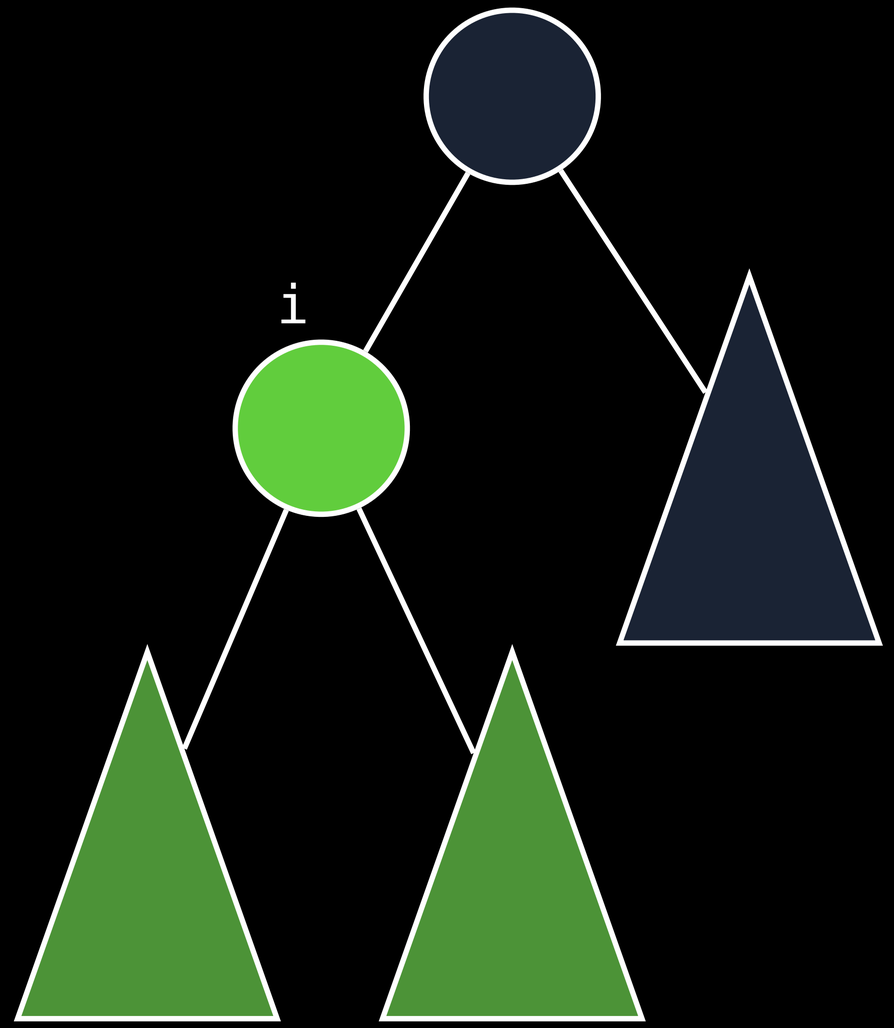
buildMaxHeap	produce a max-heap from an unordered array
maxHeapify(*i*)	correct a single violation of the heap property in a subtree with root at ***i***
maxElement	return the largest element
removeMaxElement	remove the largest element
increaseKey(*i*, *k*)	update the key of an element to a larger value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

maxHeapify(i)

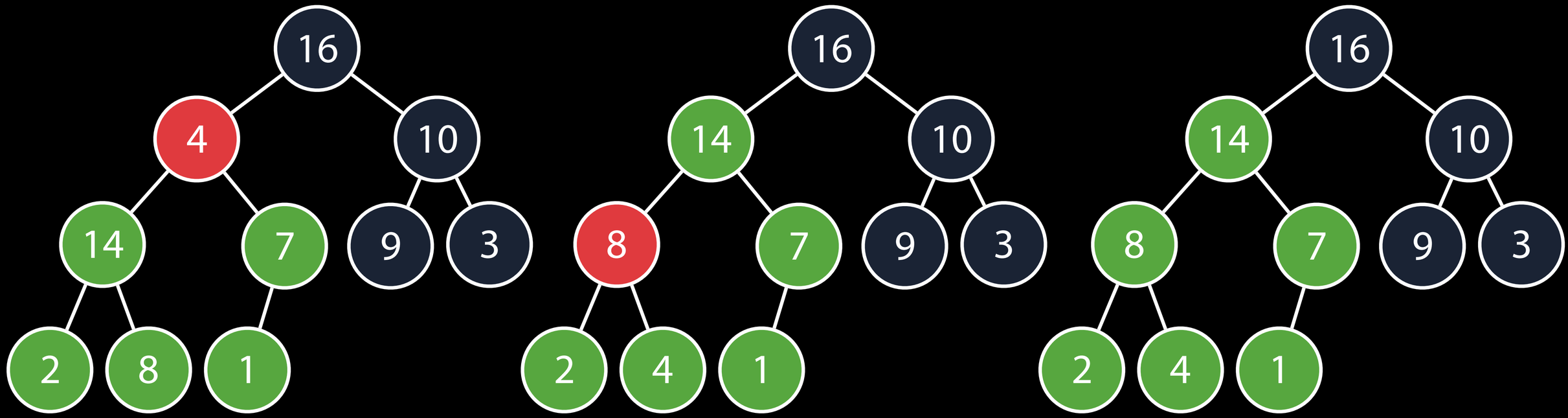
Correct a single violation of the heap property in a subtree with root at `i`

Assume the left and right children are valid max-heaps

Keep swapping with the largest child until the violation is corrected

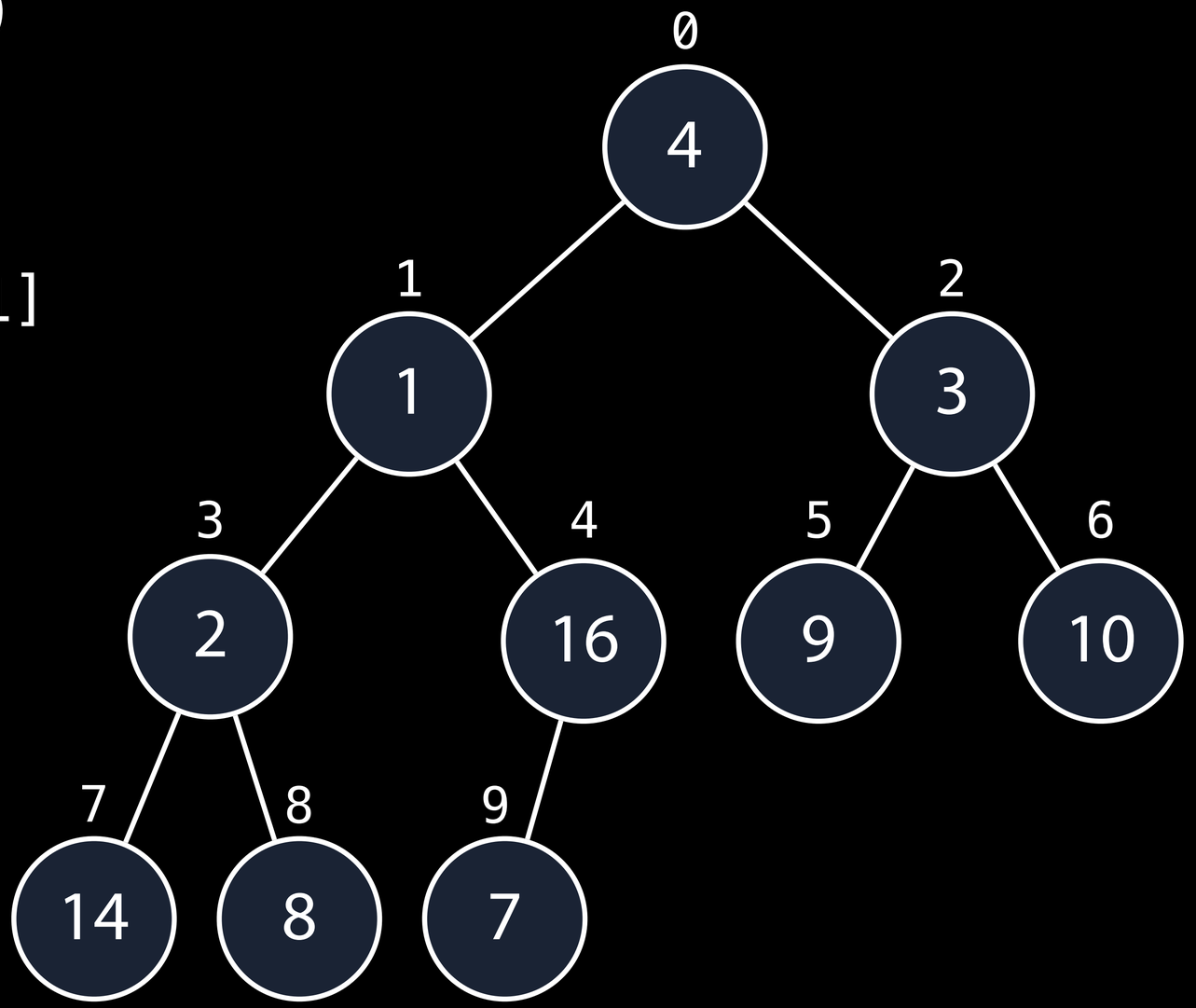


maxHeapify



buildMaxHeap

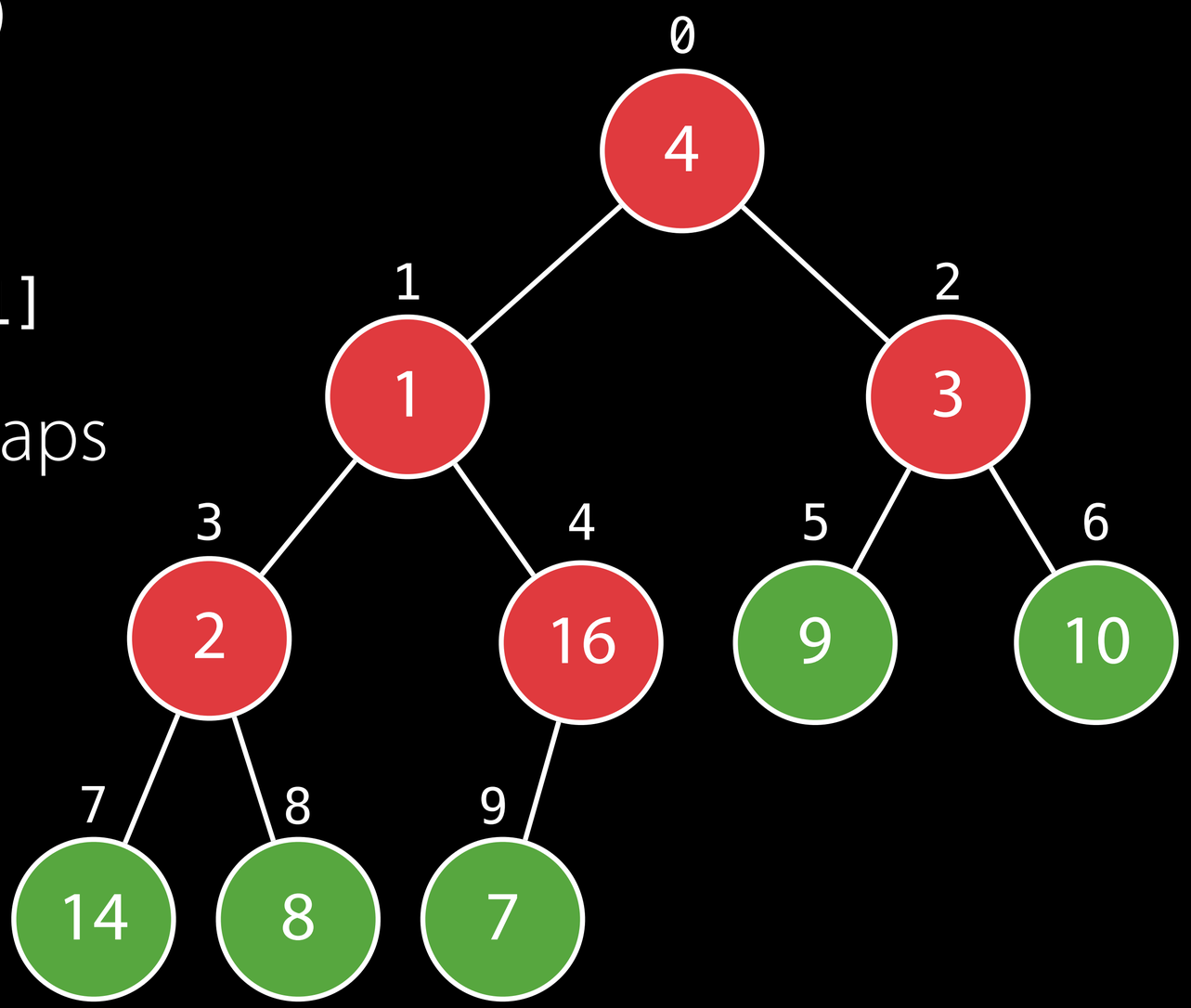
Produce a max-heap from an unordered array `A[0 ... n - 1]`



buildMaxHeap

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

Leaves are always valid max-heaps



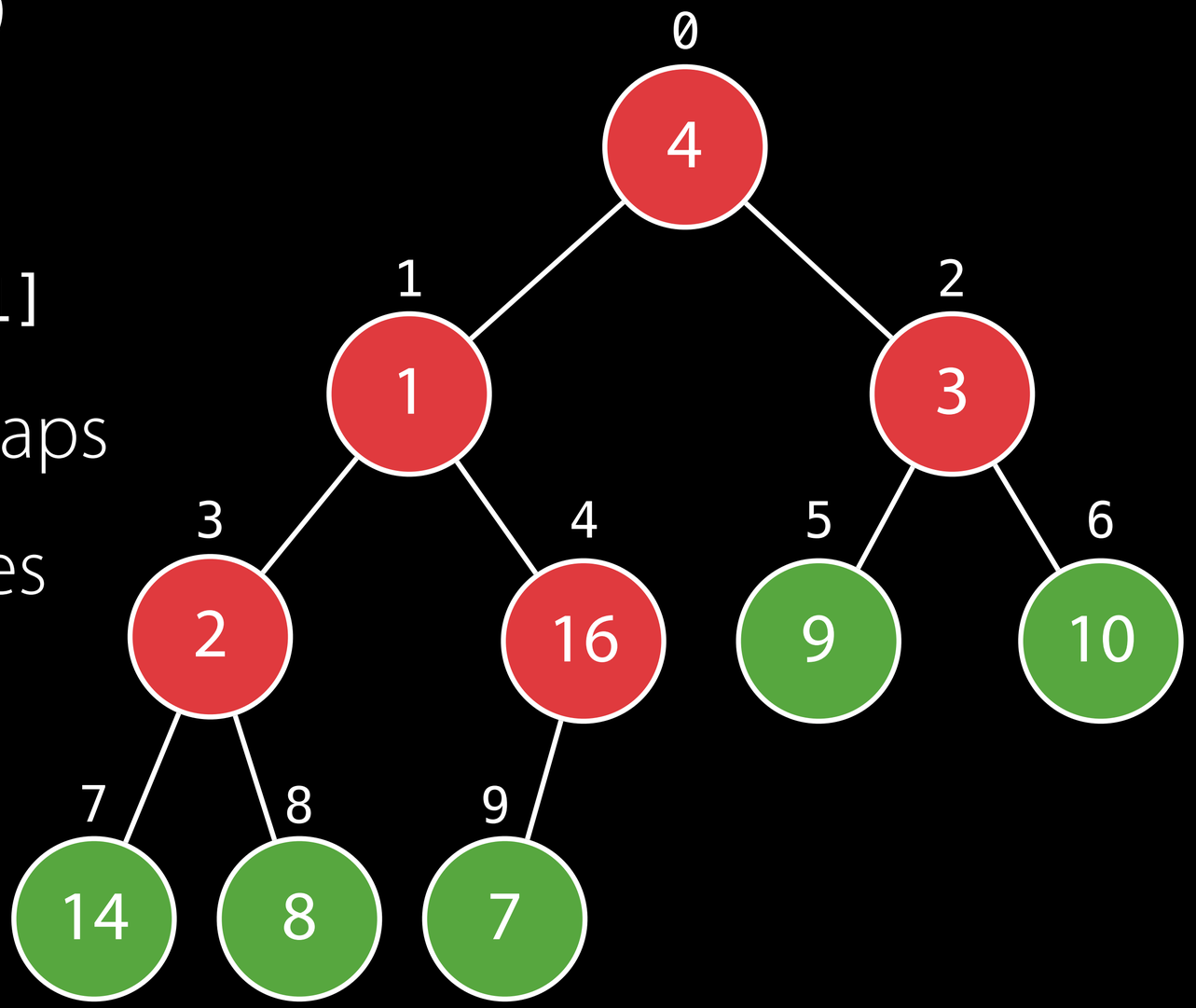
buildMaxHeap

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

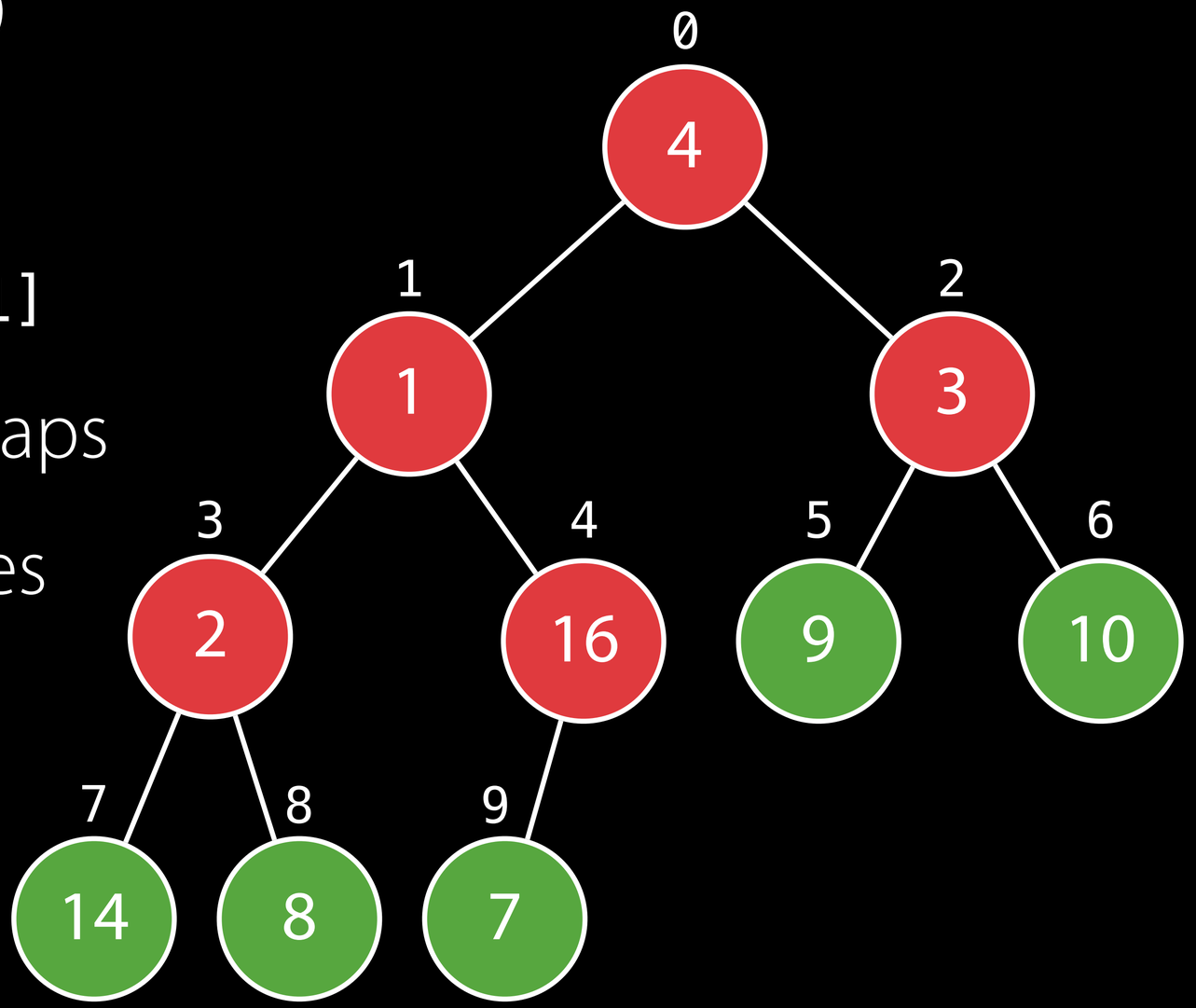
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(4)`



buildMaxHeap

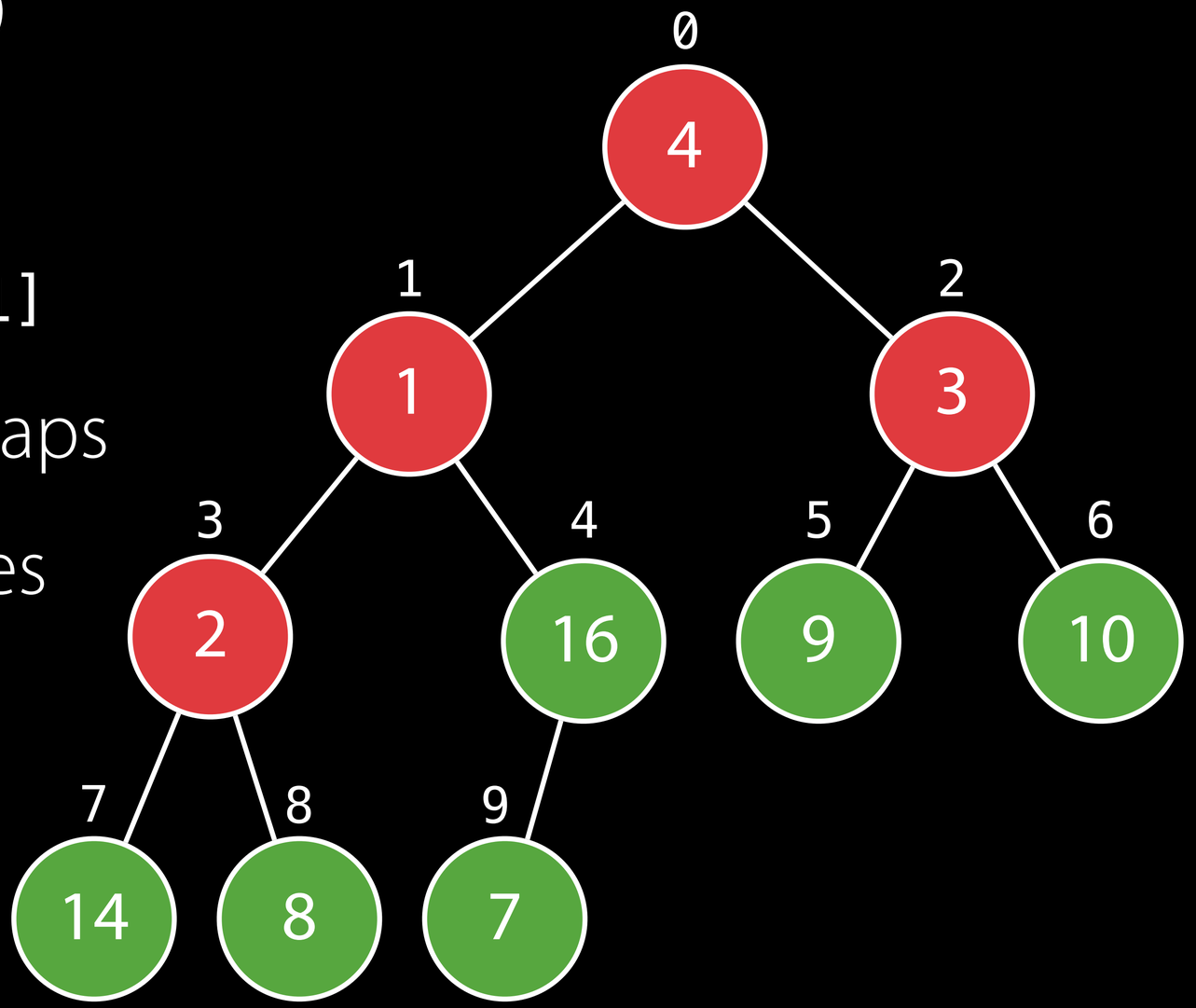
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(4)`



buildMaxHeap

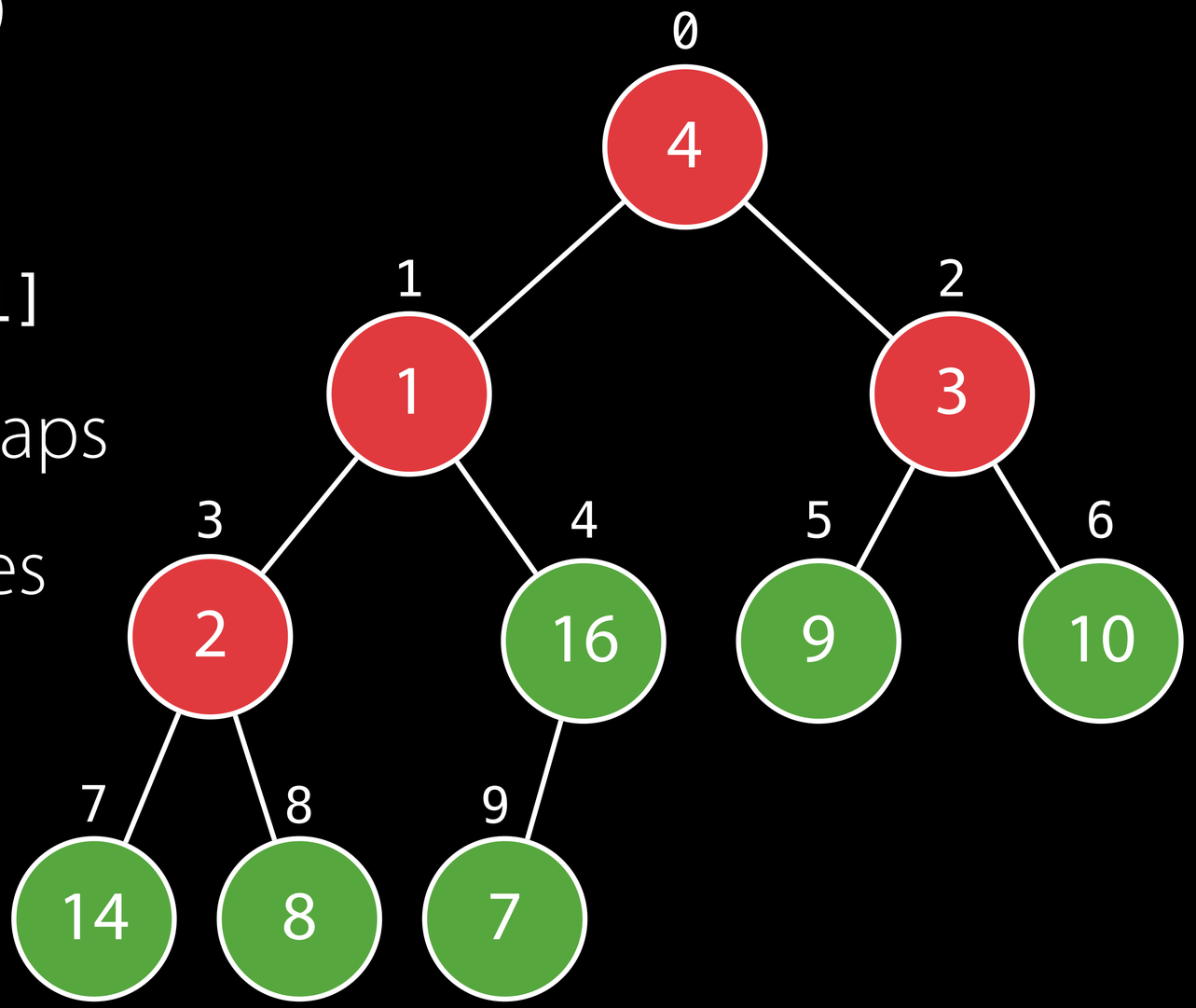
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(3)`



buildMaxHeap

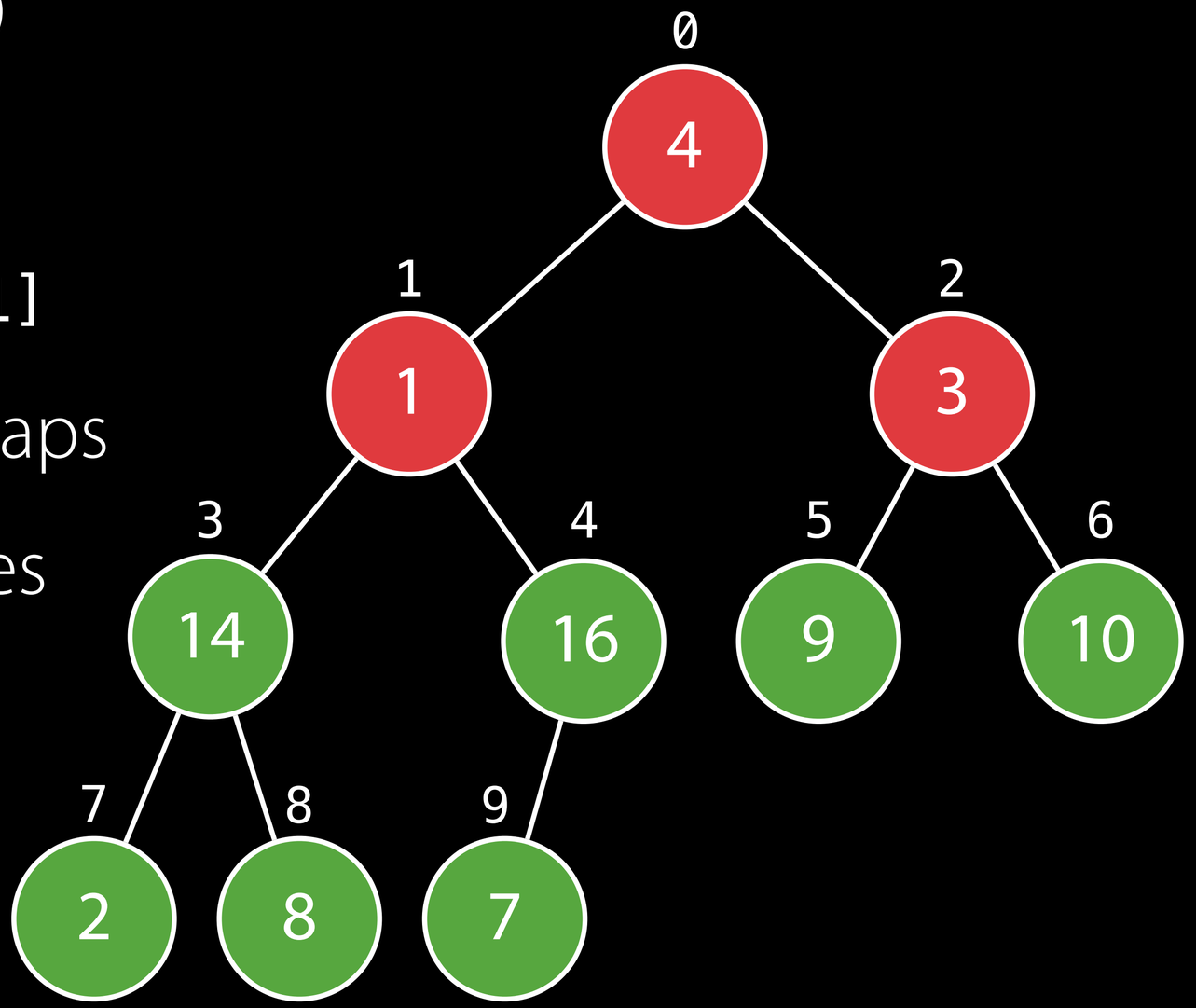
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(3)`



buildMaxHeap

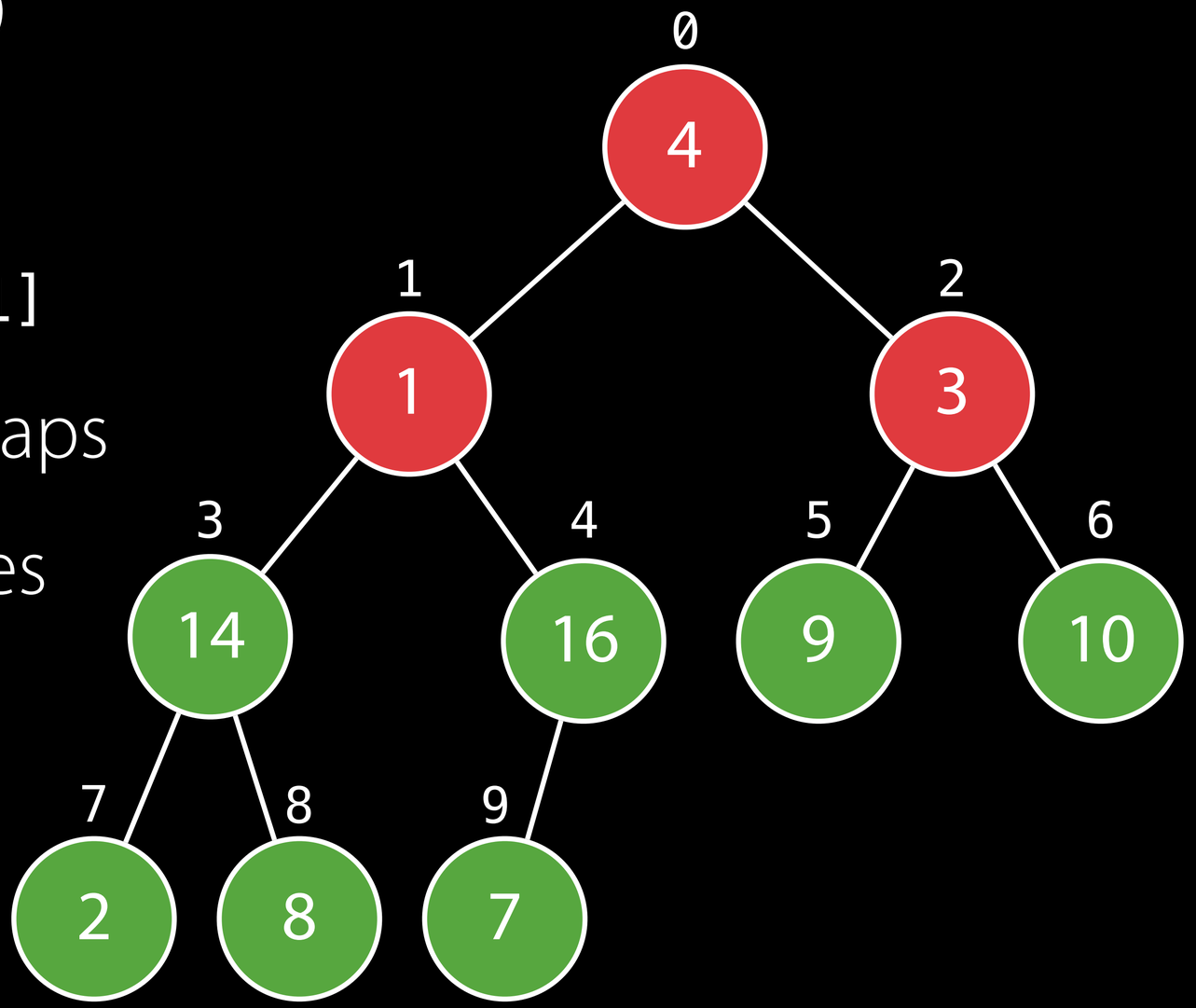
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(2)`



buildMaxHeap

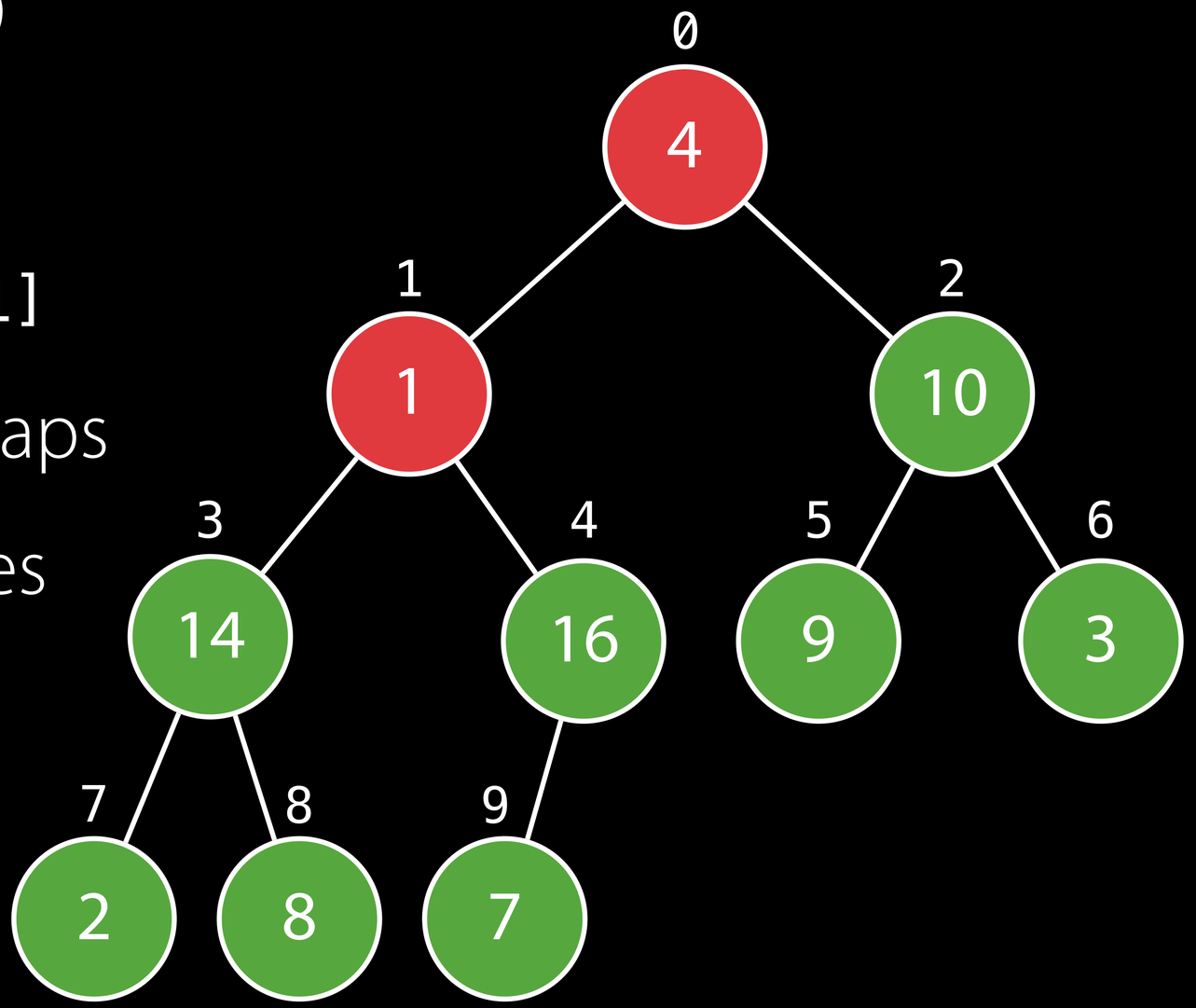
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(2)`



buildMaxHeap

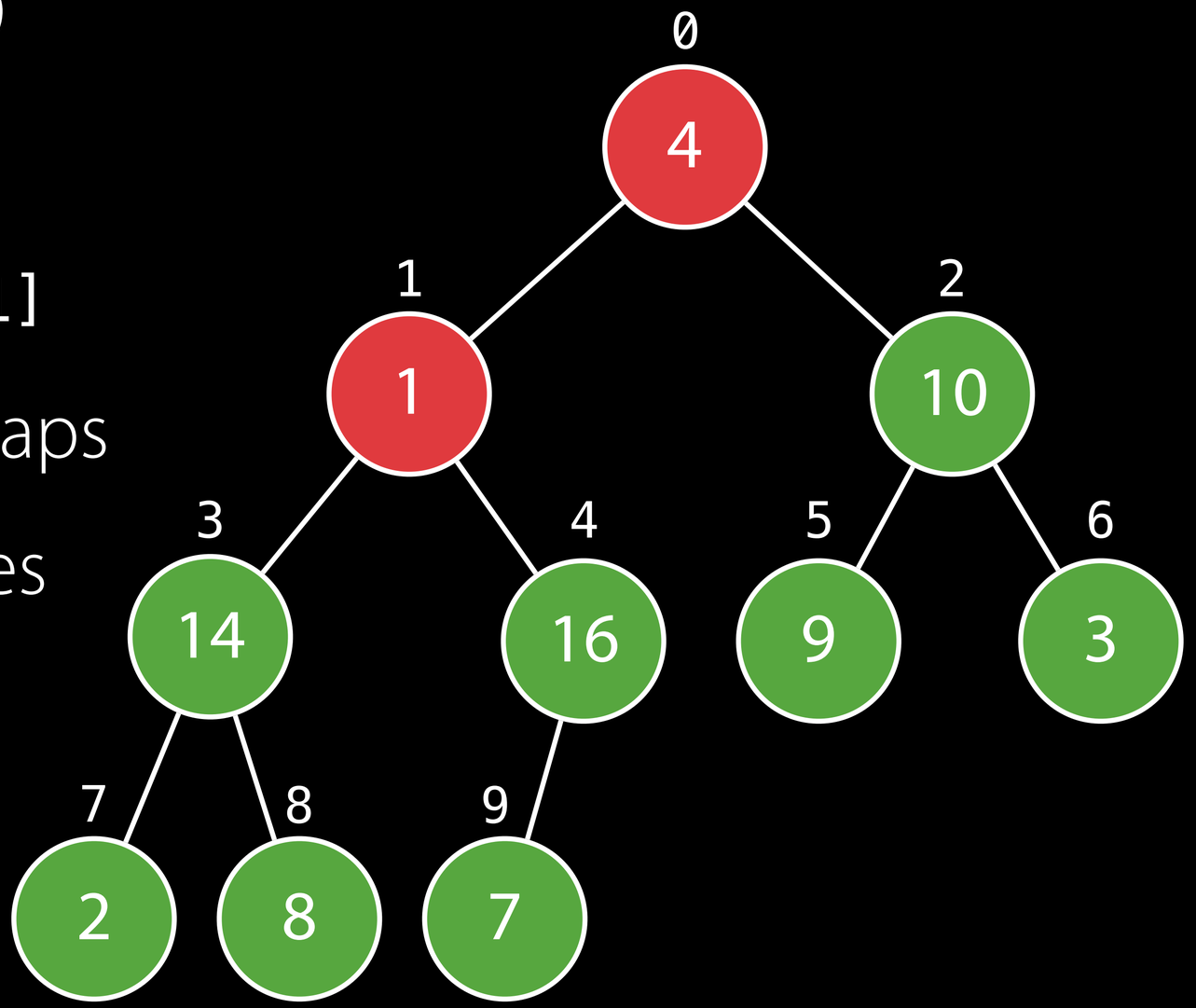
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(1)`



buildMaxHeap

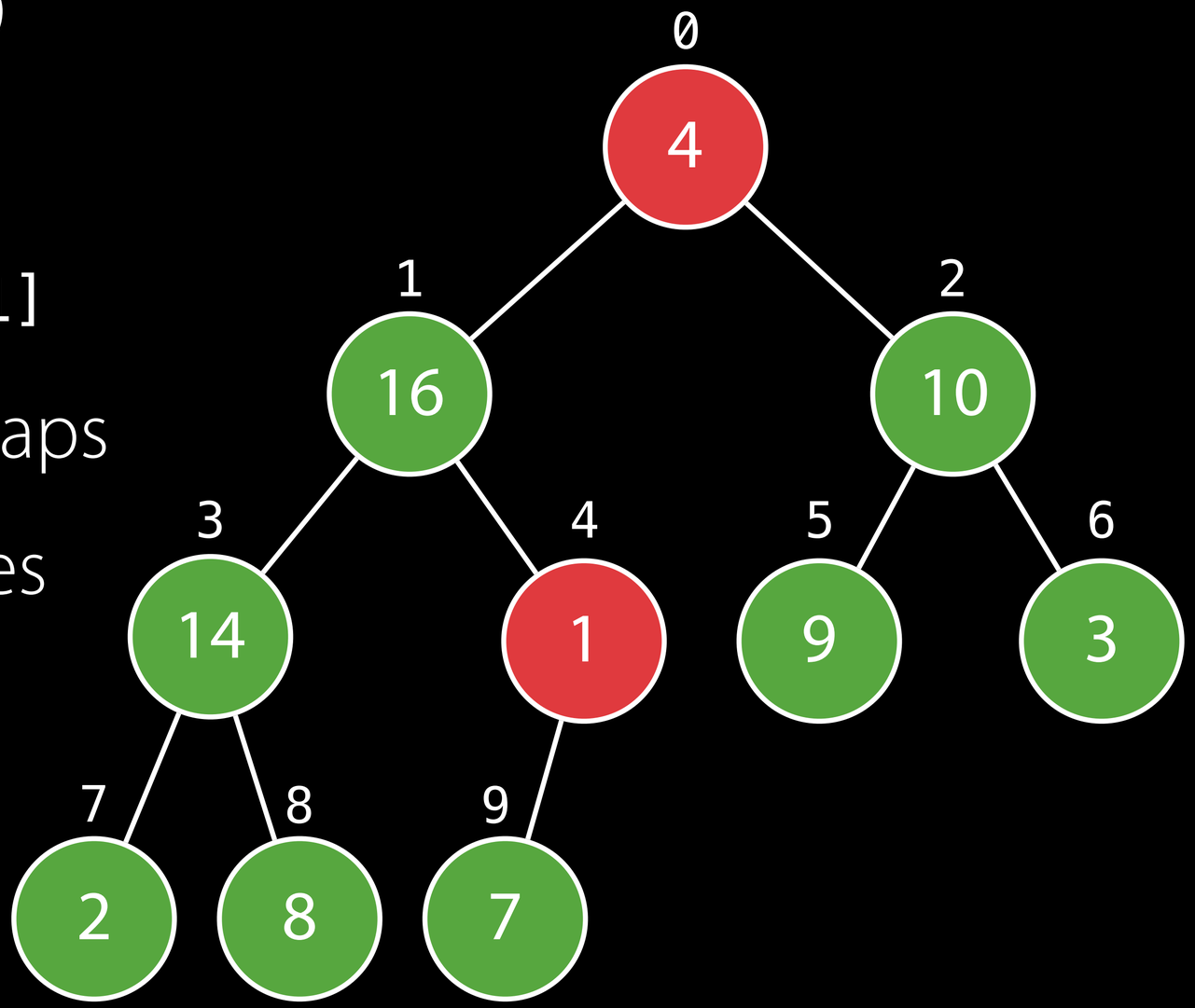
Produce a max-heap from an unordered array `A[0 ... n – 1]`

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]`

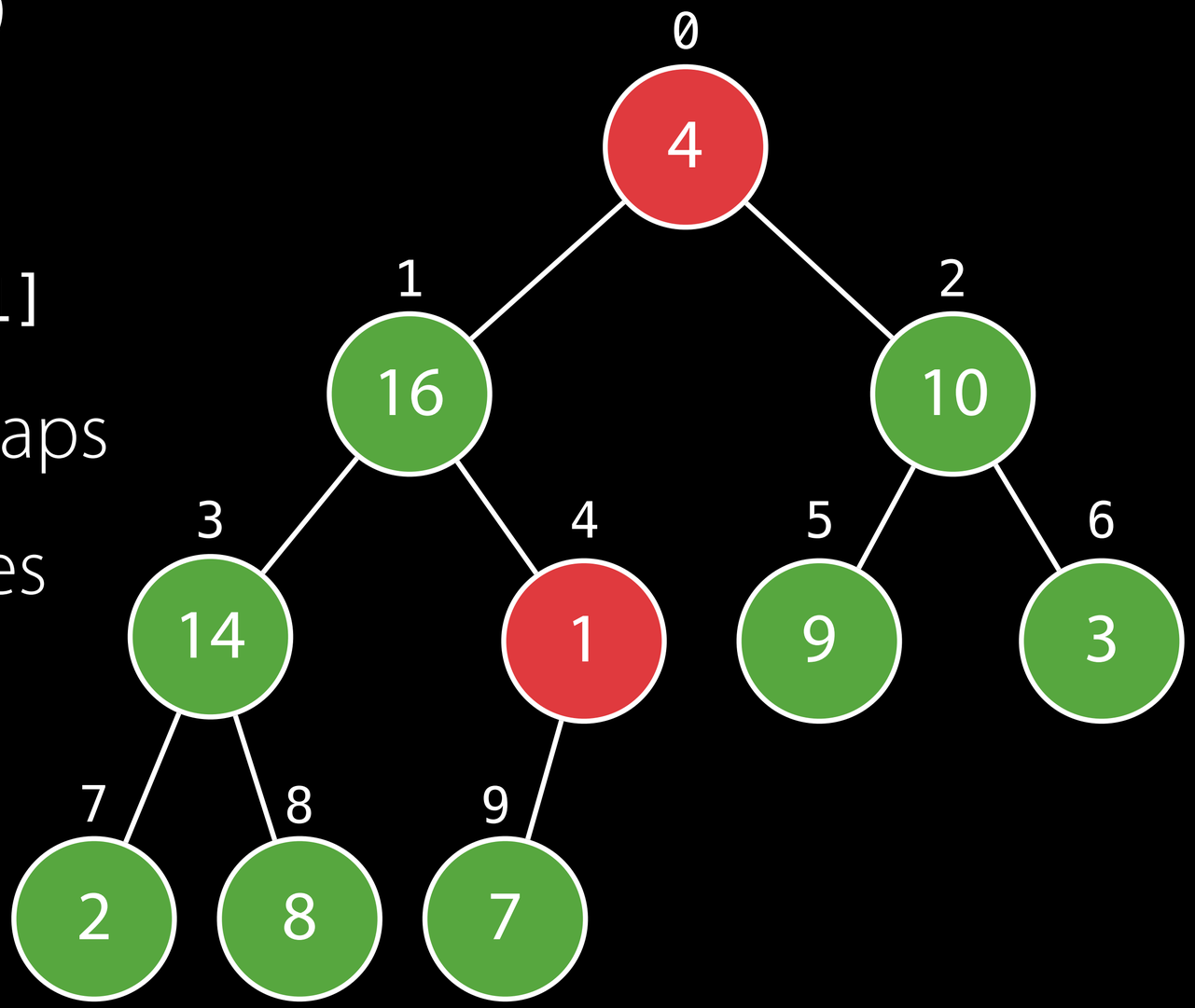
Leaves are always valid max-heaps

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]`

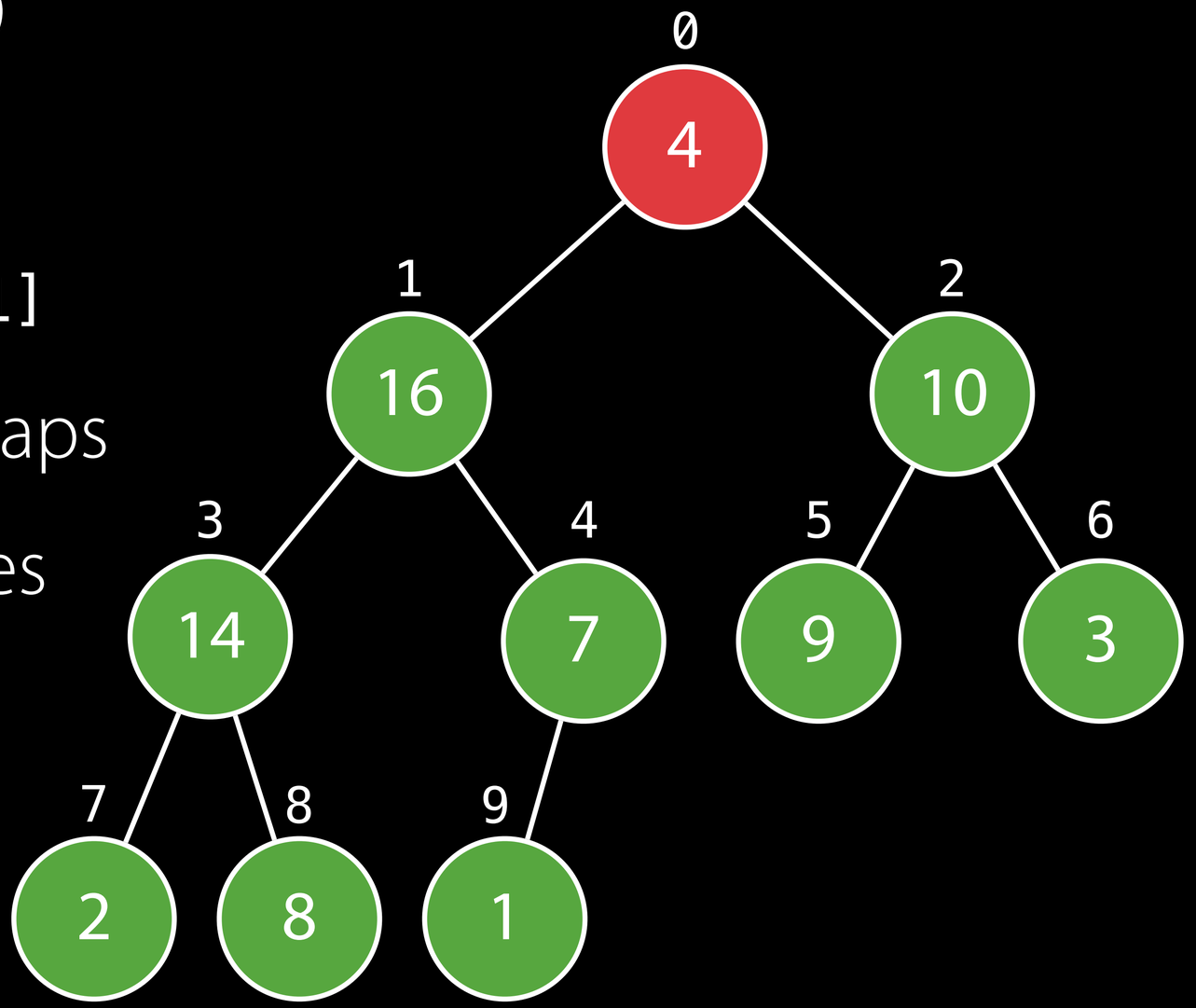
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(1)`

`maxHeapify(4)`



buildMaxHeap

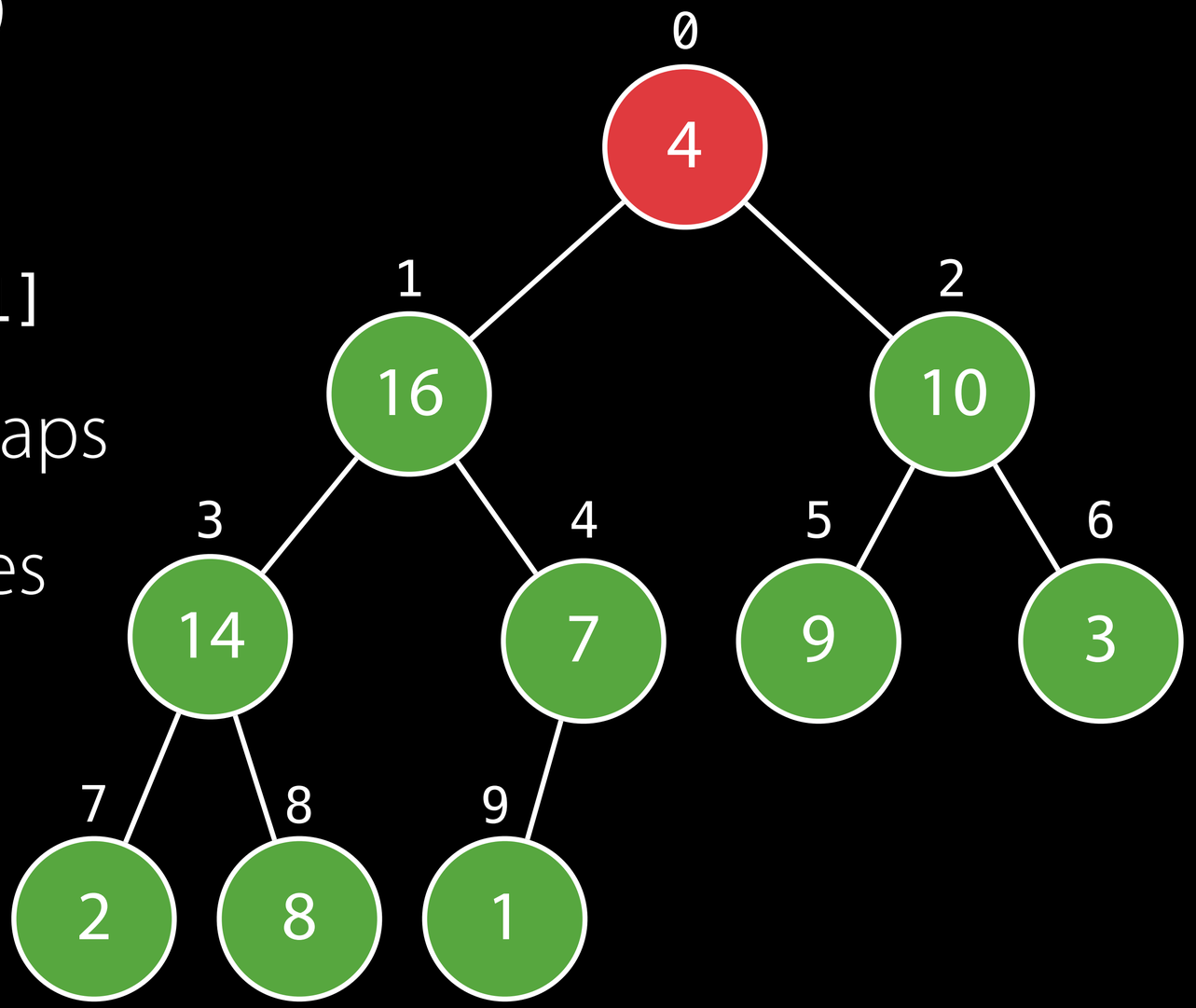
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(0)`



buildMaxHeap

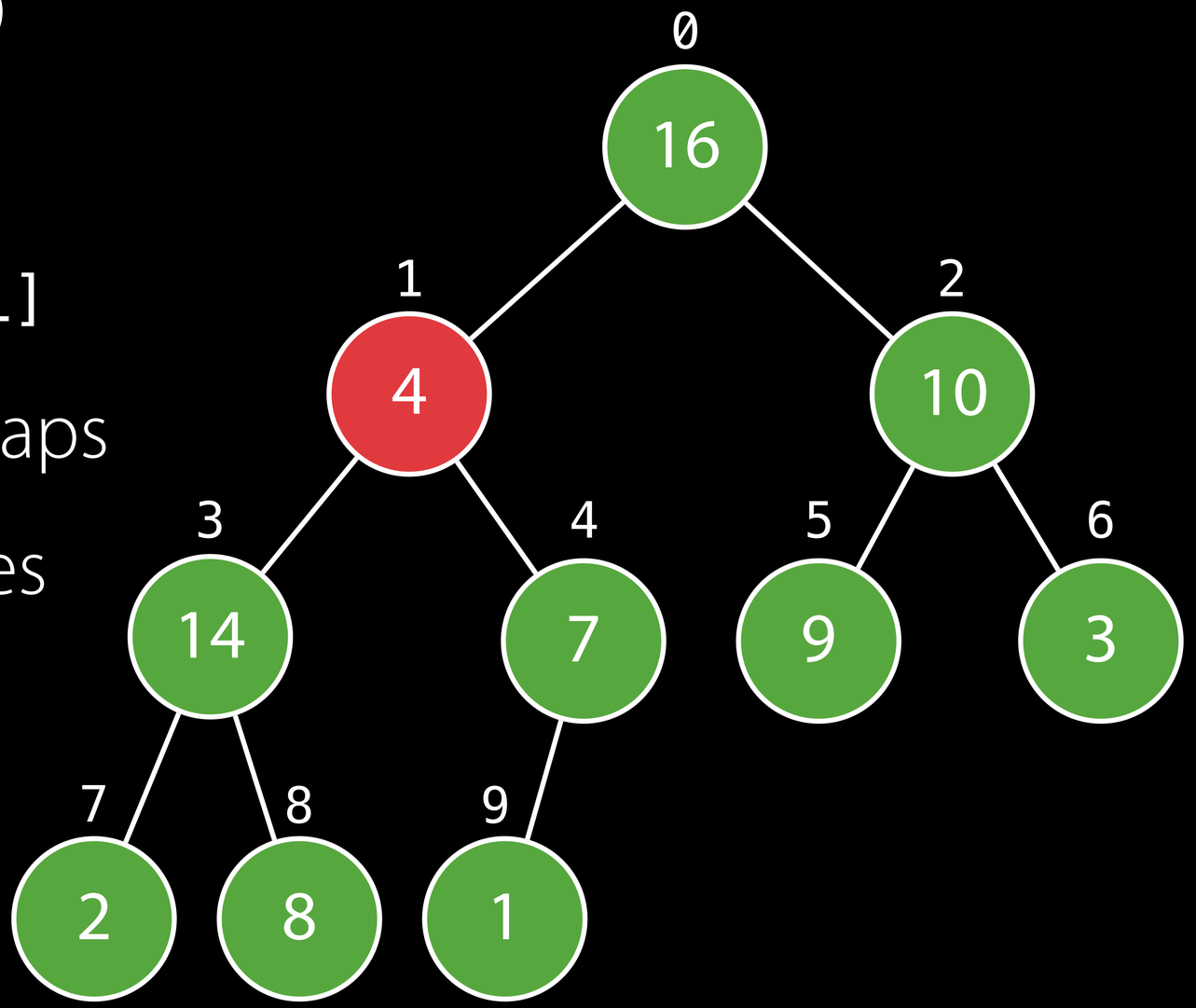
Produce a max-heap from an unordered array `A[0 ... n – 1]`

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]`

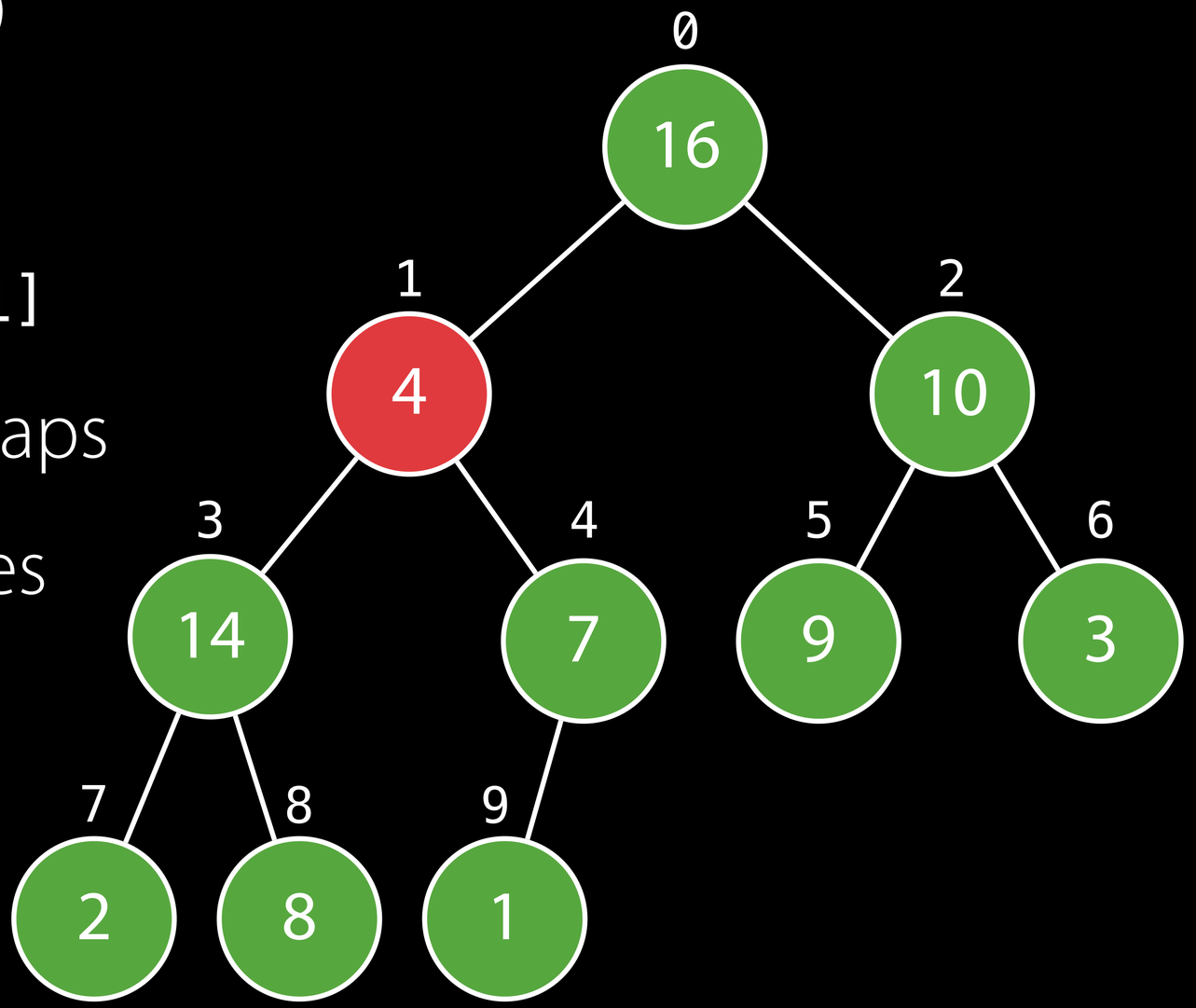
Leaves are always valid max-heaps

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]`

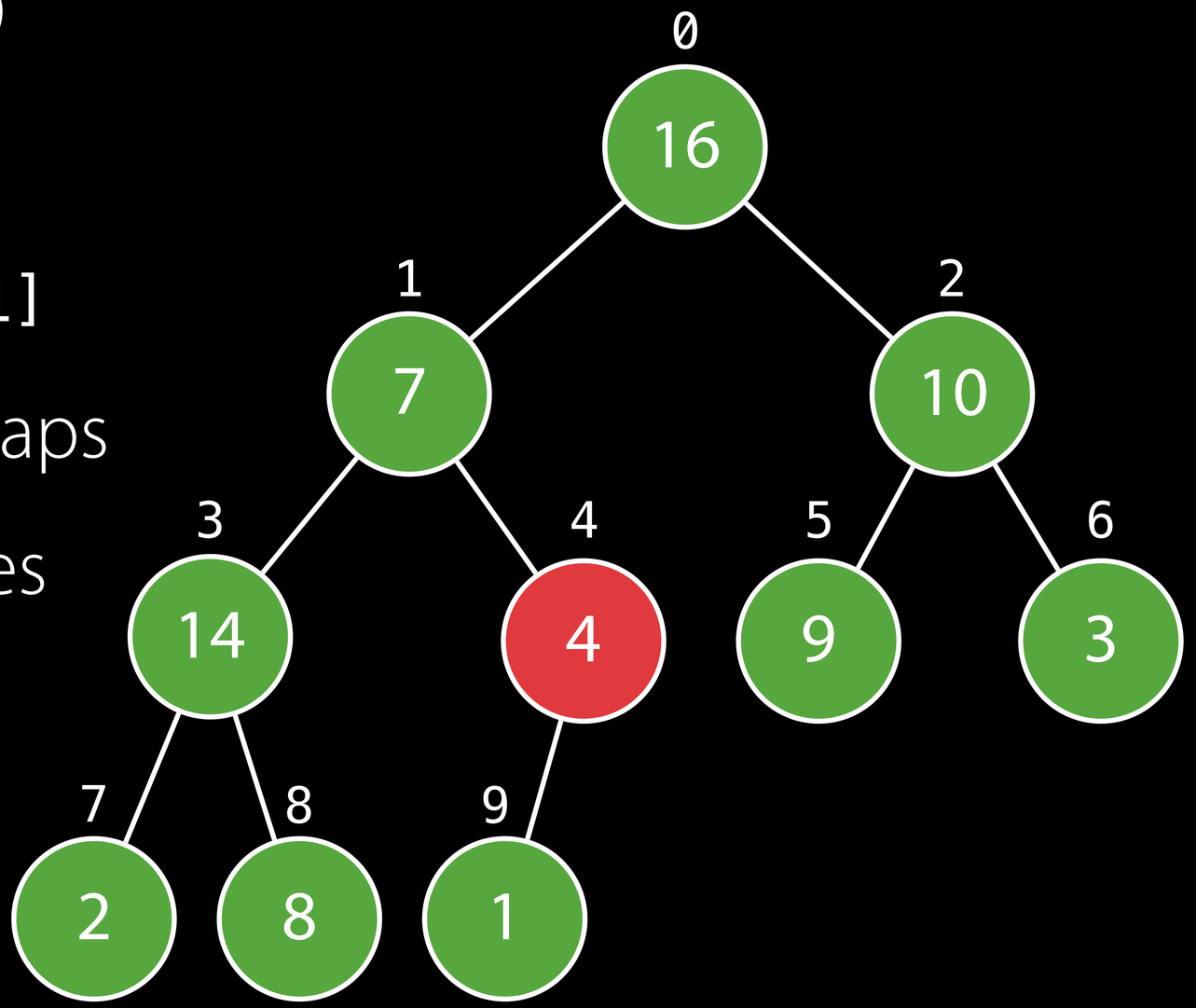
Leaves are always valid max-heaps

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]`

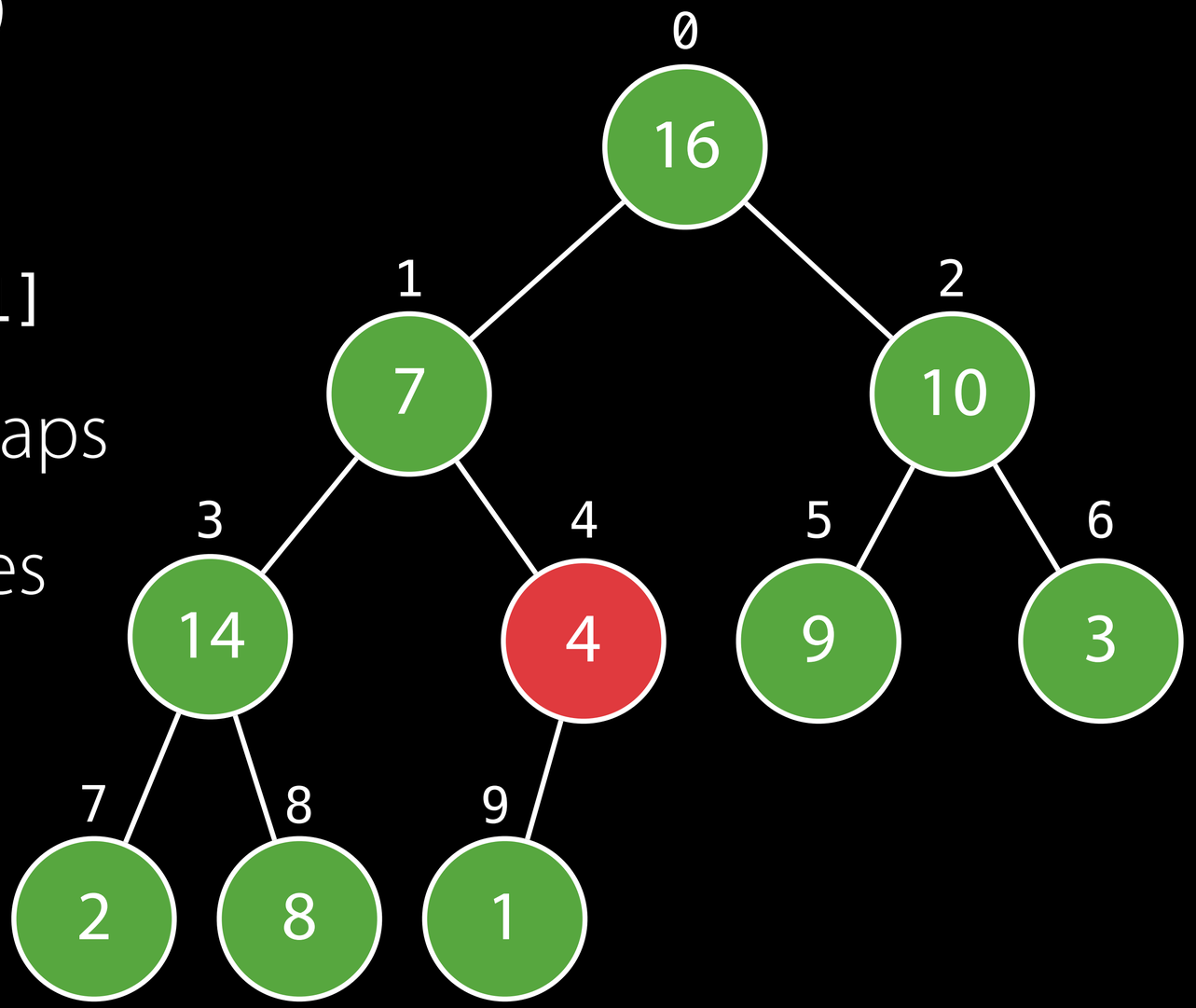
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(0)`

`maxHeapify(4)`



buildMaxHeap

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

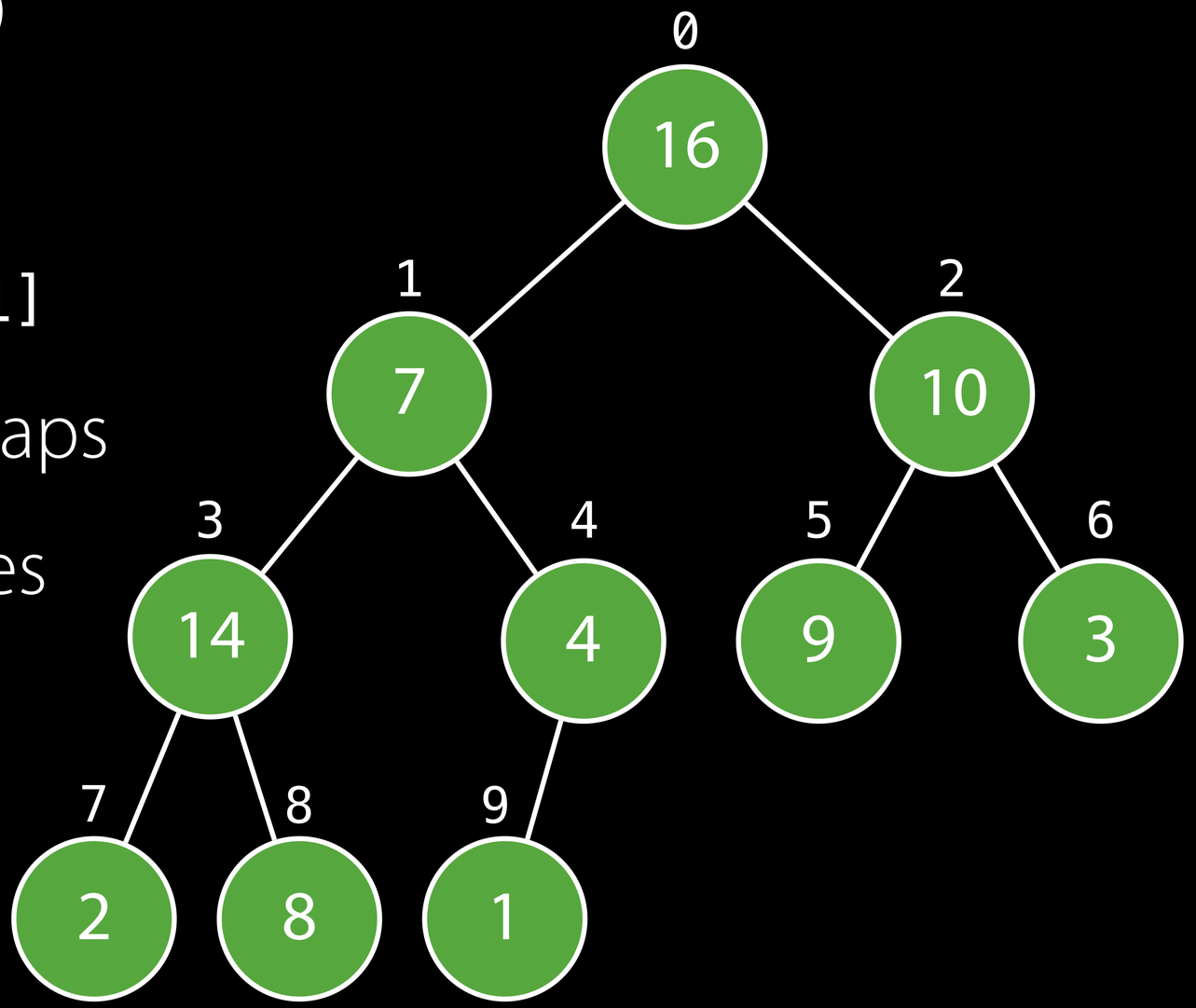
Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

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

`maxHeapify(0)`

`maxHeapify(4)`



buildMaxHeap

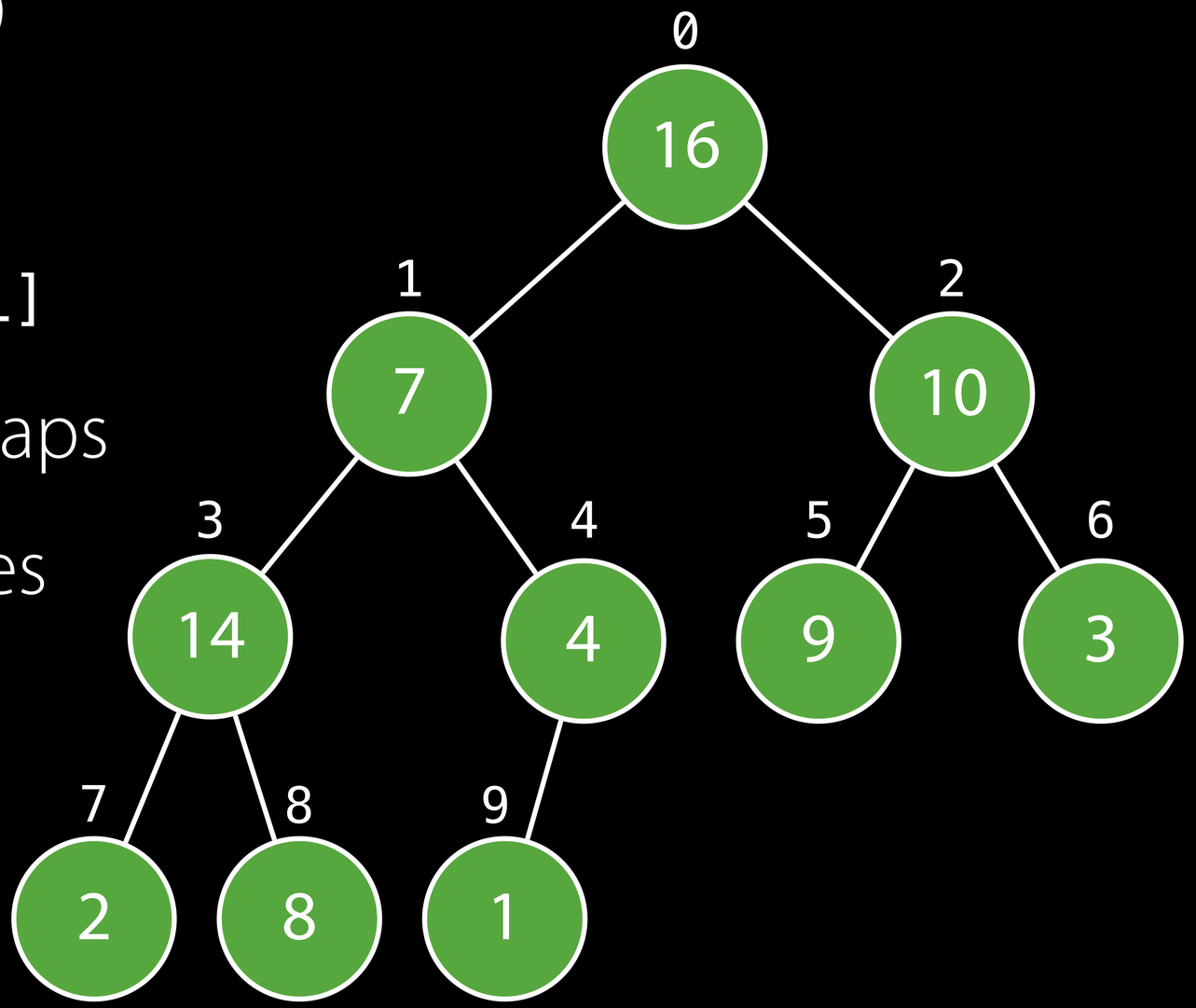
Produce a max-heap from an unordered array `A[0 ... n – 1]`

Leaves are always valid max-heaps

Call `maxHeapify` on other nodes

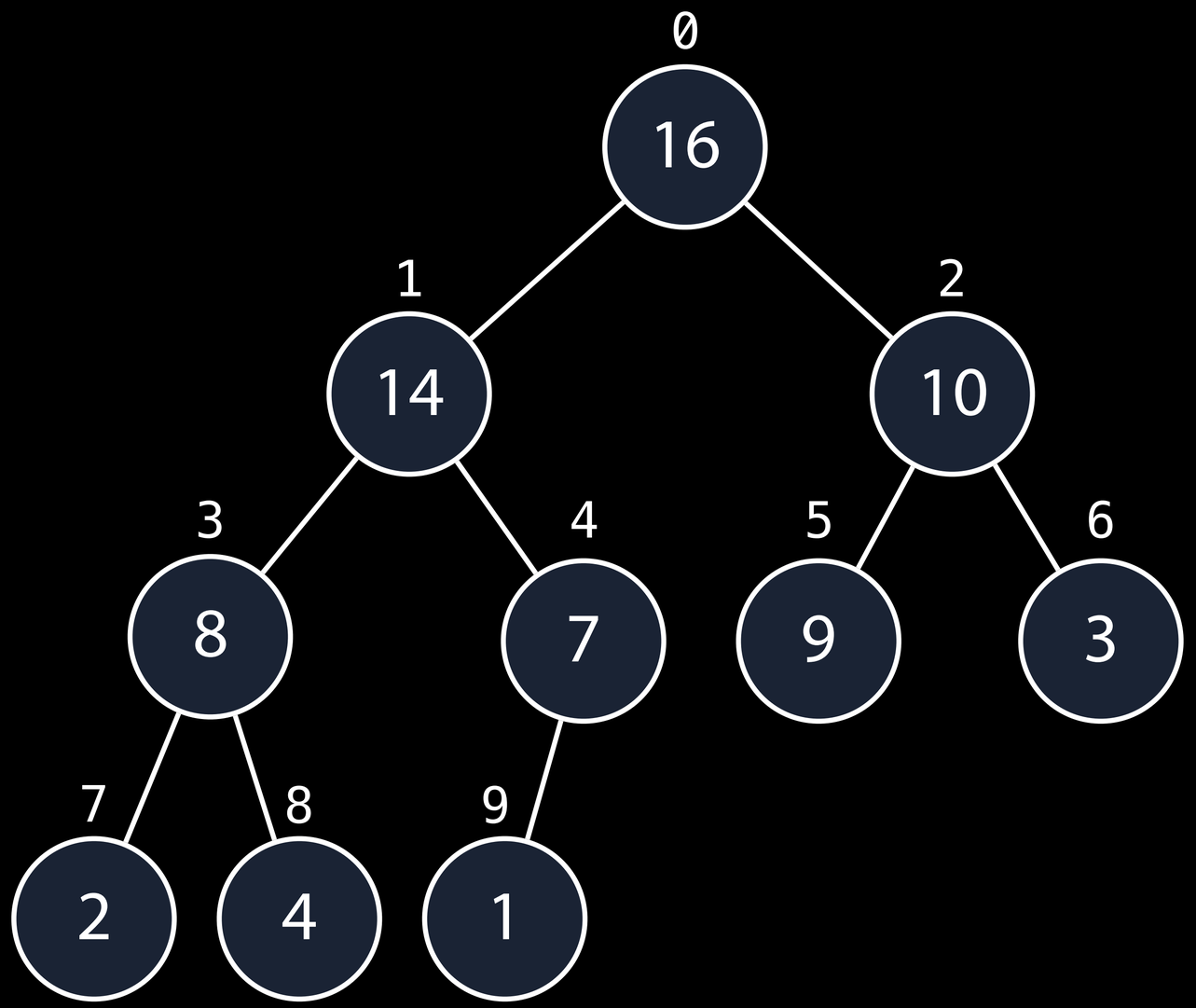
```
for i = (n – 1) / 2...0
    maxHeapify(i)
```

all nodes satisfy max-heap property



maxElement

Root of the tree

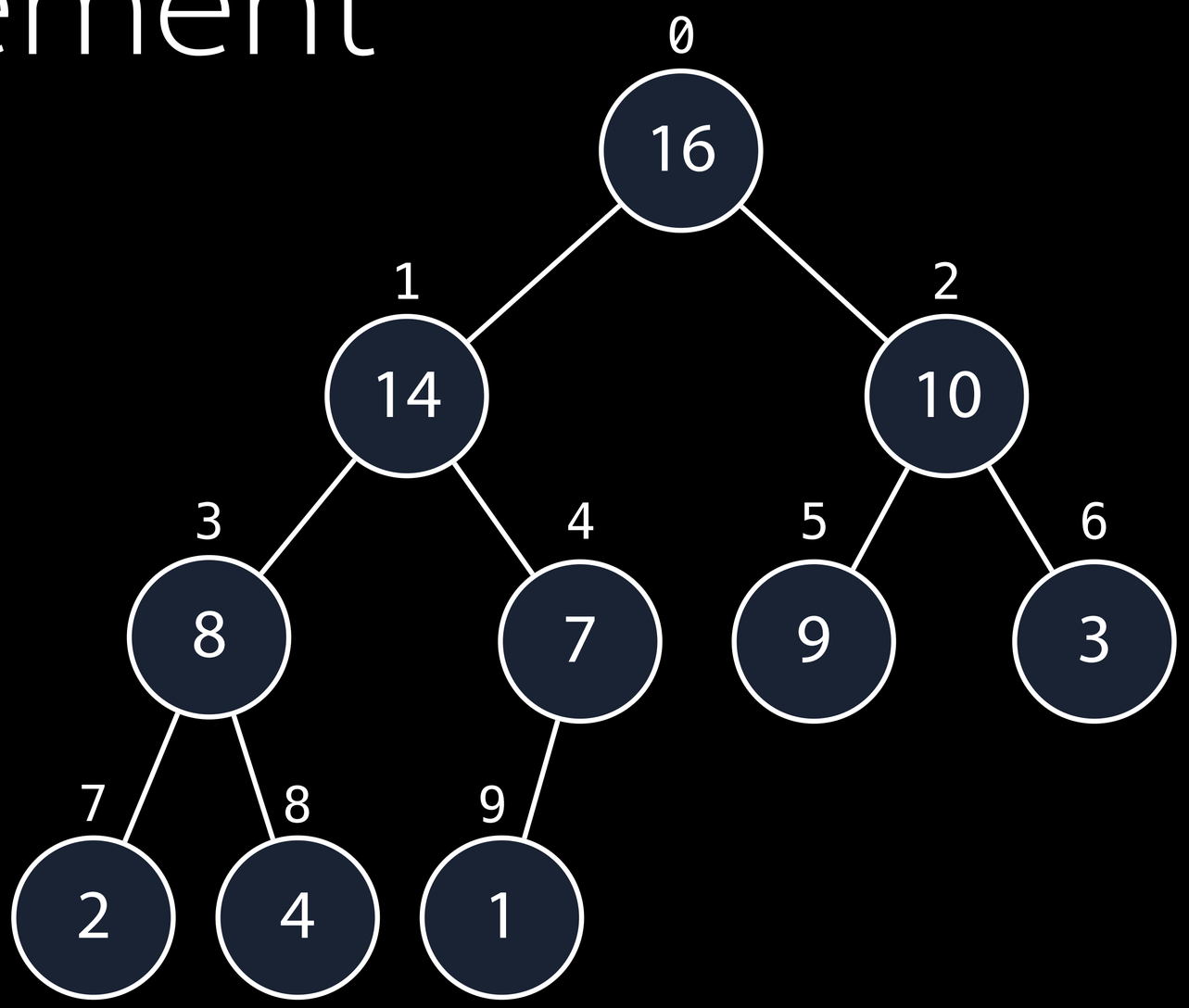


removeMaxElement

Swap root element with the last element

Decrement the size

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

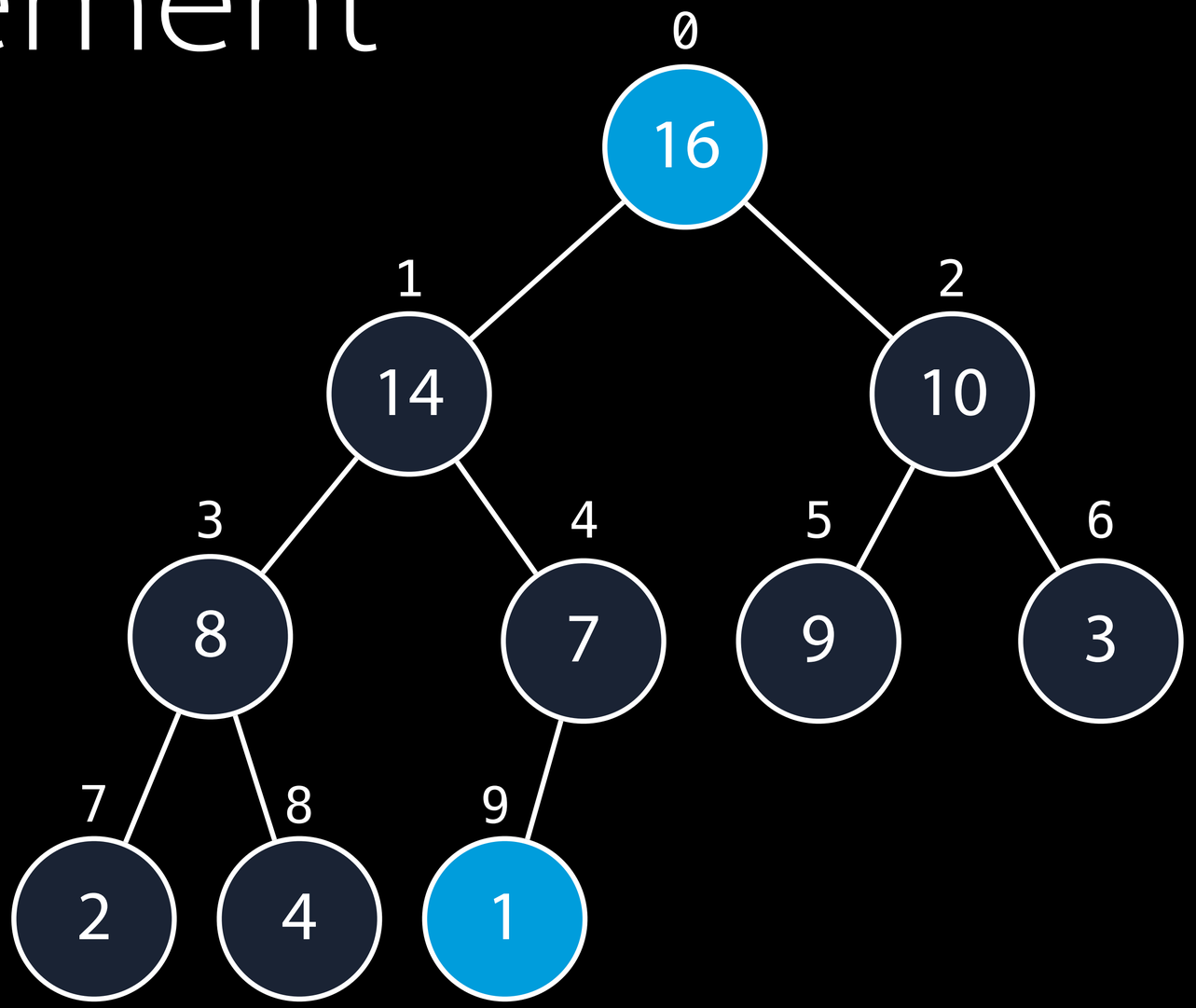


removeMaxElement

Swap root element with the last element

Decrement the size

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

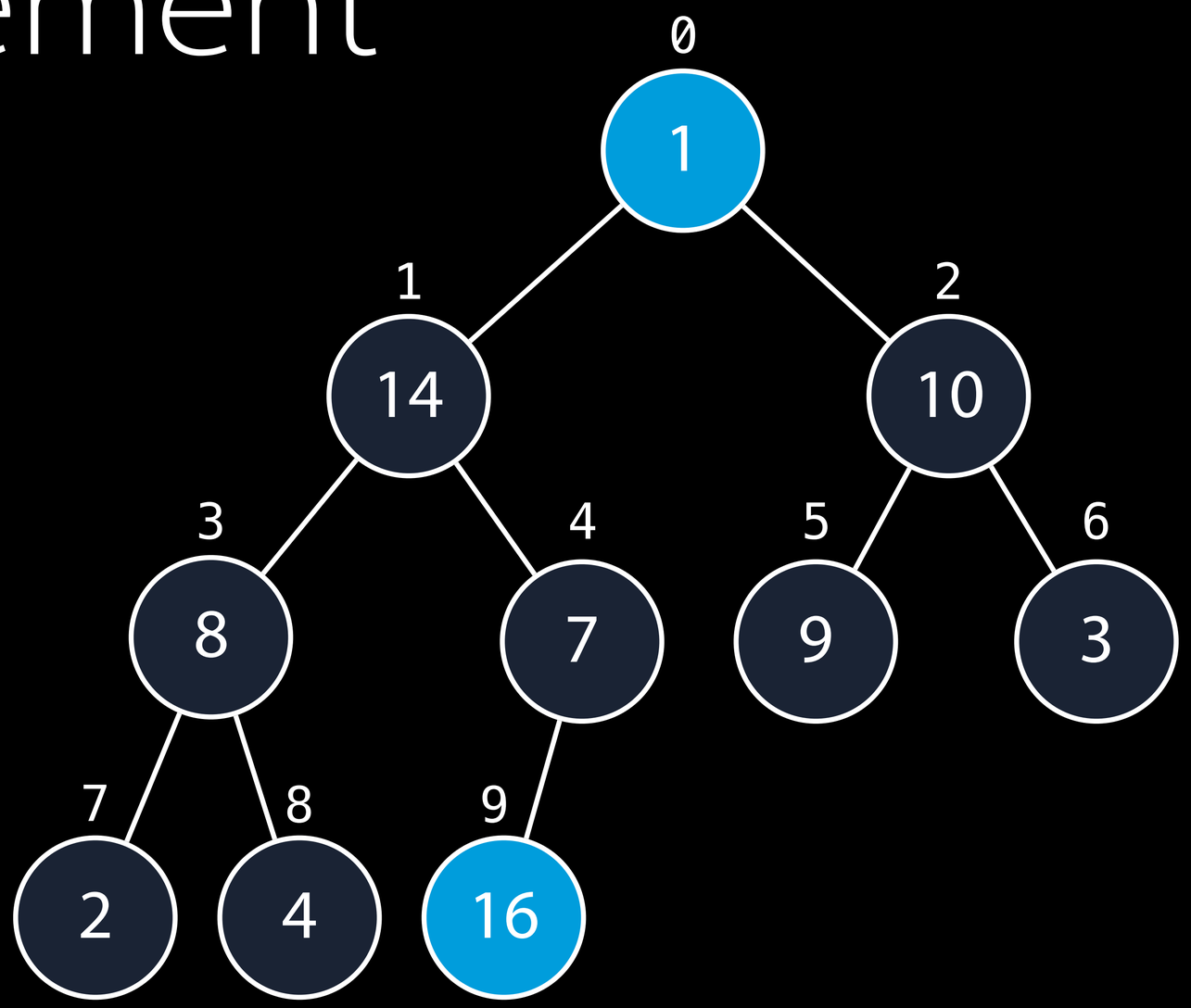


removeMaxElement

Swap root element with the last element

Decrement the size

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

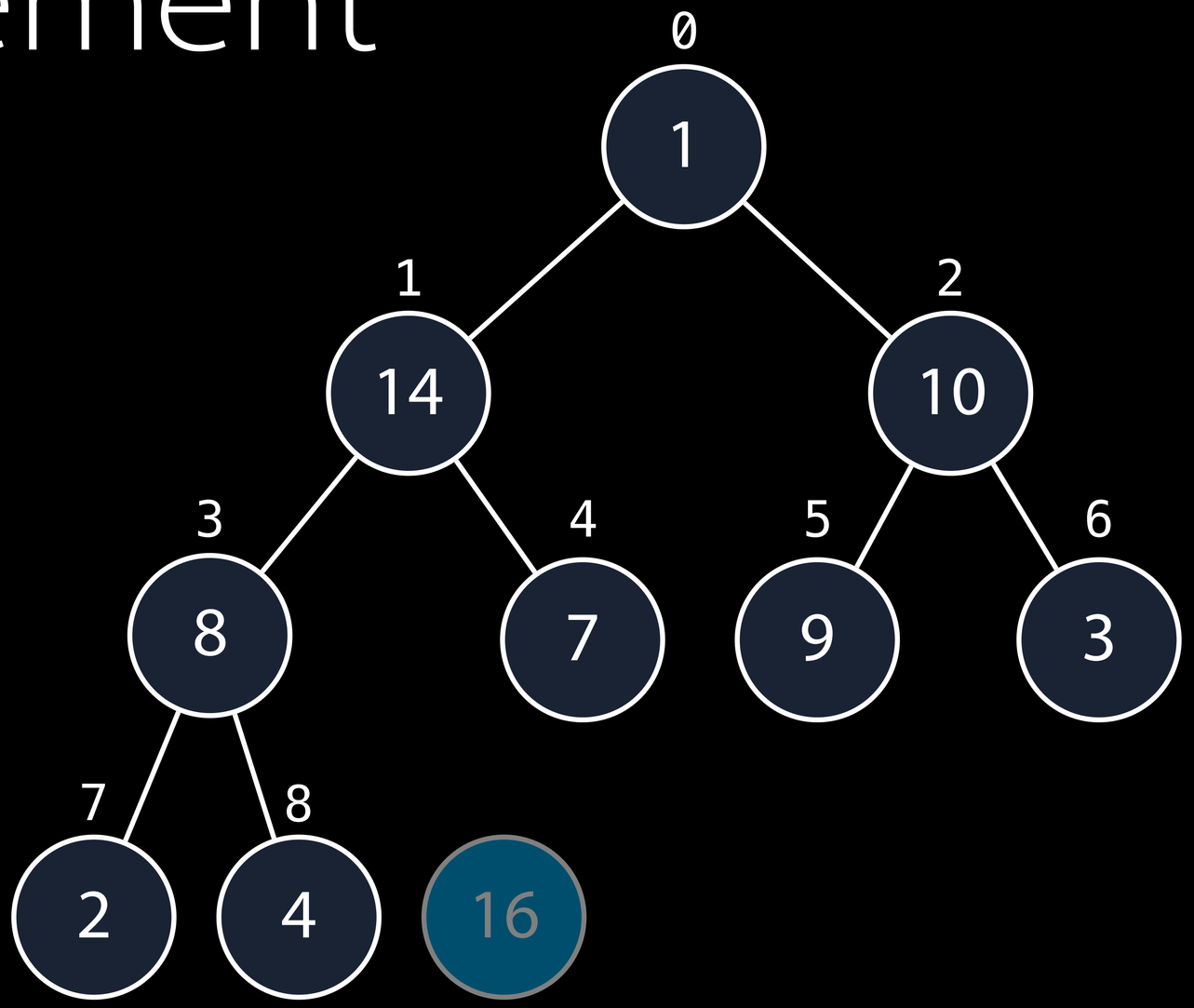


removeMaxElement

Swap root element with
the last element

Decrement the size

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

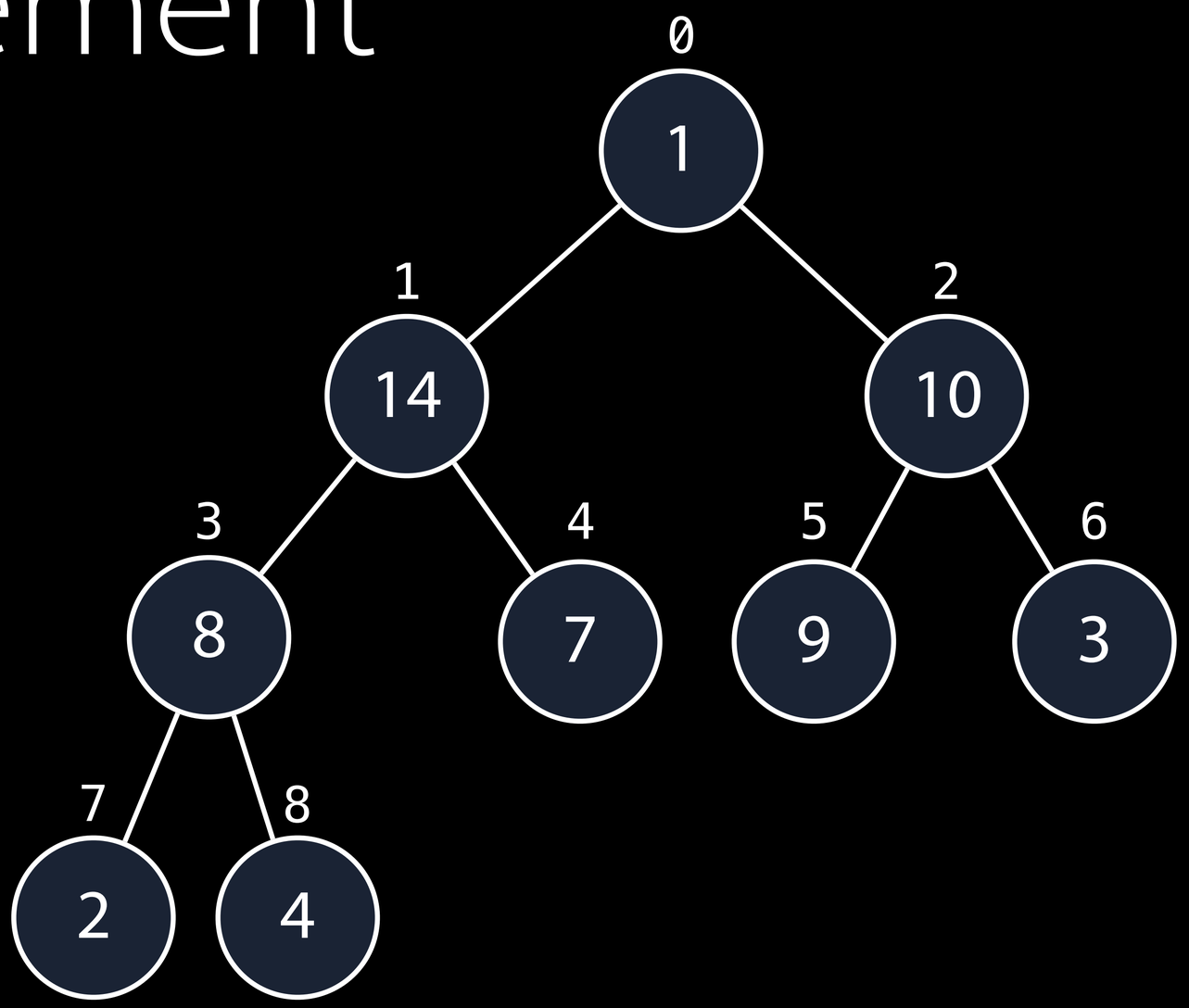


removeMaxElement

Swap root element with
the last element

Decrement the size

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

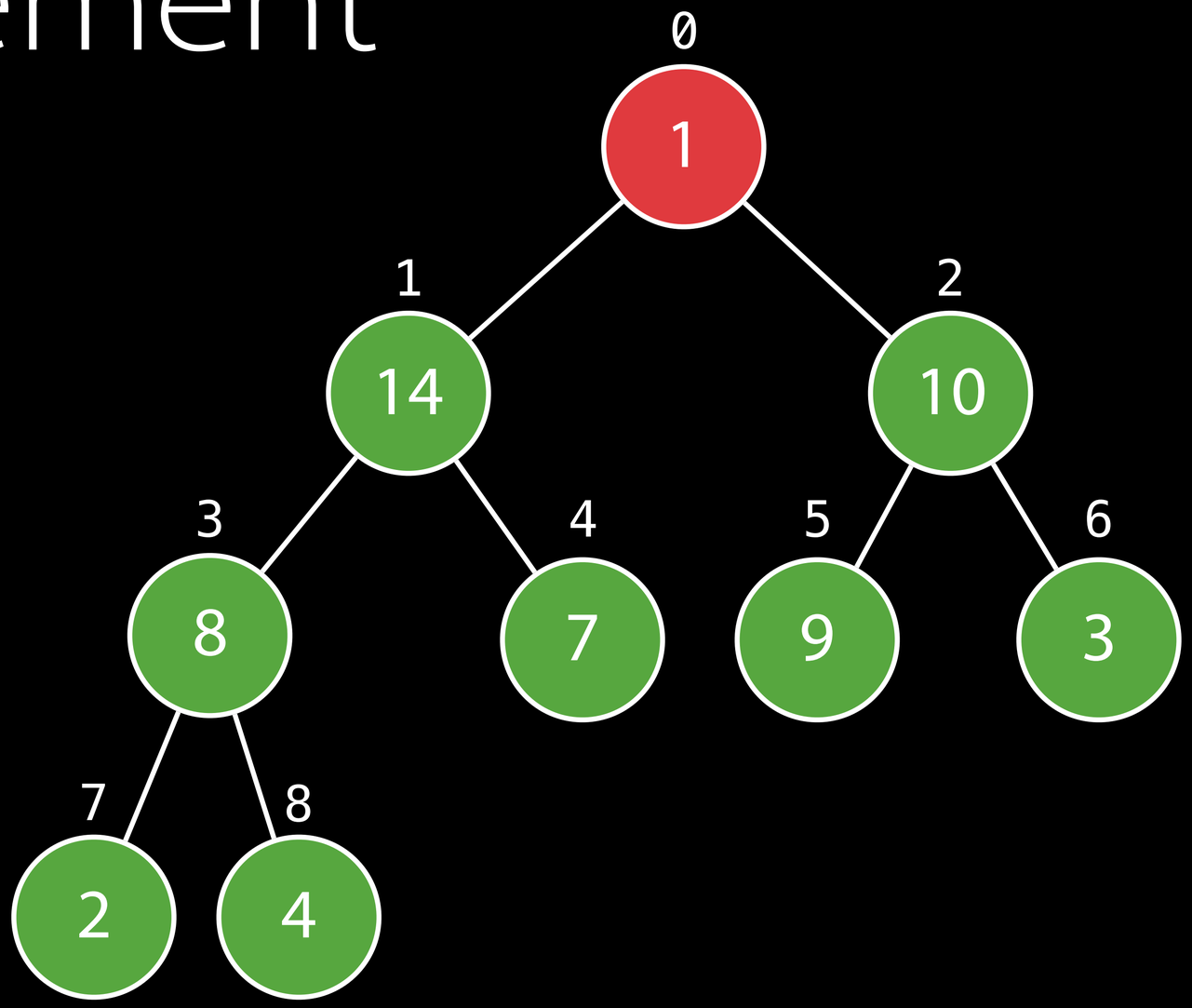


removeMaxElement

Swap root element with
the last element

Decrement the size

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

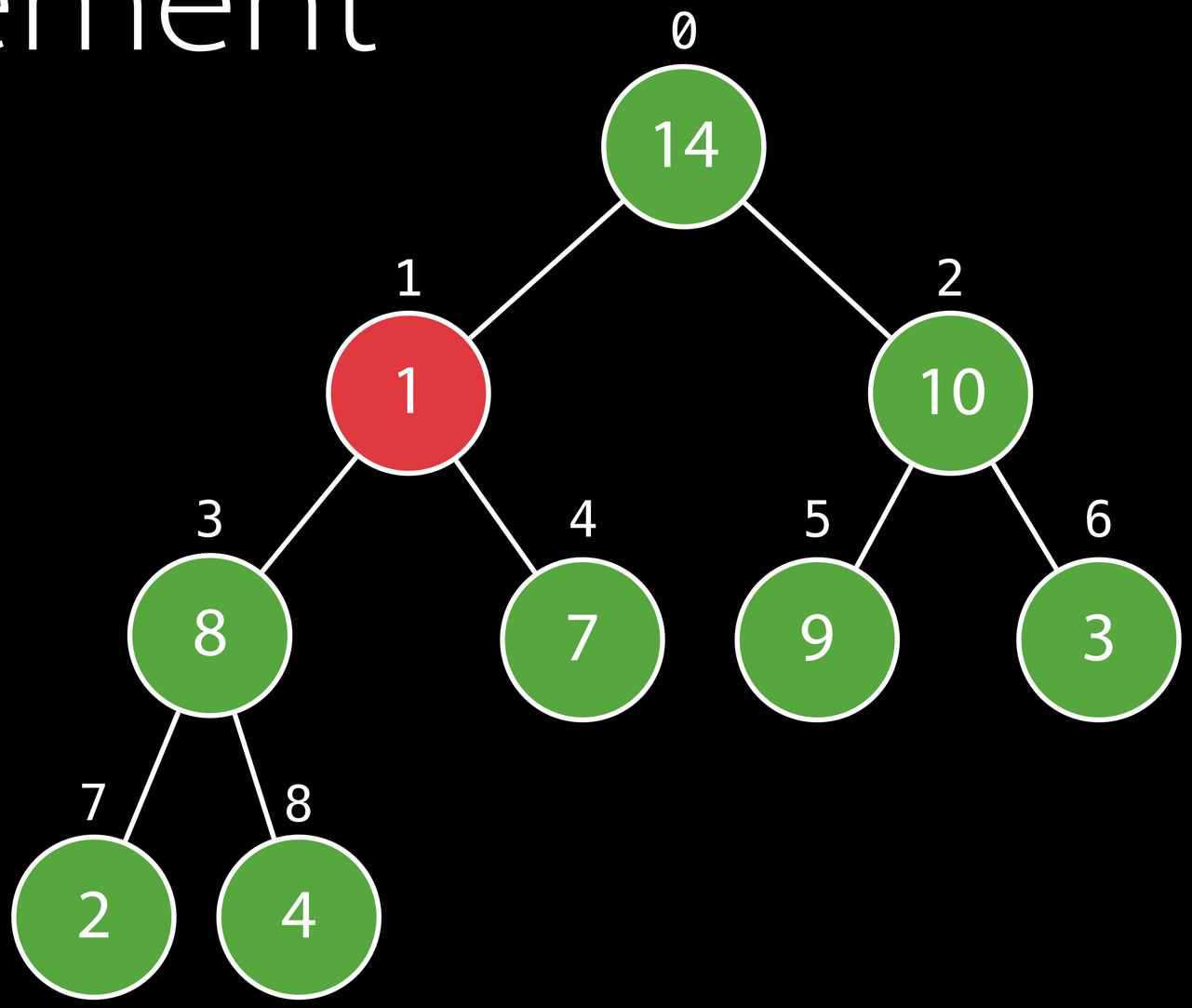


removeMaxElement

Swap root element with
the last element

Decrement the size

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

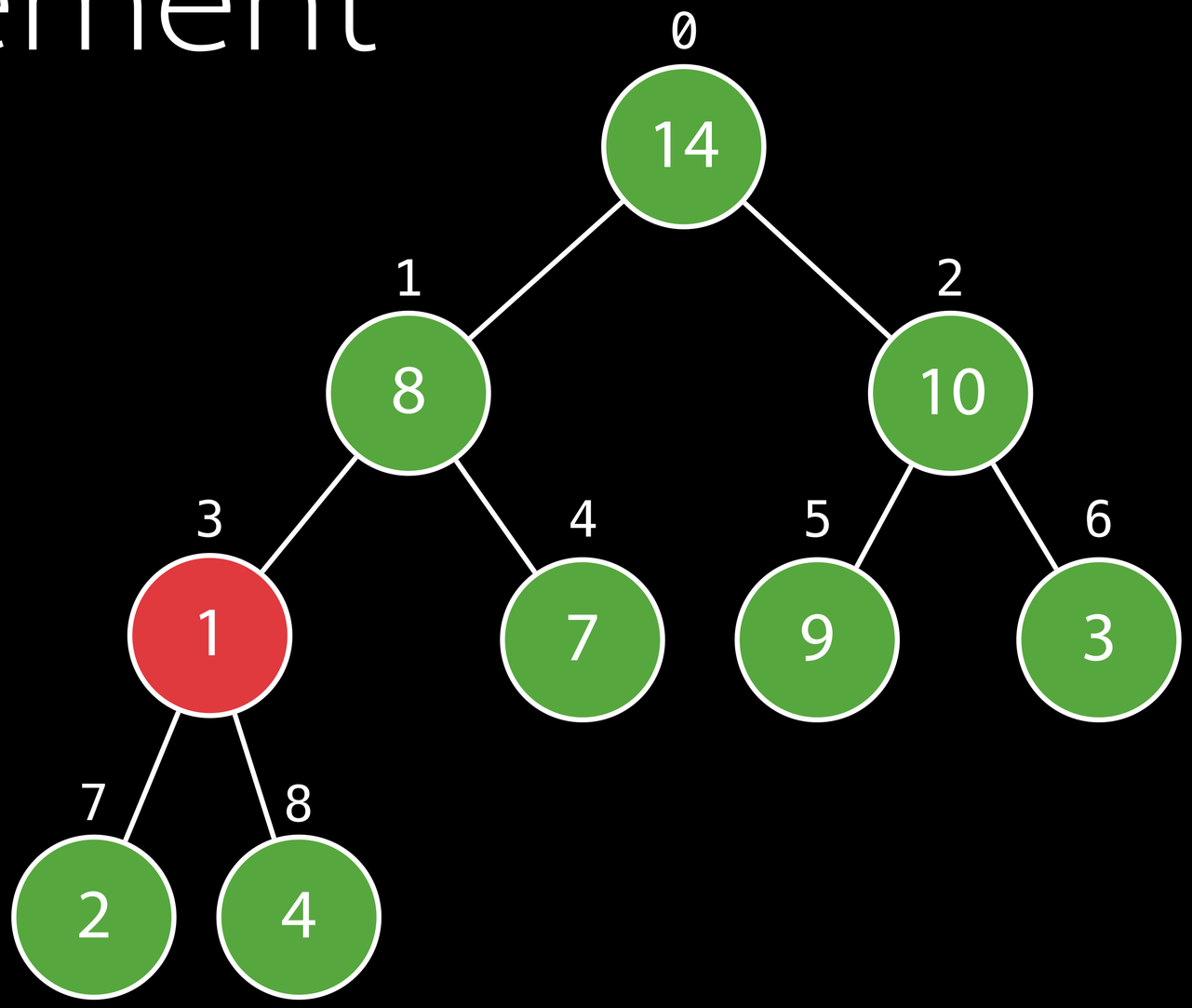


removeMaxElement

Swap root element with
the last element

Decrement the size

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

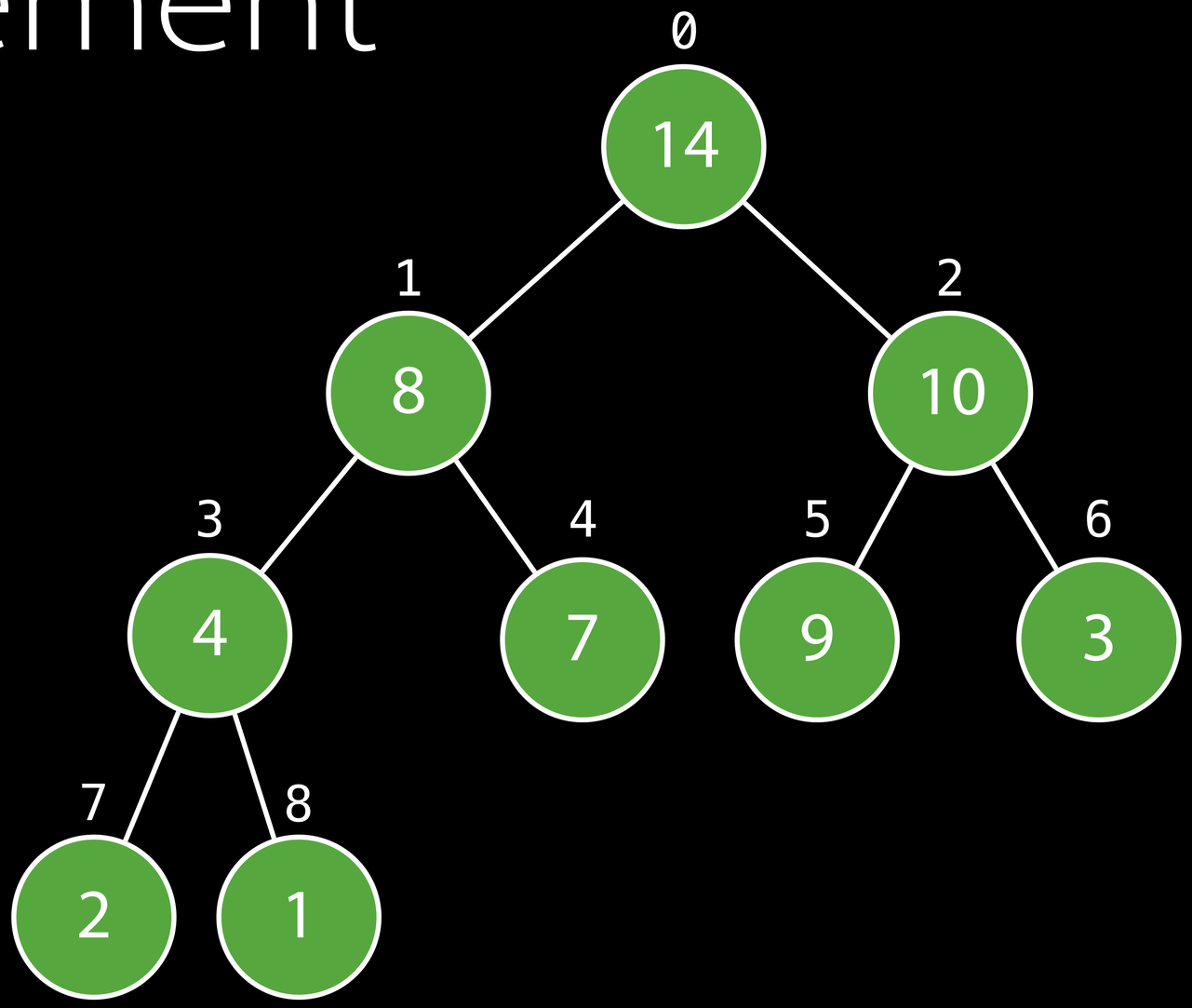


removeMaxElement

Swap root element with
the last element

Decrement the size

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

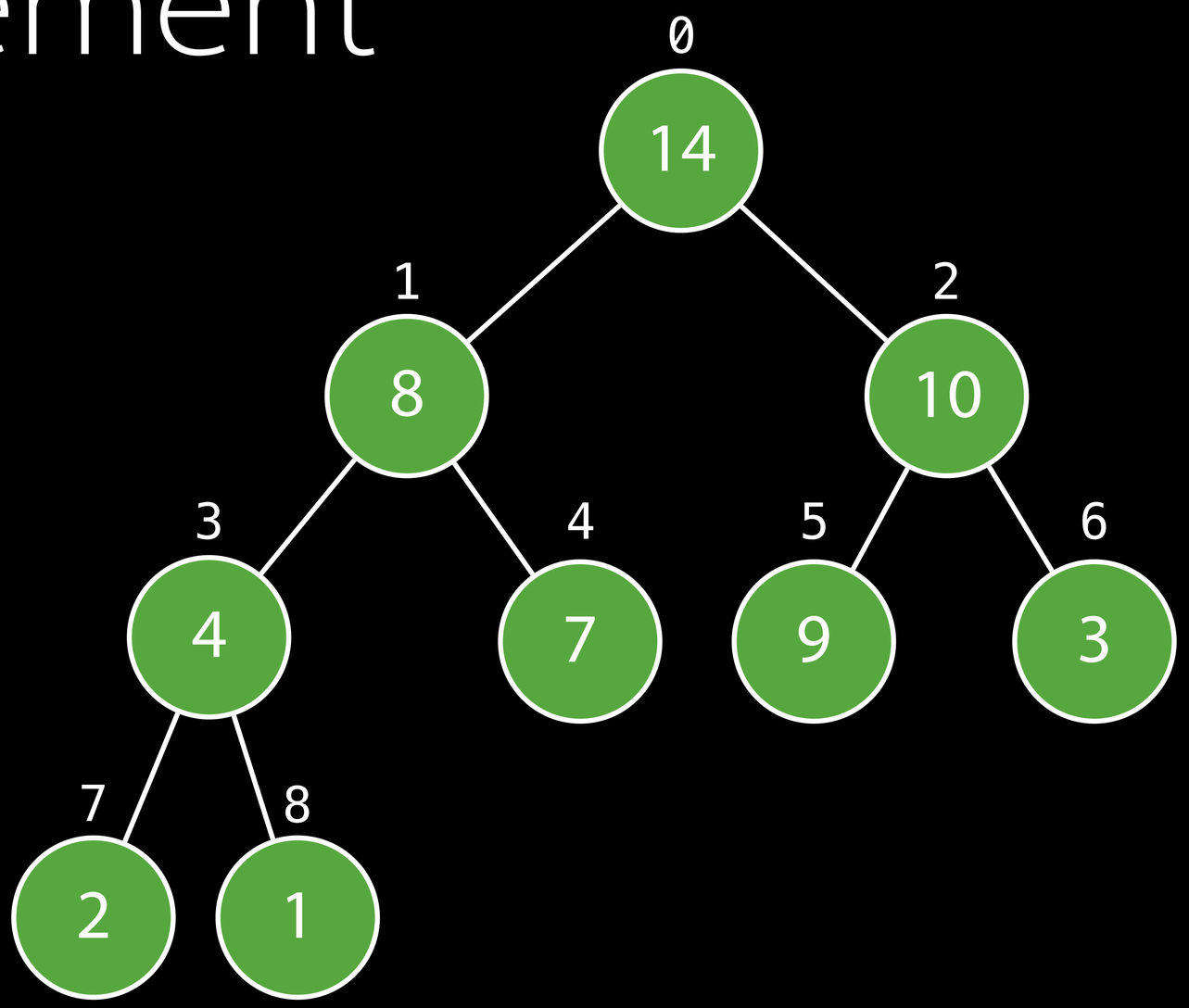


removeMaxElement

Swap root element with the last element

Decrement the size

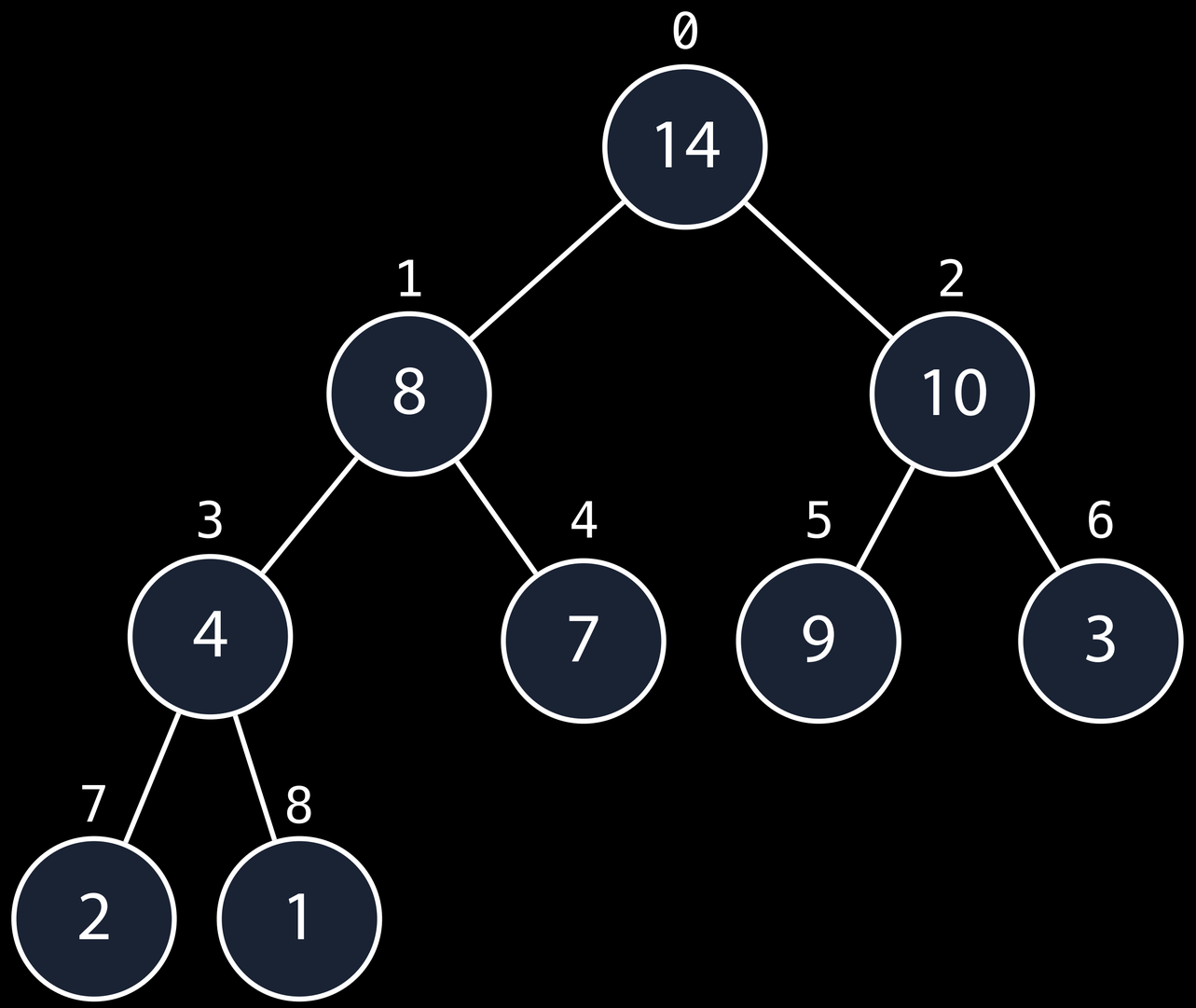
Call `maxHeapify(0)` to correct the violation at root



insert

Add an element to the end of array

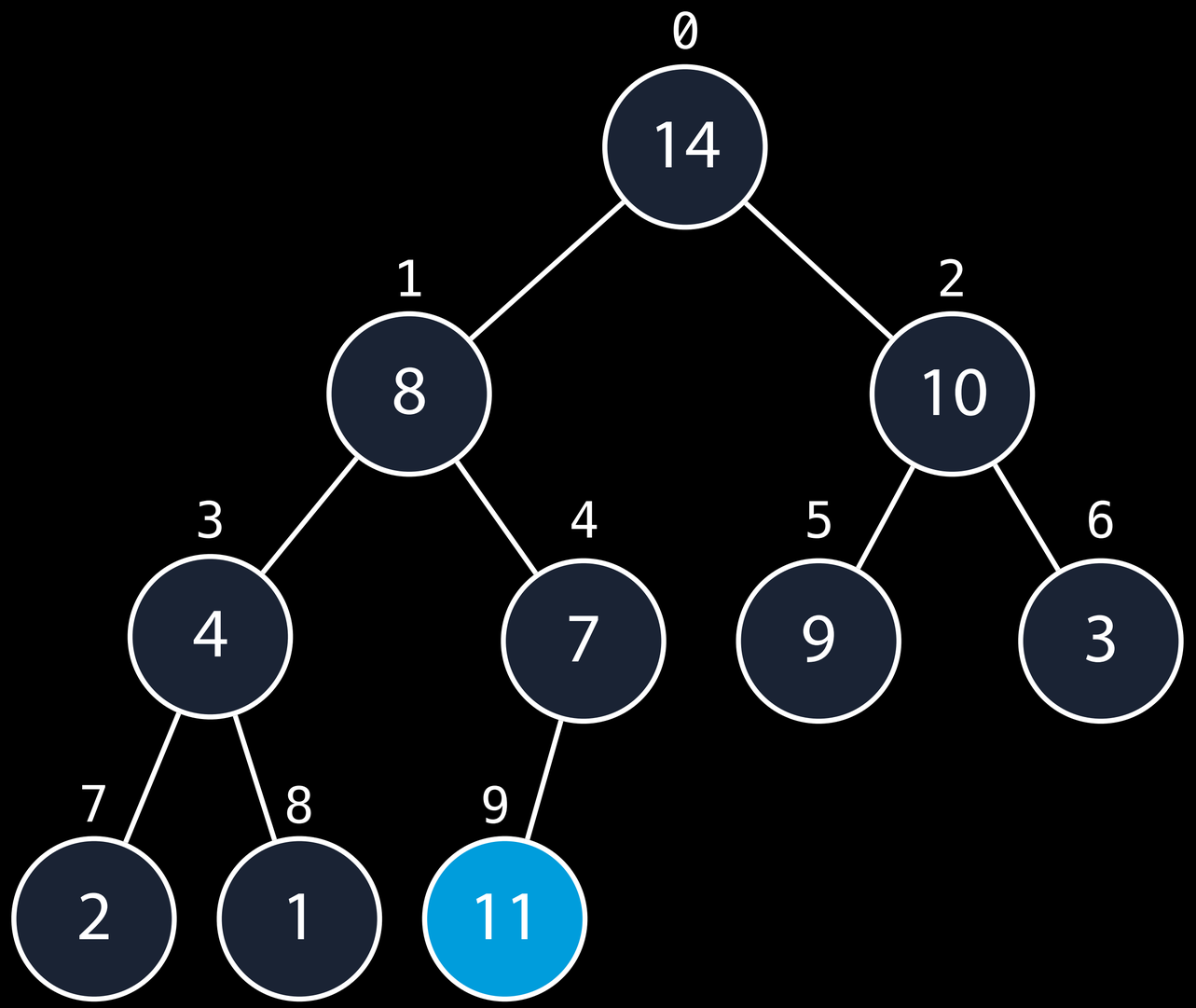
Swap with parent until correct position



insert

Add an element to the end of array

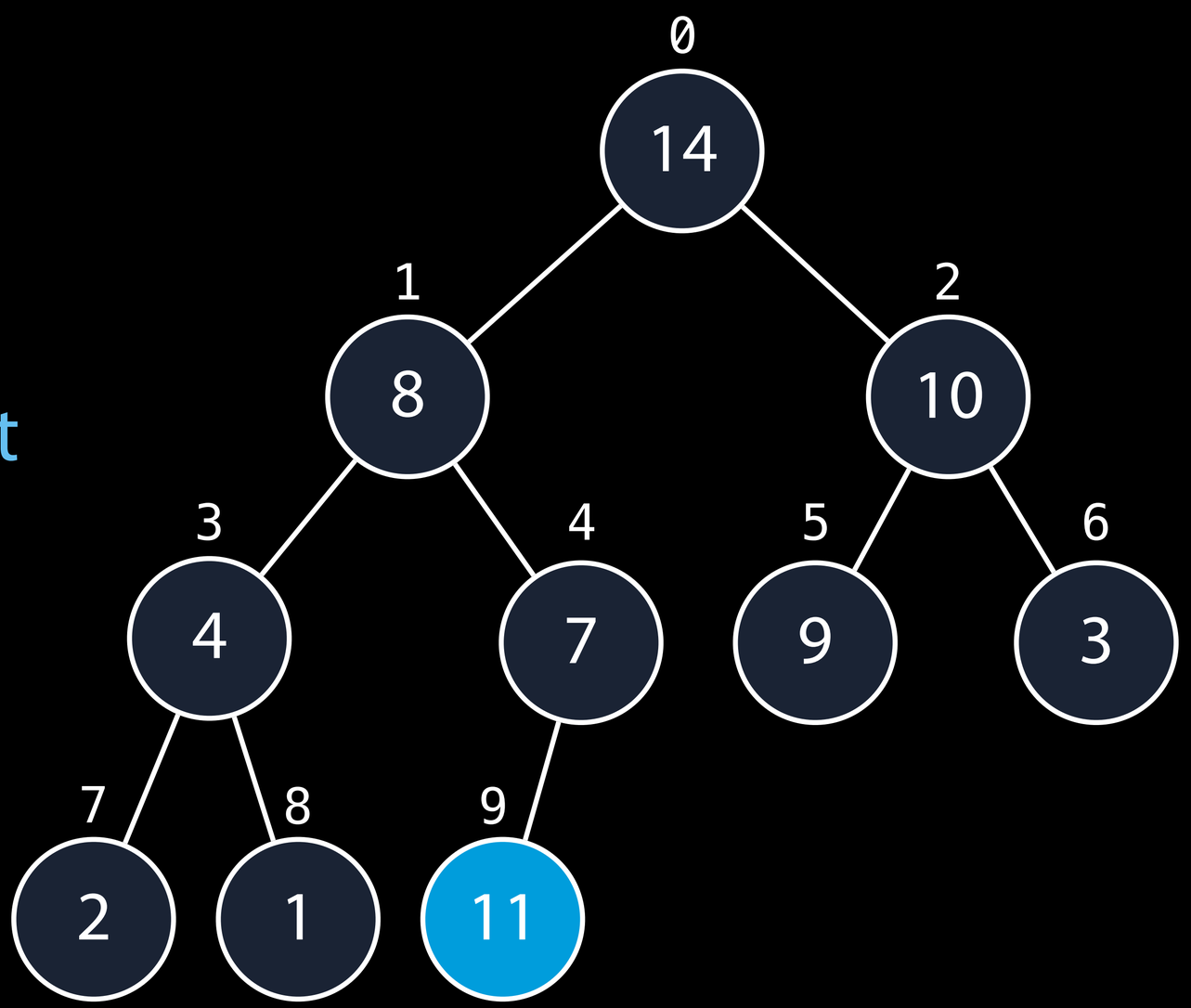
Swap with parent until correct position



insert

Add an element to the end of array

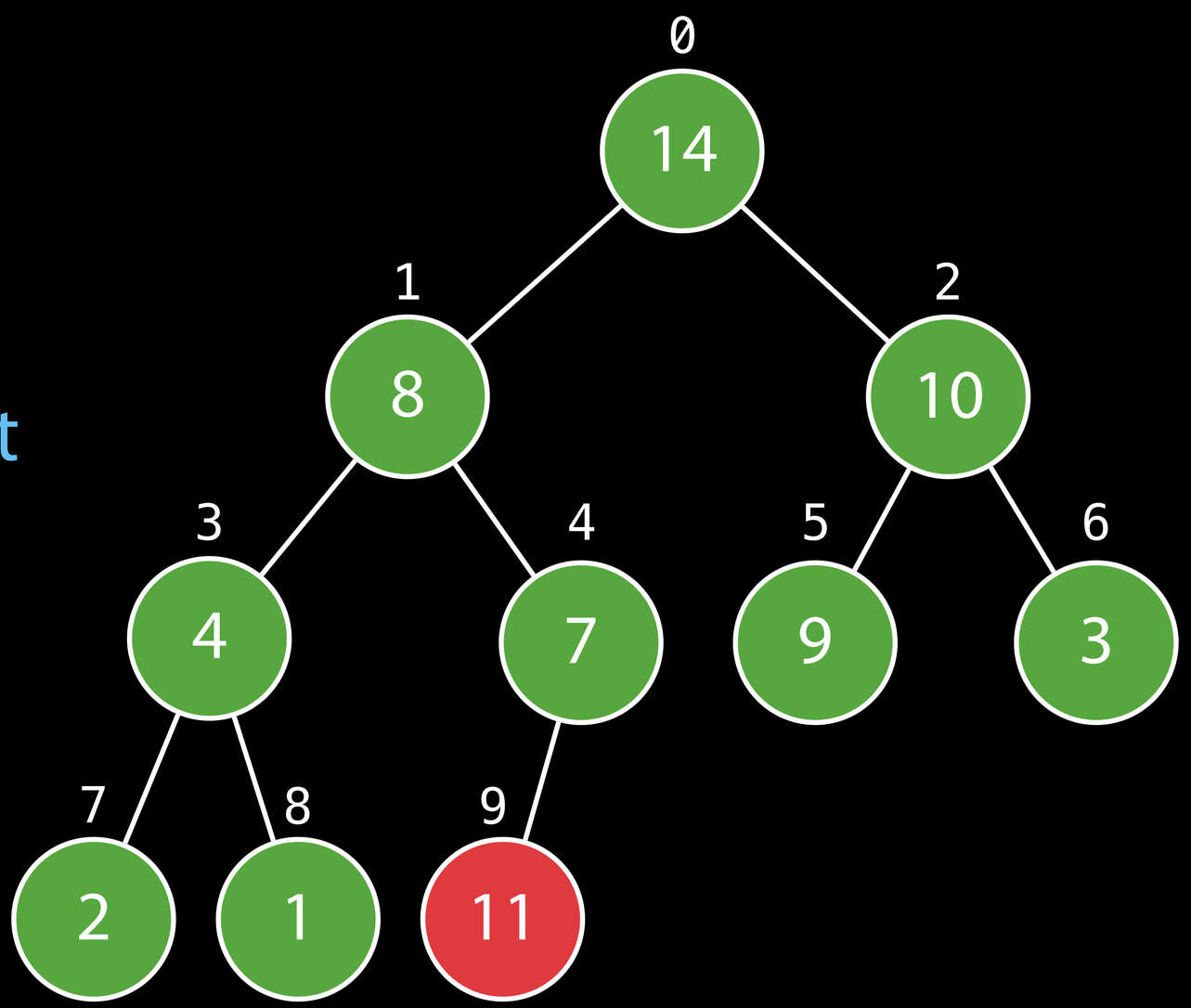
Swap with parent until correct position



insert

Add an element to the end of array

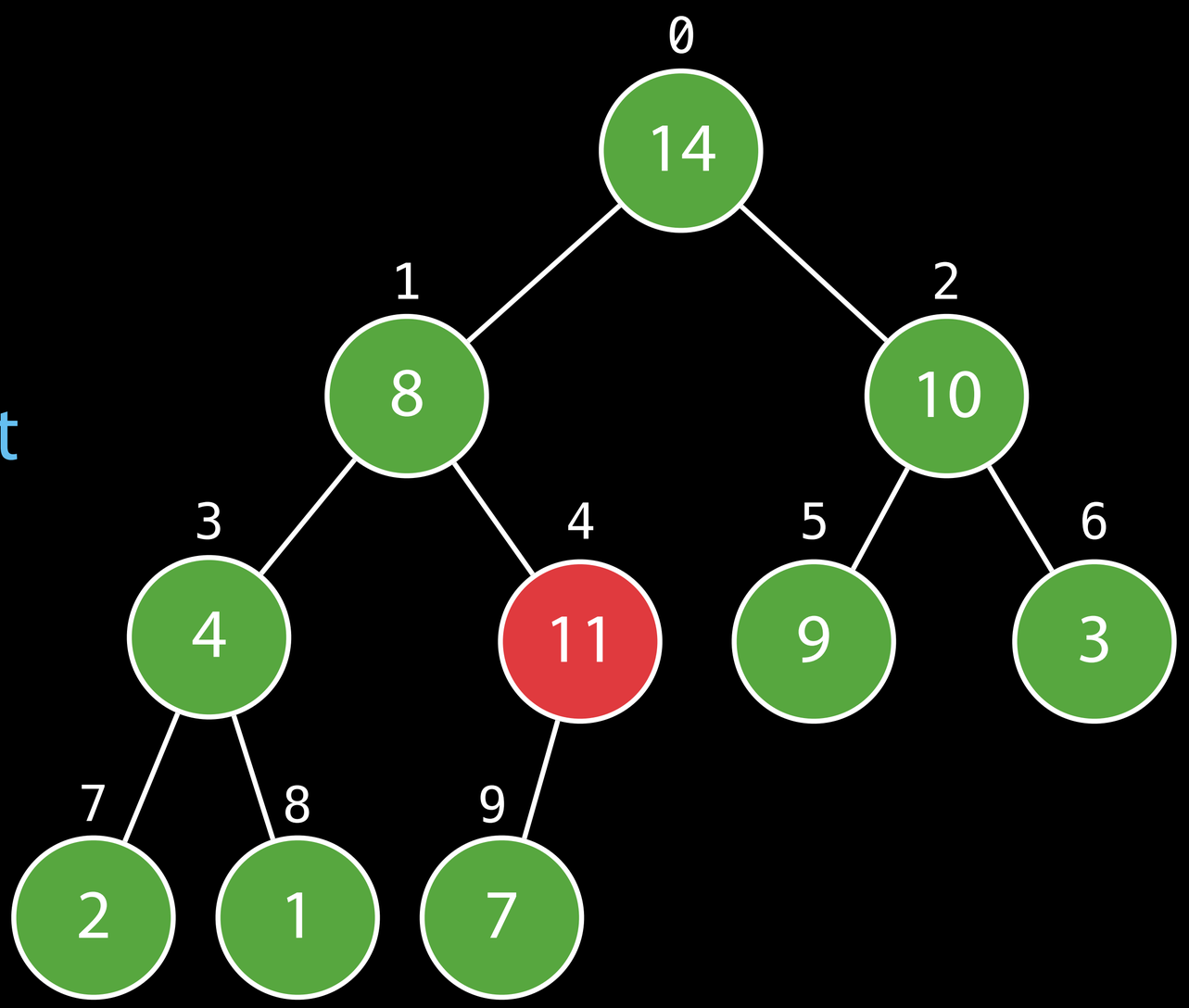
Swap with parent until correct position



insert

Add an element to the end of array

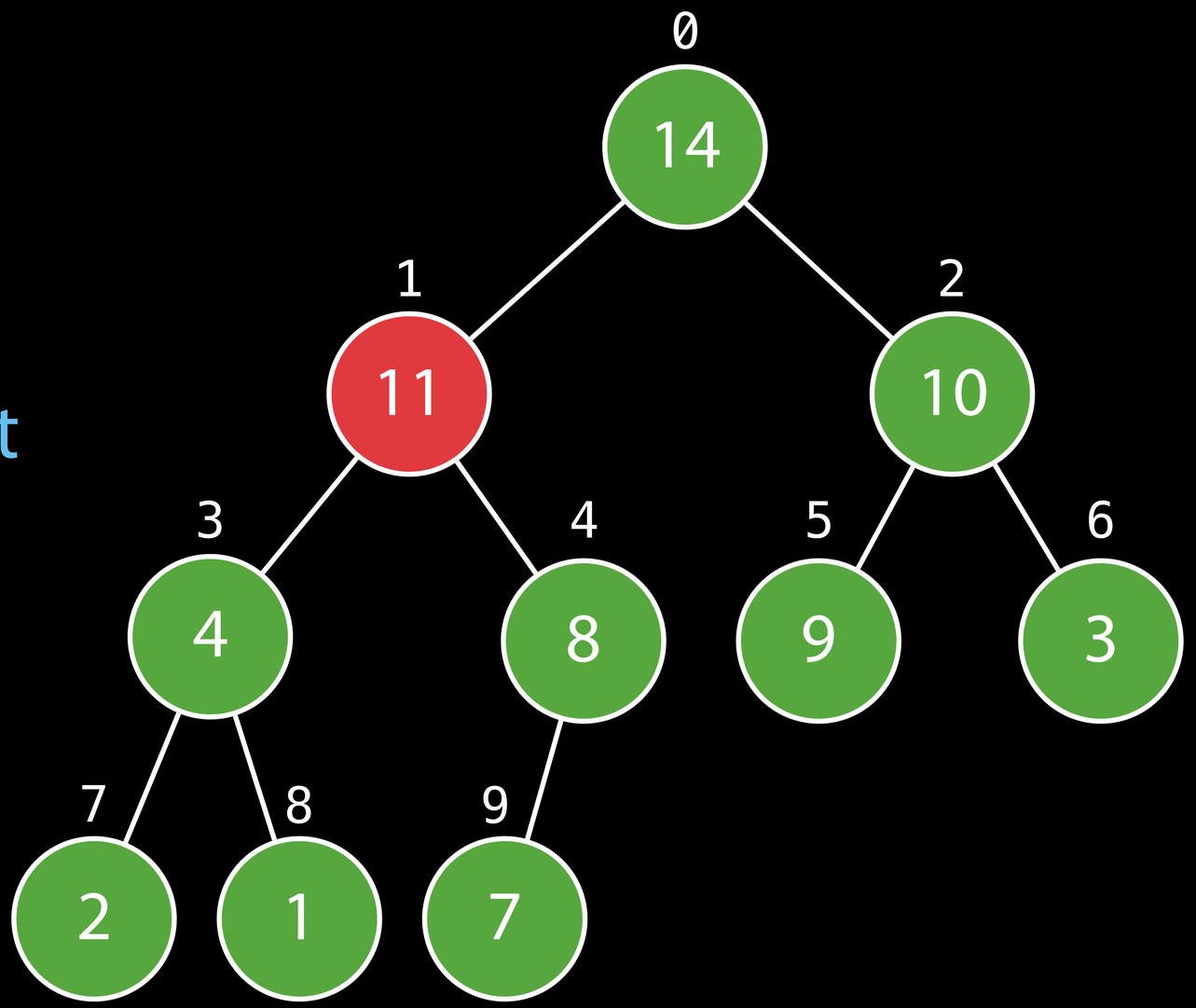
Swap with parent until correct position



insert

Add an element to the end of array

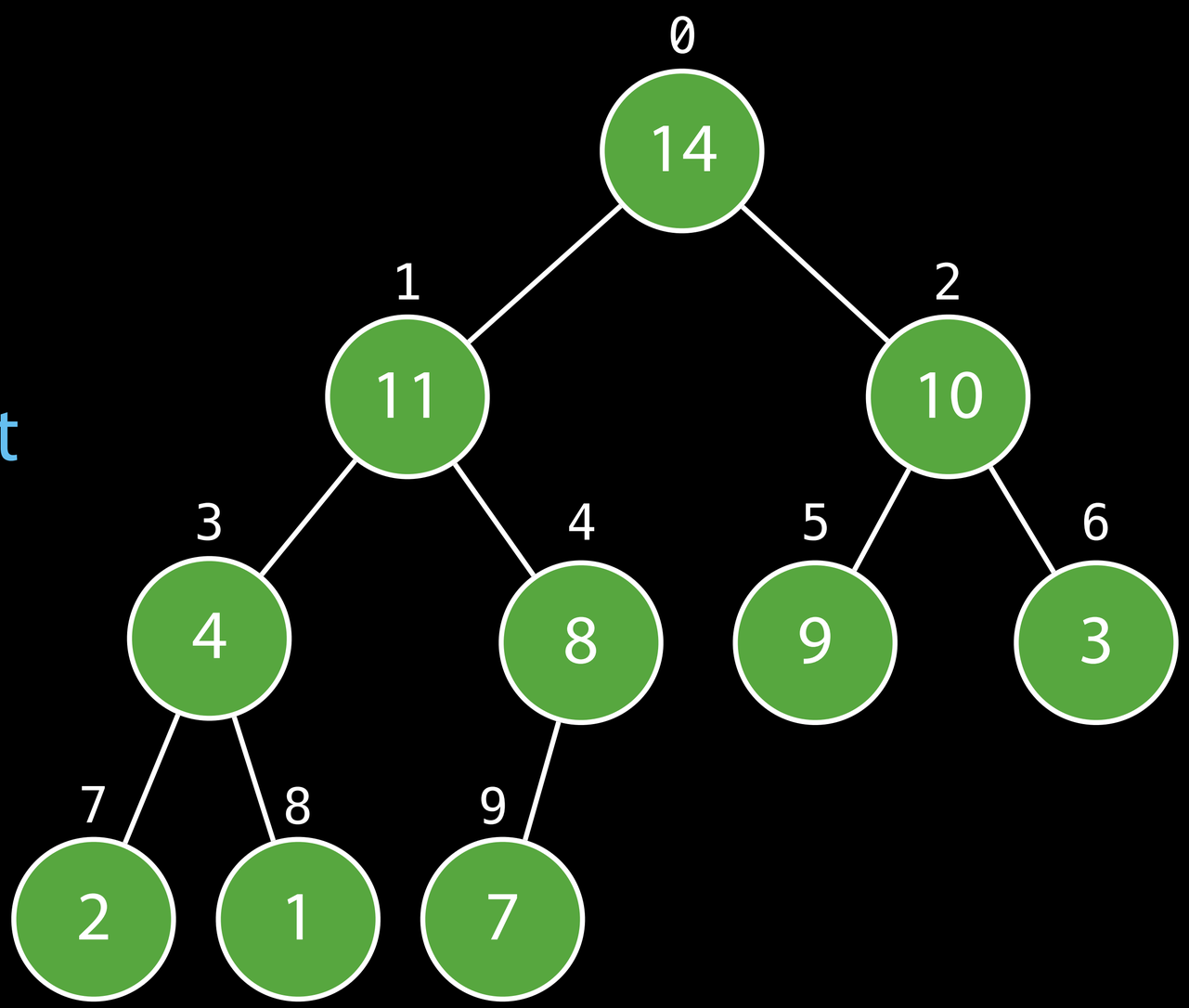
Swap with parent until correct position



insert

Add an element to the end of array

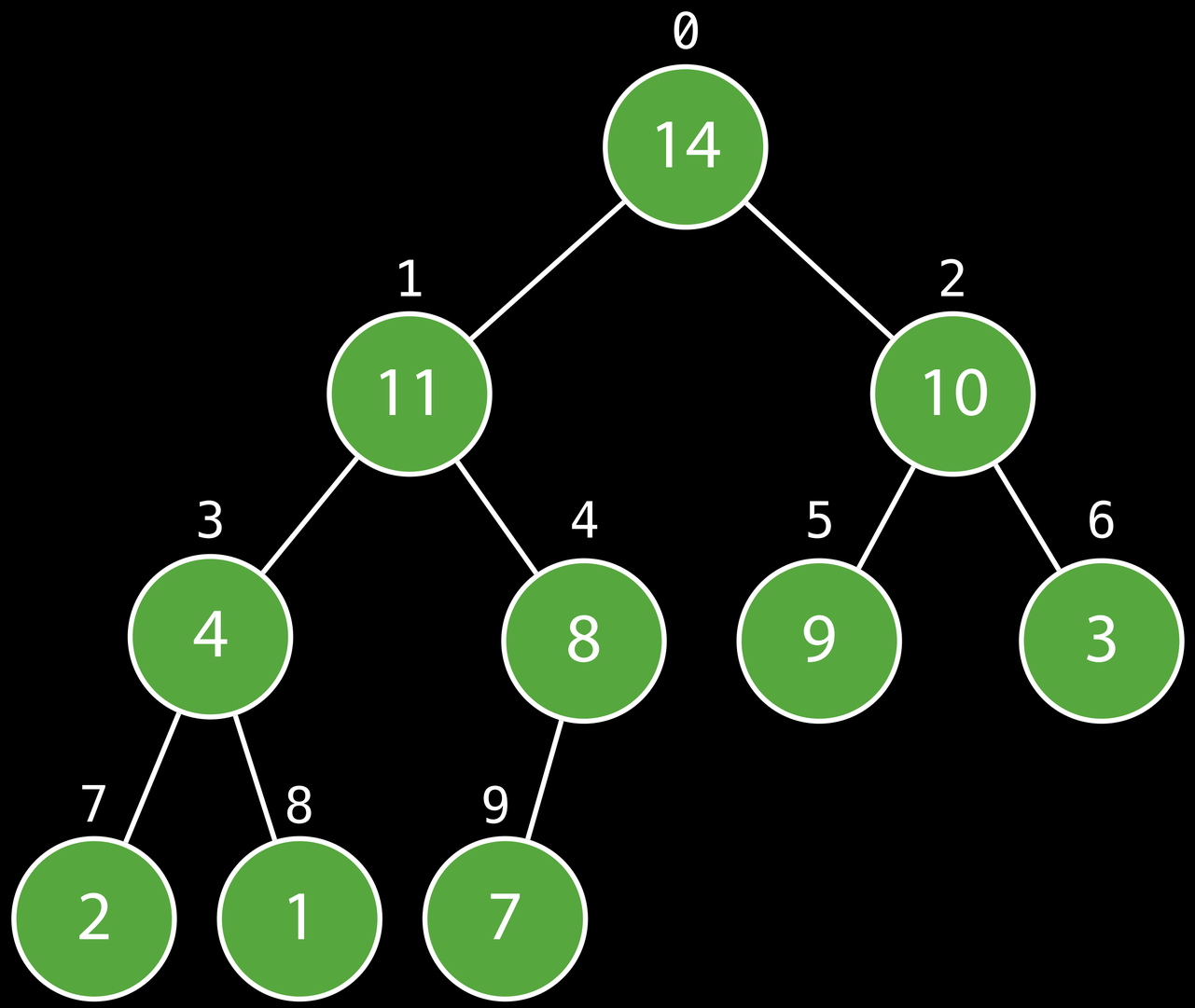
Swap with parent until correct position



insert

Add an element to the end of array

Swap with parent until correct position



Heapsort

1. Build a max-heap from unordered array.
2. Find maximum element `A[0]`.
3. Swap elements `A[n − 1]` and `A[0]`.
   Now max element is at the end!
4. Discard the last node from heap (by decrementing size variable).
5. New root may violate max heap property, but its children are max heaps. Run `maxHeapify(0)` to fix this.
6. Go to Step 2 unless heap is empty.

Pairing Heap

Project 2

due Monday 5/22

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                                  ) {
        for (int j                                  ) {
            if (                                    ) {

            }
        }
    }
    return
}
```

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.

Strings

mySpace.cpp

Write a program that prompts the user for a message, and then outputs the message with its first letter capitalized, with all letters in alternating case.

```
Give me a string: Thanks for the add!
ThAnKs FoR tHe AdD!
```

reverse.cpp

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
```

Algorithms

```
#include <algorithm>
```

Algorithms

(Mostly loops)

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

…

#include <functional>

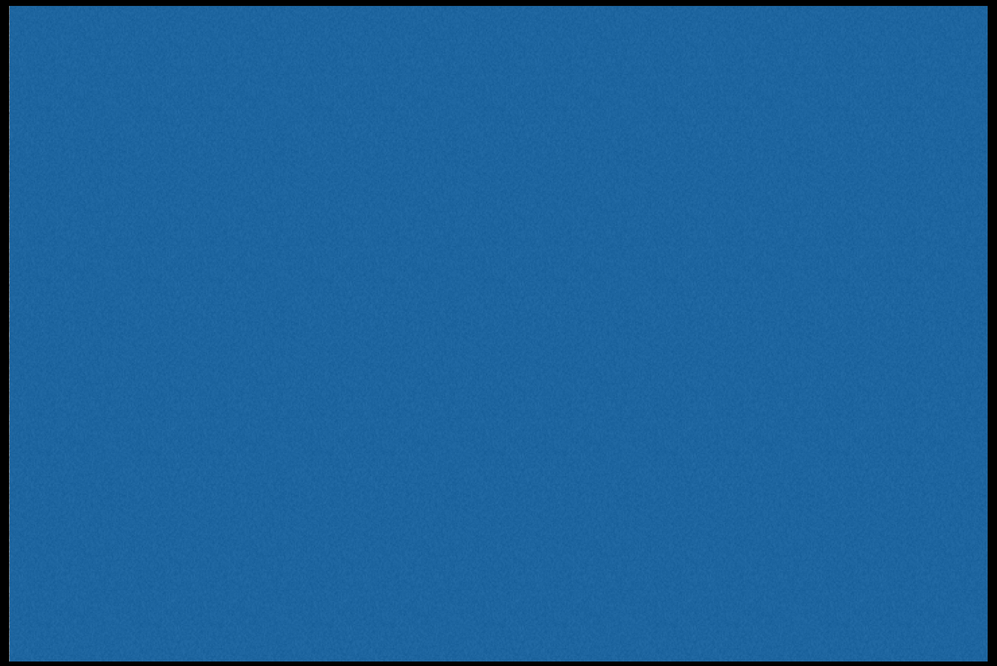
```
int result = multiplies<int>()(3, 4);

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

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));
```

Algorithms

Example



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);

}
```

```
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;
}
```

Q&A