Algorithms: Elementary Data Structures

Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 7, 11.03.2025

RECALL LAST LECTURE

Heaps

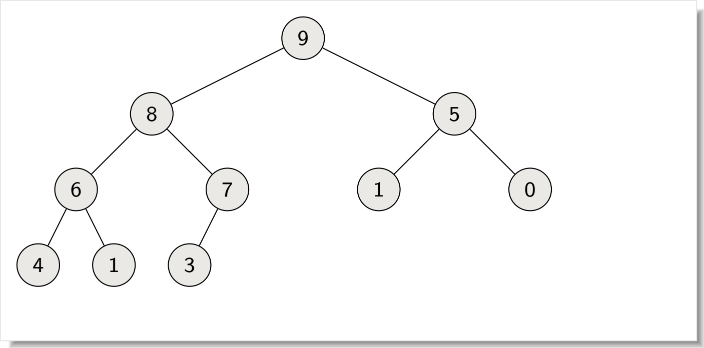
Heapsort

Priority Queues

(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

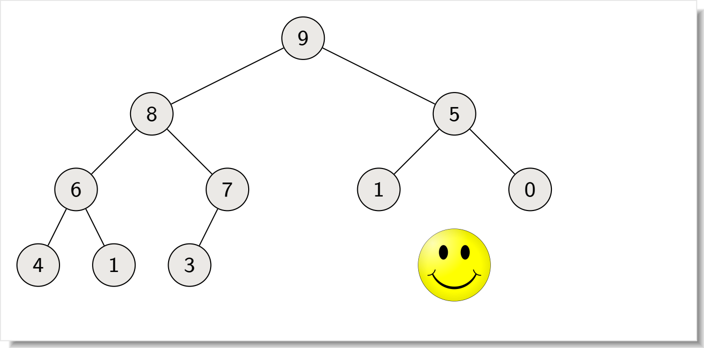
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

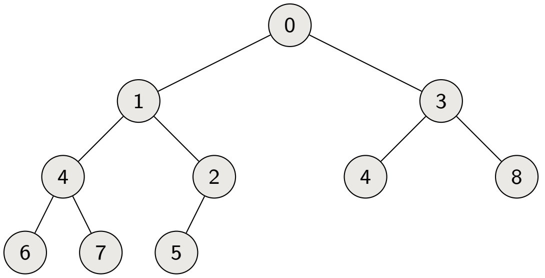
(Max)-Heap property: key of i's children is smaller or equal to i's key



(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

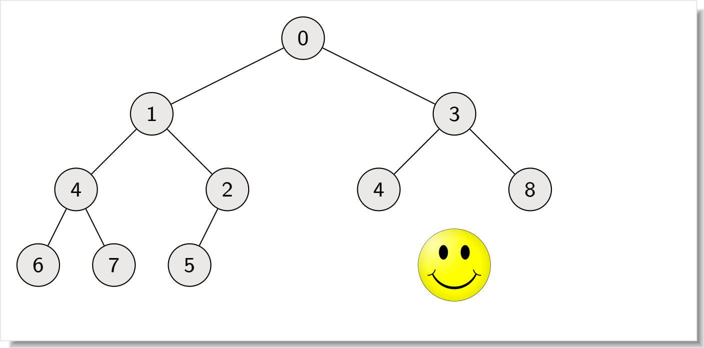
(Min)-Heap property: key of i's children is greater or equal to i's key



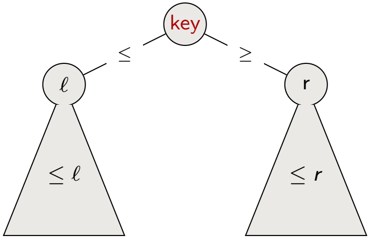
(Binary) heap data structure

Heap A (not garbage-collected storage) is a nearly complete binary tree

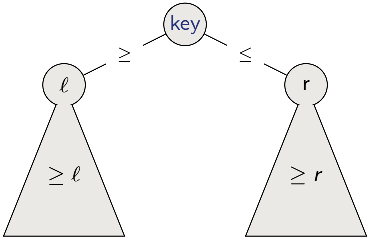
(Min)-Heap property: key of i's children is greater or equal to i's key



Max-Heap ⇒ maximum element is the root



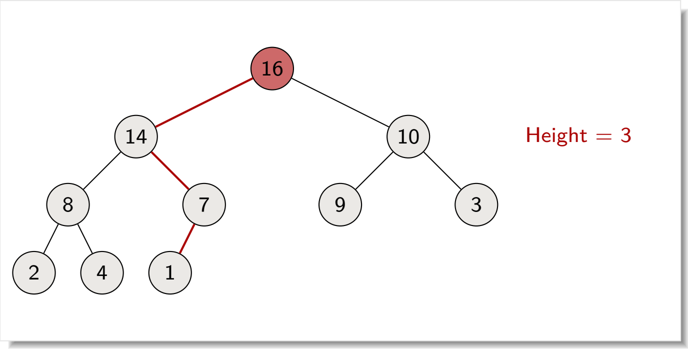
Min-Heap ⇒ minimum element is the root



Height of a heap

Height of node = # of edges on a longest simple path from the node down to a leaf

Height of heap = height of root = $\Theta(\log n)$



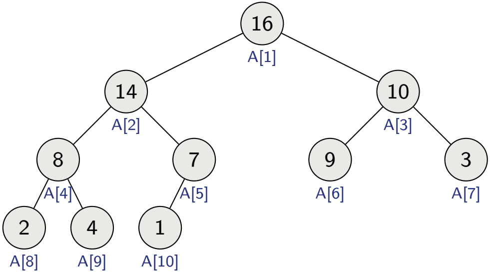
How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

A =

16	14	10	8	7	9	3	2	4	1



In this representation:

ROOT is $A[1]$

LEFT(i) =???

RIGHT(i) =???

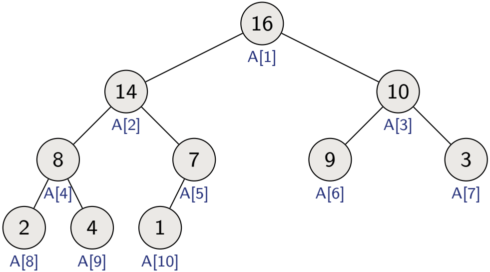
PARENT(i) =???

How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

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



In this representation:

ROOT is $A[1]$

LEFT$(i) = 2i$

RIGHT(i) =???

PARENT(i) =???

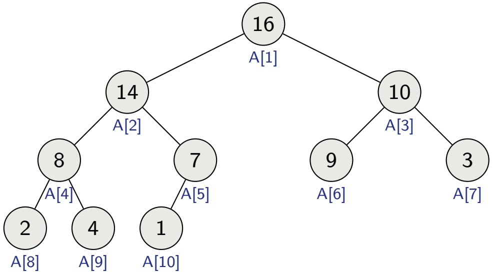
How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

A =

16	14	10	8	7	9	3	2	4	1



In this representation:

ROOT is $A[1]$

LEFT$(i) = 2i$

RIGHT$(i) = 2i + 1$

PARENT$(i) = ???$

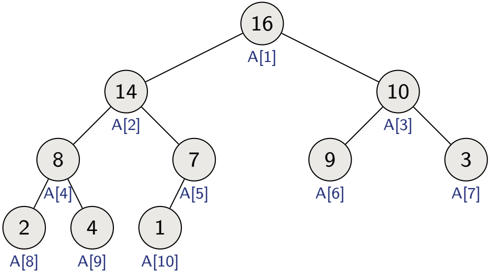
How to store a heap/tree?

~~pointer to left and right children~~

Use that tree is almost complete to store it in array

A =

16	14	10	8	7	9	3	2	4	1



In this representation:

ROOT is $A[1]$

$\text{LEFT}(i) = 2i$

$\text{RIGHT}(i) = 2i + 1$

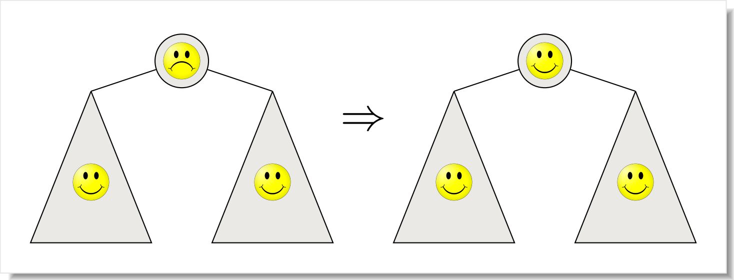
$\text{PARENT}(i) = \lfloor i/2 \rfloor$

BUILDING AND MANIPULATING HEAPS

Maintaining the heap property

MAX-HEAPIFY is important for manipulating heaps:

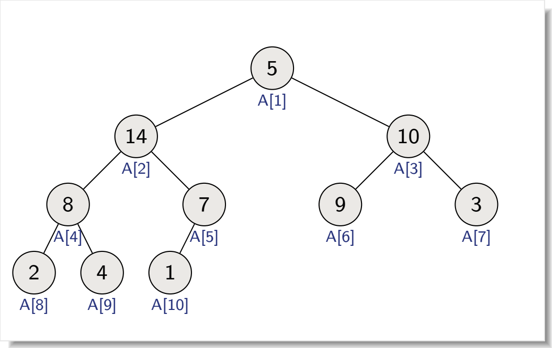
> Given an i such that the subtrees of i are heaps, it ensures that the subtree rooted at i is a heap satisfy the heap property



MAX-HEAPIFY(A, i, n)

Algorithm:

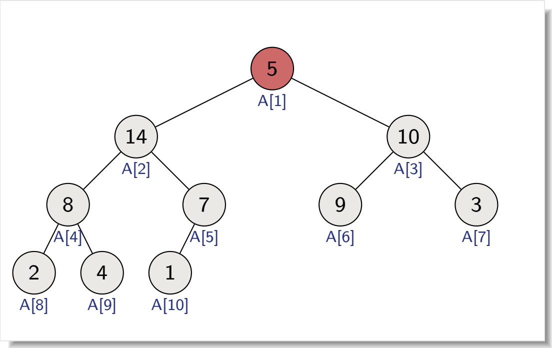
- Compare $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



MAX-HEAPIFY(A, i, n)

Algorithm:

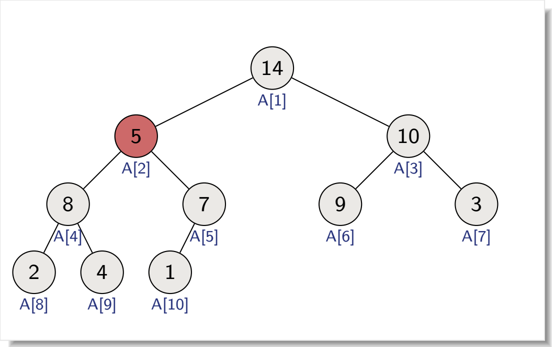
- Compare $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



MAX-HEAPIFY(A, i, n)

Algorithm:

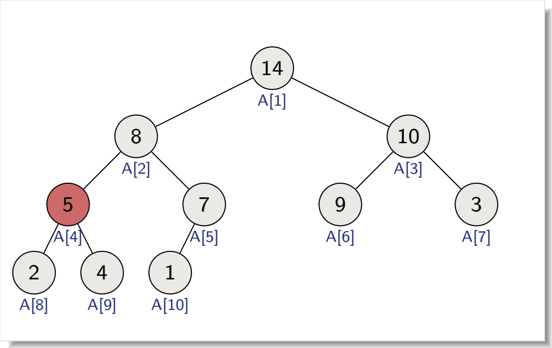
- Compare $A[i], A[\textsc{Left}(i)], A[\textsc{Right}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



MAX-HEAPIFY(A, i, n)

Algorithm:

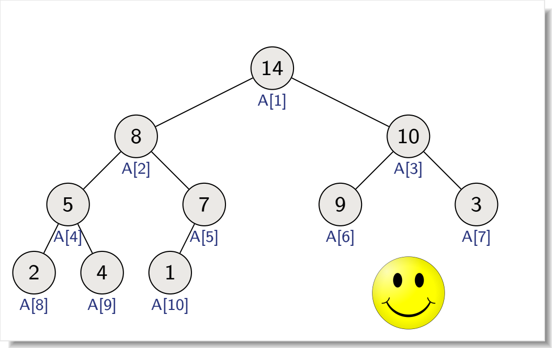
- Compare $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



Max-Heapify(A, i, n)

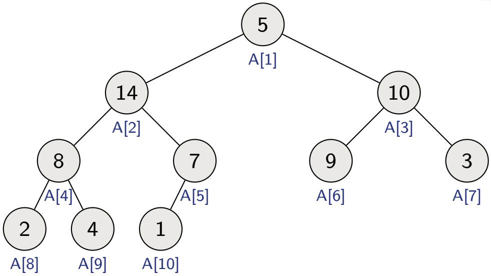
Algorithm:

- Compare $A[i], A[\textsc{Left}(i)], A[\textsc{Right}(i)]$
- If necessary, swap $A[i]$ with the largest of the two children to preserve heap property
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap



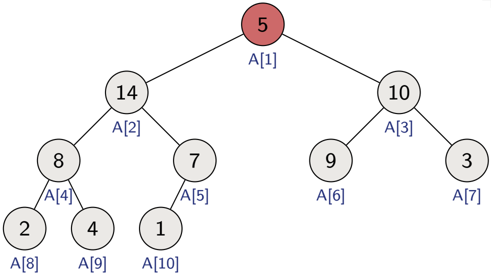
Pseudo-code and analysis

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```



Pseudo-code and analysis

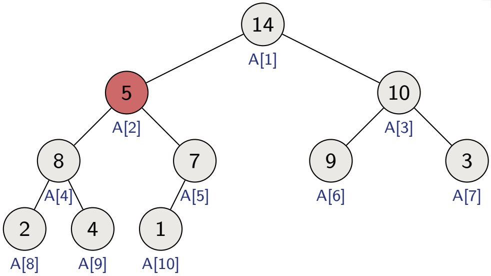
```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```



Pseudo-code and analysis

MAX-HEAPIFY(A, i, n)

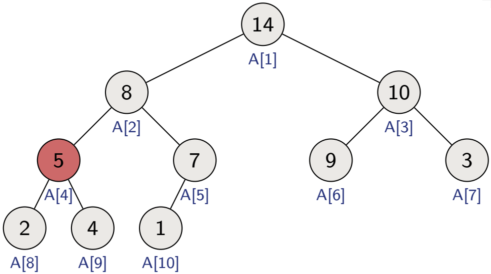
$l = $ LEFT(i)
$r = $ RIGHT(i)
if $l \leq n$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY$(A, largest, n)$



Pseudo-code and analysis

MAX-HEAPIFY(A, i, n)

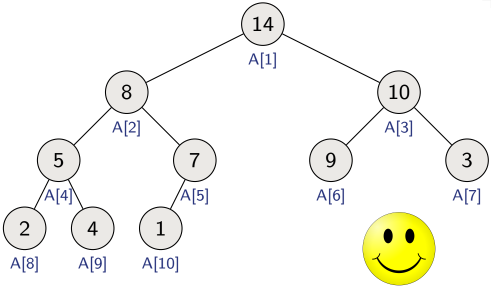
$l = $ LEFT(i)
$r = $ RIGHT(i)
if $l \leq n$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY$(A, largest, n)$



Pseudo-code and analysis

MAX-HEAPIFY(A, i, n)

$l = $ LEFT(i)
$r = $ RIGHT(i)
if $l \leq n$ and $A[l] > A[i]$
  $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
  $largest = r$
if $largest \neq i$
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY$(A, largest, n)$

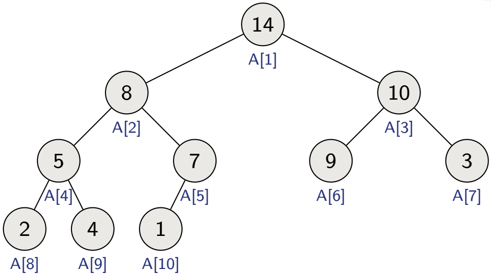


Pseudo-code and analysis

Running time?

Space?

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

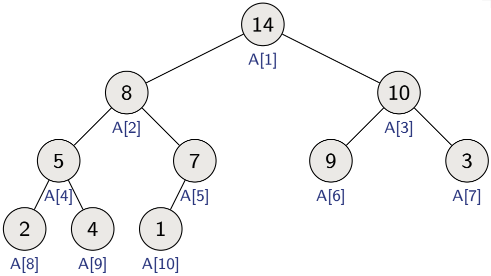


Pseudo-code and analysis

Running time?

$\Theta(\text{height of } i) = O(\log n)$

Space?



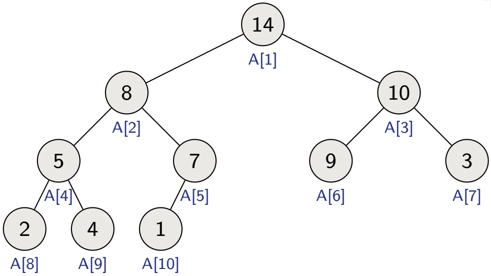

```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

Pseudo-code and analysis

Running time?

$\Theta(\text{height of } i) = O(\log n)$

Space? $\Theta(n)$




```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

Building a heap

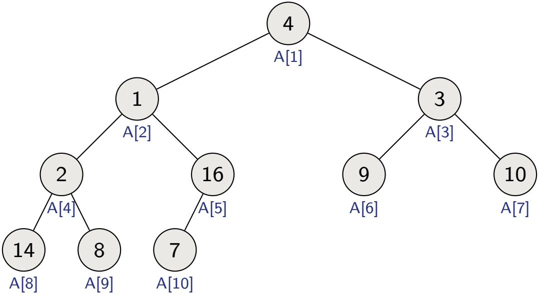
BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

  MAX-HEAPIFY(A, i, n)

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

4	1	3	2	16	9	10	14	8	7



Building a heap

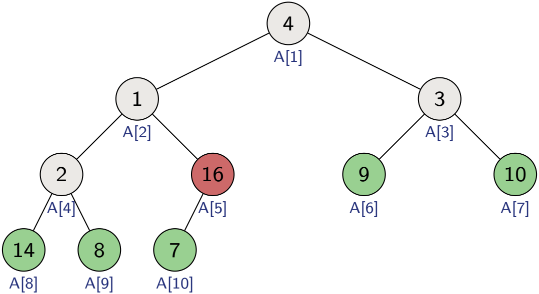
BUILD-MAX-HEAP(A, n)

  for $i = \lfloor n/2 \rfloor$ **downto** 1

      MAX-HEAPIFY(A, i, n)

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

4	1	3	2	16	9	10	14	8	7



Building a heap

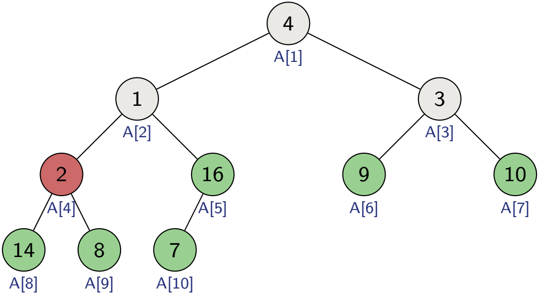
$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

  $\textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

4	1	3	2	16	9	10	14	8	7



Building a heap

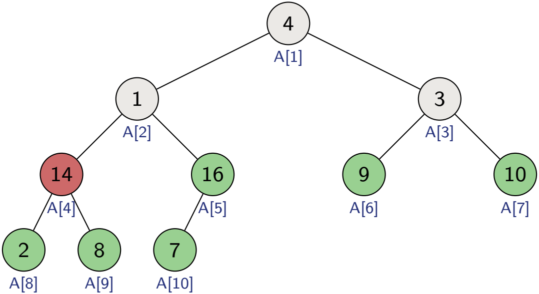
$\textsc{Build-Max-Heap}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

$\quad \textsc{Max-Heapify}(A, i, n)$

Given unordered array A of length n, $\textsc{Build-Max-Heap}$ outputs a heap

4	1	3	14	16	9	10	2	8	7

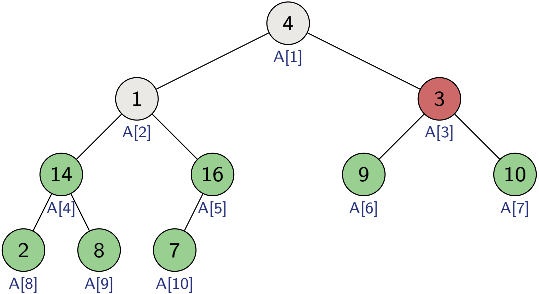


Building a heap

```
BUILD-MAX-HEAP(A, n)
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i, n)
```

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

4	1	3	14	16	9	10	2	8	7



Building a heap

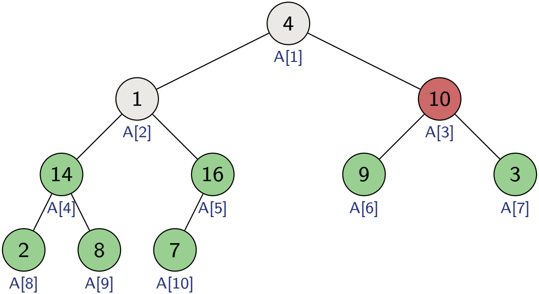
BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

4	1	10	14	16	9	3	2	8	7



Building a heap

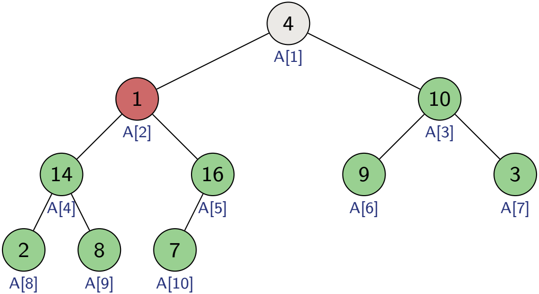
BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

4	1	10	14	16	9	3	2	8	7



Building a heap

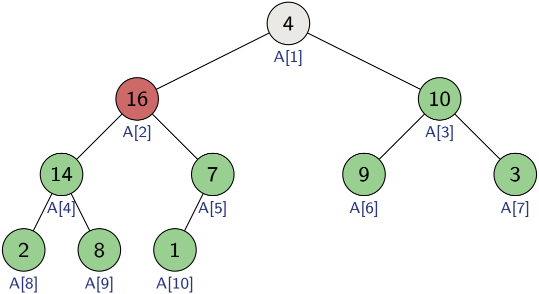
Build-Max-Heap(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

  Max-Heapify(A, i, n)

Given unordered array A of length n, Build-Max-Heap outputs a heap

4	16	10	14	7	9	3	2	8	1



Building a heap

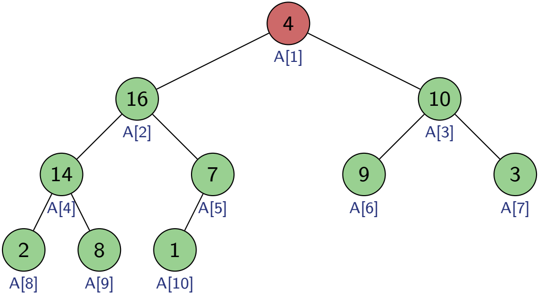
BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

4	16	10	14	7	9	3	2	8	1



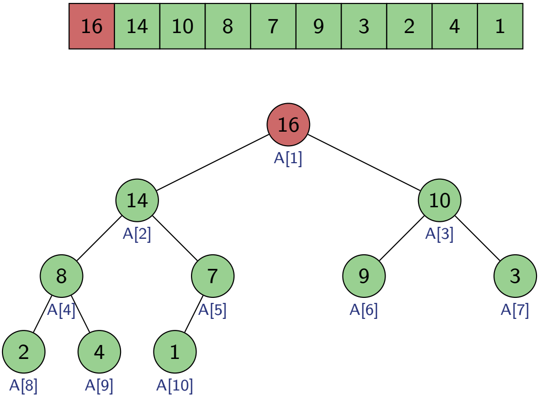
Building a heap

BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

  MAX-HEAPIFY(A, i, n)

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap



Building a heap

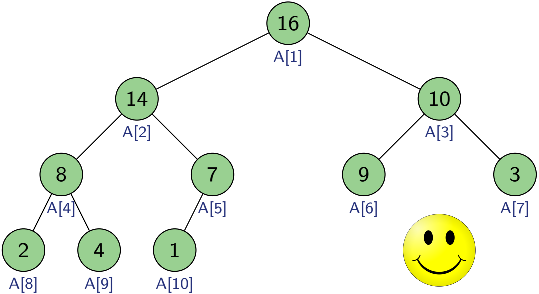
BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)

Given unordered array A of length n, BUILD-MAX-HEAP outputs a heap

16	14	10	8	7	9	3	2	4	1



Analysis

BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1
  MAX-HEAPIFY(A, i, n)

What is the worst-case running time of BUILD-MAX-HEAP?

Analysis

$\text{BUILD-MAX-HEAP}(A, n)$
for $i = \lfloor n/2 \rfloor$ **downto** 1
  $\text{MAX-HEAPIFY}(A, i, n)$

What is the worst-case running time of BUILD-MAX-HEAP?

Simple bound: $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$ in total

Analysis

$\text{BUILD-MAX-HEAP}(A, n)$
for $i = \lfloor n/2 \rfloor$ **downto** 1
    $\text{MAX-HEAPIFY}(A, i, n)$

What is the worst-case running time of BUILD-MAX-HEAP?

Simple bound: $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$ in total

Tighter analysis: Time to run MAX-HEAPIFY is linear in the height of the node it's run on. Hence, the time is bounded by

$$\sum_{h=0}^{\lg n} \{\#\text{ nodes of height } h\} O(h) = O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^h}\right),$$

which is $O(n)$ since $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$.

Correctness

$\text{BUILD-MAX-HEAP}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1

  $\text{MAX-HEAPIFY}(A, i, n)$

Correctness

```
BUILD-MAX-HEAP(A, n)
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i, n)
```

Loop invariant: At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap

Correctness

BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)

Loop invariant: At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap

Initialization:

- Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf which is the root of a trivial max-heap

Correctness

BUILD-MAX-HEAP(A, n)
for $i = \lfloor n/2 \rfloor$ **downto** 1
MAX-HEAPIFY(A, i, n)

Loop invariant: At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap

Initialization:

- Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf which is the root of a trivial max-heap
- Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true

Correctness

BUILD-MAX-HEAP(A, n)
for $i = \lfloor n/2 \rfloor$ **downto** 1
  MAX-HEAPIFY(A, i, n)

Loop invariant: At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap

Maintenance:

Correctness

$\text{BUILD-MAX-HEAP}(A, n)$

for $i = \lfloor n/2 \rfloor$ **downto** 1
  $\text{MAX-HEAPIFY}(A, i, n)$

Loop invariant: At start of every iteration of **for** loop, each node $i + 1, i + 2, \ldots, n$ is root of a max-heap

Maintenance:

- Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps

Correctness

$$\text{BUILD-MAX-HEAP}(A, n)$$
$$\textbf{for } i = \lfloor n/2 \rfloor \textbf{ downto } 1$$
$$\quad \text{MAX-HEAPIFY}(A, i, n)$$

Loop invariant: At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap

Maintenance:

- Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps
- Therefore, MAX-HEAPIFY makes node i a max-heap root (so $i, i+1, \ldots n$ are all roots of max-heaps)

Correctness

BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

  MAX-HEAPIFY(A, i, n)

Loop invariant: At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap

Maintenance:

- Children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps
- Therefore, MAX-HEAPIFY makes node i a max-heap root (so $i, i+1, \ldots n$ are all roots of max-heaps)
- Hence, the invariant stays true when decrementing i at the beginning of the next iteration

Correctness

BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)

Loop invariant: At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap

Termination:

Correctness

BUILD-MAX-HEAP(A, n)
for $i = \lfloor n/2 \rfloor$ **downto** 1
MAX-HEAPIFY(A, i, n)

Loop invariant: At start of every iteration of **for** loop, each node $i + 1, i + 2, \ldots, n$ is root of a max-heap

Termination:

- When $i = 0$, the loop terminates

Correctness

BUILD-MAX-HEAP(A, n)
for $i = \lfloor n/2 \rfloor$ **downto** 1
  MAX-HEAPIFY(A, i, n)

Loop invariant: At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap

Termination:

- When $i = 0$, the loop terminates
- By the loop invariant , each node, notably node 1, is the root of a max-heap

HEAPSORT

The heapsort algorithm

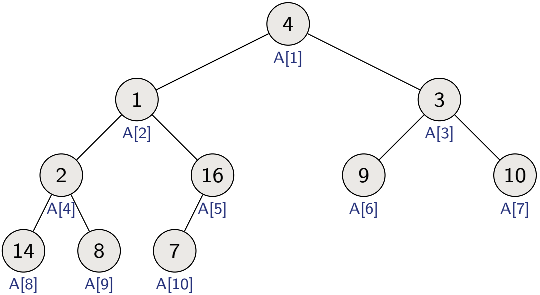
- Builds a max-heap from the array
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array
- "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root
- Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array

Example

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i-1)$

4	1	3	2	16	9	10	14	8	7



Example

HEAPSORT(A, n)

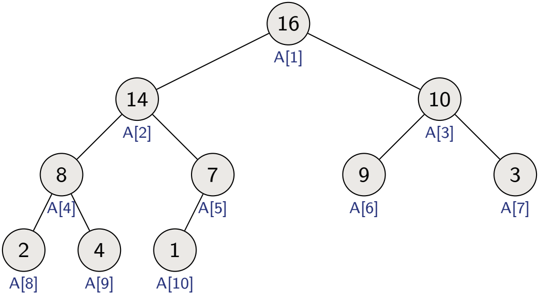
BUILD-MAX-HEAP(A, n)

for $i = n$ **downto** 2

exchange $A[1]$ with $A[i]$

MAX-HEAPIFY$(A, 1, i-1)$

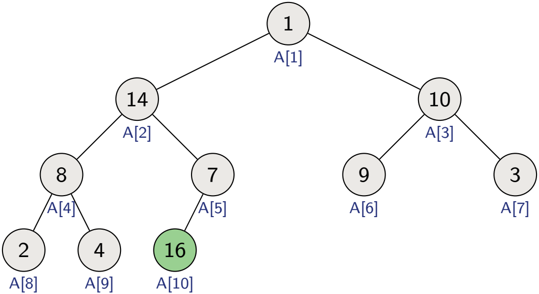
16	14	10	8	7	9	3	2	4	1



Example

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

1	14	10	8	7	9	3	2	4	16

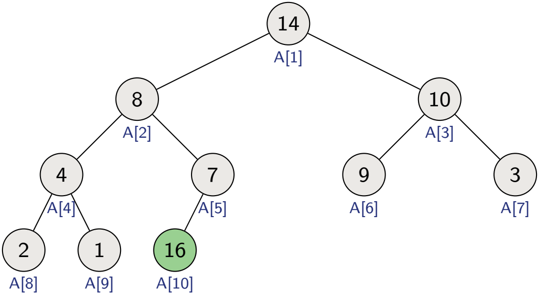


Example

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
exchange $A[1]$ with $A[i]$
MAX-HEAPIFY$(A, 1, i - 1)$

14	8	10	4	7	9	3	2	1	16



Example

HEAPSORT(A, n)

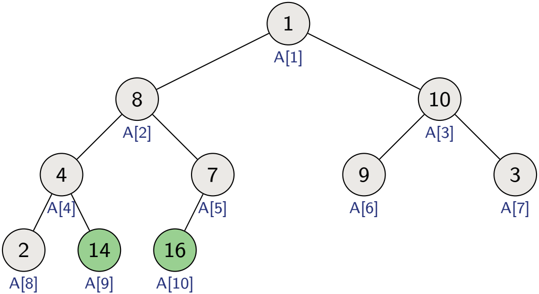
BUILD-MAX-HEAP(A, n)

for $i = n$ **downto** 2

exchange $A[1]$ with $A[i]$

MAX-HEAPIFY$(A, 1, i-1)$

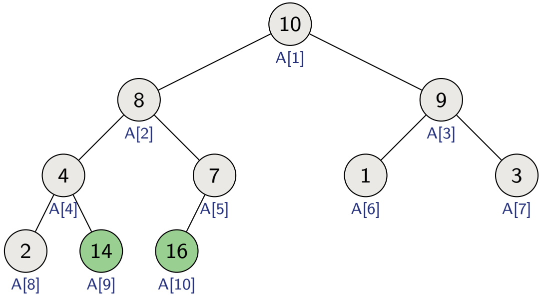
1	8	10	4	7	9	3	2	14	16



Example

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

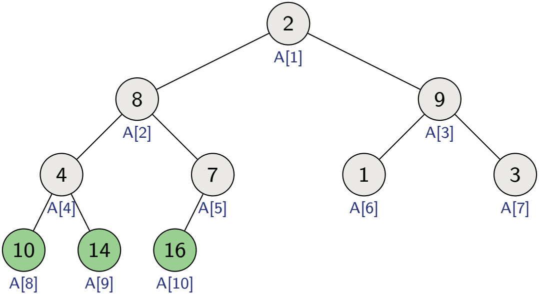
10	8	9	4	7	1	3	2	14	16



Example

HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i - 1)$

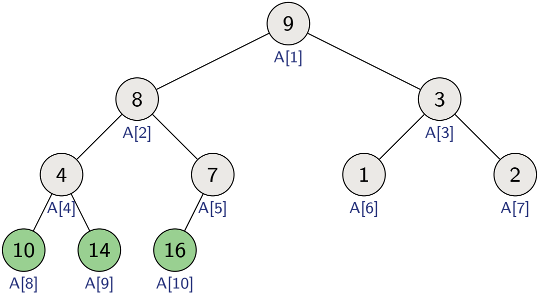
2	8	9	4	7	1	3	10	14	16



Example

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

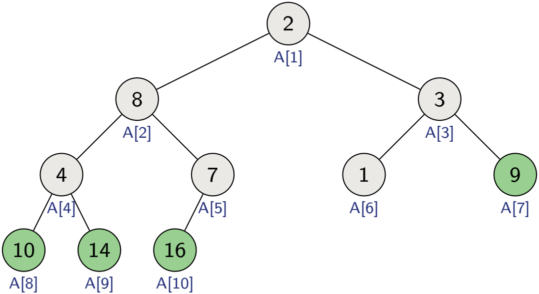
9	8	3	4	7	1	2	10	14	16



Example

HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i-1)$

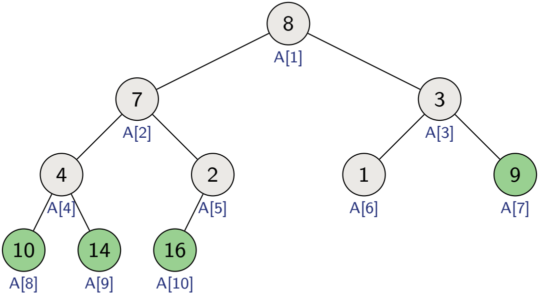
2	8	3	4	7	1	9	10	14	16



Example

HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i-1)$

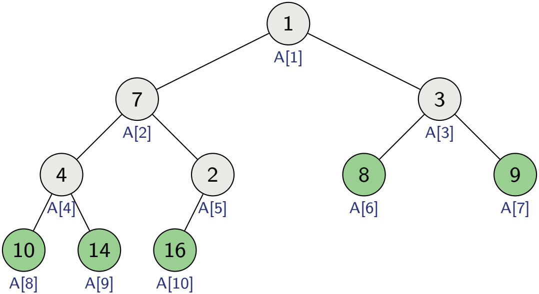
8	7	3	4	2	1	9	10	14	16



Example

HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i-1)$

1	7	3	4	2	8	9	10	14	16



Example

HEAPSORT(A, n)

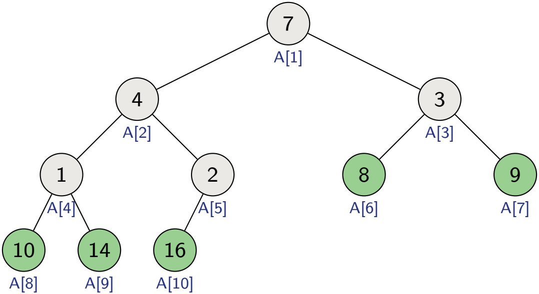
BUILD-MAX-HEAP(A, n)

for $i = n$ **downto** 2

exchange $A[1]$ with $A[i]$

MAX-HEAPIFY$(A, 1, i-1)$

7	4	3	1	2	8	9	10	14	16

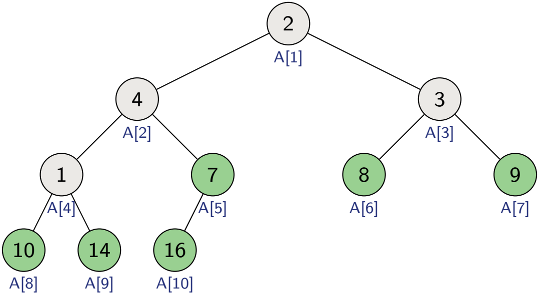


Example

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
exchange $A[1]$ with $A[i]$
MAX-HEAPIFY$(A, 1, i - 1)$

2	4	3	1	7	8	9	10	14	16

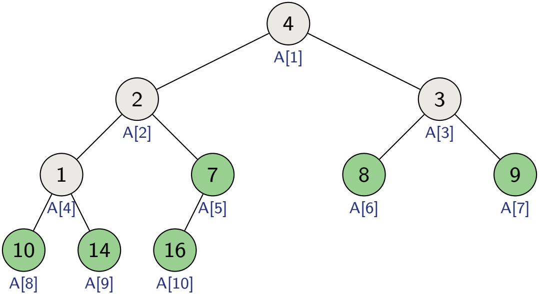


Example

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
exchange $A[1]$ with $A[i]$
MAX-HEAPIFY$(A, 1, i - 1)$

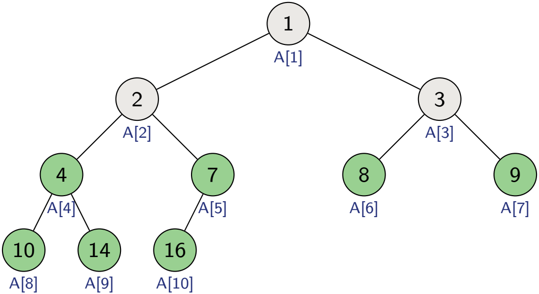
4	2	3	1	7	8	9	10	14	16



Example

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

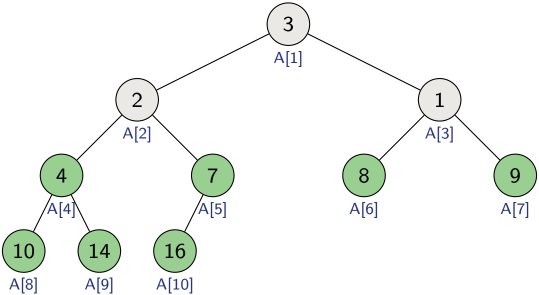
1	2	3	4	7	8	9	10	14	16



Example

HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i-1)$

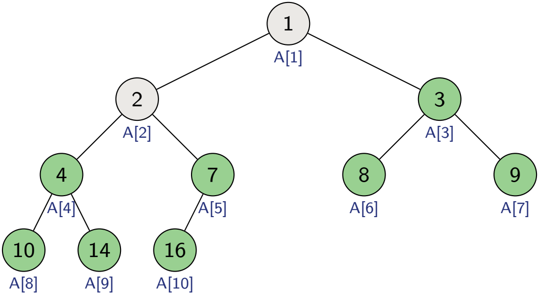
3	2	1	4	7	8	9	10	14	16



Example

HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i-1)$

1	2	3	4	7	8	9	10	14	16



Example

HEAPSORT(A, n)

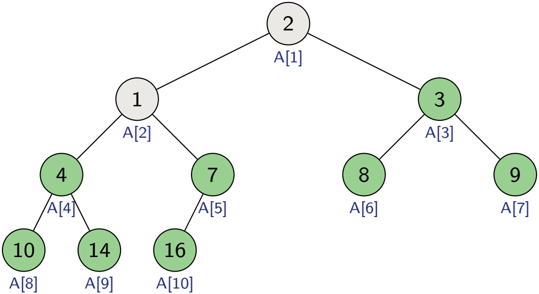
BUILD-MAX-HEAP(A, n)

for $i = n$ **downto** 2

exchange $A[1]$ with $A[i]$

MAX-HEAPIFY$(A, 1, i - 1)$

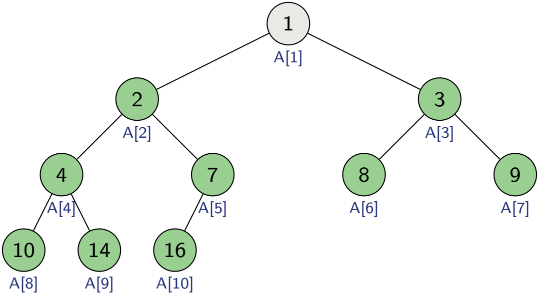
2	1	3	4	7	8	9	10	14	16



Example

HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
exchange $A[1]$ with $A[i]$
MAX-HEAPIFY$(A, 1, i-1)$

1	2	3	4	7	8	9	10	14	16

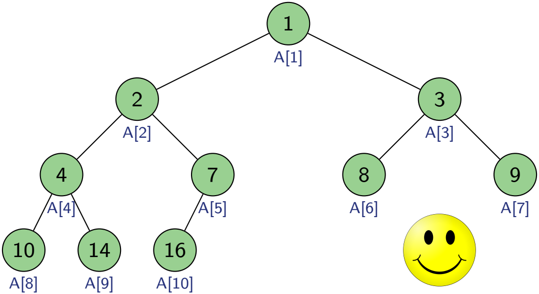


Example

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i = n$ **downto** 2
  exchange $A[1]$ with $A[i]$
  MAX-HEAPIFY$(A, 1, i-1)$

1	2	3	4	7	8	9	10	14	16



Analysis of Heapsort

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i - 1)
```

Analysis of Heapsort

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

- BUILD-MAX-HEAP: $O(n)$

Analysis of Heapsort

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i - 1)
```

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times

Analysis of Heapsort

HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    MAX-HEAPIFY$(A, 1, i - 1)$

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$

Analysis of Heapsort

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

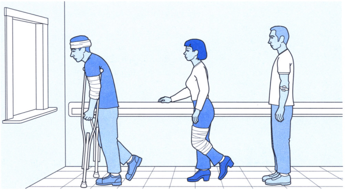
Analysis of Heapsort

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i - 1)
```

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$

HEAP IMPLEMENTATION OF PRIORITY QUEUE



Priority Queue

- Maintains a dynamic set S of elements
- Each set element has a key — an associated value that regulates its importance

Priority Queue

- Maintains a dynamic set S of elements
- Each set element has a key — an associated value that regulates its importance

What kind of operations do we want to do?

INSERT(S,x): inserts element x into S

Priority Queue

- Maintains a dynamic set S of elements
- Each set element has a key — an associated value that regulates its importance

What kind of operations do we want to do?

INSERT(S,x): inserts element x into S

MAXIMUM(S): returns element of S with largest key

Priority Queue

- Maintains a dynamic set S of elements
- Each set element has a key — an associated value that regulates its importance

What kind of operations do we want to do?

INSERT(S,x): inserts element x into S

MAXIMUM(S): returns element of S with largest key

EXTRACT-MAX(S): removes and returns element of S with largest key

Priority Queue

- Maintains a dynamic set S of elements
- Each set element has a key — an associated value that regulates its importance

What kind of operations do we want to do?

INSERT(S,x): inserts element x into S

MAXIMUM(S): returns element of S with largest key

EXTRACT-MAX(S): removes and returns element of S with largest key

INCREASE-KEY(S,x,k): increases value of element x's key to k; assume $k \geq x$'s current key value

Priority Queue

- Maintains a dynamic set S of elements
- Each set element has a key — an associated value that regulates its importance

What kind of operations do we want to do?

INSERT(S,x): inserts element x into S

MAXIMUM(S): returns element of S with largest key

EXTRACT-MAX(S): removes and returns element of S with largest key

INCREASE-KEY(S,x,k): increases value of element x's key to k; assume $k \geq x$'s current key value

Example max-priority queue application: schedule jobs on shared computer

Priority Queue

- Maintains a dynamic set S of elements
- Each set element has a key — an associated value that regulates its importance

What kind of operations do we want to do?

INSERT(S,x): inserts element x into S

MAXIMUM(S): returns element of S with largest key

EXTRACT-MAX(S): removes and returns element of S with largest key

INCREASE-KEY(S,x,k): increases value of element x's key to k; assume $k \geq x$'s current key value

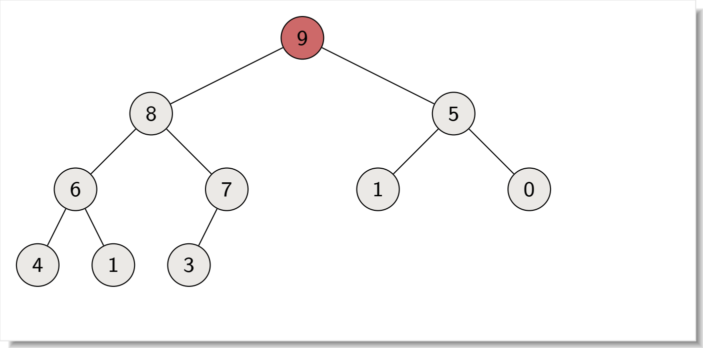
Example max-priority queue application: schedule jobs on shared computer

Heaps efficiently implement priority queues

Finding maximum element

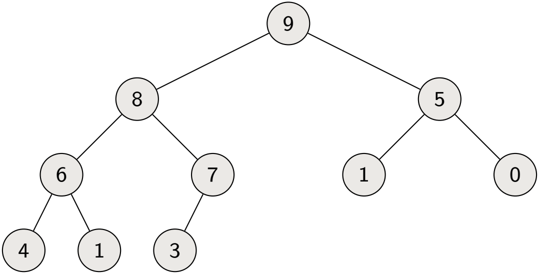
HEAP-MAXIMUM(A)
    return $A[1]$

Simply return the root in time $\Theta(1)$



Extracting maximum element

HEAP-EXTRACT-MAX(A, n)
if $n < 1$
  error "heap underflow"
$max = A[1]$
$A[1] = A[n]$
$n = n - 1$
MAX-HEAPIFY$(A, 1, n)$
return max

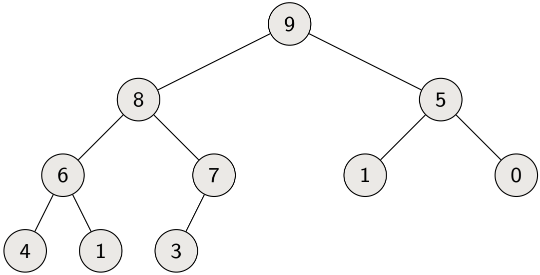


Extracting maximum element

1. Make sure heap is not empty

HEAP-EXTRACT-MAX(A, n)

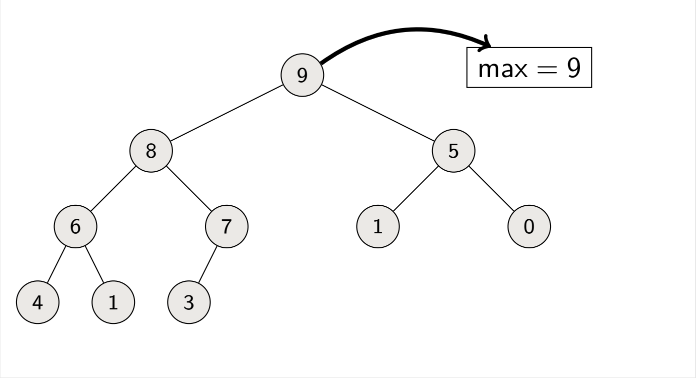
if $n < 1$
    error "heap underflow"
max $=$ $A[1]$
$A[1] = A[n]$
$n = n - 1$
MAX-HEAPIFY$(A, 1, n)$
return *max*



Extracting maximum element

1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)

```
HEAP-EXTRACT-MAX(A, n)
  if n < 1
      error "heap underflow"
  max = A[1]
  A[1] = A[n]
  n = n - 1
  MAX-HEAPIFY(A, 1, n)
  return max
```



Extracting maximum element

1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)
3. Make the last node in the tree the new root

HEAP-EXTRACT-MAX(A, n)

if $n < 1$

  error "heap underflow"

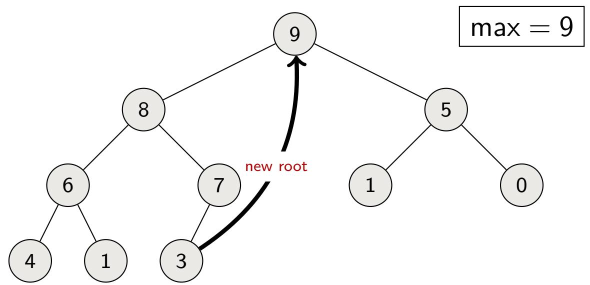
$max = A[1]$

$A[1] = A[n]$

$n = n - 1$

MAX-HEAPIFY$(A, 1, n)$

return *max*



Extracting maximum element

1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)
3. Make the last node in the tree the new root

HEAP-EXTRACT-MAX(A, n)

  if $n < 1$

      error "heap underflow"

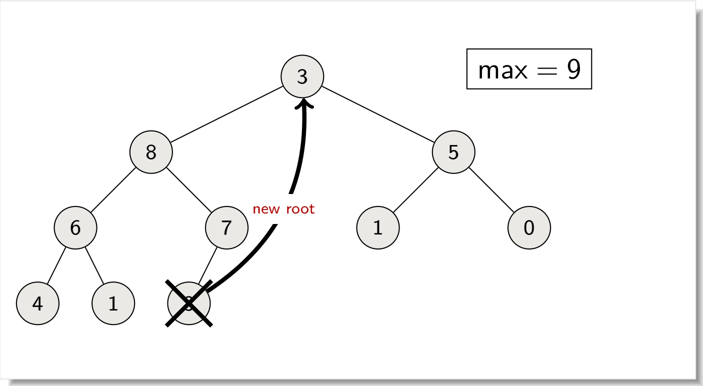
  $max = A[1]$

  $A[1] = A[n]$

  $n = n - 1$

  MAX-HEAPIFY$(A, 1, n)$

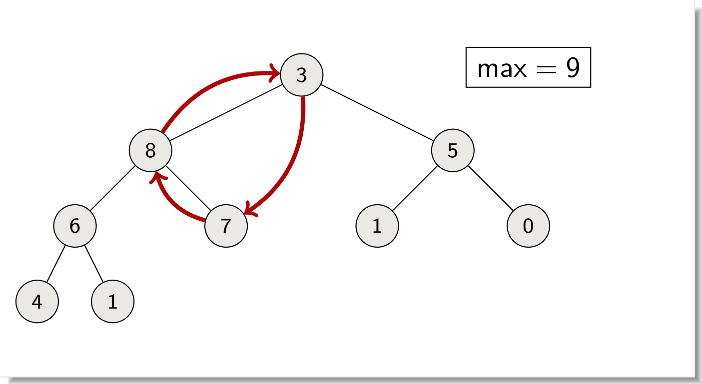
  return max



Extracting maximum element

1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)
3. Make the last node in the tree the new root
4. Re-heapify the heap, with one fewer node

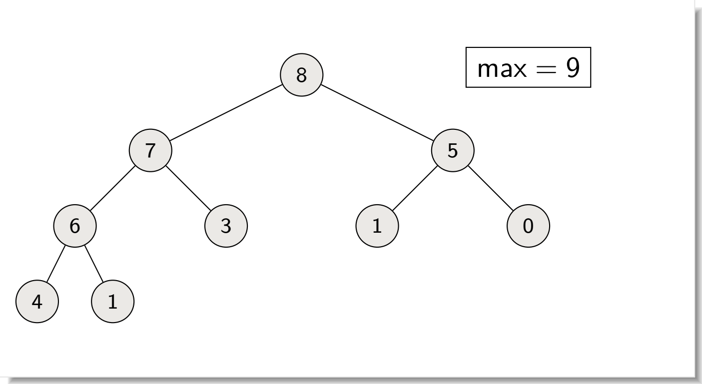
```
HEAP-EXTRACT-MAX(A, n)
  if n < 1
      error "heap underflow"
  max = A[1]
  A[1] = A[n]
  n = n - 1
  MAX-HEAPIFY(A, 1, n)
  return max
```



Extracting maximum element

1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)
3. Make the last node in the tree the new root
4. Re-heapify the heap, with one fewer node

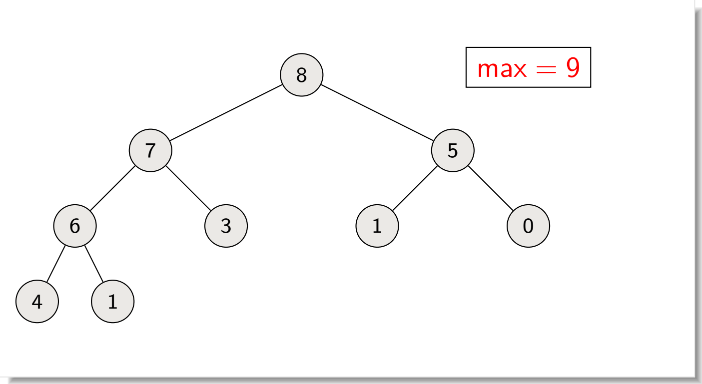
HEAP-EXTRACT-MAX(A, n)
  if $n < 1$
    error "heap underflow"
  $max = A[1]$
  $A[1] = A[n]$
  $n = n - 1$
  MAX-HEAPIFY$(A, 1, n)$
  return max



Extracting maximum element

1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)
3. Make the last node in the tree the new root
4. Re-heapify the heap, with one fewer node
5. Return the copy of the maximum element

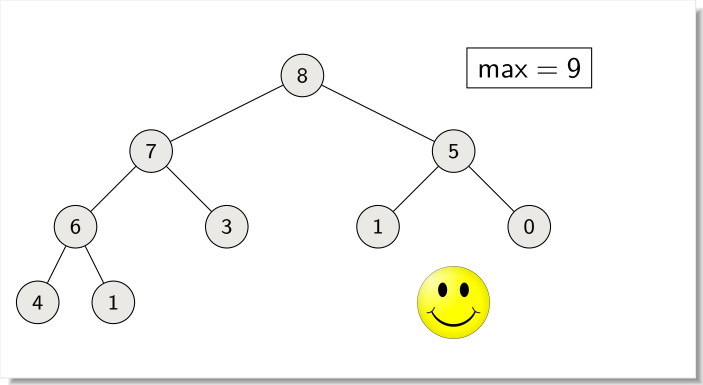
HEAP-EXTRACT-MAX(A, n)
  if $n < 1$
    error "heap underflow"
  $max = A[1]$
  $A[1] = A[n]$
  $n = n - 1$
  MAX-HEAPIFY$(A, 1, n)$
  return max



Extracting maximum element

1. Make sure heap is not empty
2. Make a copy of the maximum element (the root)
3. Make the last node in the tree the new root
4. Re-heapify the heap, with one fewer node
5. Return the copy of the maximum element

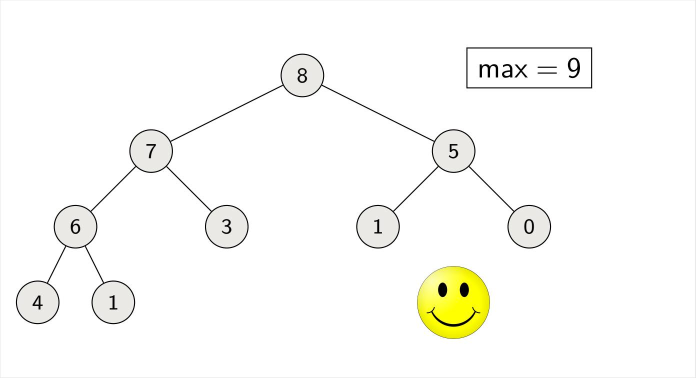
```
HEAP-EXTRACT-MAX(A, n)
  if n < 1
      error "heap underflow"
  max = A[1]
  A[1] = A[n]
  n = n - 1
  MAX-HEAPIFY(A, 1, n)
  return max
```



Extracting maximum element

Analysis:

```
HEAP-EXTRACT-MAX(A, n)
  if n < 1
      error "heap underflow"
  max = A[1]
  A[1] = A[n]
  n = n − 1
  MAX-HEAPIFY(A, 1, n)
  return max
```

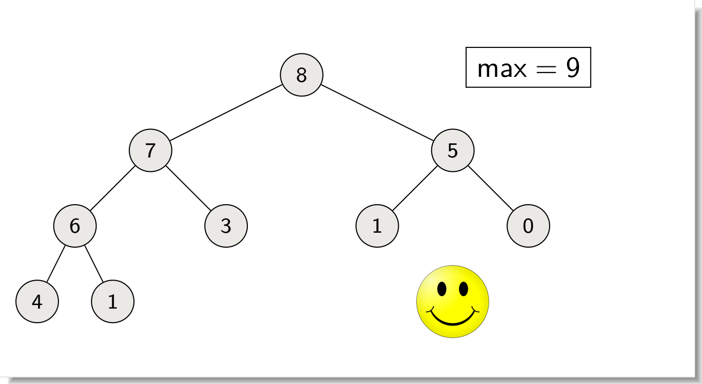


Extracting maximum element

Analysis: Constant-time assignments plus time for MAX-HEAPIFY

HEAP-EXTRACT-MAX(A, n)

if $n < 1$
    error "heap underflow"
$max = A[1]$
$A[1] = A[n]$
$n = n - 1$
MAX-HEAPIFY$(A, 1, n)$
return max



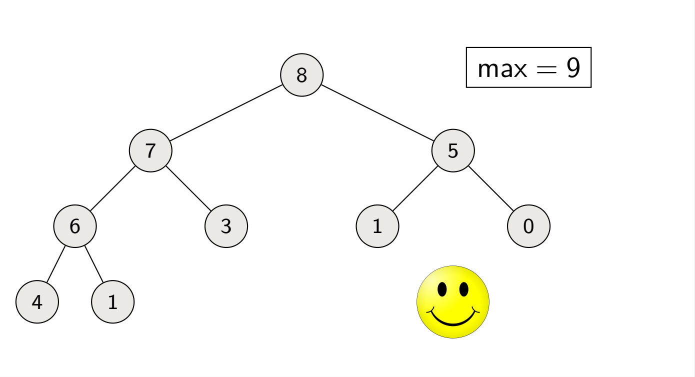
Extracting maximum element

Analysis: Constant-time assignments plus time for MAX-HEAPIFY

Hence, it runs in time $O(\lg n)$

HEAP-EXTRACT-MAX(A, n)

```
if n < 1
    error "heap underflow"
max = A[1]
A[1] = A[n]
n = n - 1
MAX-HEAPIFY(A, 1, n)
return max
```

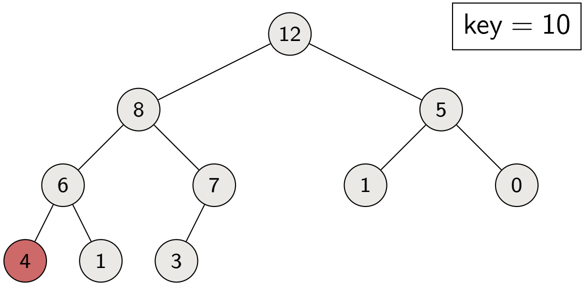


Increasing key value

Given a heap A, index i, and new value *key*

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
  error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  exchange $A[i]$ with $A[\text{PARENT}(i)]$
  $i = \text{PARENT}(i)$



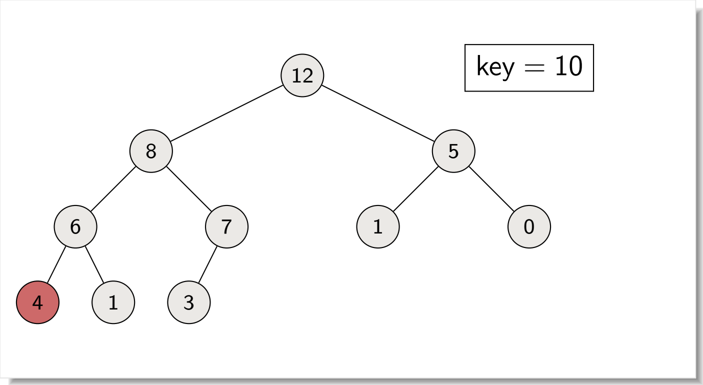
Increasing key value

Given a heap A, index i, and new value *key*

1. Make sure $key \geq A[i]$

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
  error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  exchange $A[i]$ with $A[\text{PARENT}(i)]$
  $i = \text{PARENT}(i)$



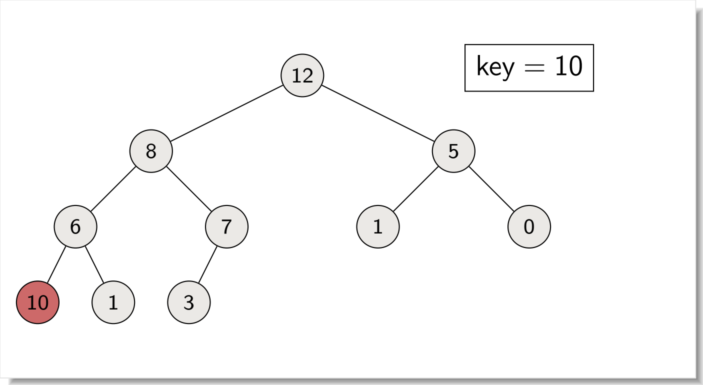
Increasing key value

Given a heap A, index i, and new value *key*

1. Make sure $key \geq A[i]$
2. Update $A[i]$'s value to *key*

HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

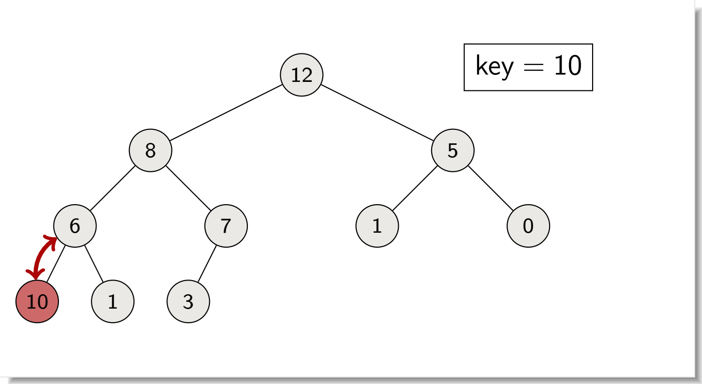


Increasing key value

Given a heap A, index i, and new value *key*

1. Make sure $key \geq A[i]$
2. Update $A[i]$'s value to *key*
3. Traverse the tree upward comparing new key to the parent and swapping keys if necessary, until the new key is smaller than the parent's key

```
HEAP-INCREASE-KEY(A, i, key)
  if key < A[i]
      error "new key is smaller than current key"
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
      exchange A[i] with A[PARENT(i)]
      i = PARENT(i)
```

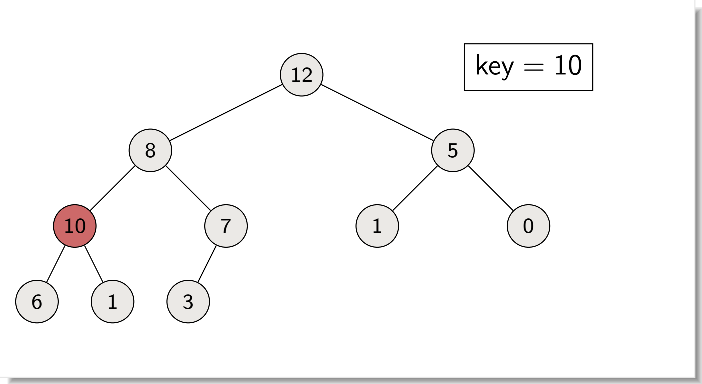


Increasing key value

```
HEAP-INCREASE-KEY(A, i, key)
  if key < A[i]
      error "new key is smaller than current key"
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
      exchange A[i] with A[PARENT(i)]
      i = PARENT(i)
```

Given a heap A, index i, and new value *key*

1. Make sure $key \geq A[i]$
2. Update $A[i]$'s value to *key*
3. Traverse the tree upward comparing new key to the parent and swapping keys if necessary, until the new key is smaller than the parent's key

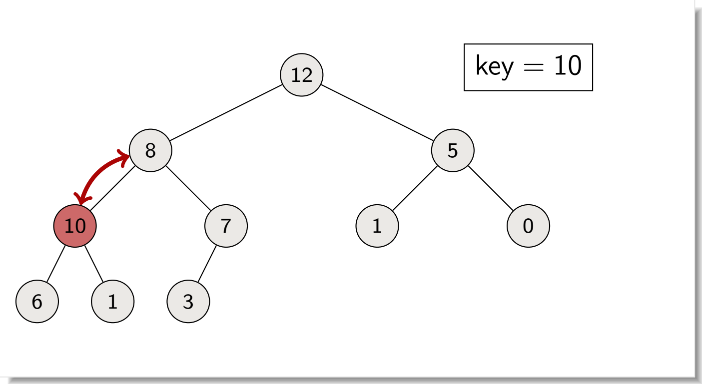


Increasing key value

Given a heap A, index i, and new value *key*

1. Make sure $key \geq A[i]$
2. Update $A[i]$'s value to *key*
3. Traverse the tree upward comparing new key to the parent and swapping keys if necessary, until the new key is smaller than the parent's key

```
HEAP-INCREASE-KEY(A, i, key)
  if key < A[i]
      error "new key is smaller than current key"
  A[i] = key
  while i > 1 and A[PARENT(i)] < A[i]
      exchange A[i] with A[PARENT(i)]
      i = PARENT(i)
```



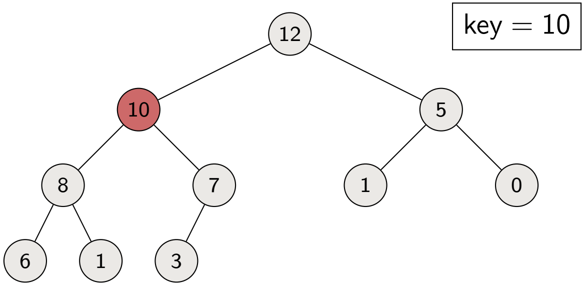
Increasing key value

Given a heap A, index i, and new value *key*

1. Make sure $key \geq A[i]$
2. Update $A[i]$'s value to *key*
3. Traverse the tree upward comparing new key to the parent and swapping keys if necessary, until the new key is smaller than the parent's key

HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```



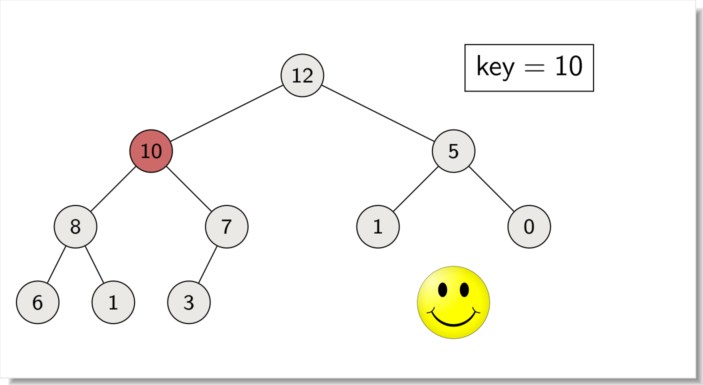
Increasing key value

Given a heap A, index i, and new value *key*

1. Make sure $key \geq A[i]$
2. Update $A[i]$'s value to *key*
3. Traverse the tree upward comparing new key to the parent and swapping keys if necessary, until the new key is smaller than the parent's key

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
  error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  exchange $A[i]$ with $A[\text{PARENT}(i)]$
  $i = \text{PARENT}(i)$

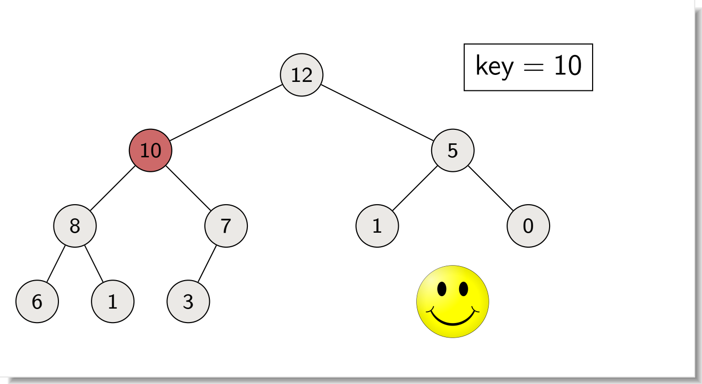


Increasing key value

Analysis:

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

if $key < A[i]$
  error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{Parent}(i)] < A[i]$
  exchange $A[i]$ with $A[\text{Parent}(i)]$
  $i = \text{Parent}(i)$

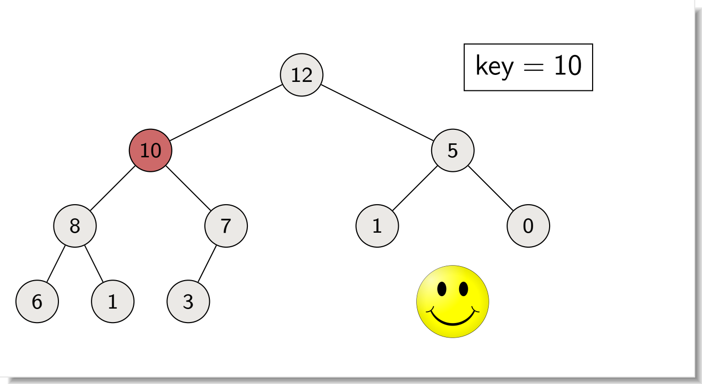


Increasing key value

HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

Analysis:
Upward path from node i has length $O(\lg n)$ in an n-element heap



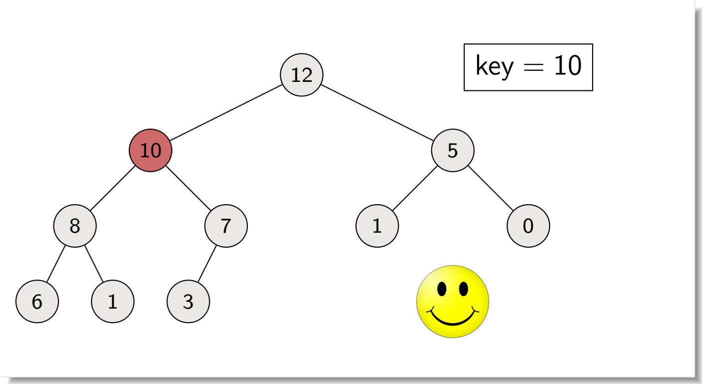
Increasing key value

HEAP-INCREASE-KEY(A, i, key)

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

Analysis:
Upward path from node i has length $O(\lg n)$ in an n-element heap

Hence, it runs in time $O(\lg n)$

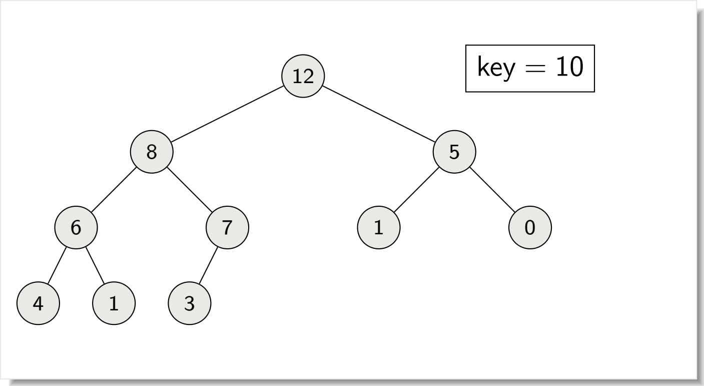


Inserting into the heap

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)

Given a new *key* to insert into heap



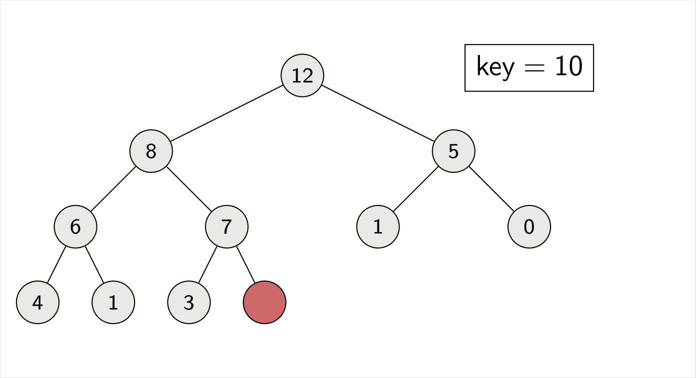
Inserting into the heap

Given a new *key* to insert into heap

1. Increment the heap size

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)



Inserting into the heap

Given a new *key* to insert into heap

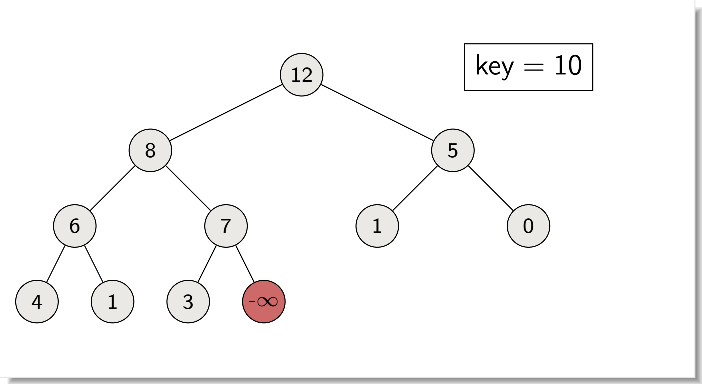
1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$

$A[n] = -\infty$

HEAP-INCREASE-KEY(A, n, key)



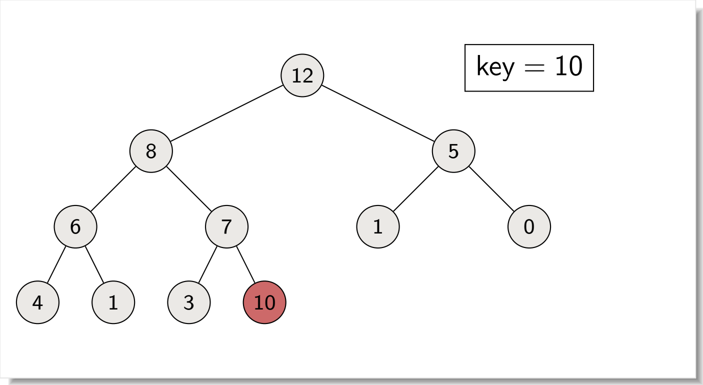
Inserting into the heap

Given a new *key* to insert into heap

1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$
3. Increase the $-\infty$ value to *key* using HEAP-INCREASE-KEY

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)



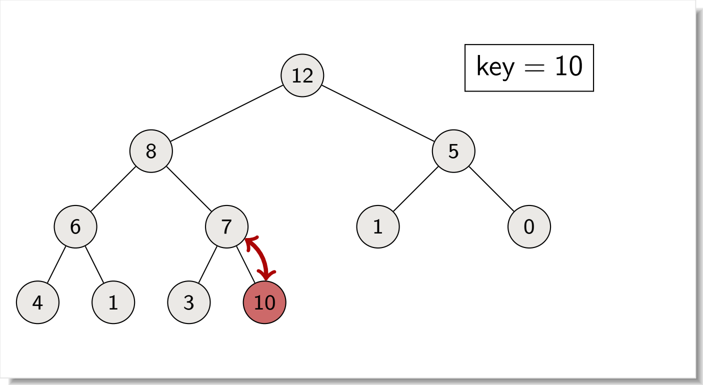
Inserting into the heap

Given a new *key* to insert into heap

1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$
3. Increase the $-\infty$ value to *key* using HEAP-INCREASE-KEY

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)



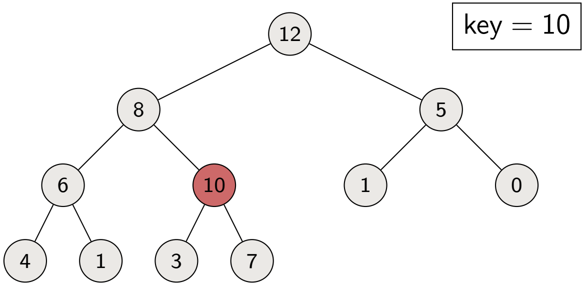
Inserting into the heap

$\textsc{Max-Heap-Insert}(A, key, n)$

$n = n + 1$
$A[n] = -\infty$
$\textsc{Heap-Increase-Key}(A, n, key)$

Given a new *key* to insert into heap

1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$
3. Increase the $-\infty$ value to *key* using $\textsc{Heap-Increase-Key}$



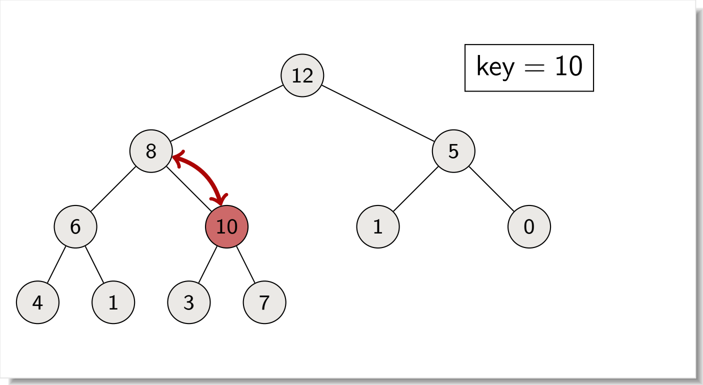
Inserting into the heap

Given a new *key* to insert into heap

1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$
3. Increase the $-\infty$ value to *key* using HEAP-INCREASE-KEY

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)



Inserting into the heap

Given a new *key* to insert into heap

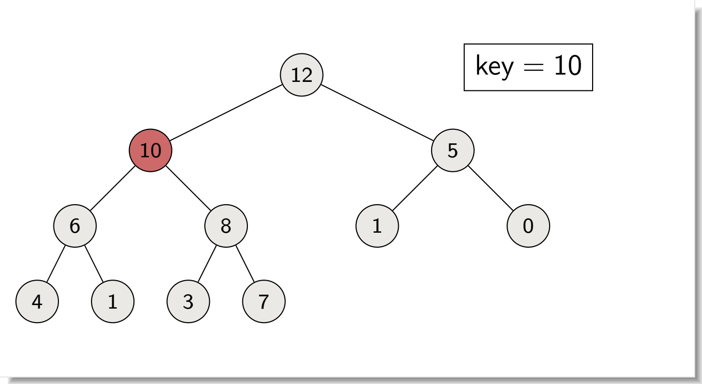
1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$
3. Increase the $-\infty$ value to *key* using HEAP-INCREASE-KEY

MAX-HEAP-INSERT (A, key, n)

$n = n + 1$

$A[n] = -\infty$

HEAP-INCREASE-KEY (A, n, key)



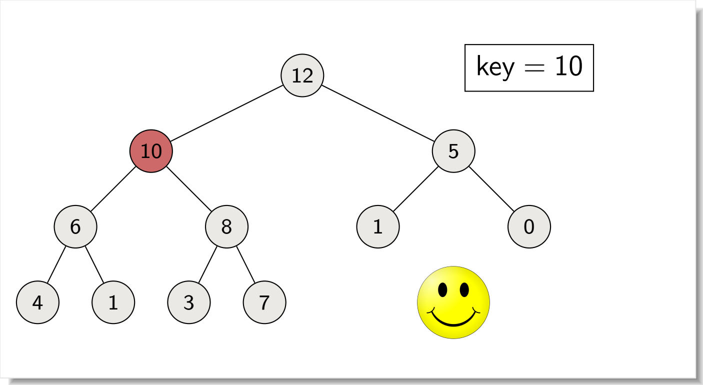
Inserting into the heap

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)

Given a new *key* to insert into heap

1. Increment the heap size
2. Insert a new node in the last position in the heap, with key $-\infty$
3. Increase the $-\infty$ value to *key* using HEAP-INCREASE-KEY

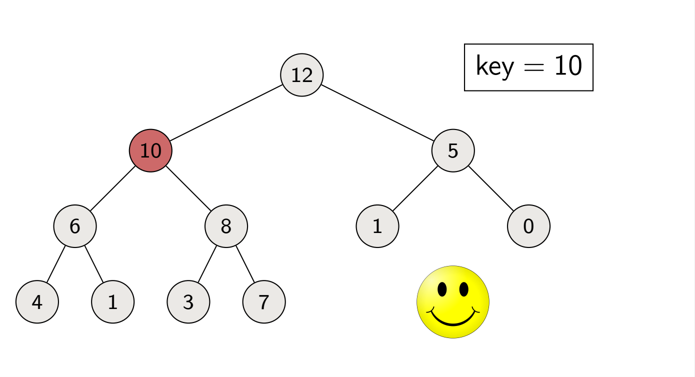


Inserting into the heap

Analysis:

MAX-HEAP-INSERT (A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY (A, n, key)



Inserting into the heap

MAX-HEAP-INSERT (A, key, n)

$n = n + 1$

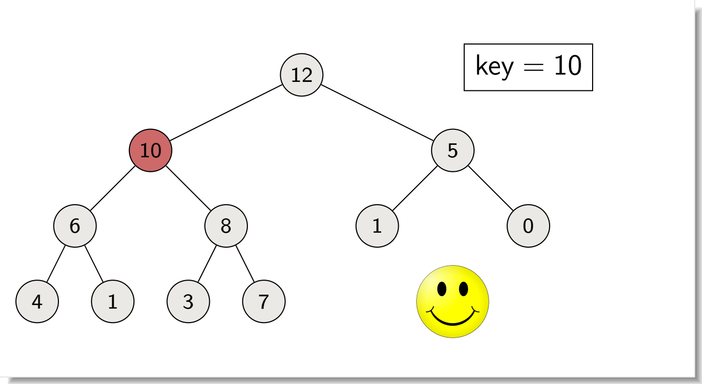
$A[n] = -\infty$

HEAP-INCREASE-KEY (A, n, key)

Analysis: Constant time assignments

$+$

Time for HEAP-INCREASE-KEY



Inserting into the heap

Analysis: Constant time assignments

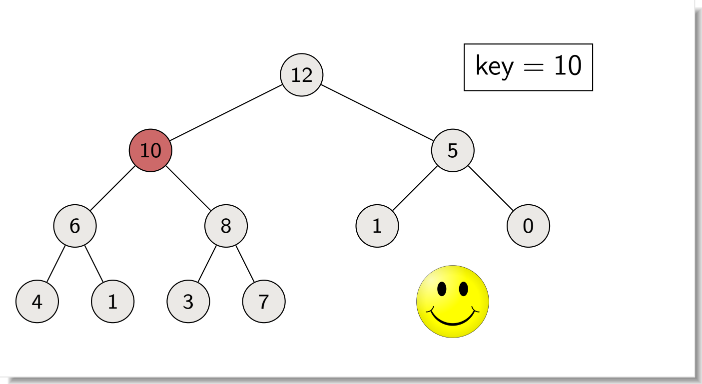
$+$

Time for HEAP-INCREASE-KEY

Hence, it runs in time $O(\lg n)$

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)



Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  INSERT(S,x):

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  $\textsc{Insert}(S,x)$: $O(\lg n)$

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S):

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S): $O(1)$

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S): $O(1)$
  EXTRACT-MAX(S):

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  Insert(S,x): $O(\lg n)$
  Maximum(S): $O(1)$
  Extract-Max(S): $O(\lg n)$

Summary

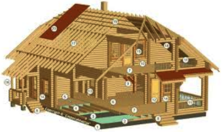
- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  - INSERT(S,x): $O(\lg n)$
  - MAXIMUM(S): $O(1)$
  - EXTRACT-MAX(S): $O(\lg n)$
  - INCREASE-KEY(S,x,k):

Summary

- Heapsort runs in time $O(n \log n)$ and is in-place
- Great algorithm but a well-implemented quicksort usually beats it in practice
- Heaps efficiently implement priority queues
  - $\textsc{Insert}(S,x)$: $O(\lg n)$
  - $\textsc{Maximum}(S)$: $O(1)$
  - $\textsc{Extract-Max}(S)$: $O(\lg n)$
  - $\textsc{Increase-Key}(S,x,k)$: $O(\lg n)$
- Min-priority queues are implemented with min-heaps similarly

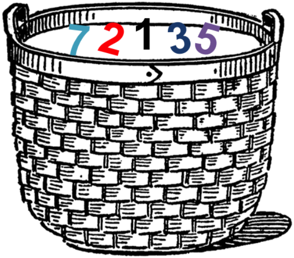
Elementary Data Structures

Algorithm →



Algorithm →

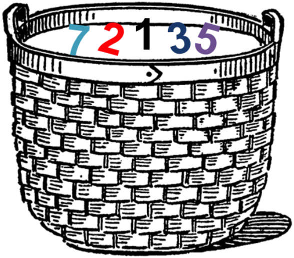
Data structures = dynamic sets of items



Data structure containing numbers

Data structures = dynamic sets of items

What kind of operations do we want to do?

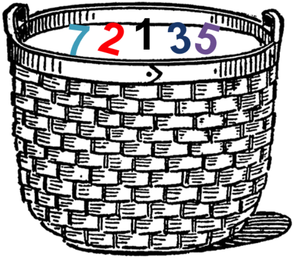


Data structure containing numbers

Data structures = dynamic sets of items

What kind of operations do we want to do?

- Modifying operations: insertion, deletion, . . .
- Query operations: search, maximum, minimum, . . .

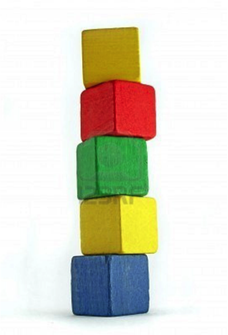


Data structure containing numbers

Stacks (last-in, first-out)

- Insert operation called PUSH(S,x)

- Delete operation called POP(S)



Stacks (last-in, first-out)

- Insert operation called PUSH(S,x)
- Delete operation called POP(S)

Example:

PUSH(S,2), PUSH(S,1), POP(S), PUSH(S,3), POP(S), POP(S)

S

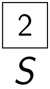
Stacks (last-in, first-out)

- Insert operation called PUSH(S,x)

- Delete operation called POP(S)

Example:

PUSH(S,2), PUSH(S,1), POP(S), PUSH(S,3), POP(S), POP(S)



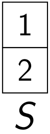
Stacks (last-in, first-out)

- Insert operation called PUSH(S,x)

- Delete operation called POP(S)

Example:

PUSH(S,2), PUSH(S,1), POP(S), PUSH(S,3), POP(S), POP(S)



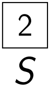
Stacks (last-in, first-out)

- Insert operation called PUSH(S,x)

- Delete operation called POP(S)

Example:

PUSH(S,2), PUSH(S,1), POP(S), PUSH(S,3), POP(S), POP(S)



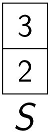
Stacks (last-in, first-out)

- Insert operation called PUSH(S,x)

- Delete operation called POP(S)

Example:

PUSH(S,2), PUSH(S,1), POP(S), PUSH(S,3), POP(S), POP(S)



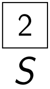
Stacks (last-in, first-out)

- Insert operation called PUSH(S,x)

- Delete operation called POP(S)

Example:

PUSH(S,2), PUSH(S,1), POP(S), PUSH(S,3), POP(S), POP(S)



Stacks (last-in, first-out)

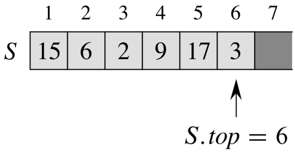
- Insert operation called PUSH(S,x)

- Delete operation called POP(S)

Example:
PUSH(S,2), PUSH(S,1), POP(S), PUSH(S,3), POP(S), POP(S)

S

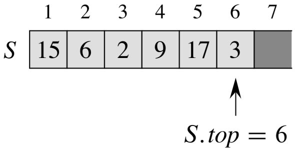
Stacks Implementation



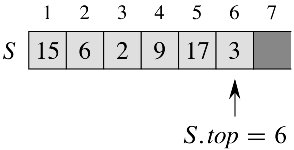
Implementation using arrays: S consists of elements $S[1, \ldots, S.top]$

- $S[1]$ element at the bottom
- $S[S.top]$ element at the top

Stacks Implementation

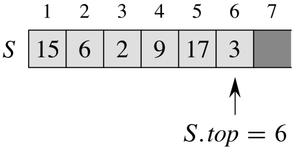


Stacks Implementation




```
STACK-EMPTY(S)
if S.top = 0
    return TRUE
else return FALSE
```

Stacks Implementation



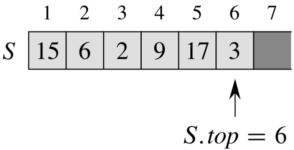
STACK-EMPTY(S)

1. **if** $S.top = 0$
2.   **return** TRUE
3. **else return** FALSE

PUSH(S,x)

1. $S.top \leftarrow S.top + 1$
2. $S[S.top] \leftarrow x$

Stacks Implementation



Stack-Empty(S)

```
if S.top = 0
    return TRUE
else return FALSE
```

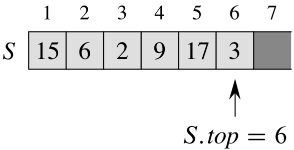
Push(S,x)

```
S.top ← S.top + 1
S[S.top] ← x
```

Pop(S)

```
if Stack-Empty(S)
    error "underflow"
else
    S.top ← S.top − 1
    return S[S.top + 1]
```

Stacks Implementation



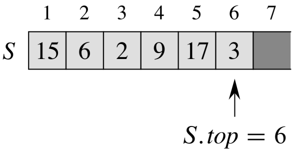
What is the running time of these operations?

```
STACK-EMPTY(S)
if S.top = 0
    return TRUE
else return FALSE
```

```
PUSH(S,x)
S.top ← S.top + 1
S[S.top] ← x
```

```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else
    S.top ← S.top − 1
    return S[S.top + 1]
```

Stacks Implementation



What is the running time of these operations? $O(1)$

STACK-EMPTY(S)

```
if S.top = 0
    return TRUE
else return FALSE
```

PUSH(S,x)

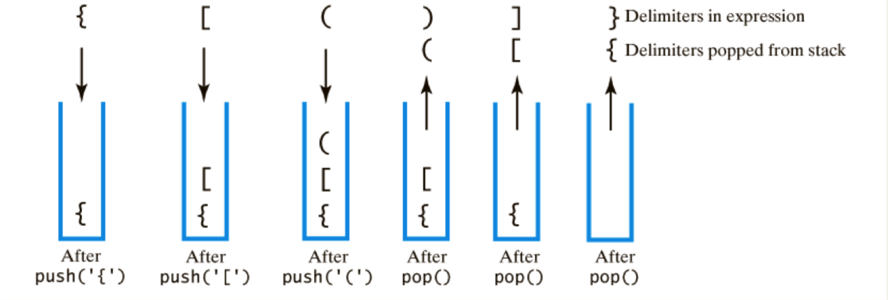
```
S.top ← S.top + 1
S[S.top] ← x
```

POP(S)

```
if STACK-EMPTY(S)
    error "underflow"
else
    S.top ← S.top − 1
    return S[S.top + 1]
```

Stacks are everywhere in every software

Stacks are everywhere in every software



The contents of a stack during the scan of an expression that contains the balanced delimiters $\{\ [\ (\ )\ ]\ \}$

$$a\{b[c(d+e)/2-f]+1\}$$

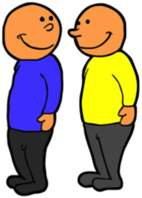
Queues (first-in, first-out)

- Insert operation called ENQUEUE(Q,x)
- Delete operation called DEQUEUE(Q)

Queues (first-in, first-out)

- Insert operation called $\textsc{Enqueue}(Q,x)$
- Delete operation called $\textsc{Dequeue}(Q)$

Example:



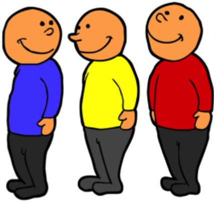
Head *Tail*

Queues (first-in, first-out)

- Insert operation called ENQUEUE(Q,x)
- Delete operation called DEQUEUE(Q)

Example:

ENQUEUE(Q, ),



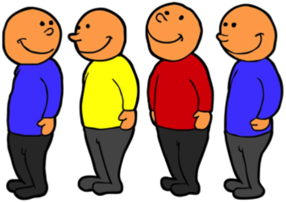
Head *Tail*

Queues (first-in, first-out)

- Insert operation called $\textsc{Enqueue}(Q,x)$
- Delete operation called $\textsc{Dequeue}(Q)$

Example:

$\textsc{Enqueue}(Q,$  $)$, $\textsc{Enqueue}(Q,$  $)$,



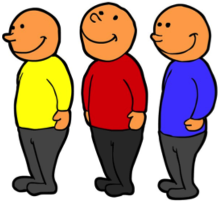
Head *Tail*

Queues (first-in, first-out)

- Insert operation called ENQUEUE(Q,X)
- Delete operation called DEQUEUE(Q)

Example:

ENQUEUE(Q, ), ENQUEUE(Q, ), DEQUEUE(Q),



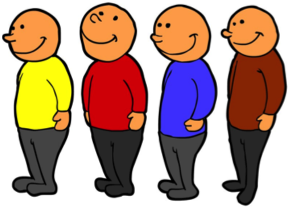
Head *Tail*

Queues (first-in, first-out)

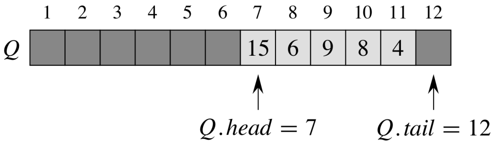
- Insert operation called $\textsc{Enqueue}(Q,x)$
- Delete operation called $\textsc{Dequeue}(Q)$

Example:

$\textsc{Enqueue}(Q,$  $)$, $\textsc{Enqueue}(Q,$  $)$, $\textsc{Dequeue}(Q)$, $\textsc{Enqueue}(Q,$  $)$



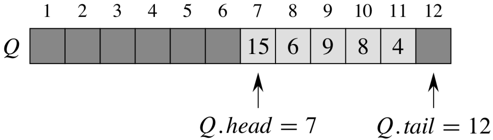
Queue Implementation



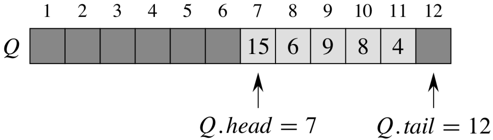
Implementation using arrays: Q consists of elements $S[Q.head, \ldots, Q.tail - 1]$

- $Q.head$ points at the first element
- $Q.tail$ points at the next location where a newly arrived element will be placed

Queue Implementation



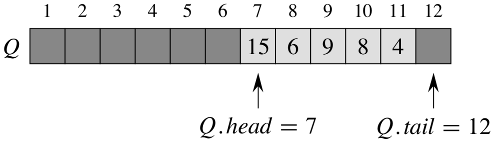
Queue Implementation



ENQUEUE(Q,x)

$Q[Q.tail] = x$
if $Q.tail = Q.length$
    $Q.tail \leftarrow 1$
else $Q.tail \leftarrow Q.tail + 1$

Queue Implementation



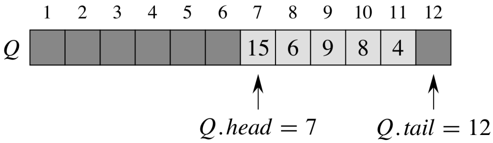
ENQUEUE(Q,x)

```
Q[Q.tail] = x
if Q.tail = Q.length
    Q.tail ← 1
else Q.tail ← Q.tail + 1
```

DEQUEUE(Q)

```
x = Q[Q.head]
if Q.head = Q.length
    Q.head ← 1
else Q.head ← Q.head + 1
return x
```

Queue Implementation

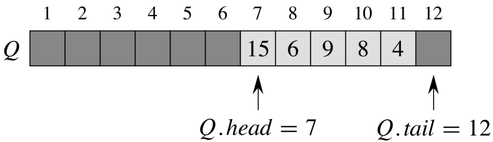


What is the running time of these operations?

```
ENQUEUE(Q,x)
Q[Q.tail] = x
if Q.tail = Q.length
    Q.tail ← 1
else Q.tail ← Q.tail + 1
```

```
DEQUEUE(Q)
x = Q[Q.head]
if Q.head = Q.length
    Q.head ← 1
else Q.head ← Q.head + 1
return x
```

Queue Implementation



What is the running time of these operations? $O(1)$

ENQUEUE(Q,x)

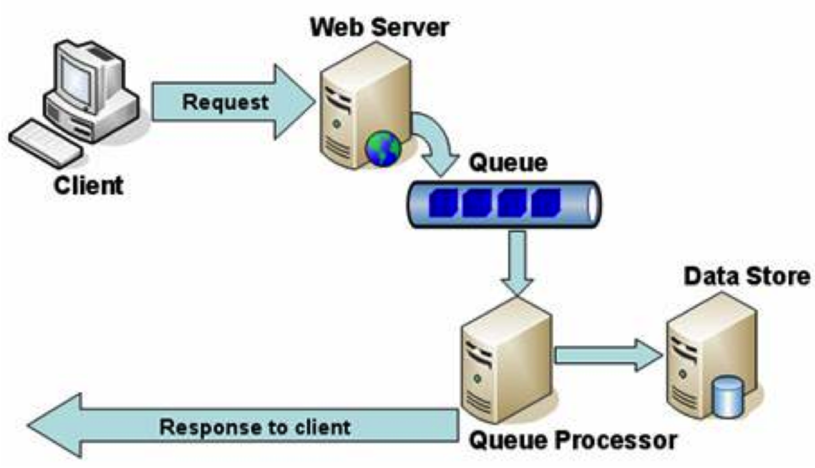
1. $Q[Q.tail] = x$
2. **if** $Q.tail = Q.length$
3. $\quad Q.tail \leftarrow 1$
4. **else** $Q.tail \leftarrow Q.tail + 1$

DEQUEUE(Q)

1. $x = Q[Q.head]$
2. **if** $Q.head = Q.length$
3. $\quad Q.head \leftarrow 1$
4. **else** $Q.head \leftarrow Q.head + 1$
5. **return** x

Applications of Queues

One example: **Web server**



Stacks and Queues

Positives

Negatives

Stacks and Queues

Positives

- Very efficient
- Natural operations

Negatives

Stacks and Queues

Positives

- Very efficient
- Natural operations

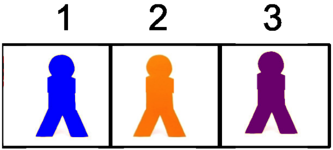
Negatives

- Limited support: for example, no search
- Implementations using arrays have a *fixed* capacity

Linked List

Objects are arranged in a linear order

Not indexes in array

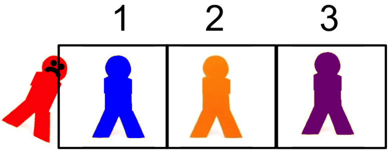


But pointers in each object

Linked List

Objects are arranged in a linear order

Not indexes in array

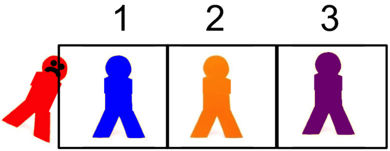


But pointers in each object

Linked List

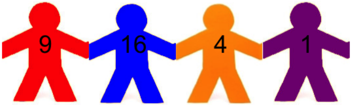
Objects are arranged in a linear order

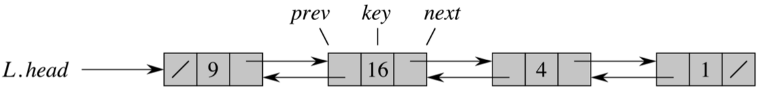
Not indexes in array



But pointers in each object

Linked List

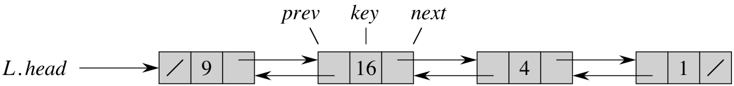




A list can be

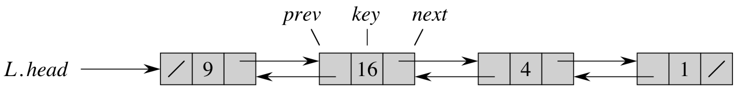
- Single linked or double linked
- Sorted or unsorted
- etc.

Searching a Linked List



Task: Given k return pointer to first element with key k

Searching a Linked List

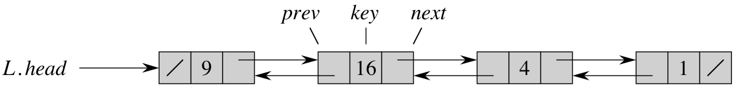


Task: Given k return pointer to first element with key k

LIST-SEARCH(L,K)

1. $x \leftarrow L.head$
2. **while** $x \neq nil$ and $x.key \neq k$
3. $\quad x \leftarrow x.next$
4. **return** x

Searching a Linked List



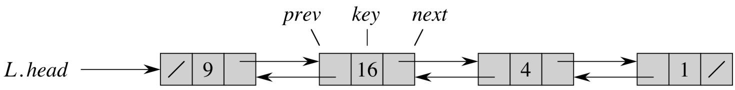
Task: Given k return pointer to first element with key k

LIST-SEARCH(L,K)

```
x ← L.head
while x ≠ nil and x.key ≠ k
    x ← x.next
return x
```

Running time?

Searching a Linked List



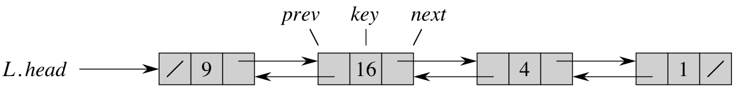
Task: Given k return pointer to first element with key k

LIST-SEARCH(L,K)

1. $x \leftarrow L.head$
2. **while** $x \neq nil$ and $x.key \neq k$
3. $\quad x \leftarrow x.next$
4. **return** x

Running time? $O(n)$

Searching a Linked List



Task: Given k return pointer to first element with key k

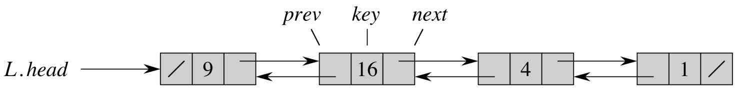
LIST-SEARCH(L,K)

1. $x \leftarrow L.head$
2. **while** $x \neq nil$ and $x.key \neq k$
3. $\quad x \leftarrow x.next$
4. **return** x

Running time? $O(n)$

What if no element with key k exists?

Searching a Linked List



Task: Given k return pointer to first element with key k

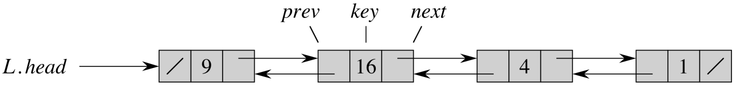
List-Search(L,k)

1. $x \leftarrow L.head$
2. **while** $x \neq nil$ and $x.key \neq k$
3. $\quad x \leftarrow x.next$
4. **return** x

Running time? $O(n)$

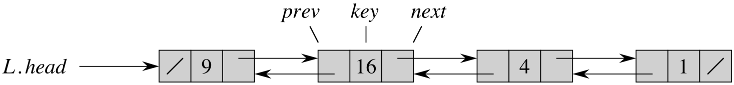
What if no element with key k exists? returns nil

Inserting into a Linked List



Task: Insert a new element x

Inserting into a Linked List

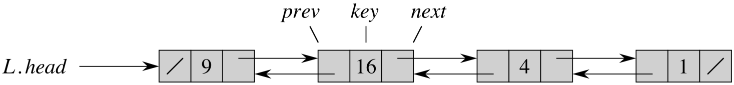


Task: Insert a new element x

LIST-INSERT(L,x)

1. $x.next \leftarrow L.head$
2. **if** $L.head \neq nil$
3. $\quad L.head.prev \leftarrow x$
4. $L.head \leftarrow x$
5. $x.prev = NIL$

Inserting into a Linked List



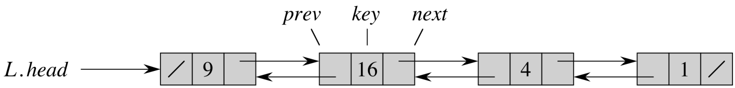
Task: Insert a new element x

List-Insert(L,x)

1. $x.next \leftarrow L.head$
2. **if** $L.head \neq nil$
3. $\quad L.head.prev \leftarrow x$
4. $L.head \leftarrow x$
5. $x.prev = NIL$

Running time?

Inserting into a Linked List



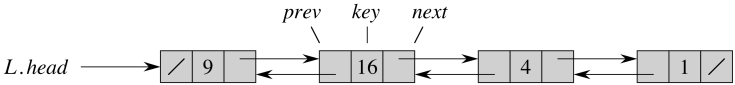
Task: Insert a new element x

LIST-INSERT(L,x)

1. $x.next \leftarrow L.head$
2. **if** $L.head \neq nil$
3. $\quad L.head.prev \leftarrow x$
4. $L.head \leftarrow x$
5. $x.prev = NIL$

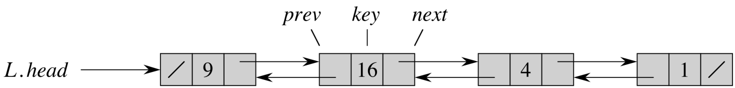
Running time? $O(1)$

Deleting From a Linked List



Task: Given a pointer to an element x remove it from L

Deleting From a Linked List

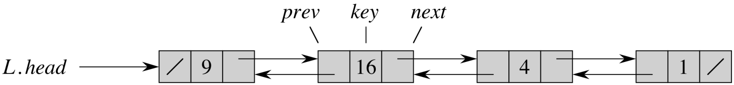


Task: Given a pointer to an element x remove it from L

LIST-DELETE(L,x)

1. **if** $x.prev \neq nil$
2. $\quad x.prev.next \leftarrow x.next$
3. **else** $L.head \leftarrow x.next$
4. **if** $x.next \neq nil$
5. $\quad x.next.prev \leftarrow x.prev$

Deleting From a Linked List



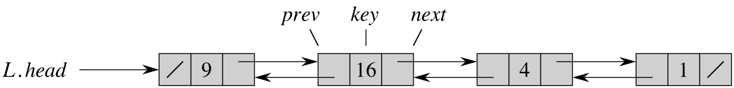
Task: Given a pointer to an element x remove it from L

LIST-DELETE(L,x)

1. **if** $x.prev \neq nil$
2. $\quad x.prev.next \leftarrow x.next$
3. **else** $L.head \leftarrow x.next$
4. **if** $x.next \neq nil$
5. $\quad x.next.prev \leftarrow x.prev$

Running time?

Deleting From a Linked List

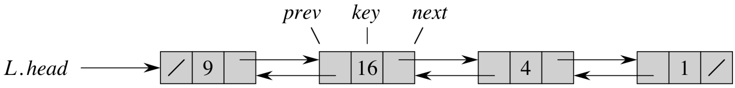


Task: Given a pointer to an element x remove it from L

```
LIST-DELETE(L,x)
if x.prev ≠ nil
    x.prev.next ← x.next
else L.head ← x.next
if x.next ≠ nil
    x.next.prev ← x.prev
```

Running time? $O(1)$

Sentinels



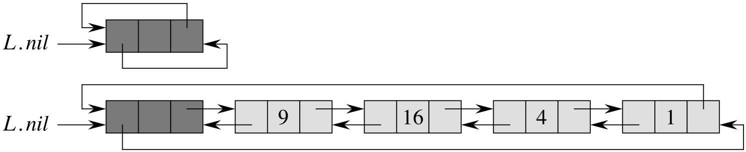
Note: If x is in the middle of the list then

simplified

```
LIST-DELETE(L,x)
if x.prev ≠ nil
    x.prev.next ← x.next
else L.head ← x.next
if x.next ≠ nil
    x.next.prev ← x.prev
```

```
LIST-DELETE'(L,x)
x.prev.next ← x.next
x.next.prev ← x.prev
```

Sentinels



LIST-DELETE(L,x)

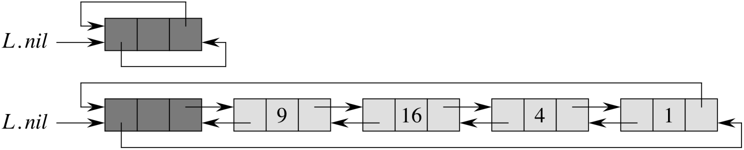
1. **if** $x.prev \neq nil$
2. $\quad x.prev.next \leftarrow x.next$
3. **else** $L.head \leftarrow x.next$
4. **if** $x.next \neq nil$
5. $\quad x.next.prev \leftarrow x.prev$

simplified

LIST-DELETE'(L,x)

1. $x.prev.next \leftarrow x.next$
2. $x.next.prev \leftarrow x.prev$

Sentinels



simplified

LIST-INSERT(L,x)

1. $x.next \leftarrow L.head$
2. **if** $L.head \neq nil$
3. $\quad L.head.prev \leftarrow x$
4. $L.head \leftarrow x$
5. $x.prev = NIL$

LIST-INSERT'(L,x)

1. $x.next \leftarrow L.nil.next$
2. $L.nil.next.prev \leftarrow x$
3. $L.nil.next \leftarrow x$
4. $x.prev \leftarrow L.nil$

Summary Linked List

- Dynamic data structure without predefined capacity
- Insertion: $O(1)$
- Deletion: $O(1)$ (if double linked)
  - Question in book: can you do it for single linked?
- Search: $O(n)$

Summary Linked List

- Dynamic data structure without predefined capacity
- Insertion: $O(1)$
- Deletion: $O(1)$ (if double linked)
  - Question in book: can you do it for single linked?
- Search: $O(n)$

Summary

- Heaps efficiently implement priority queues

Summary

- Heaps efficiently implement priority queues
  Insert(S,x):

Summary

- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$

Summary

- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S):

Summary

- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S): $O(1)$

Summary

- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S): $O(1)$
  EXTRACT-MAX(S):

Summary

- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S): $O(1)$
  EXTRACT-MAX(S): $O(\lg n)$

Summary

- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S): $O(1)$
  EXTRACT-MAX(S): $O(\lg n)$
  INCREASE-KEY(S,x,k):

Summary

- Heaps efficiently implement priority queues
  INSERT(S,x): $O(\lg n)$
  MAXIMUM(S): $O(1)$
  EXTRACT-MAX(S): $O(\lg n)$
  INCREASE-KEY(S,x,k): $O(\lg n)$
- Min-priority queues are implemented with min-heaps similarly
- Stacks, Queues and Linked lists
  - Good at specific operations for specific uses
  - Bad at search