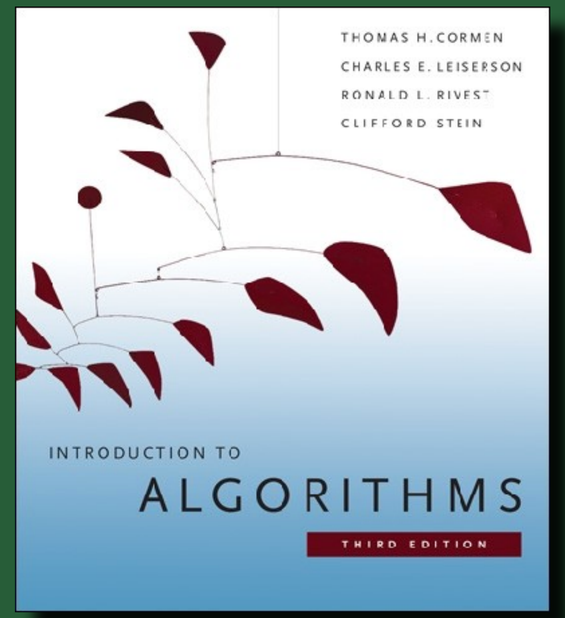
UNIVERSITY OF ALBERTA

CMPUT 204
ALGORITHMS I

Section A1, Fall 2022



Instructor: Martin Müller
Slides by Xiaoqi Tan

Recap: Heaps

Array indices start from 1

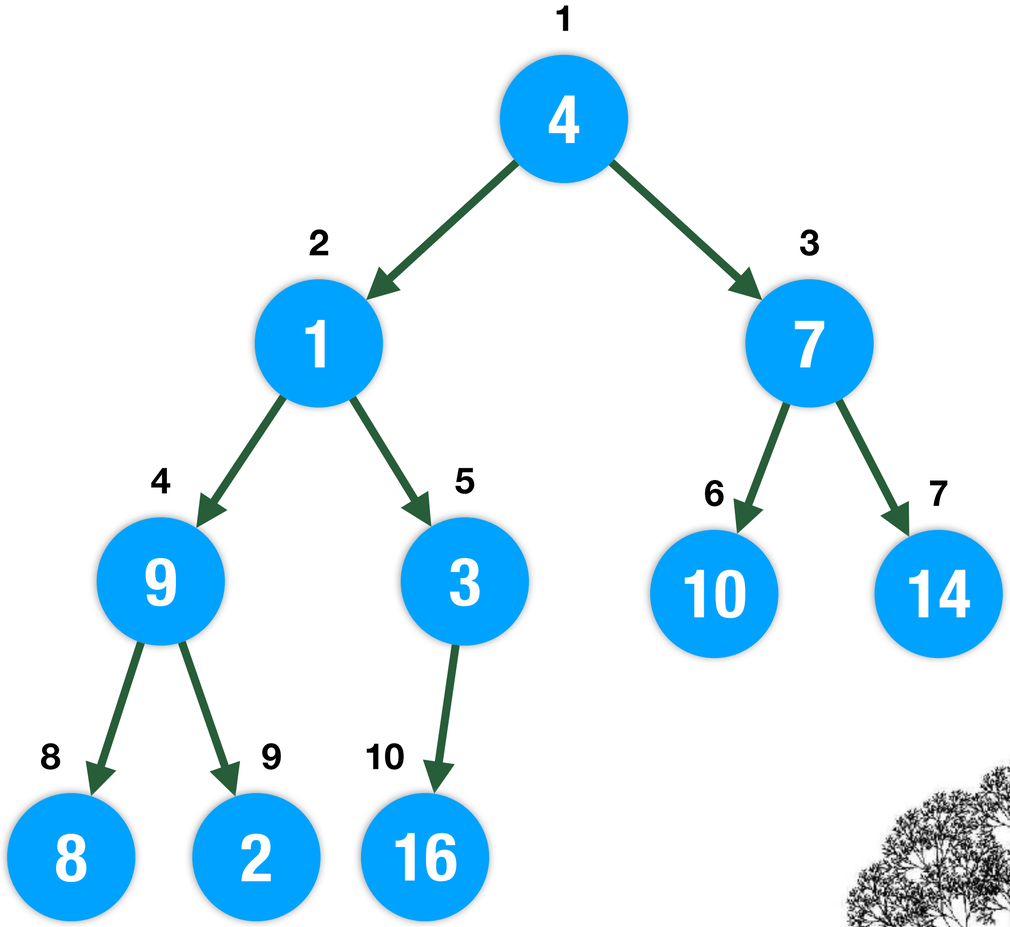
$$\texttt{leftchild}(i) = 2i$$

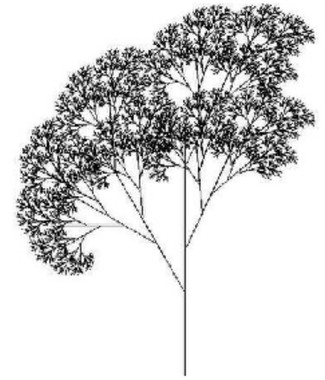
$$\texttt{rightchild}(i) = 2i + 1$$

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

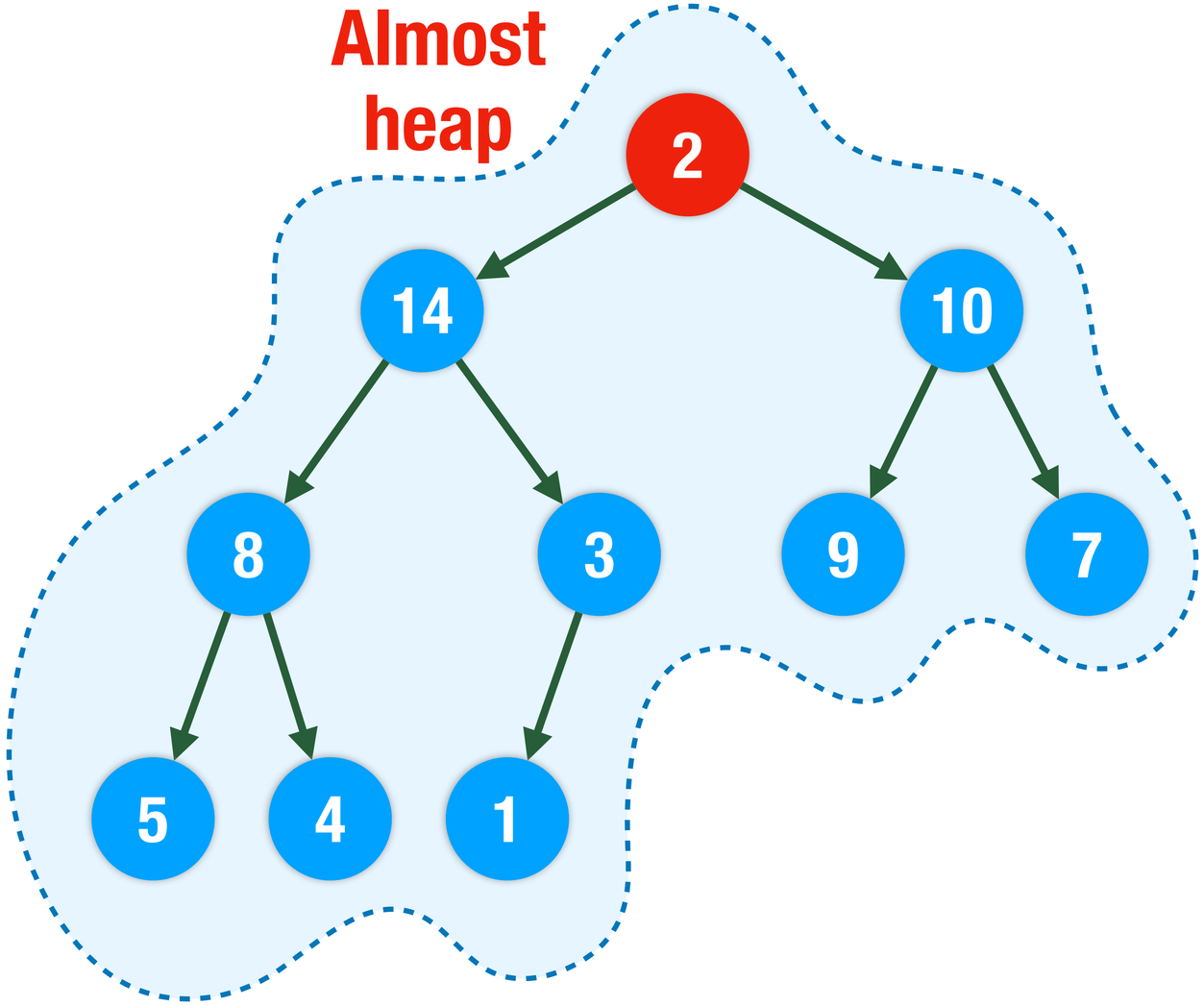
Keys

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



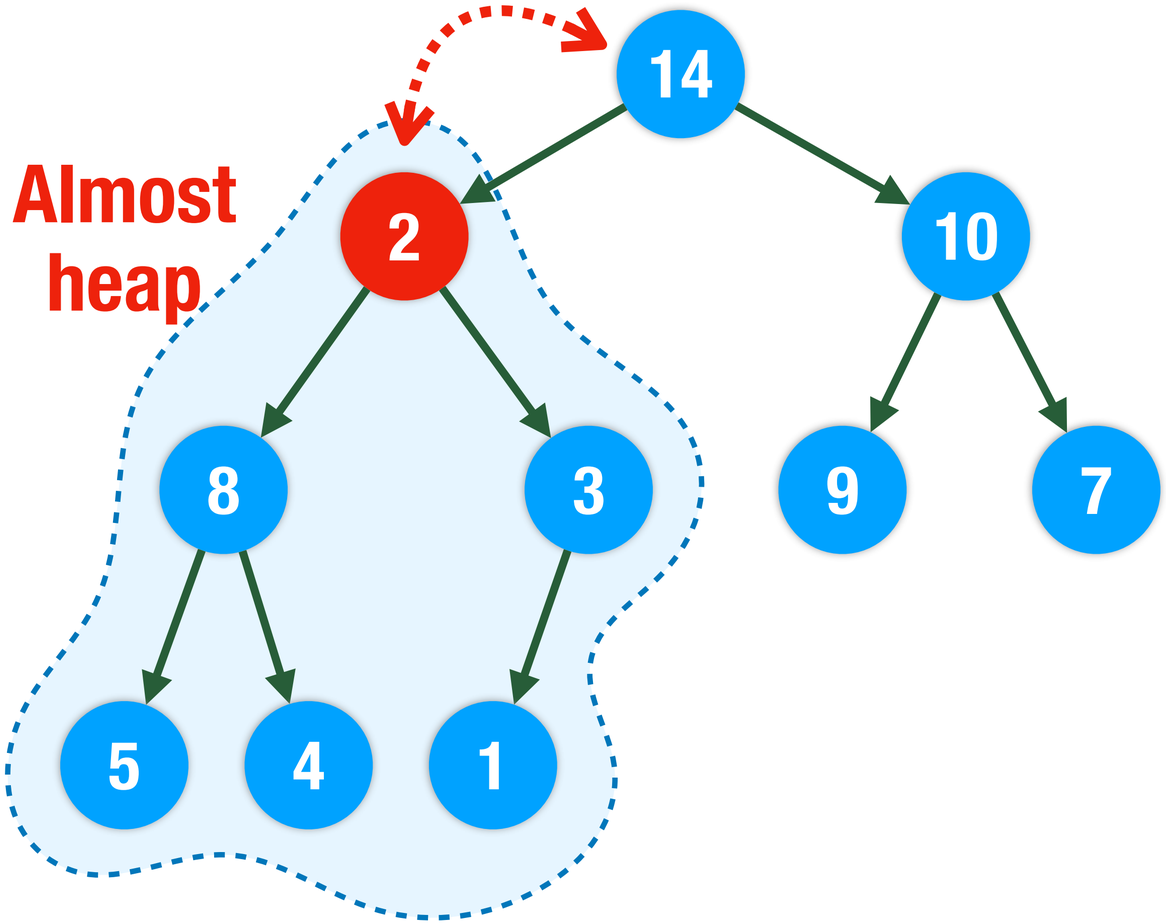


Recap: Max-Heapify(A, i)



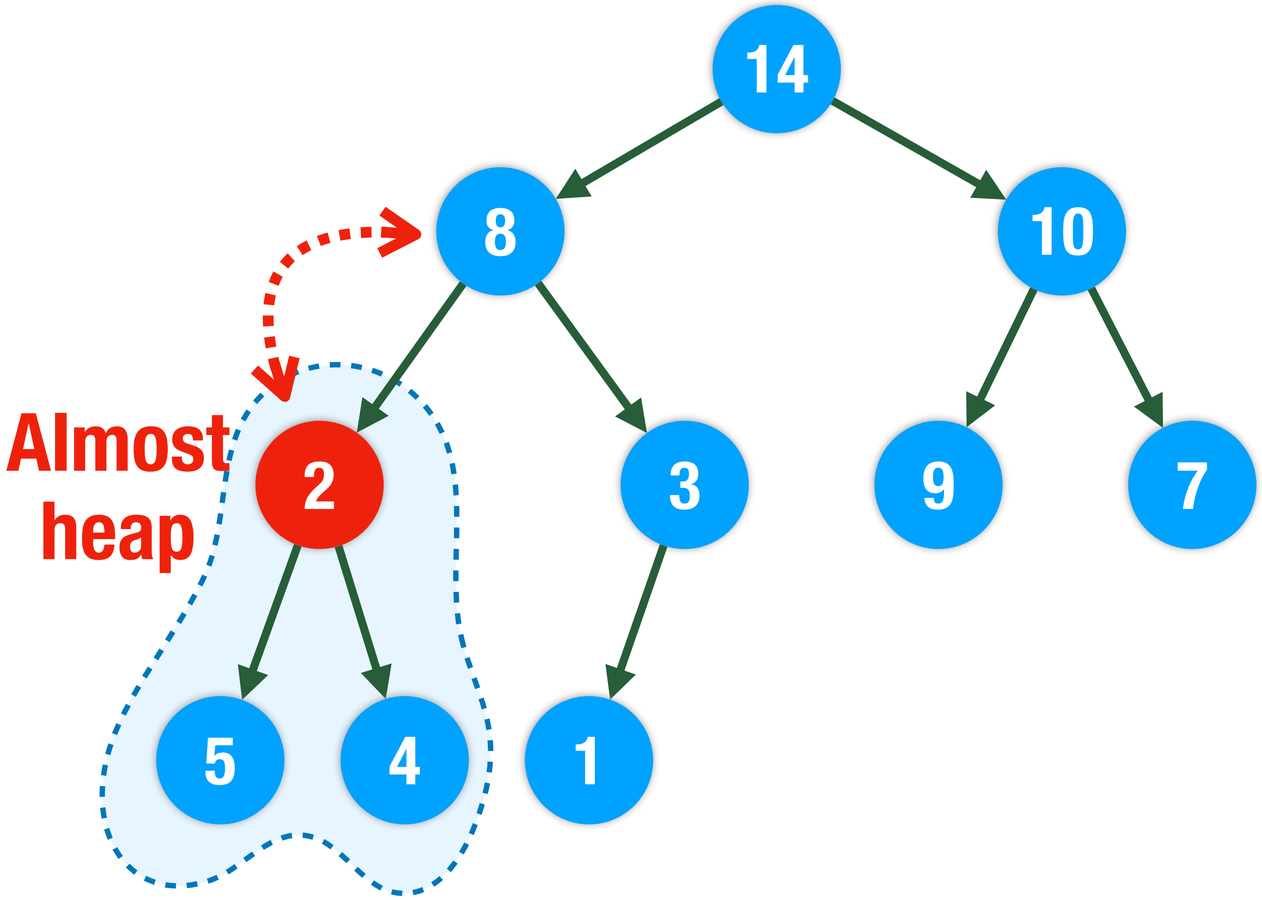

```
procedure Max-Heapify(A, i)
        **turns almost-heap into a heap
        **pre-condition:  tree rooted at A[i] is an almost-heap
        **post-condition:  tree rooted at A[i] is a heap
    lc ← leftchild(i)
    rc ← rightchild(i)
    largest ← i
    if (lc ≤ heapsize(A) and A[lc] > A[largest]) then
        largest ← lc
    if (rc ≤ heapsize(A) and A[rc] > A[largest]) then
        largest ← rc          **largest = index of max{A[i], A[rc], A[lc]}
    if (largest ≠ i) then
        exchange A[i] ↔ A[largest]
        Max-Heapify(A, largest)
```

Recap: Max-Heapify(A, i)




```
procedure Max-Heapify(A, i)
      **turns almost-heap into a heap
      **pre-condition:  tree rooted at A[i] is an almost-heap
      **post-condition:  tree rooted at A[i] is a heap
   lc ← leftchild(i)
   rc ← rightchild(i)
   largest ← i
   if (lc ≤ heapsize(A) and A[lc] > A[largest]) then
      largest ← lc
   if (rc ≤ heapsize(A) and A[rc] > A[largest]) then
      largest ← rc          **largest = index of max{A[i], A[rc], A[lc]}
   if (largest ≠ i) then
      exchange A[i] ↔ A[largest]
      Max-Heapify(A, largest)
```

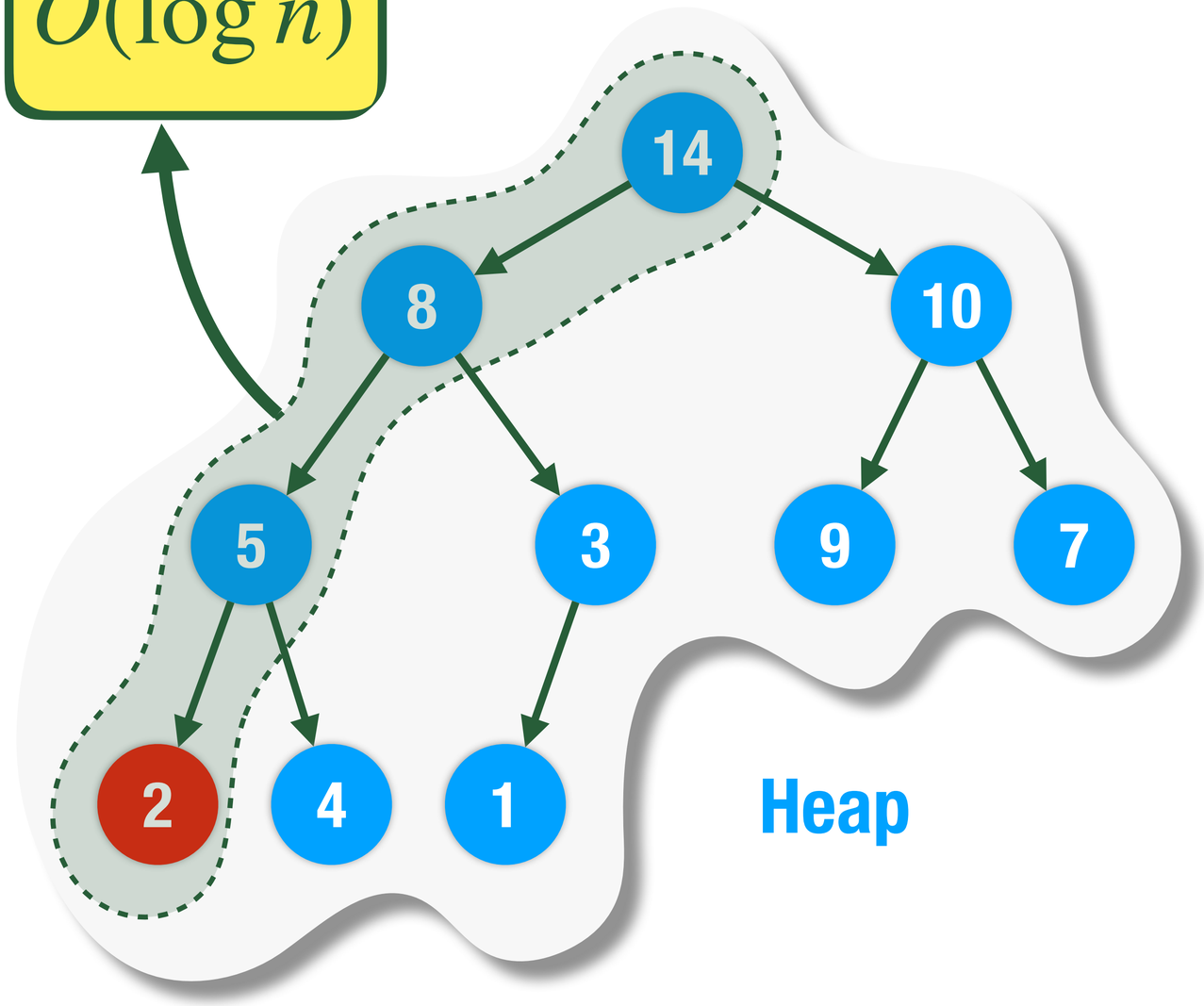
Recap: Max-Heapify(A, i)




```
procedure Max-Heapify(A, i)
      **turns almost-heap into a heap
      **pre-condition:  tree rooted at A[i] is an almost-heap
      **post-condition:  tree rooted at A[i] is a heap
   lc ← leftchild(i)
   rc ← rightchild(i)
   largest ← i
   if (lc ≤ heapsize(A) and A[lc] > A[largest]) then
      largest ← lc
   if (rc ≤ heapsize(A) and A[rc] > A[largest]) then
      largest ← rc          **largest = index of max{A[i], A[rc], A[lc]}
   if (largest ≠ i) then
      exchange A[i] ↔ A[largest]
      Max-Heapify(A, largest)
```

Recap: Max-Heapify(A, i)

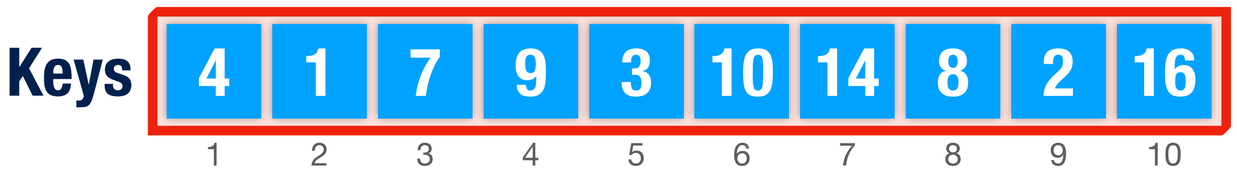
$O(\log n)$

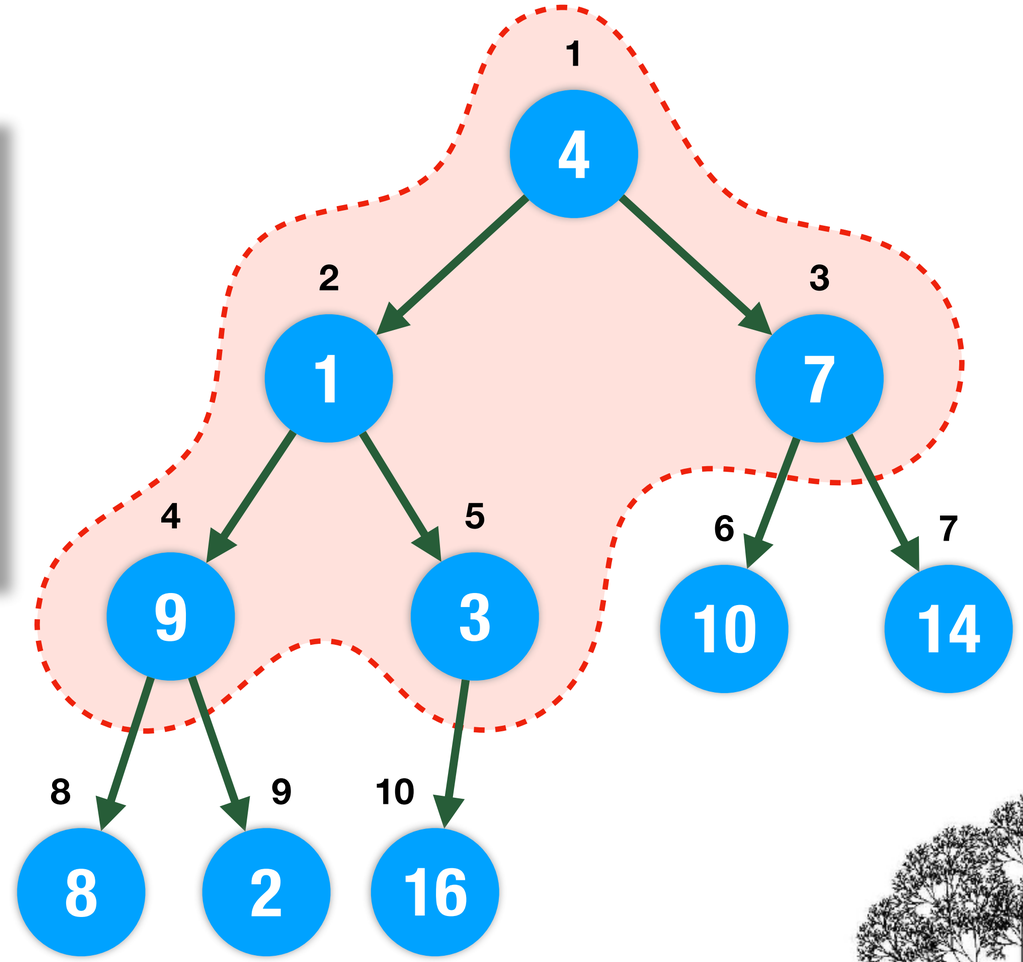


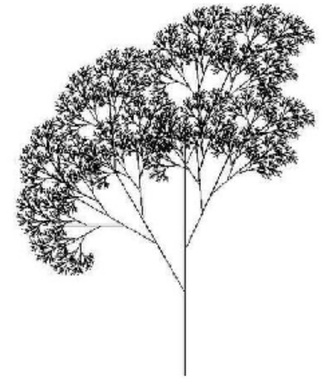

```
procedure Max-Heapify(A, i)
      **turns almost-heap into a heap
      **pre-condition:  tree rooted at A[i] is an almost-heap
      **post-condition:  tree rooted at A[i] is a heap
   lc ← leftchild(i)
   rc ← rightchild(i)
   largest ← i
   if (lc ≤ heapsize(A) and A[lc] > A[largest]) then
      largest ← lc
   if (rc ≤ heapsize(A) and A[rc] > A[largest]) then
      largest ← rc          **largest = index of max{A[i], A[rc], A[lc]}
   if (largest ≠ i) then
      exchange A[i] ↔ A[largest]
      Max-Heapify(A, largest)
```

Recap: Build-Max-Heap(A)

```
procedure Build-Max-Heap(A)
        **turn an array into a heap
    heapsize(A) ← length[A]
    for (i ← ⌊length[A]/2⌋ downto 1) do
        Max-Heapify(A, i)
```

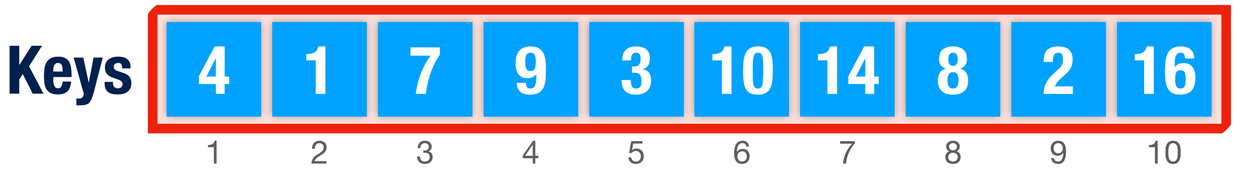


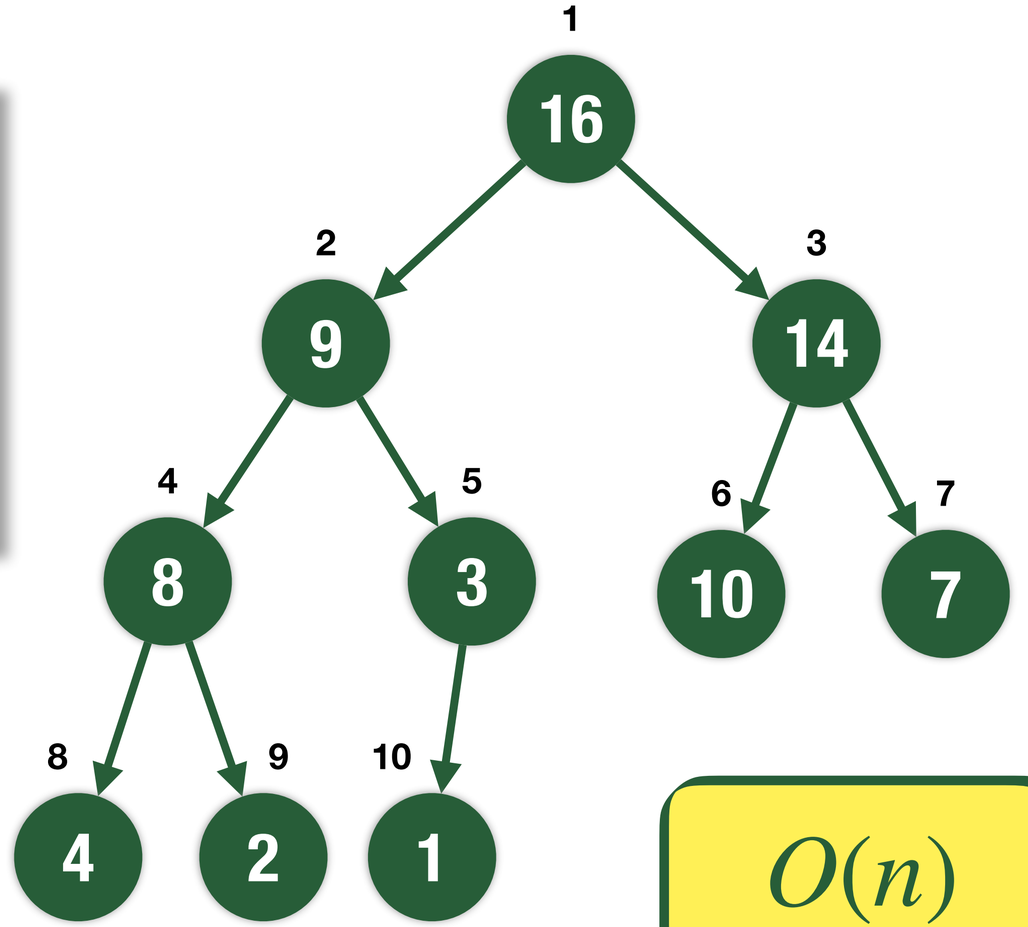




Recap: Build-Max-Heap(A)

```
procedure Build-Max-Heap(A)
        **turn an array into a heap
    heapsize(A) ← length[A]
    for (i ← ⌊length[A]/2⌋ downto 1) do
        Max-Heapify(A, i)
```

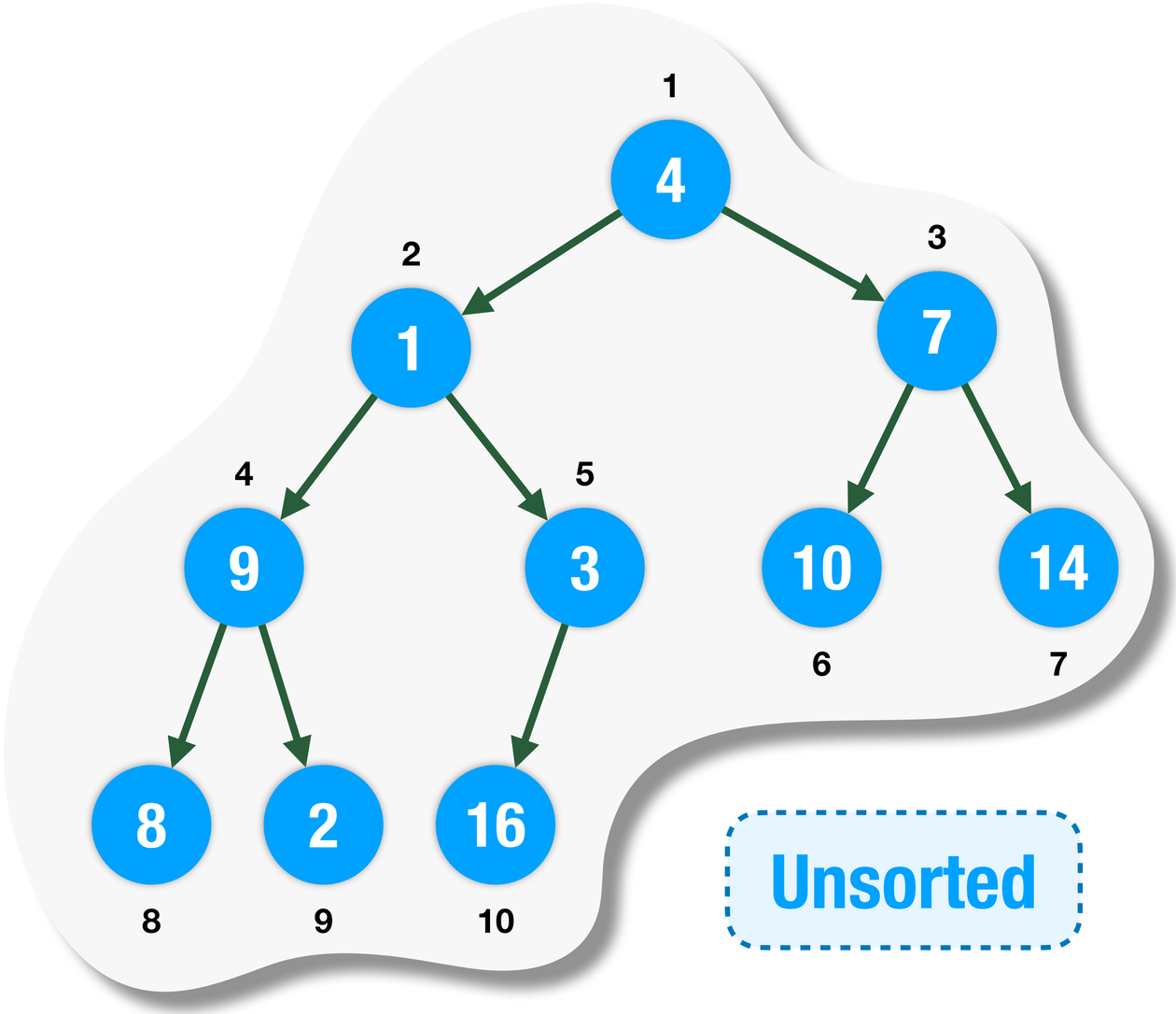




$O(n)$

Recap: Heapsort(A)

```
procedure Heapsort(A)
        **post-condition:  sorted array
    Build-Max-Heap(A)
    for (i ← heapsize(A) downto 2) do
        exchange A[1] ↔ A[i]
        heapsize(A) ← heapsize(A) − 1
        Max-Heapify(A, 1)
```

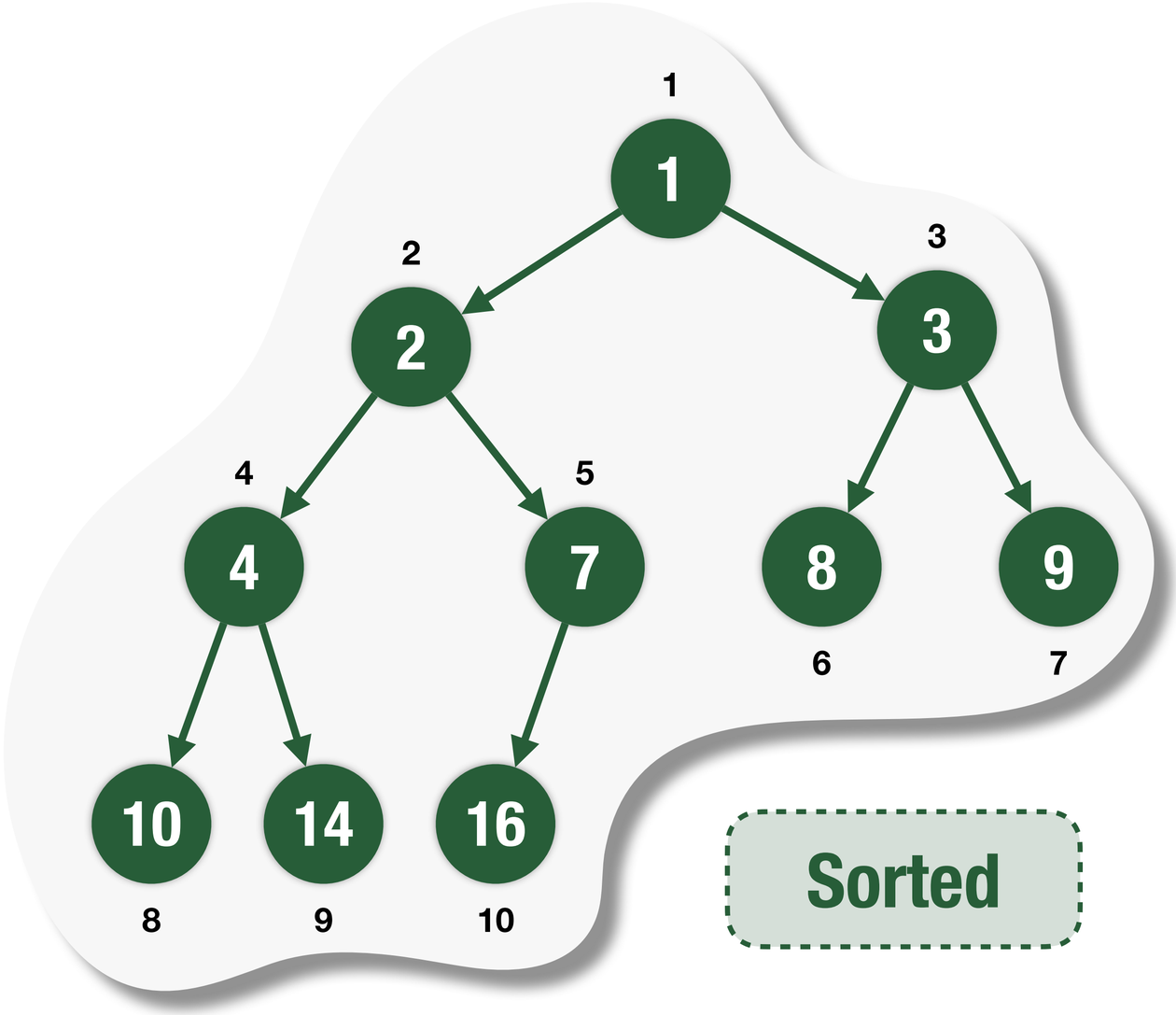


Recap: Heapsort(A)

$\Theta(n \log n)$

```
procedure Heapsort(A)
      **post-condition:  sorted array
   Build-Max-Heap(A)
   for (i ← heapsize(A) downto 2) do
      exchange A[1] ↔ A[i]
      heapsize(A) ← heapsize(A) − 1
      Max-Heapify(A, 1)
```

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



Lecture 14

Heaps; Priority Queues

Priority Queues; Examples

Python code:
priority_queue.py

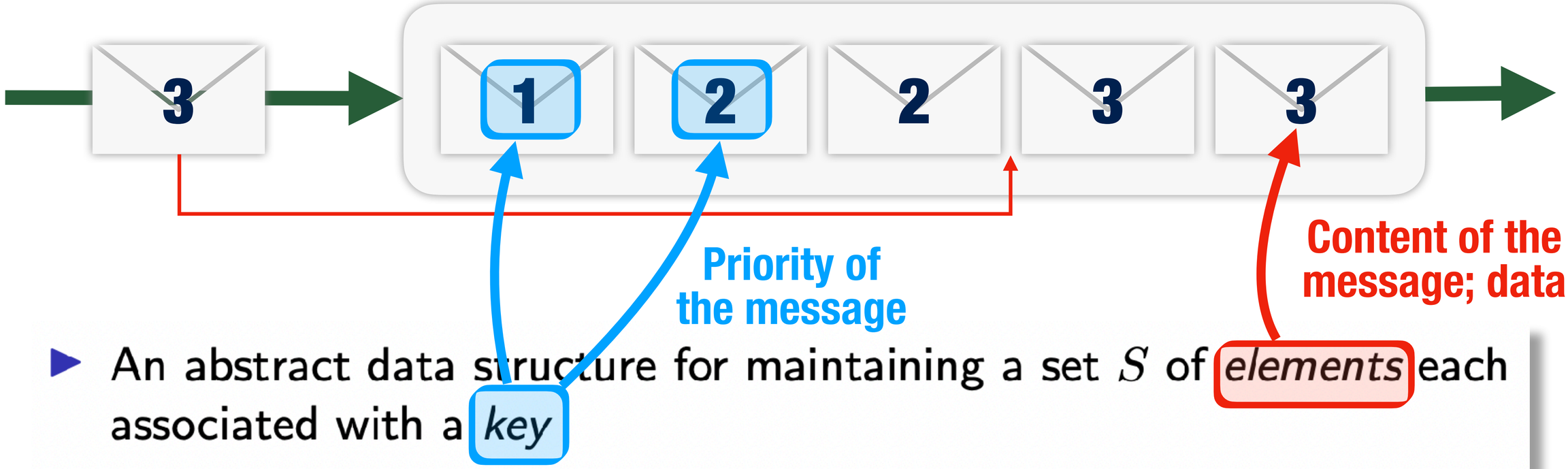
Motivating Examples

Messages are ordered in the queue based on “priority” specified

Examples:

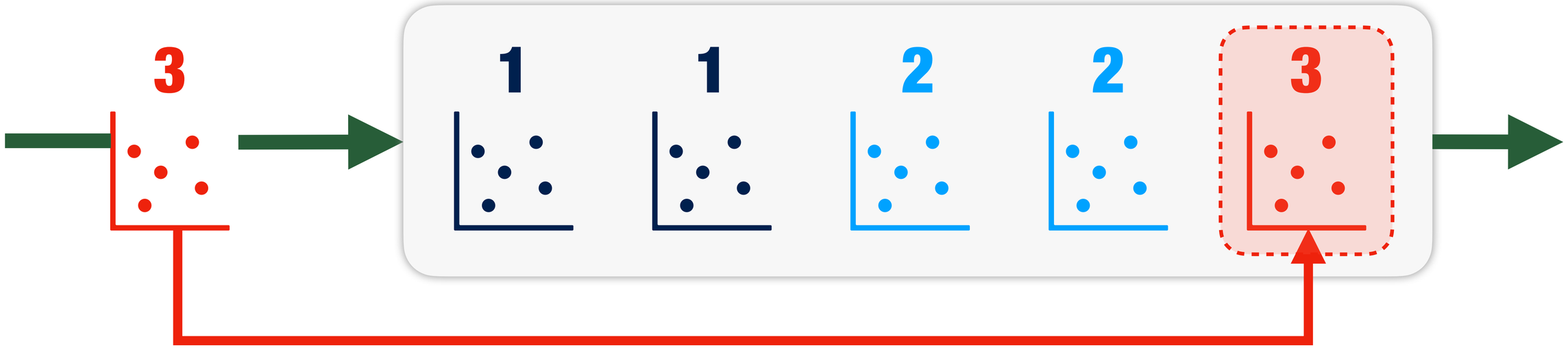
- events in calendar, ordered by time, what’s the next event?
- messages sent over a cable with limited bandwidth, what to send first?

Priority Queue: Key and Element



- An abstract data structure for maintaining a set S of *elements* each associated with a *key*
- Key — represents the priority of the element

Priority Queue: Another Example



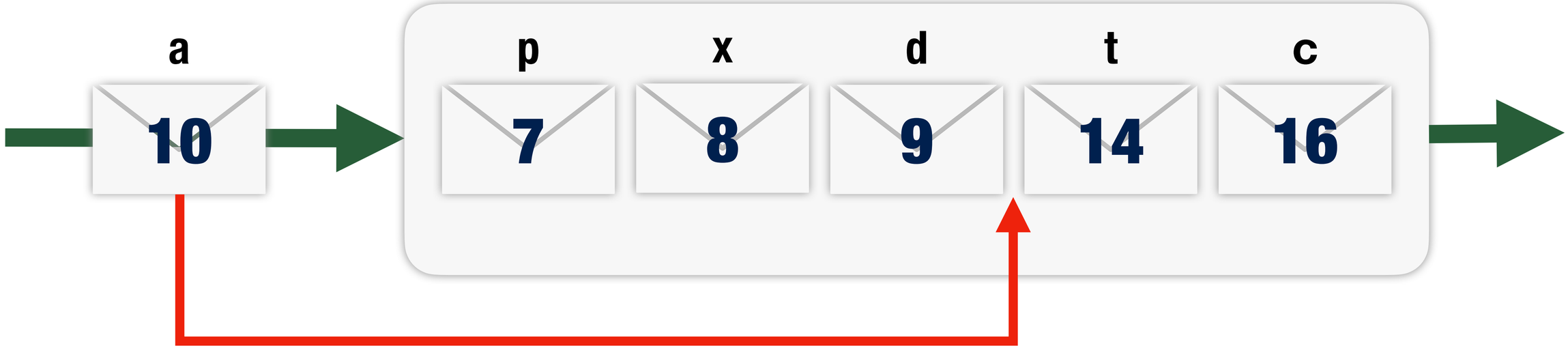
- Example: a set of jobs to be scheduled on a shared computer.
  - The jobs arrive and should be placed in the queue.
  - Each has a priority. Queue should be with respect to this.
  - To perform a job, we "extract" the one in the queue with highest priority.

Priority Queue Operations

- In general, a PQ supports these operations:
  - `initialize` — insert all keys at once
  - `insert` — a new element — **Enqueue**
  - `maximum` — return the element with the maximum key
  - `extract maximum` — return the maximum and remove the element from the queue — **Dequeue**
  - `increase key` — increase the priority for an element
- Implementation? Heap !!!

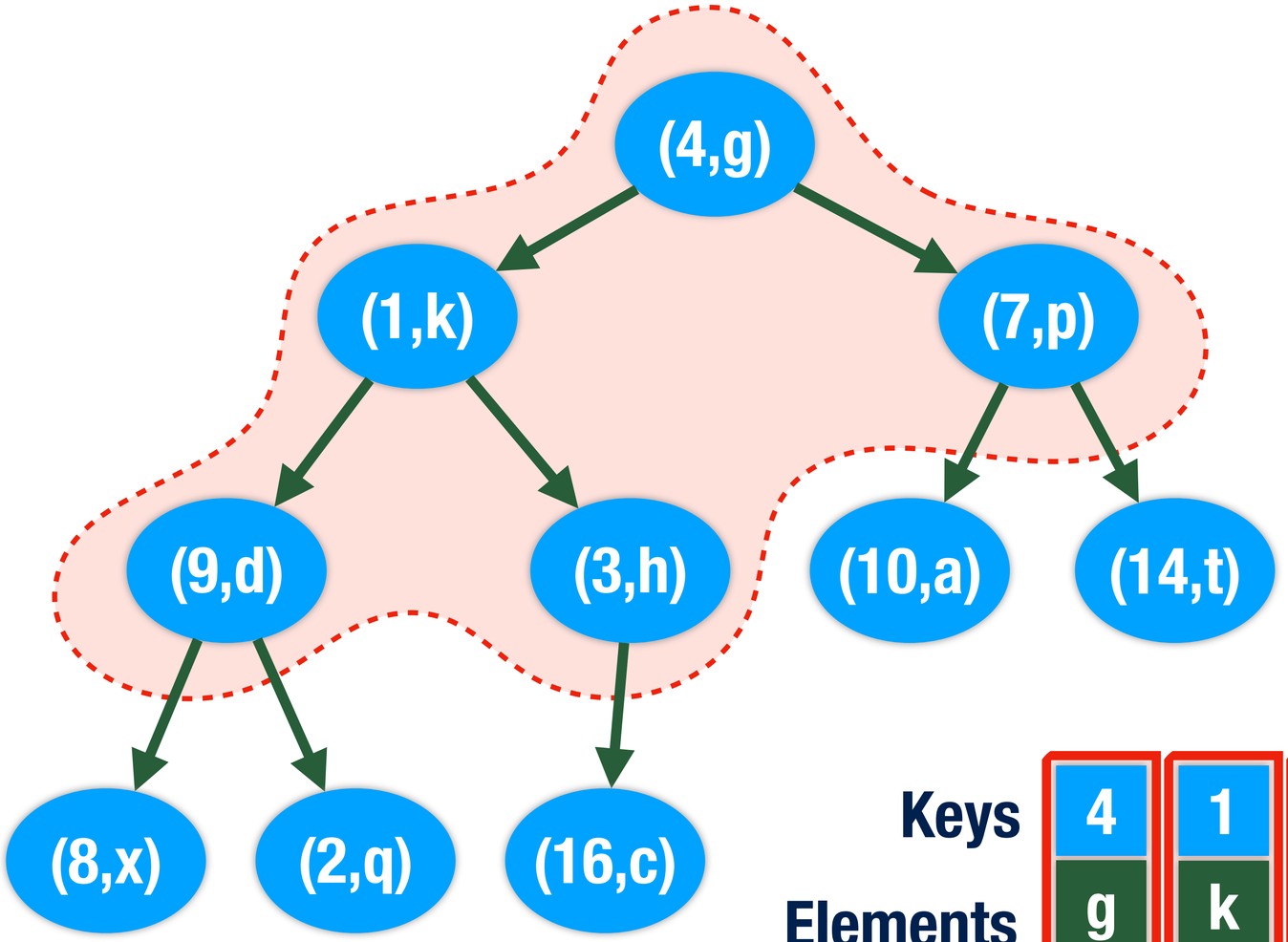
Python code: heap.py,
priority_queue.py

PQ: Heap Implementation



	1	2	3	4	5	6	7	8	9	10
Keys	4	1	7	9	3	10	14	8	2	16
Elements	g	k	p	d	h	a	t	x	q	c

PQ Initialization



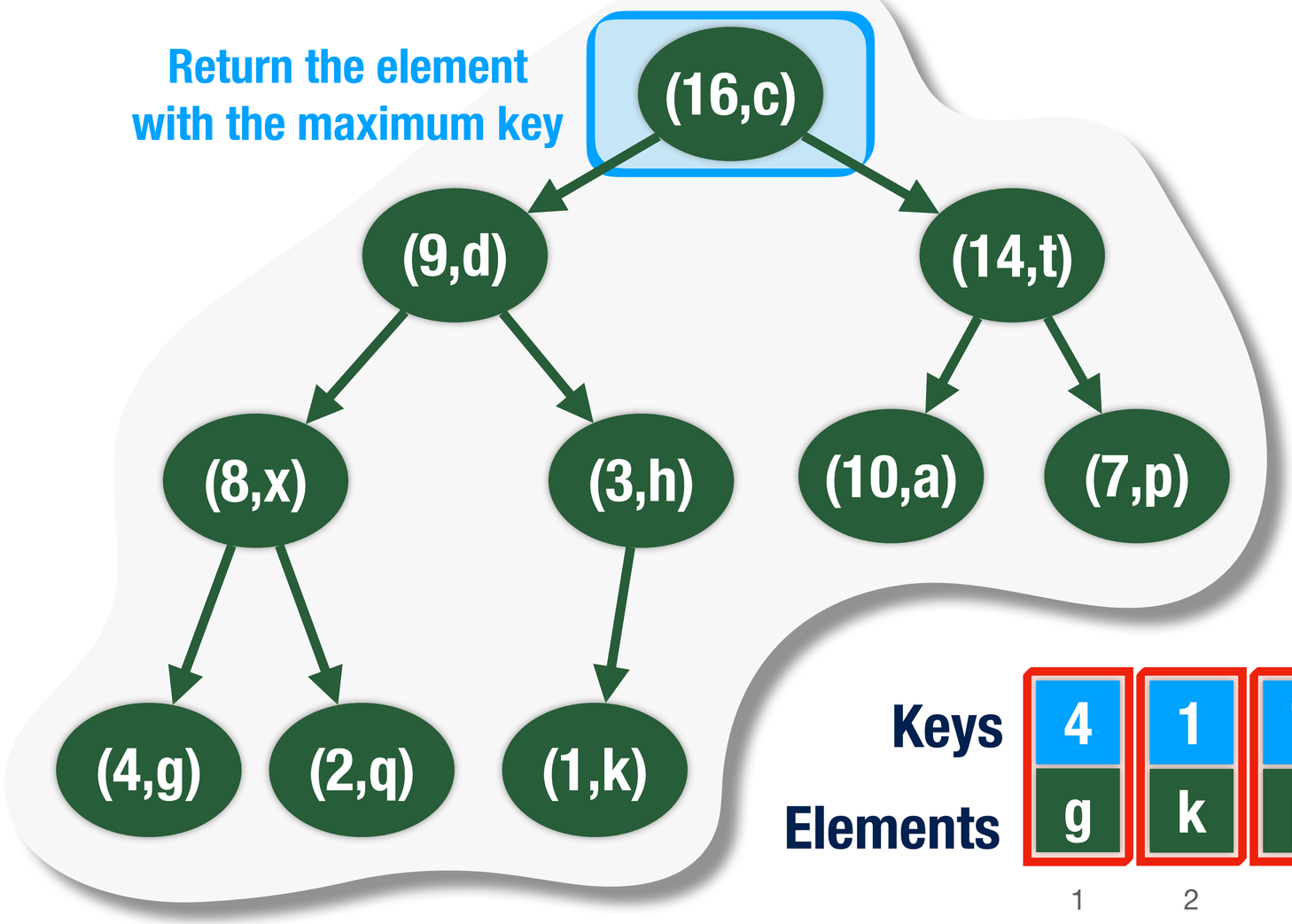

```
procedure Build-Max-Heap(A)
        **turn an array into a heap
    heapsize(A) ← length[A]
    for (i ← ⌊length[A]/2⌋ downto 1) do
        Max-Heapify(A, i)
```

	1	2	3	4	5	6	7	8	9	10
Keys	4	1	7	9	3	10	14	8	2	16
Elements	g	k	p	d	h	a	t	x	q	c

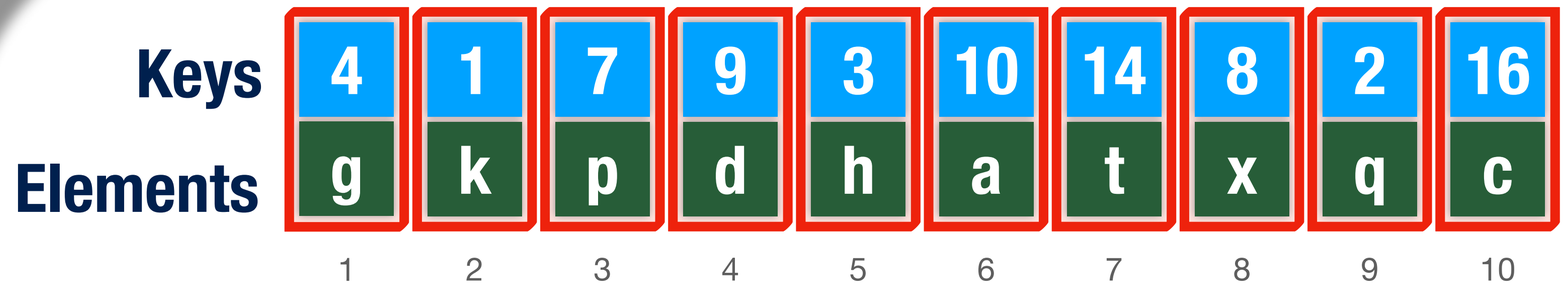
Initialize: Build-Max-Heap(A)

Maximize(A) = $A[1]$

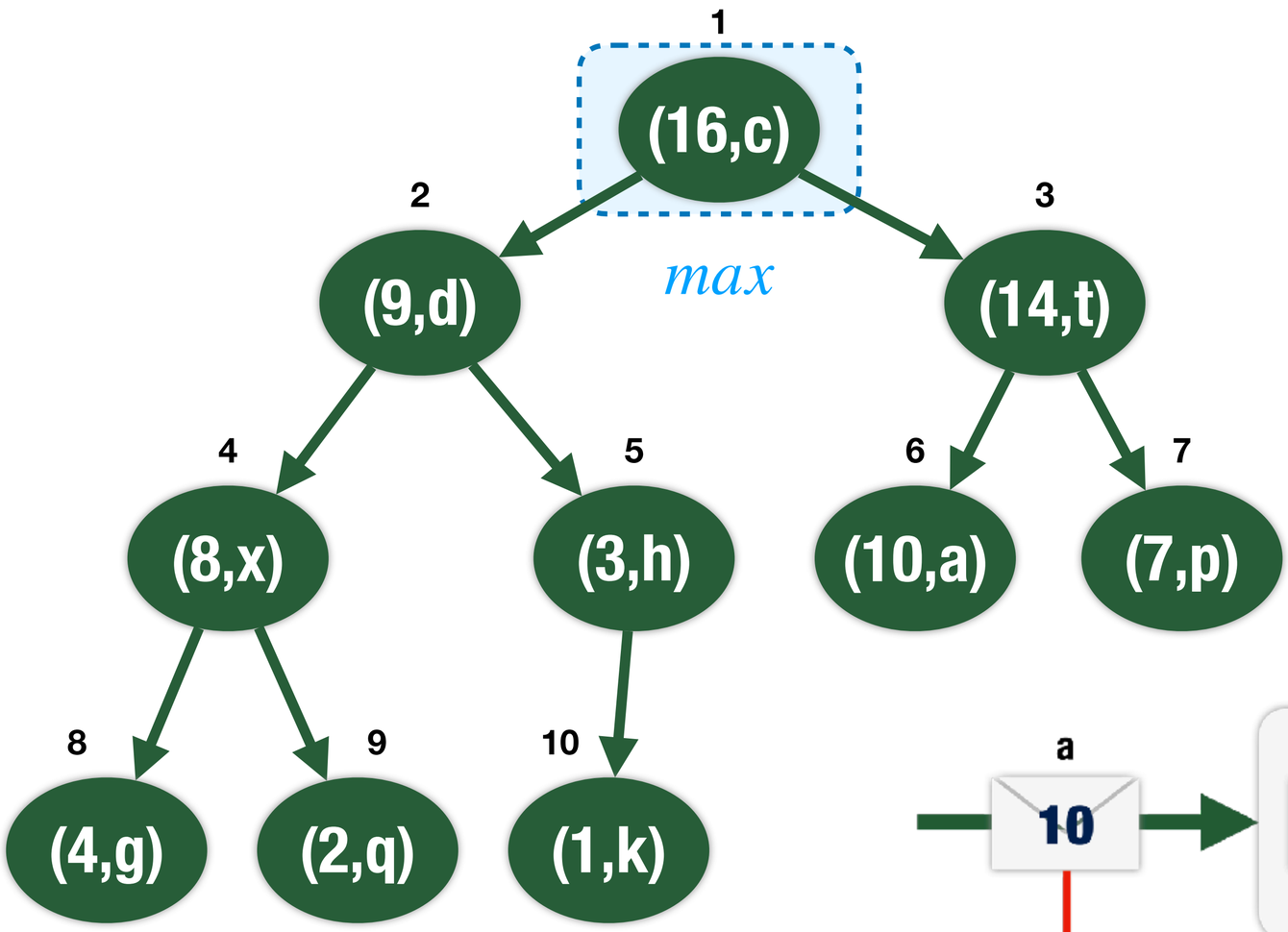
Return the element with the maximum key



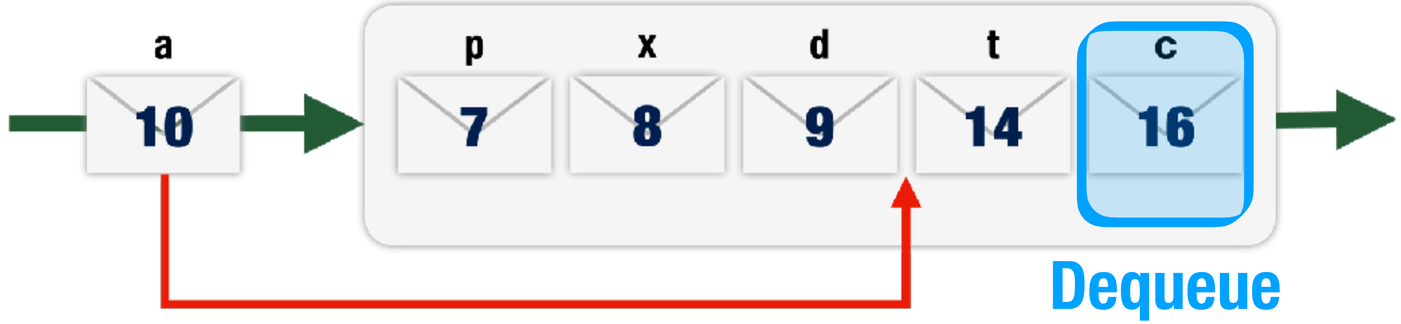

```
procedure Build-Max-Heap(A)
        **turn an array into a heap
    heapsize(A) ← length[A]
    for (i ← ⌊length[A]/2⌋ downto 1) do
        Max-Heapify(A, i)
```



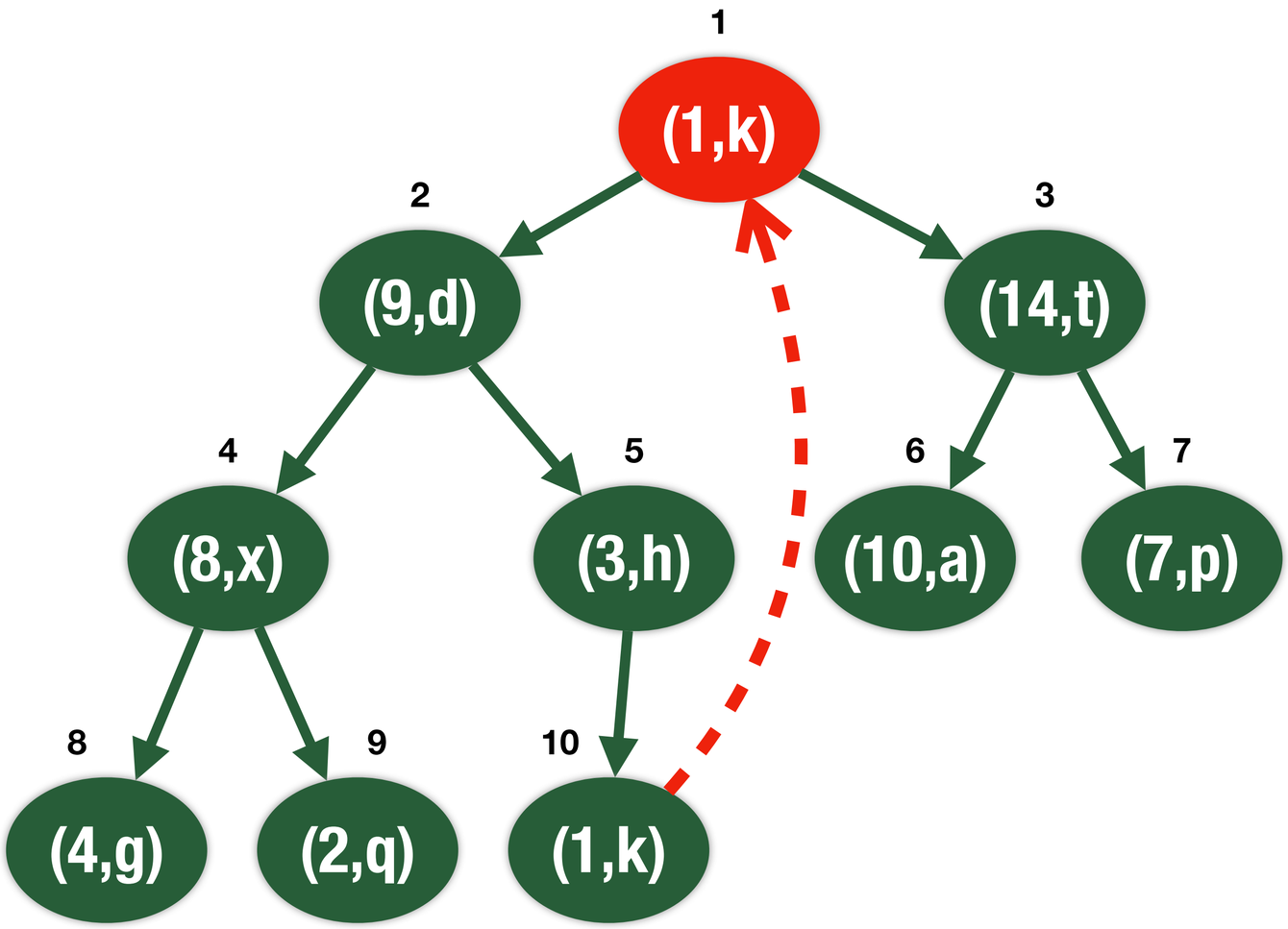
Extract Maximum: Heap-Extract-Max(A)



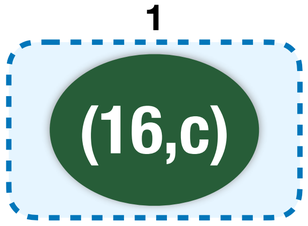

```
procedure Heap-Extract-Max(A)
    ** precondition: A isn't empty
    max ← A[1]
    A[1] ← A[heapsize[A]]
    heapsize[A] ← heapsize[A] − 1
    if (heapsize[A] > 0) then
        Max-Heapify(A, 1)
    return max
```



Extract Maximum: Heap-Extract-Max(A)

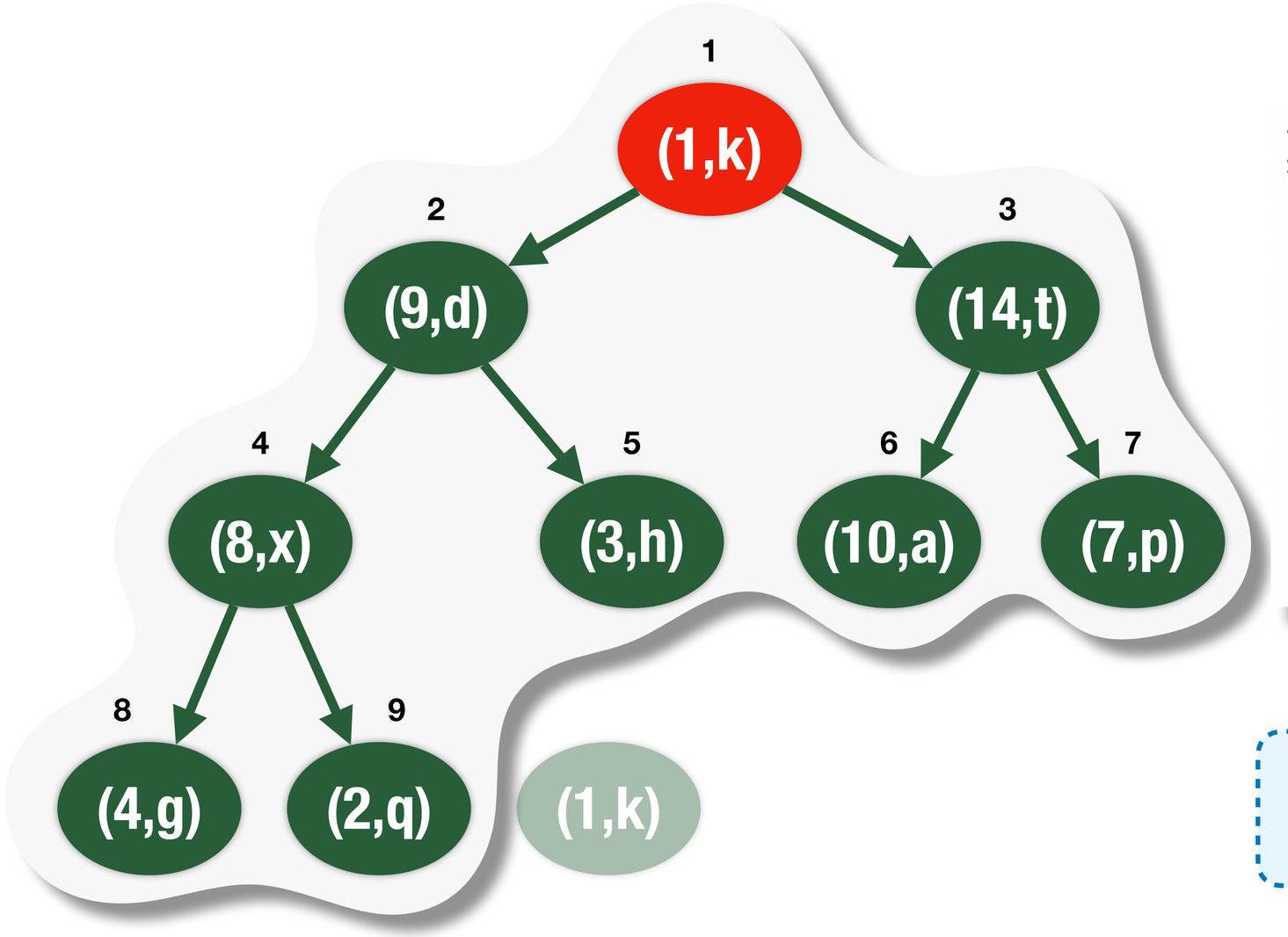



```
procedure Heap-Extract-Max(A)
    ** precondition: A isn't empty
    max ← A[1]
    A[1] ← A[heapsize[A]]
    heapsize[A] ← heapsize[A] − 1
    if (heapsize[A] > 0) then
        Max-Heapify(A, 1)
    return max
```

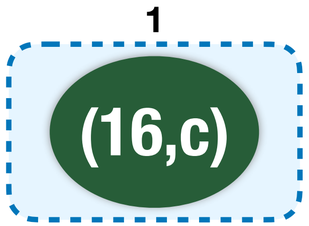


max

Extract Maximum: Heap-Extract-Max(A)

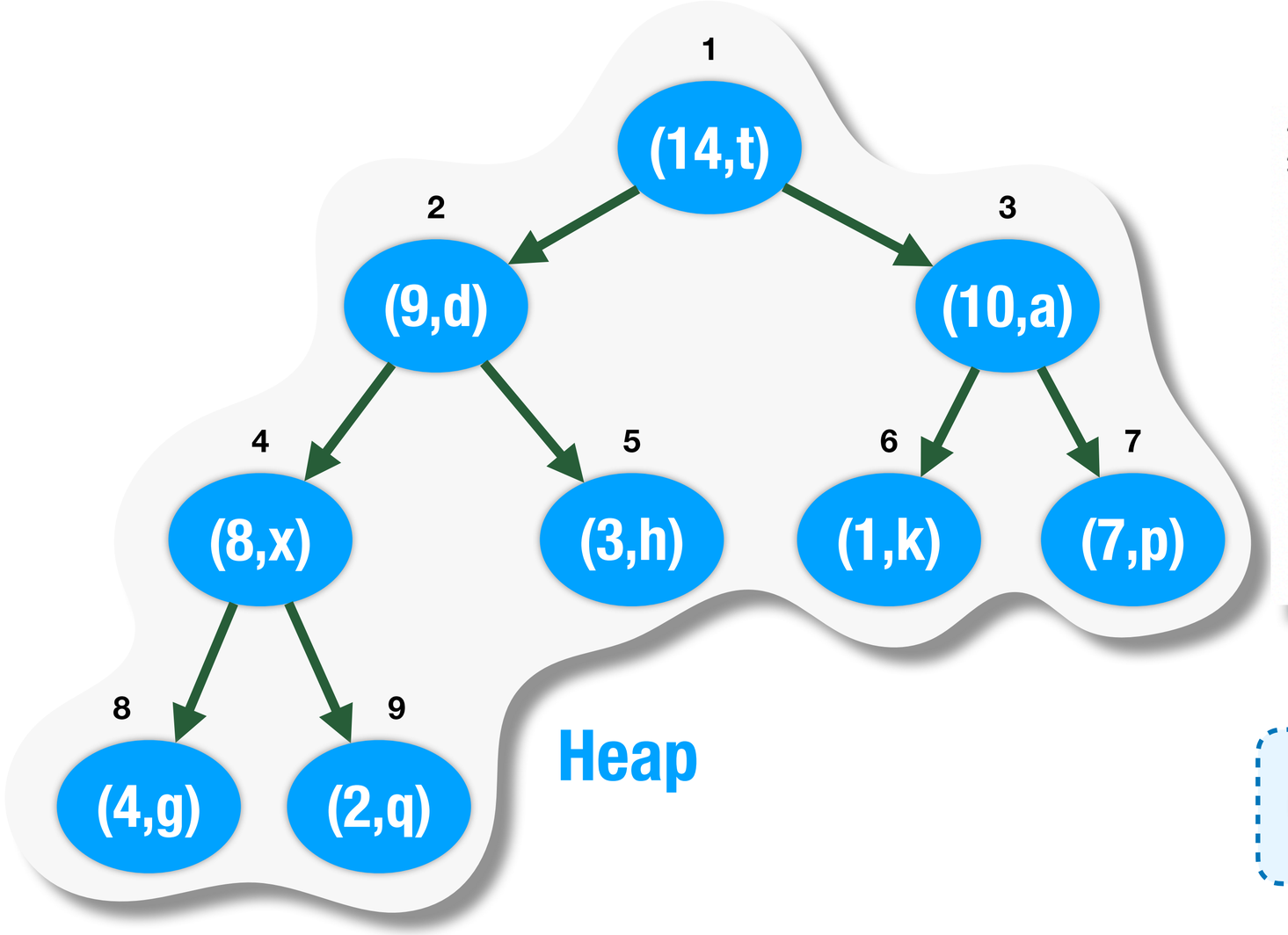



```
procedure Heap-Extract-Max(A)
    ** precondition: A isn't empty
    max ← A[1]
    A[1] ← A[heapsize[A]]
    heapsize[A] ← heapsize[A] − 1
    if (heapsize[A] > 0) then
        Max-Heapify(A, 1)
    return max
```

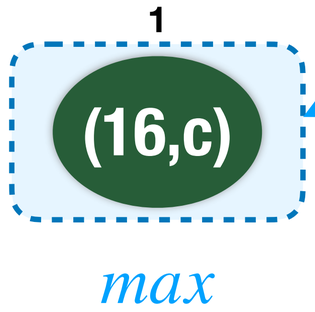


max

Extract Maximum: Heap-Extract-Max(A)

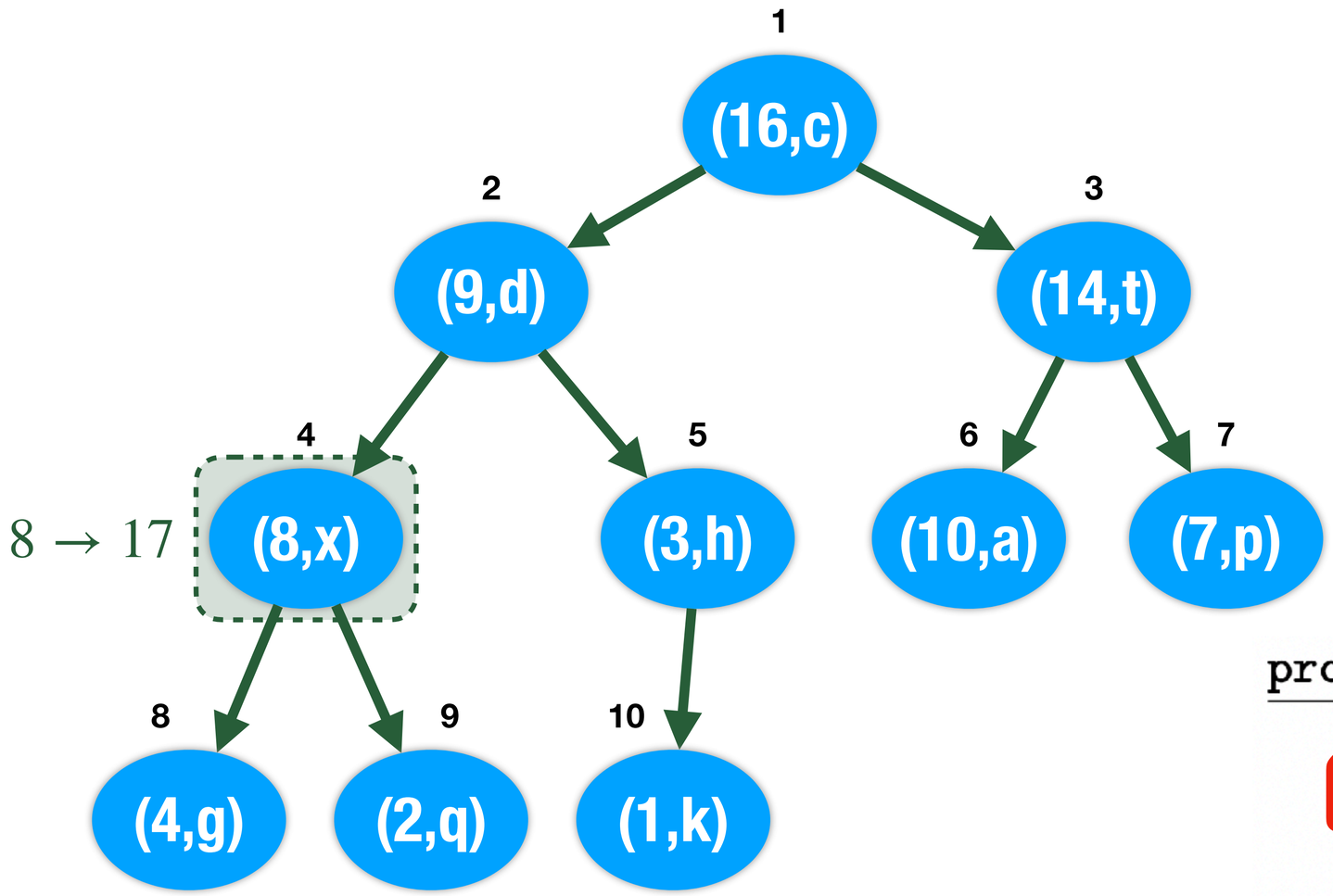



```
procedure Heap-Extract-Max(A)
    ** precondition: A isn't empty
    max ← A[1]
    A[1] ← A[heapsize[A]]
    heapsize[A] ← heapsize[A] − 1
    if (heapsize[A] > 0) then
        Max-Heapify(A, 1)
    return max
```



$\Theta(\log n)$

Increase Key: Heap-Increase-Key(A, i, key)



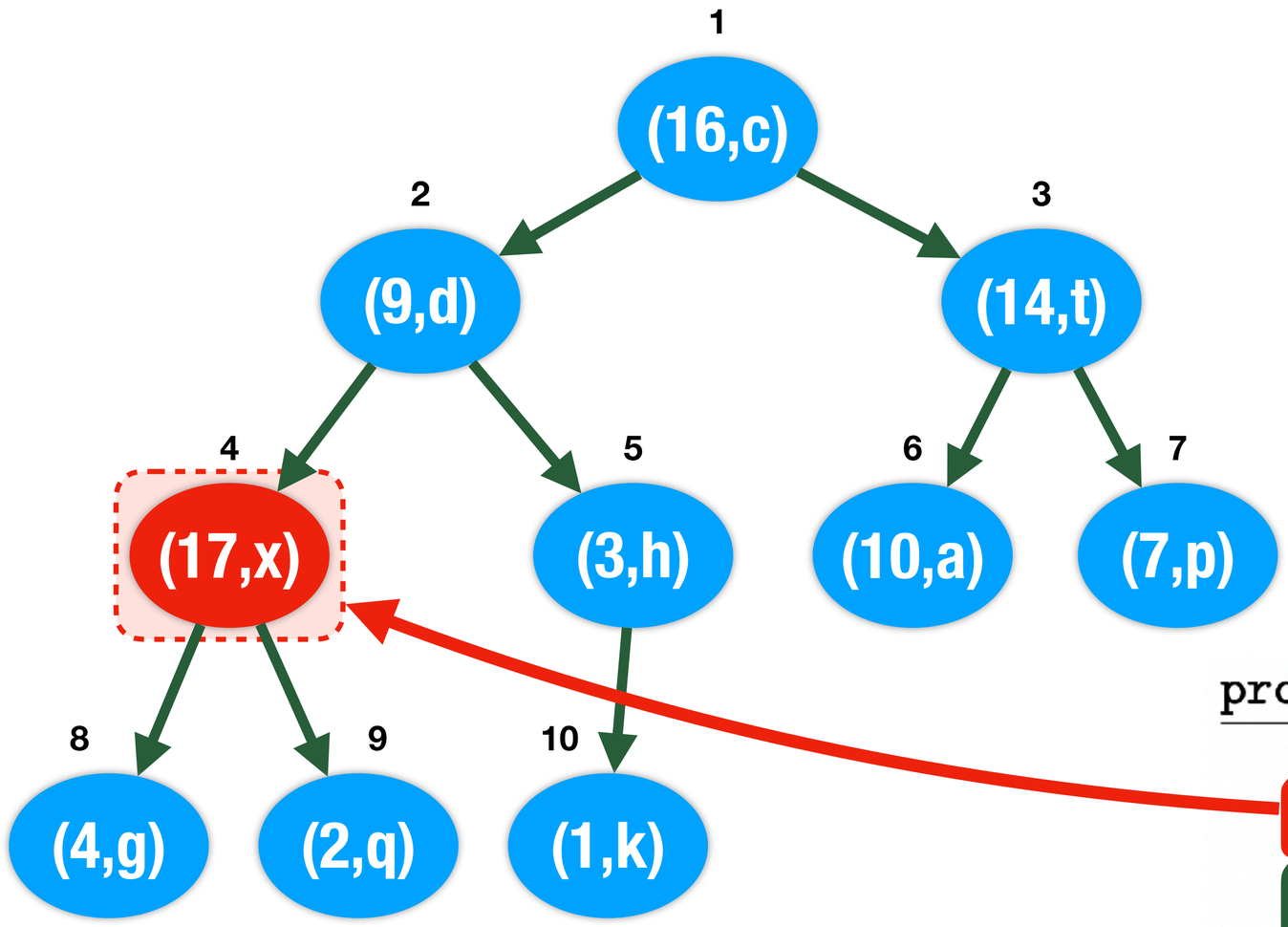
The inverse of Max-Heapify

Increase the priority value for $A[i]$ and "bubble up" until max-heap property is restored.

$(A,4,17)$

```
procedure Heap-Increase-Key(A, i, key)
    ** Precondition: key ≥ A[i]
    A[i] ← key
    while (i > 1 and A[Parent(i)] < A[i]) do
        exchange A[i] ↔ A[Parent(i)]
        i ← Parent(i)
```

Increase Key: Heap-Increase-Key(A, i, key)

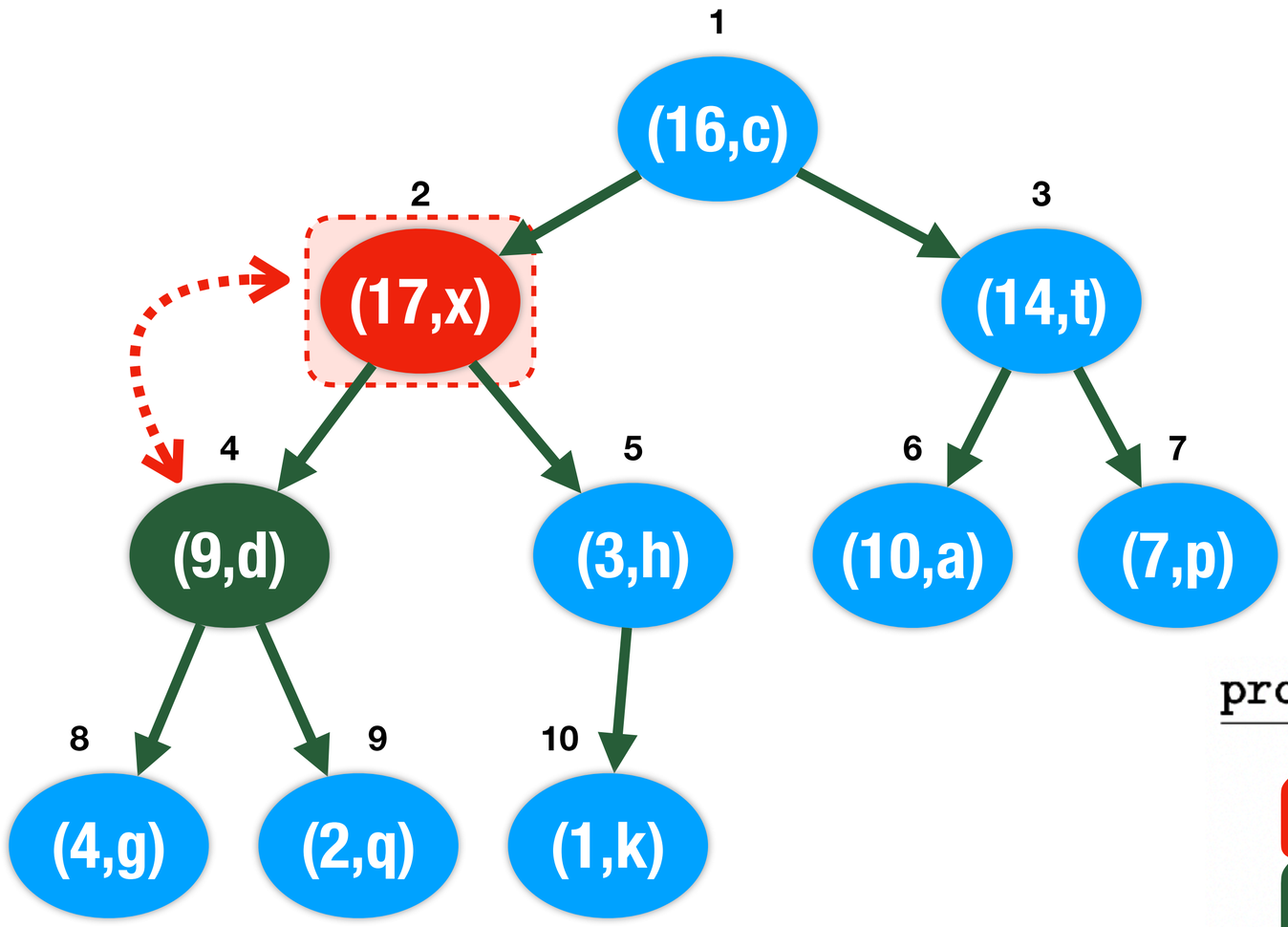


The inverse of Max-Heapify

Increase the priority value for $A[i]$ and "bubble up" until max-heap property is restored.

```
procedure Heap-Increase-Key(A, i, key)
    ** Precondition: key ≥ A[i]
    A[i] ← key
    while (i > 1 and A[Parent(i)] < A[i]) do
        exchange A[i] ↔ A[Parent(i)]
        i ← Parent(i)
```

Increase Key: Heap-Increase-Key(A, i, key)

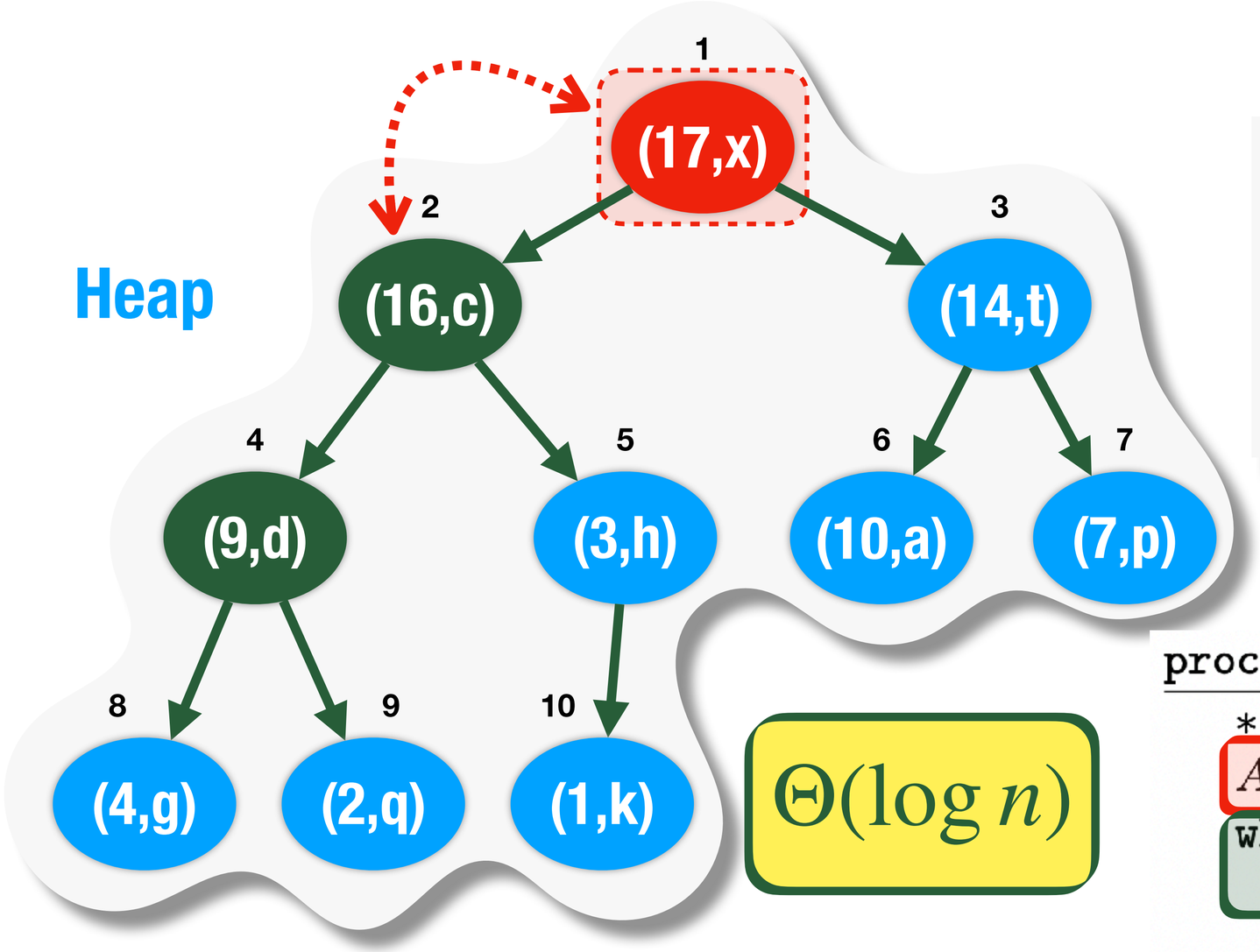


The inverse of Max-Heapify

Increase the priority value for $A[i]$ and "bubble up" until max-heap property is restored.

```
procedure Heap-Increase-Key(A, i, key)
    ** Precondition: key ≥ A[i]
    A[i] ← key
    while (i > 1 and A[Parent(i)] < A[i]) do
        exchange A[i] ↔ A[Parent(i)]
        i ← Parent(i)
```

Increase Key: Heap-Increase-Key(A, i, key)

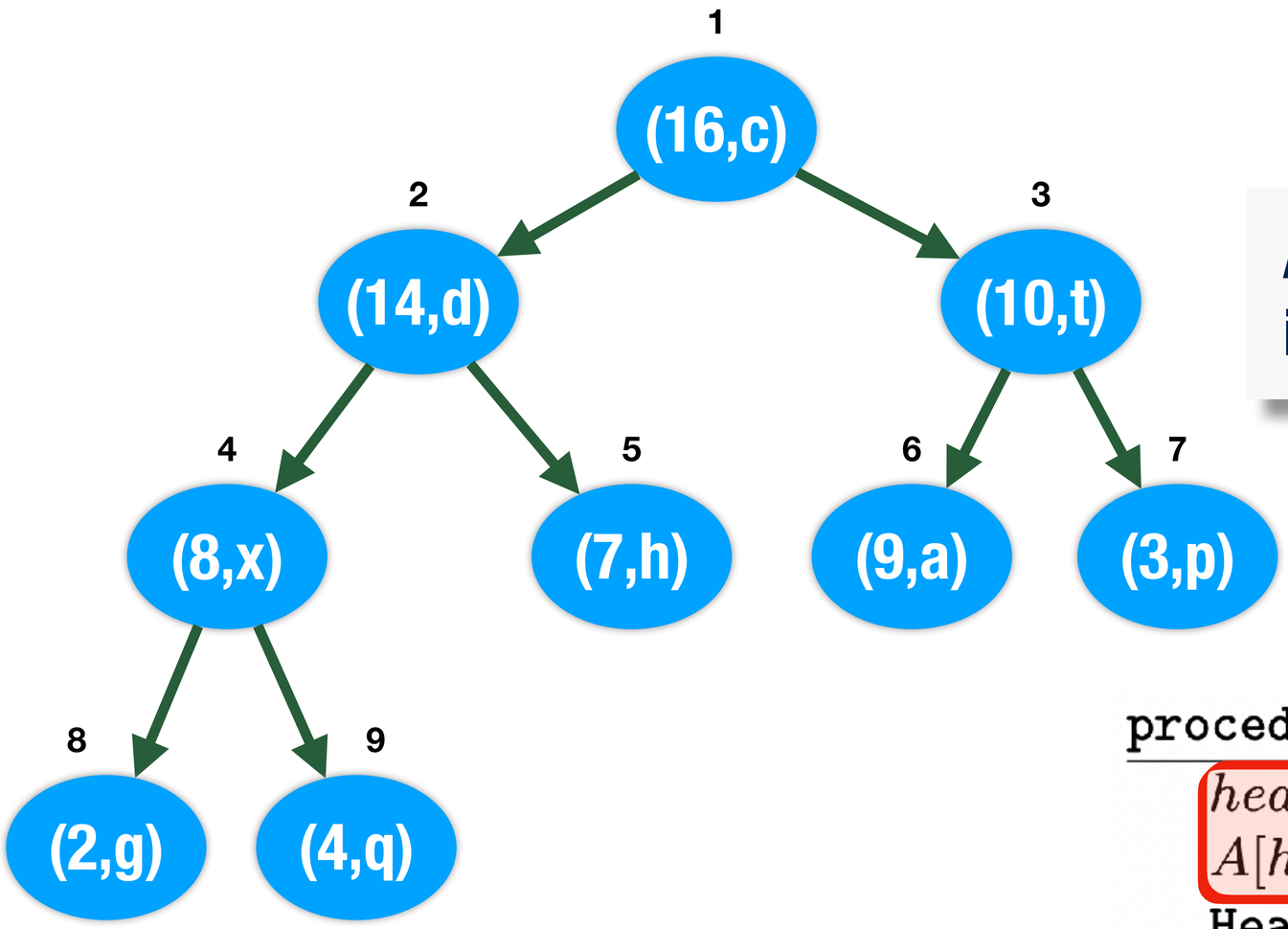


The inverse of Max-Heapify

Increase the priority value for $A[i]$ and "bubble up" until max-heap property is restored.

```
procedure Heap-Increase-Key(A, i, key)
    ** Precondition: key ≥ A[i]
    A[i] ← key
    while (i > 1 and A[Parent(i)] < A[i]) do
        exchange A[i] ↔ A[Parent(i)]
        i ← Parent(i)
```

Insert: Heap-Insert(A, key)

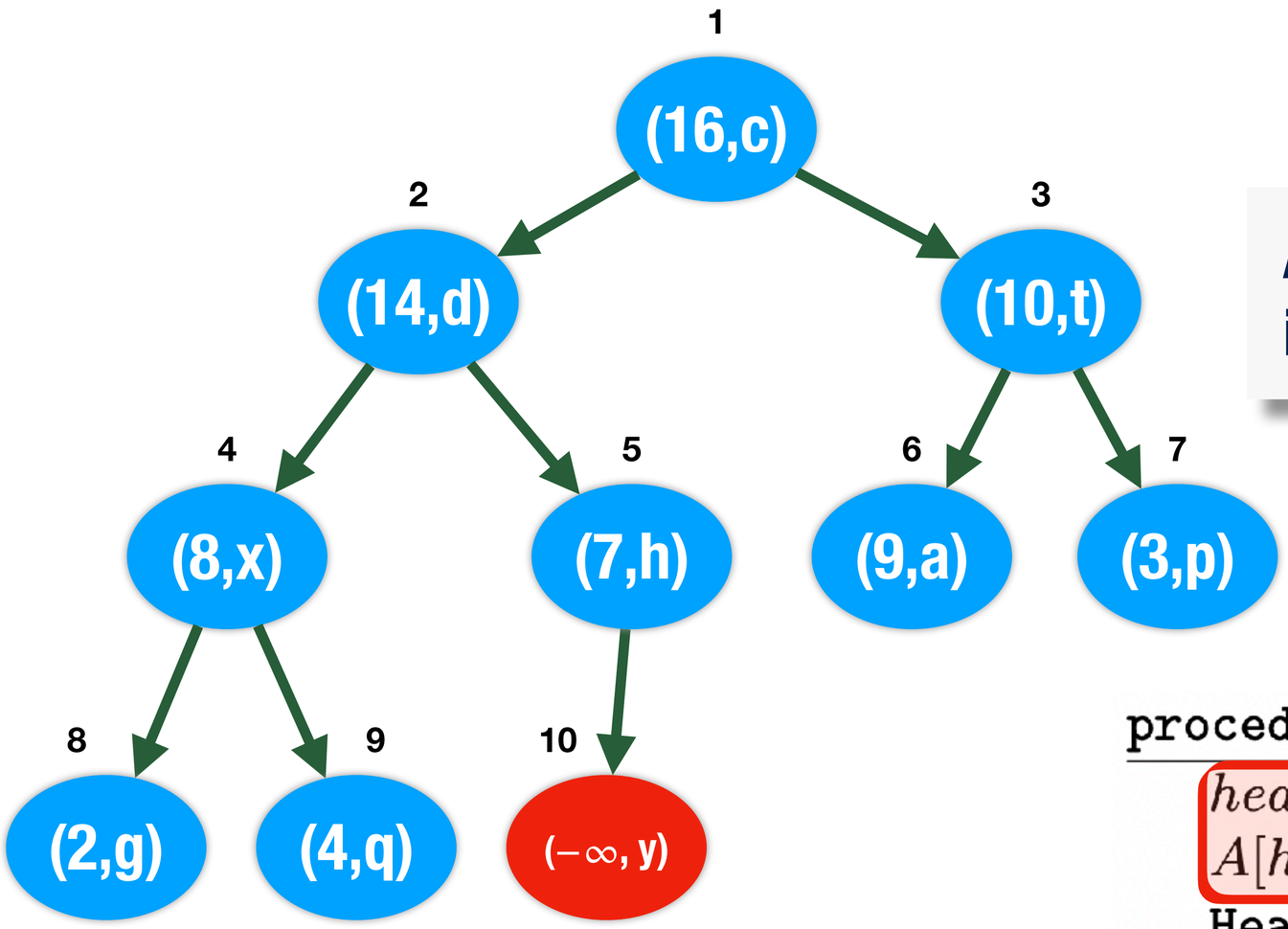


Add a new key with lowest priority, increase its priority to new key

$(A, 15)$

```
procedure Heap-Insert(A, key)
    heapsize[A] ← heapsize[A] + 1
    A[heapsize[A]] ← −∞          ** or any value
    Heap-Increase-key (A, heapsize[A], key)
```

Insert: Heap-Insert(A, key)

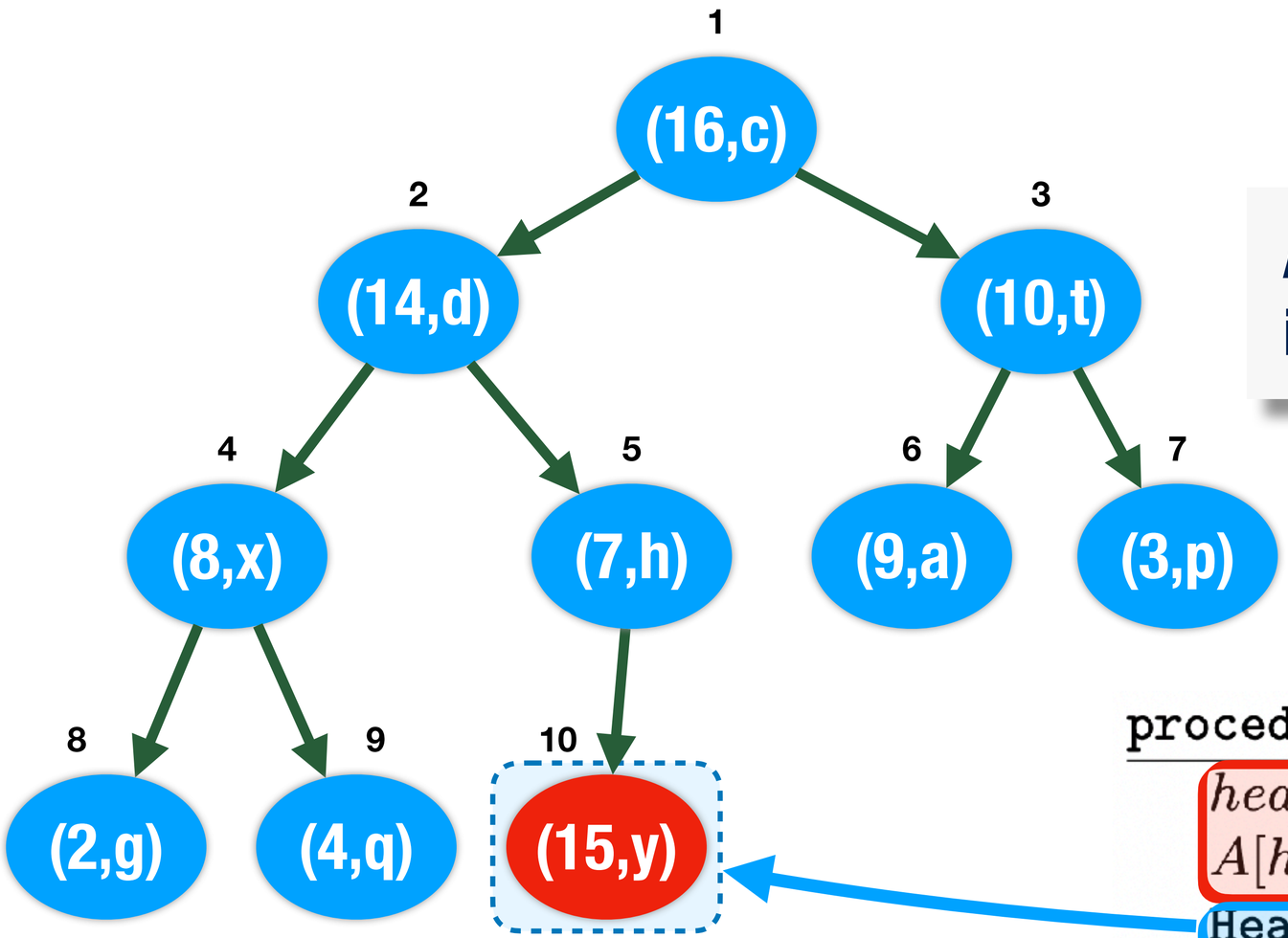


Add a new key with lowest priority, increase its priority to new key

$(A, 15)$

```
procedure Heap-Insert(A, key)
    heapsize[A] ← heapsize[A] + 1
    A[heapsize[A]] ← −∞          ** or any value
    Heap-Increase-key (A, heapsize[A], key)
```

Insert: Heap-Insert(A, key)

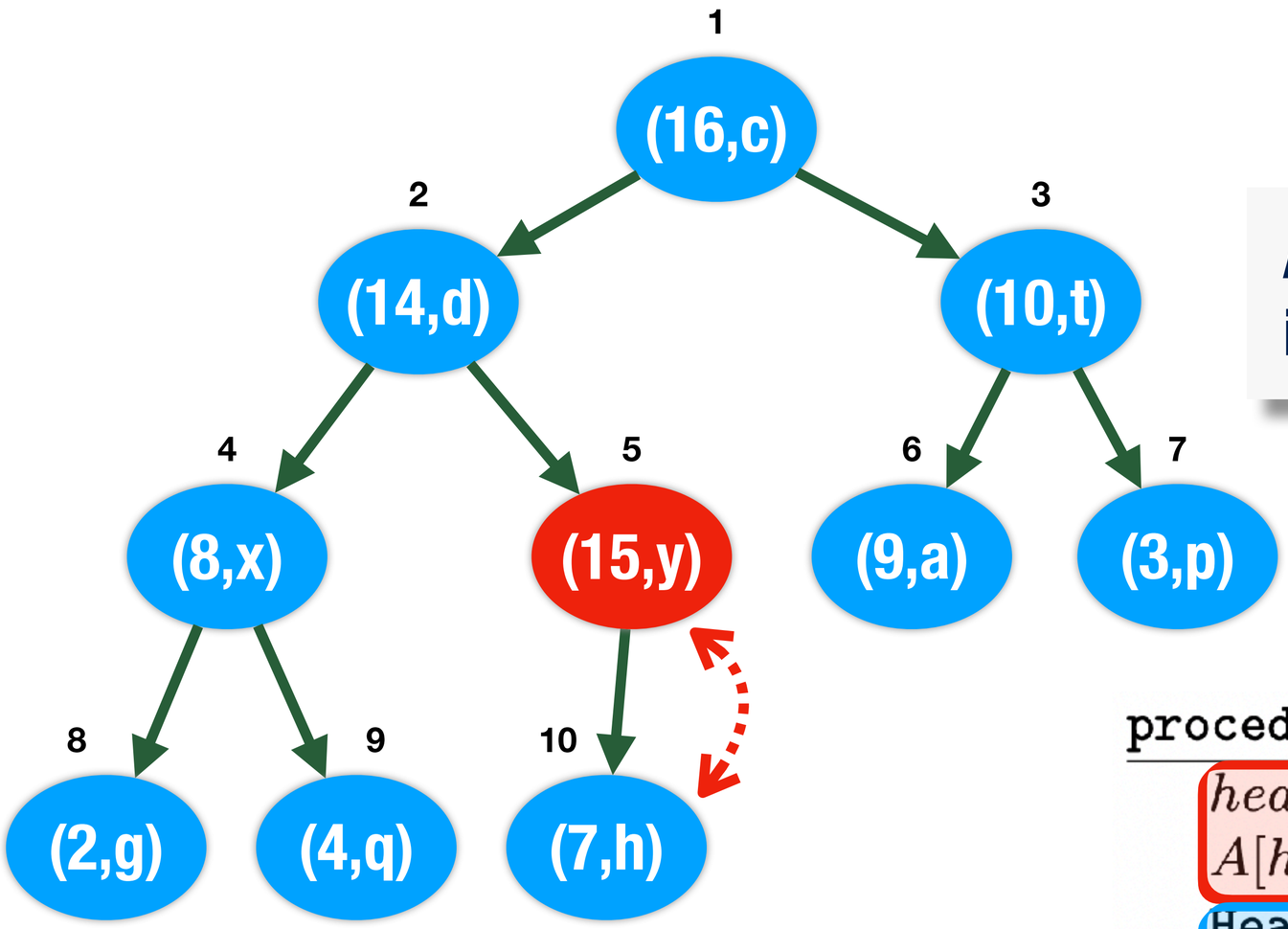


Add a new key with lowest priority, increase its priority to new key

$(A, 15)$

```
procedure Heap-Insert(A, key)
    heapsize[A] ← heapsize[A] + 1
    A[heapsize[A]] ← −∞        ** or any value
    Heap-Increase-key (A, heapsize[A], key)
```

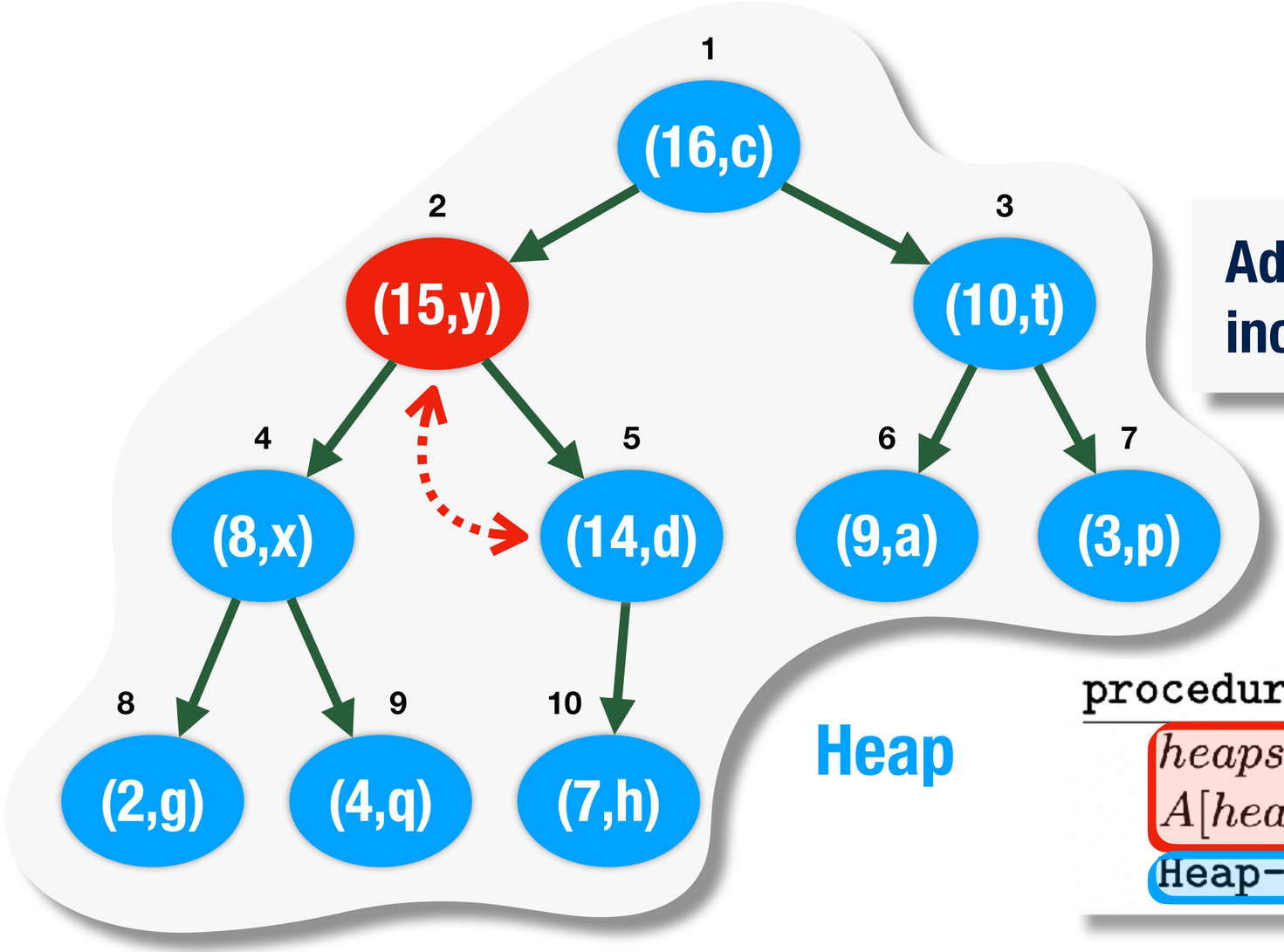
Insert: Heap-Insert(A, key)



Add a new key with lowest priority, increase its priority to new key

```
procedure Heap-Insert(A, key)
    heapsize[A] ← heapsize[A] + 1
    A[heapsize[A]] ← −∞          ** or any value
    Heap-Increase-key (A, heapsize|A|, key)
```

Insert: Heap-Insert(A, key)



Add a new key with lowest priority, increase its priority to new key

$\Theta(\log n)$

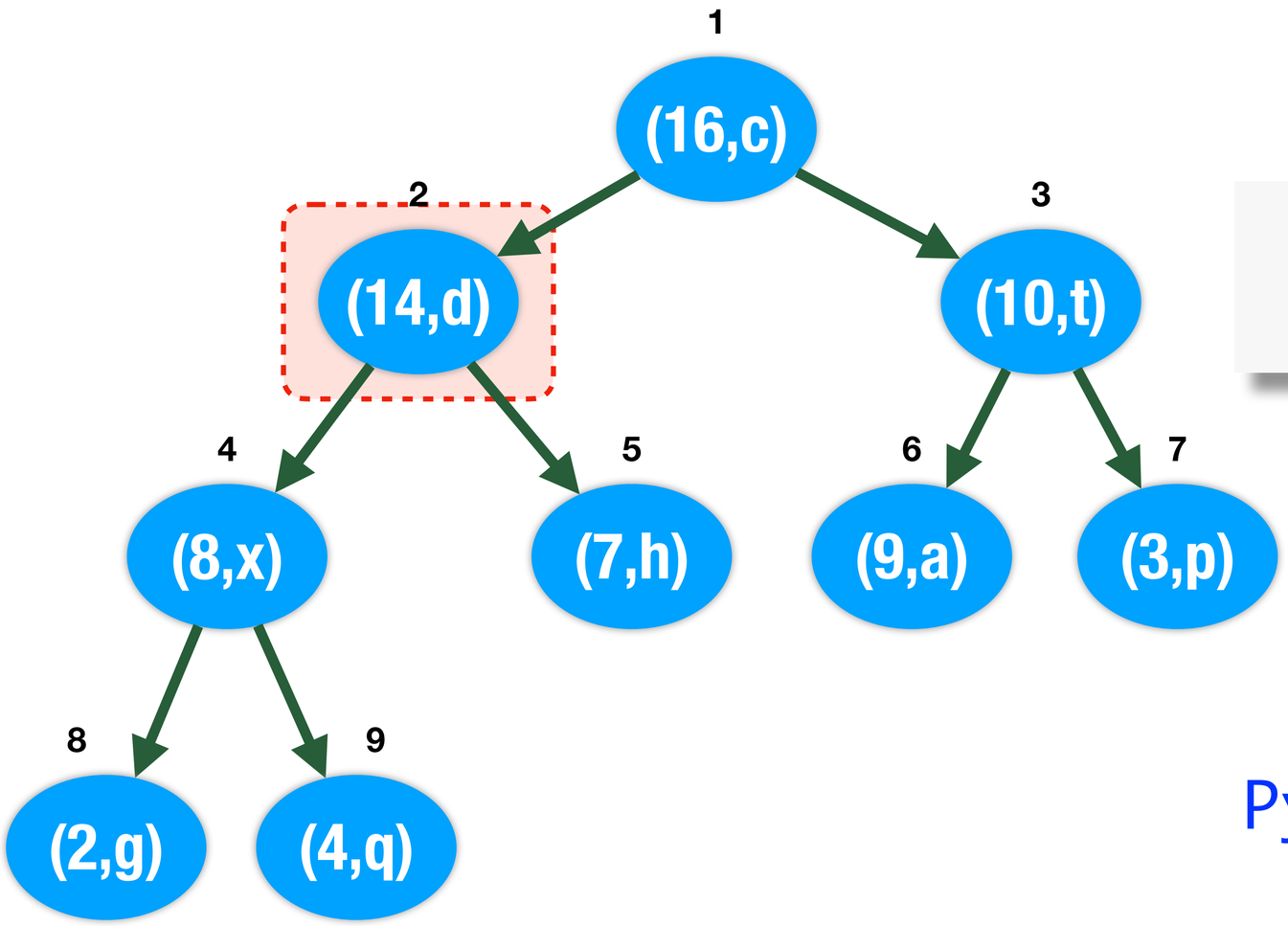
```
procedure Heap-Insert(A, key)
    heapsize[A] ← heapsize[A] + 1
    A[heapsize[A]] ← −∞          ** or any value
    Heap-Increase-key (A, heapsize[A], key)
```

PQ Operations: Summary

Python code: heap.py, priority_queue.py

- `Initialize`(A) — Build-Max-Heap. Takes $\Theta(n)$ time.
- `Maximum`(A) — Return $A[1]$. Takes $\Theta(1)$ time.
- `Extract-Maximum`(A) — Like deleting from an array: put $A[n]$ as the new first element before returning the max. The difference: we `Max-Heapify`$(A, 1)$ to make this array into a heap. $\Theta(\lg n)$ time.
- `Increase-Key`(A, i, new_key) — The inverse of `Max-Heapify`: Increase the priority value for $A[i]$ and bubble up to till max-heap property is restored. $\Theta(\lg n)$ time.
- `Insert`(A, new_key) — Add a new key with lowest priority, increase its priority to new_key. $\Theta(\lg n)$ time.

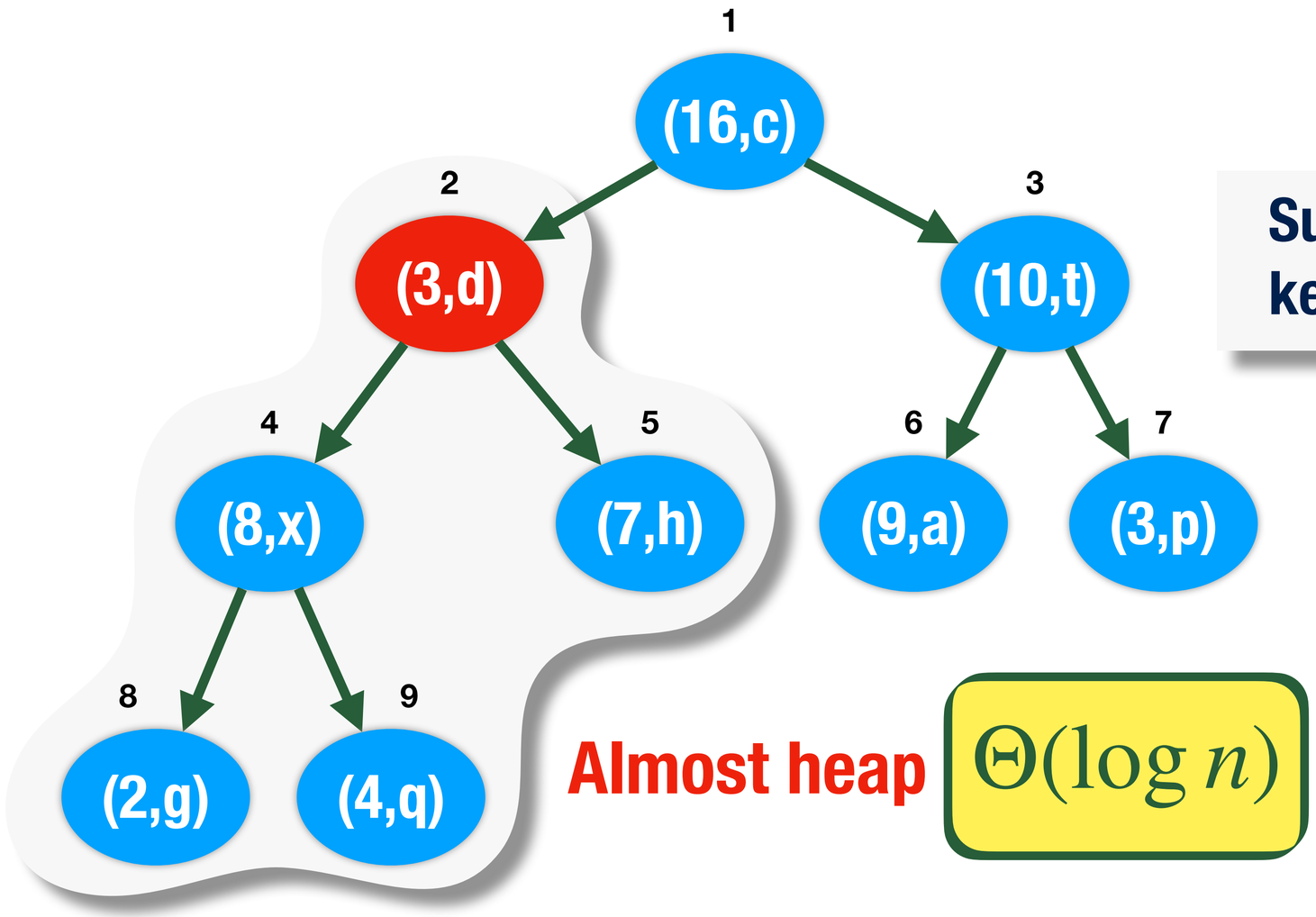
About Decrease Key



Suppose we want to decrease the key of $A[2]$ from 14 to 3

Python code: decrease_key in priority_queue.py

About Decrease Key

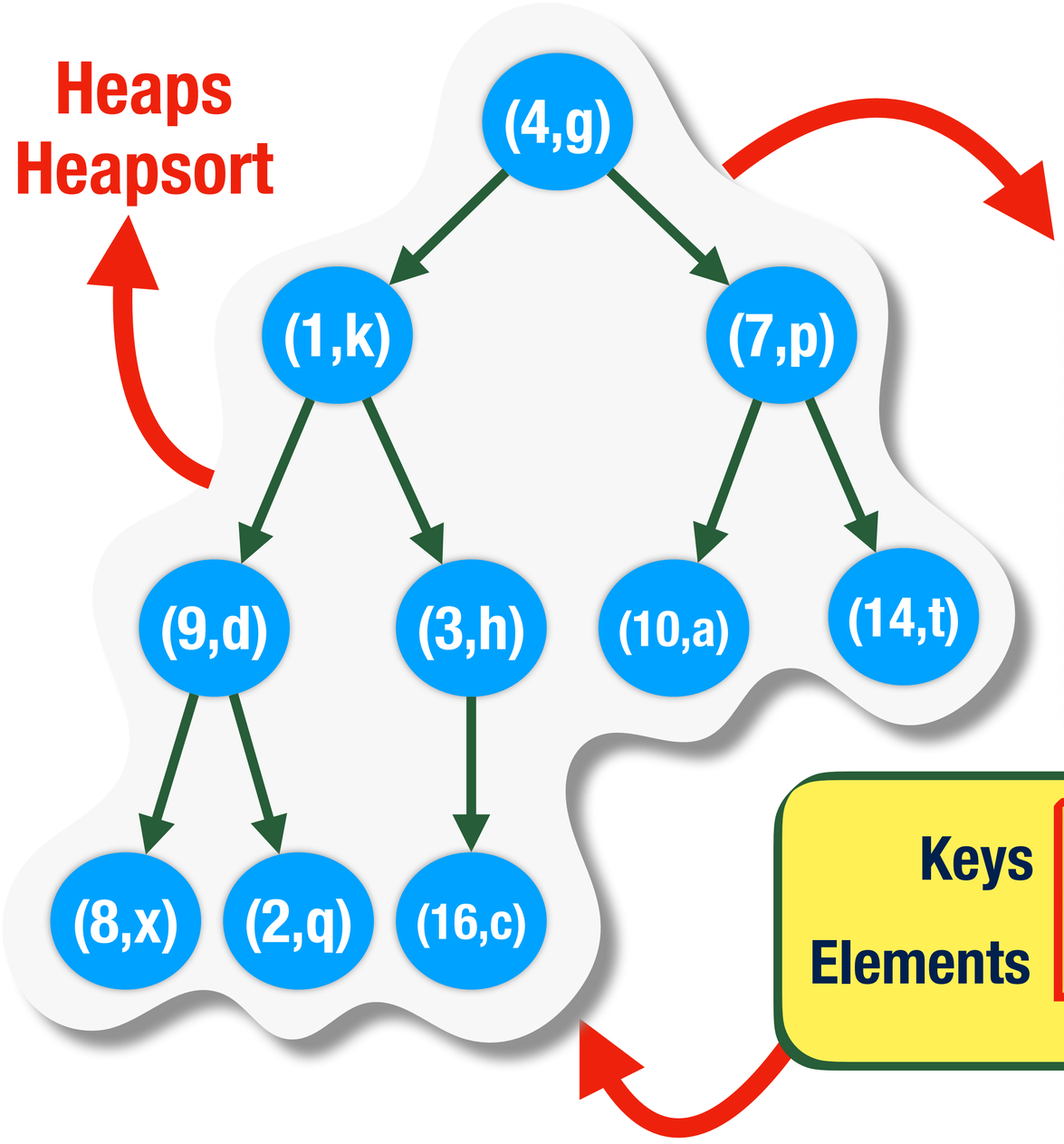


Suppose we want to decrease the key of $A[2]$ from 14 to 3

Almost heap $\Theta(\log n)$

```
procedure Max-Heapify(A, i)
      **turns almost-heap into a heap
      **pre-condition:  tree rooted at A[i] is an almost-heap
      **post-condition:  tree rooted at A[i] is a heap
   lc ← leftchild(i)
   rc ← rightchild(i)
   largest ← i
   if (lc ≤ heapsize(A) and A[lc] > A[largest]) then
      largest ← lc
   if (rc ≤ heapsize(A) and A[rc] > A[largest]) then
      largest ← rc          **largest = index of max{A[i], A[rc], A[lc]}
   if (largest ≠ i) then
      exchange A[i] ↔ A[largest]
      Max-Heapify(A, largest)
```

Heaps
Heapsort



PQ: Operations and RT analysis

- `Initialize`(A) — Build-Max-Heap. Takes $\Theta(n)$ time.
- `Maximum`(A) — Return $A[1]$. Takes $\Theta(1)$ time.
- `Extract-Maximum`(A) — Like deleting from an array: put $A[n]$ as the new first element before returning the max. The difference: we `Max-Heapify`$(A, 1)$ to make this array into a heap. $\Theta(\lg n)$ time.
- `Increase-Key`(A, i, new_key) — The inverse of `Max-Heapify`: Increase the priority value for $A[i]$ and bubble up to till max-heap property is restored. $\Theta(\lg n)$ time.
- `Insert`(A, new_key) — Add a new key with lowest priority, increase its priority to new_key. $\Theta(\lg n)$ time.

	1	2	3	4	5	6	7	8	9	10
Keys	4	1	7	9	3	10	14	8	2	16
Elements	g	k	p	d	h	a	t	x	q	c

Sorting Algorithms Summary (So Far)

First: Why Sorting?

Sorting: Given n elements, rearrange in ascending order

Obvious applications

- Sort names in eClass
- Organize a music library

Make things easier once sorted

- Find the median
- Find duplicate emails
- Binary search

Non-obvious applications

- Artificial intelligence
- Item recommendation on Amazon
- Supply chain management
- Cloud computing
- Energy sustainability

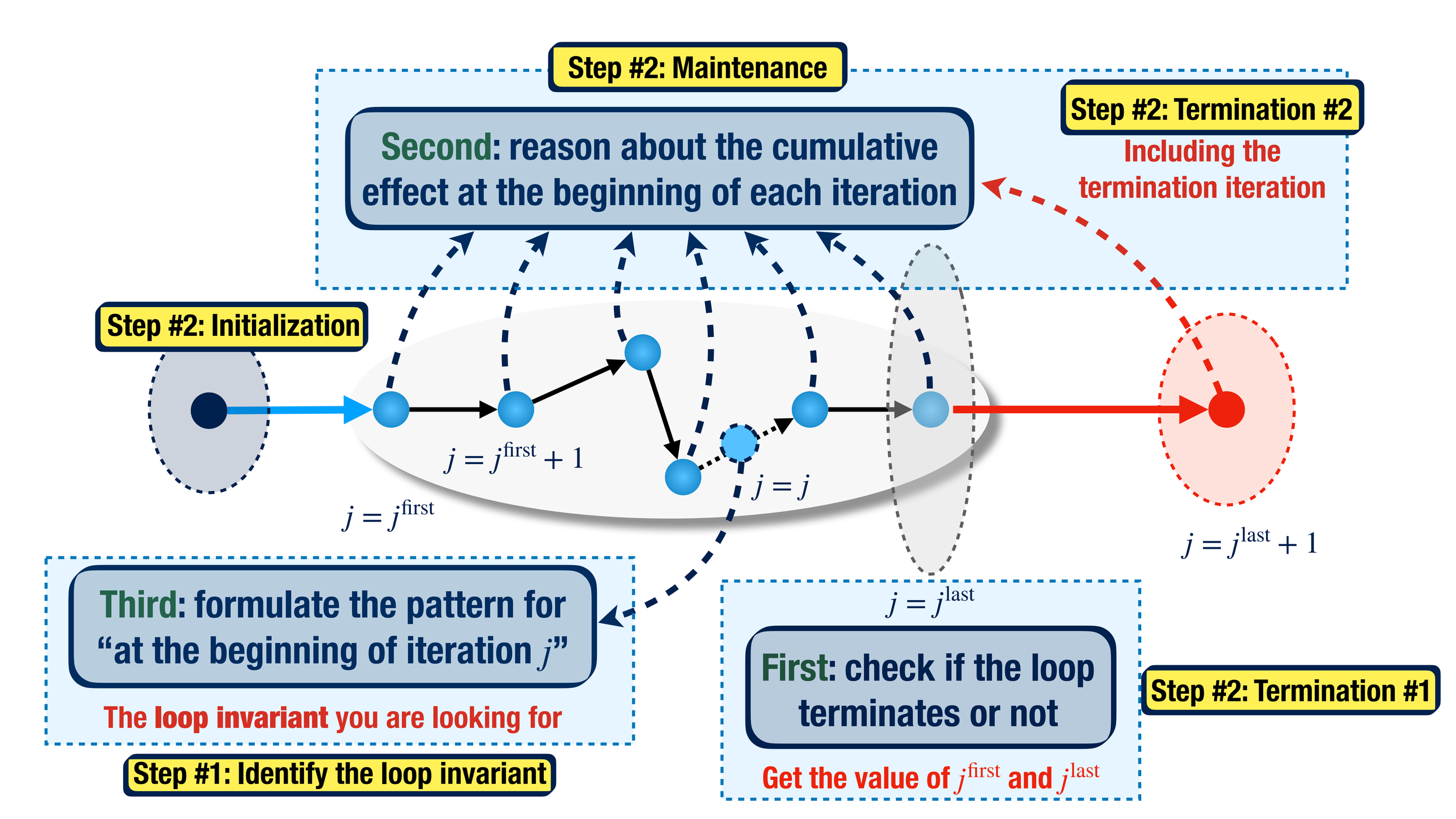
Sorting #1: Insertion Sort

```
InsertionSort(A)

for j = 2 to n do
    key = A[j]
    i = j − 1
    while (i > 0 and A[i] > key) do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

6 5 3 1 8 7 2 4

$\Theta(n^2)$

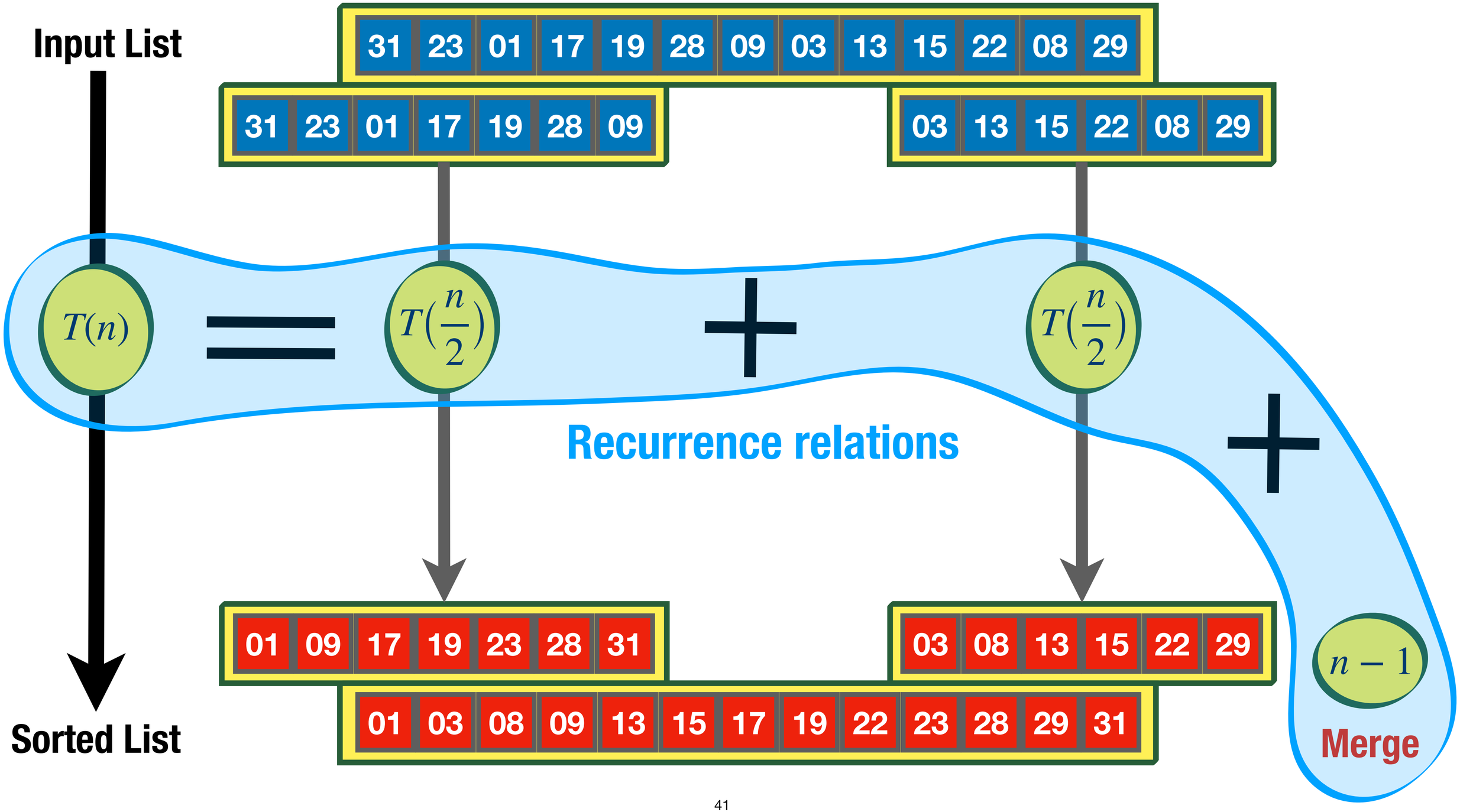

Step #2: Maintenance
Second: reason about the cumulative effect at the beginning of each iteration
Step #2: Termination #2
Including the termination iteration
Step #2: Initialization
$j = j^{\text{first}}$
$j = j^{\text{first}} + 1$
$j = j$
$j = j^{\text{last}} + 1$
$j = j^{\text{last}}$
Third: formulate the pattern for "at the beginning of iteration j"
The loop invariant you are looking for
Step #1: Identify the loop invariant
First: check if the loop terminates or not
Get the value of j^{first} and j^{last}
Step #2: Termination #1

Sorting #2: Merge Sort

```
Merge-Sort(A; lo, hi)
    if lo < hi then
        mid = ⌊(lo + hi)/2⌋
        Merge-Sort(A; lo, mid)
        Merge-Sort(A; mid + 1, hi)
        Merge(A; lo, mid, hi)
```

$\Theta(n \log n)$

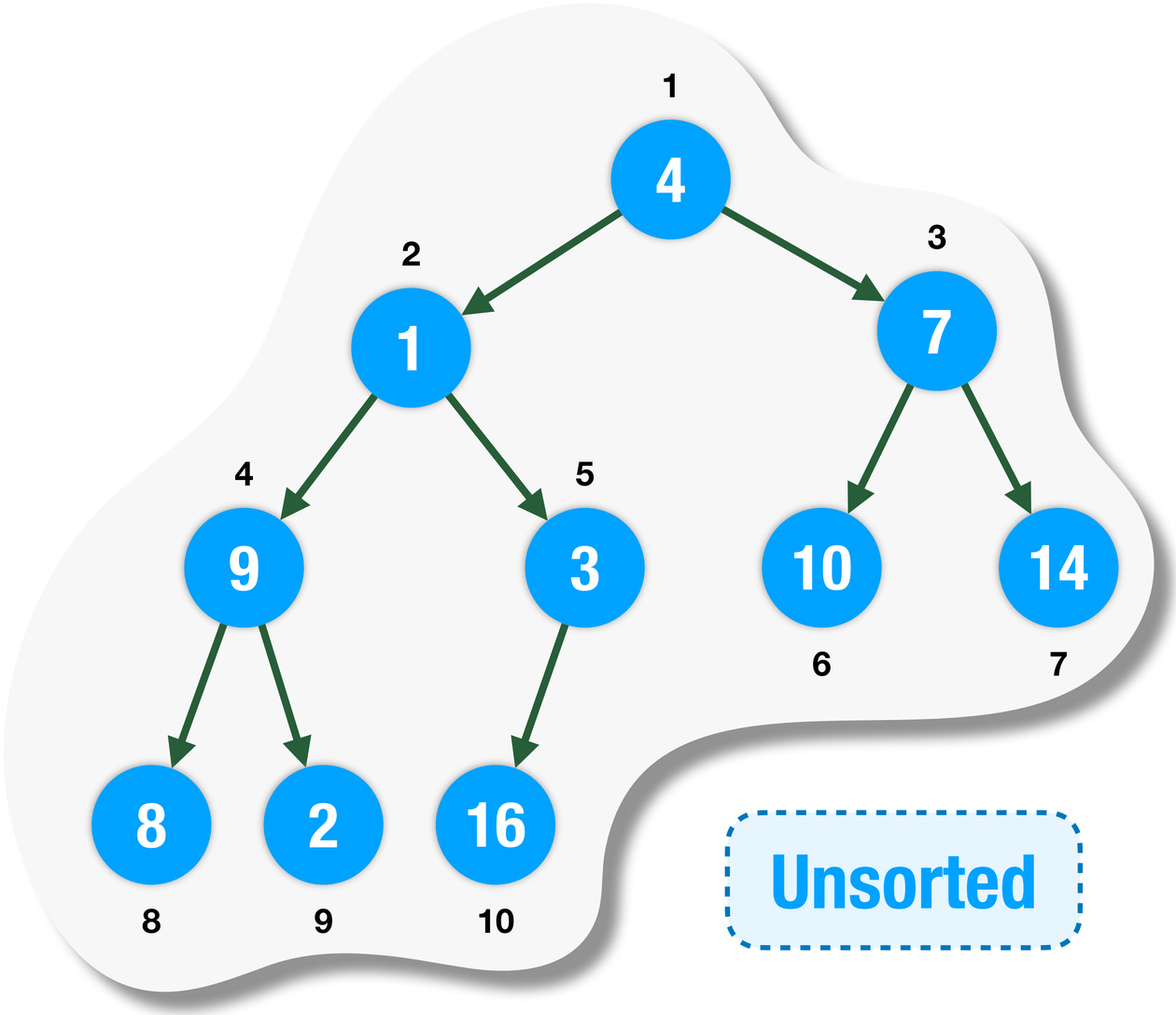
Design idea matters!!


Input List
31 23 01 17 19 28 09 03 13 15 22 08 29
31 23 01 17 19 28 09
03 13 15 22 08 29
$T(n)$
$= =$
$T(\frac{n}{2})$
$+$
$T(\frac{n}{2})$
Recurrence relations
$+$
$n-1$
Merge
01 09 17 19 23 28 31
03 08 13 15 22 29
01 03 08 09 13 15 17 19 22 23 28 29 31
Sorted List

Sorting #3: Heapsort

```
procedure Heapsort(A)
      **post-condition:  sorted array
   Build-Max-Heap(A)
   for (i ← heapsize(A) downto 2) do
      exchange A[1] ↔ A[i]
      heapsize(A) ← heapsize(A) − 1
      Max-Heapify(A, 1)
```

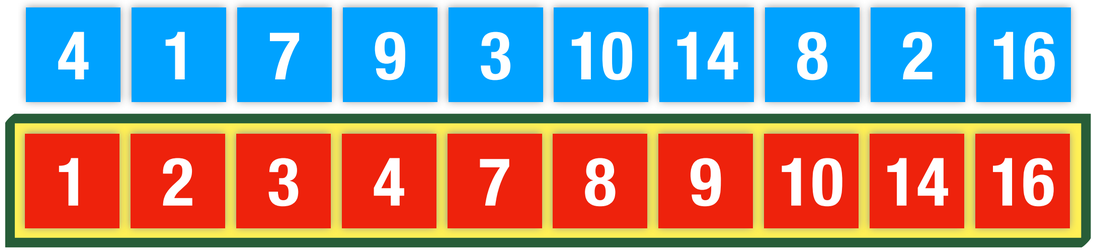
4	1	7	9	3	10	14	8	2	16
1	2	3	4	5	6	7	8	9	10

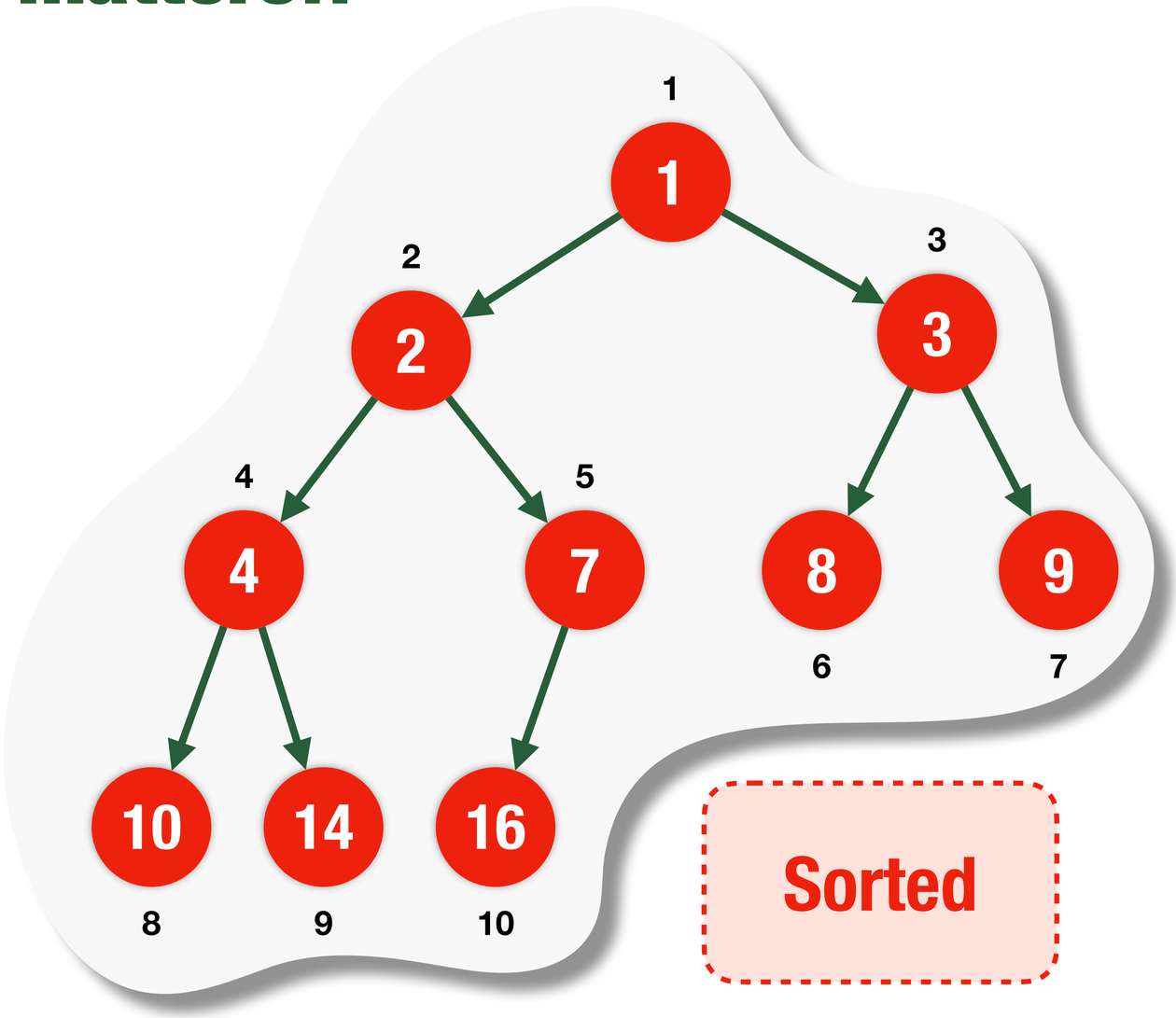


$\Theta(n \log n)$

Data structure matters!!

```
procedure Heapsort(A)
      **post-condition:  sorted array
   Build-Max-Heap(A)
   for (i ← heapsize(A) downto 2) do
      exchange A[1] ↔ A[i]
      heapsize(A) ← heapsize(A) − 1
      Max-Heapify(A, 1)
```





Sorted

Keeping Data Sorted - Static vs Dynamic Setting

- So far — we have considered the notion of a static problem: someone gives you an array of n items and we have to sort them.
- However, the problem can be studied also in the dynamic setting: the set of items changes, keys are added and removed (inserted and deleted), and every now and then, we wish to sort them (or find a value x among them).

- Option 1: use a data-structure that does insertion and deletion fast (array, list, hash-table), and upon a sorting request - run a sorting algorithm.
- Option 2: use a data-structure that keeps the elements sorted. (a binary search tree)

Martin's comment:
Priority queue is an interesting compromise
Keeps data partially sorted in heap
Insert and delete are pretty fast - O(log n), but not O(1)

Quicksort

Hash table

Binary search tree

Next Topics

Next Topics

Quicksort;

Binary Search Tree;

Hash Table