Mergeable Heaps

David Kauchak
cs302
Spring 2013

Admin

- Homework 7?

Binary heap

A binary tree where the value of a parent is greater than or equal to the value of it's children

Additional restriction: all levels of the tree are **complete** except the last

Max heap vs. min heap

Binary heap - operations

Maximum(S) - return the largest element in the set

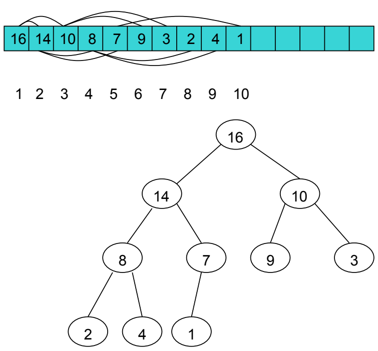
ExtractMax(S) – Return and remove the largest element in the set

Insert(S, val) – insert val into the set

IncreaseElement(S, x, val) – increase the value of element x to val

BuildHeap(A) – build a heap from an array of elements

Binary heap representations



Heapify

Assume left and right children are heaps, turn current set into a valid heap

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  largest ← i
  if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
  if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
  if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

Heapify

Assume left and right children are heaps, turn current set into a valid heap

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  largest ← i
  if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
  if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
  if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

Heapify

Assume left and right children are heaps, turn current set into a valid heap

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  largest ← i
  if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
  if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
  if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

find out which is largest: current, left of right

Heapify

Assume left and right children are heaps, turn current set into a valid heap

HEAPIFY(A, i)
$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
$largest \leftarrow i$
if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
  $largest \leftarrow l$
if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
  $largest \leftarrow r$
if $largest \neq i$
  swap $A[i]$ and $A[largest]$
  HEAPIFY$(A, largest)$

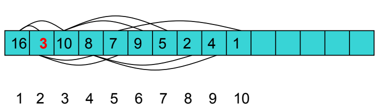
Heapify

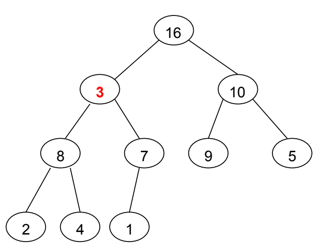
Assume left and right children are heaps, turn current set into a valid heap

HEAPIFY(A, i)
$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
$largest \leftarrow i$
if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
  $largest \leftarrow l$
if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
  $largest \leftarrow r$
if $largest \neq i$
  swap $A[i]$ and $A[largest]$
  HEAPIFY$(A, largest)$

if a child is larger, swap and recurse

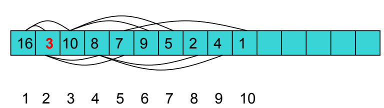
Heapify

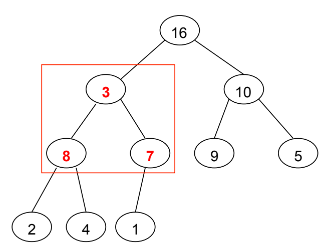




HEAPIFY(A, i)
$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
$largest \leftarrow i$
if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
  $largest \leftarrow l$
if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
  $largest \leftarrow r$
if $largest \neq i$
  swap $A[i]$ and $A[largest]$
  HEAPIFY$(A, largest)$

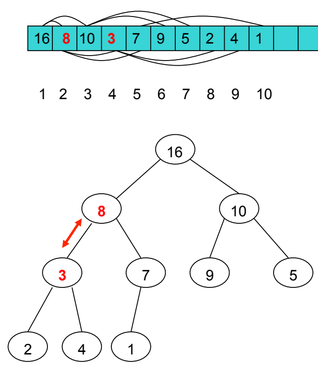
Heapify





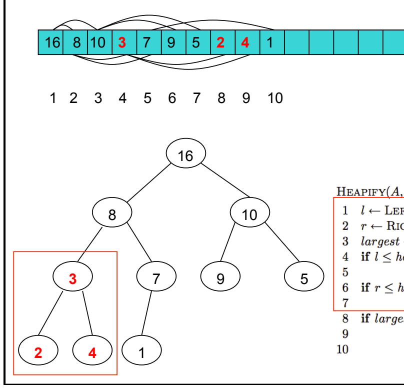
HEAPIFY(A, i)
$l \leftarrow$ LEFT(i)
$r \leftarrow$ RIGHT(i)
$largest \leftarrow i$
if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
  $largest \leftarrow l$
if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
  $largest \leftarrow r$
if $largest \neq i$
  swap $A[i]$ and $A[largest]$
  HEAPIFY$(A, largest)$

Heapify



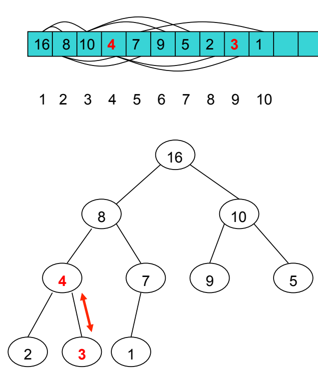

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  largest ← i
  if l ≤ heap-size[A] and A[l] > A[i]
          largest ← l
  if r ≤ heap-size[A] and A[r] > A[largest]
          largest ← r
  if largest ≠ i
          swap A[i] and A[largest]
          HEAPIFY(A, largest)
```

Heapify



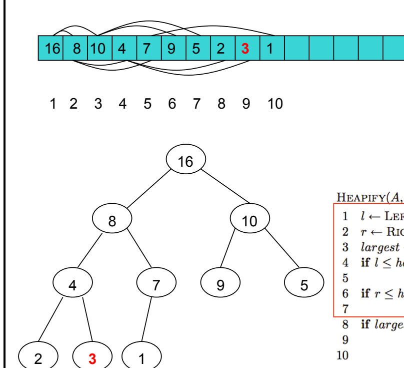

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  largest ← i
  if l ≤ heap-size[A] and A[l] > A[i]
          largest ← l
  if r ≤ heap-size[A] and A[r] > A[largest]
          largest ← r
  if largest ≠ i
          swap A[i] and A[largest]
          HEAPIFY(A, largest)
```

Heapify



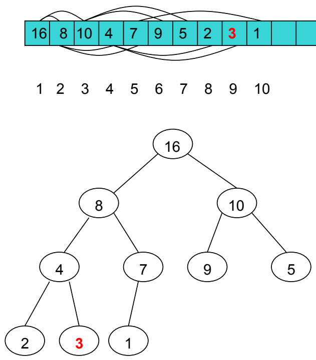

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  largest ← i
  if l ≤ heap-size[A] and A[l] > A[i]
          largest ← l
  if r ≤ heap-size[A] and A[r] > A[largest]
          largest ← r
  if largest ≠ i
          swap A[i] and A[largest]
          HEAPIFY(A, largest)
```

Heapify




```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  largest ← i
  if l ≤ heap-size[A] and A[l] > A[i]
          largest ← l
  if r ≤ heap-size[A] and A[r] > A[largest]
          largest ← r
  if largest ≠ i
          swap A[i] and A[largest]
          HEAPIFY(A, largest)
```

Heapify




```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

Correctness of Heapify

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

Correctness of Heapify

Base case:

- Heap with a single element
- Trivially a heap

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

Correctness of Heapify

Both children are valid heaps

Three cases:

Case 1: A[i] (current node) is the largest

```
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

- parent is greater than both children
- both children are heaps
- current node is a valid heap

Correctness of Heapify

Case 2: left child is the largest

```
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

- When Heapify returns:
  - Left child is a valid heap
  - Right child is unchanged and therefore a valid heap
  - Current node is larger than both children since we selected the largest node of current, left and right
  - current node is a valid heap

Case 3: right child is largest

- similar to above

Running time of Heapify

What is the cost of each individual call to Heapify (not counting recursive calls)?

- Θ(1)

How many calls are made to Heapify?

- O(height of the tree)

What is the height of the tree?

- Complete binary tree, except for the last level

$$2^h \le n$$

$$h \le \log_2 n$$

O(log n)

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

Binary heap - operations

Maximum(S) - return the largest element in the set

ExtractMax(S) – Return and remove the largest element in the set

Insert(S, val) – insert val into the set

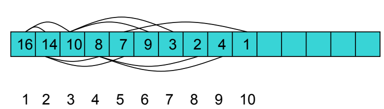
IncreaseElement(S, x, val) – increase the value of element x to val

BuildHeap(A) – build a heap from an array of elements

Maximum

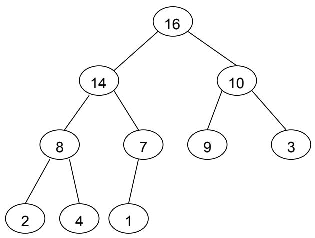
Return the largest element from the set

Return A[1]



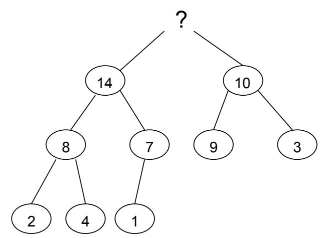
ExtractMax

Return and remove the largest element in the set



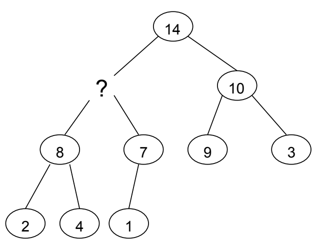
ExtractMax

Return and remove the largest element in the set



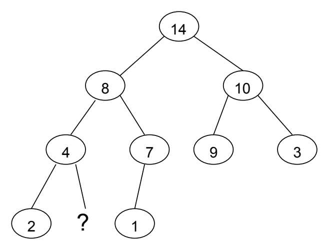
ExtractMax

Return and remove the largest element in the set



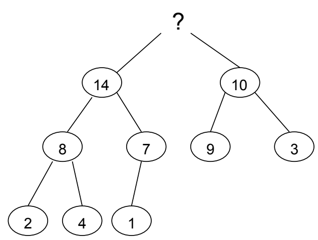
ExtractMax

Return and remove the largest element in the set



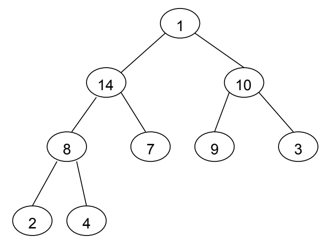
ExtractMax

Return and remove the largest element in the set



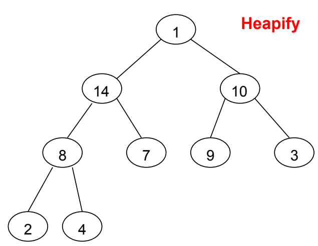
ExtractMax

Return and remove the largest element in the set



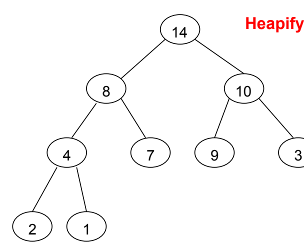
ExtractMax

Return and remove the largest element in the set



ExtractMax

Return and remove the largest element in the set



ExtractMax

Return and remove the largest element in the set

```
EXTRACT-MAX(A)
  if heap-size[A] < 1
        error
  max ← A[1]
  A[1] ← A[HEAP-SIZE[A]] − 1
  heap-size[A] ← heap-size[A] − 1
  HEAPIFY(A, 1)
  return max
```

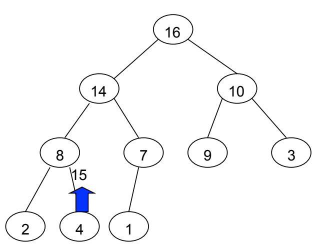
ExtractMax running time

Constant amount of work plus one call to Heapify – O(log n)

```
EXTRACT-MAX(A)
  if heap-size[A] < 1
        error
  max ← A[1]
  A[1] ← A[HEAP-SIZE[A]] − 1
  heap-size[A] ← heap-size[A] − 1
  HEAPIFY(A, 1)
  return max
```

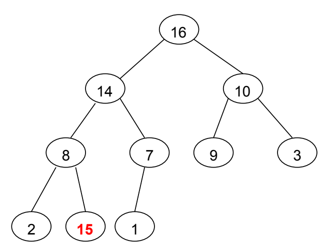
IncreaseElement

Increase the value of element *x* to *val*



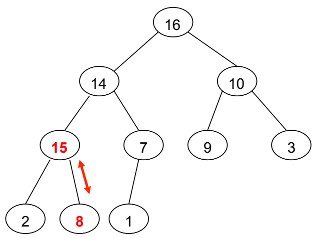
IncreaseElement

Increase the value of element *x* to *val*



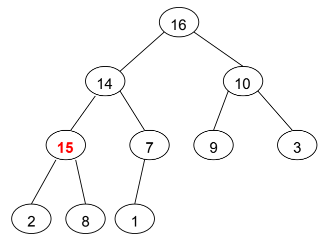
IncreaseElement

Increase the value of element *x* to *val*



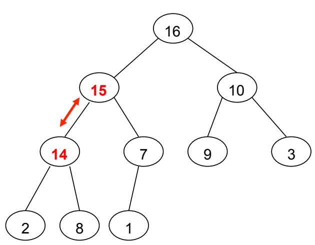
IncreaseElement

Increase the value of element *x* to *val*



IncreaseElement

Increase the value of element *x* to *val*



IncreaseElement

Increase the value of element *x* to *val*

```
INCREASE-ELEMENT(A, i, val)
  if val < A[i]
        error
  A[i] ← val
  while i > 1 and A[PARENT(i)] < A[i]
        swap A[i] and A[PARENT(i)]
        i ← PARENT(i)
```

Correctness of IncreaseElement

Why is it ok to swap values with parent?

```
INCREASE-ELEMENT(A, i, val)
if val < A[i]
        error
A[i] ← val
while i > 1 and A[PARENT(i)] < A[i]
        swap A[i] and A[PARENT(i)]
        i ← PARENT(i)
```

Correctness of IncreaseElement

Stop when heap property is satisfied

```
INCREASE-ELEMENT(A, i, val)
if val < A[i]
        error
A[i] ← val
while i > 1 and A[PARENT(i)] < A[i]
        swap A[i] and A[PARENT(i)]
        i ← PARENT(i)
```

Running time of IncreaseElement

Follows a path from a node to the root

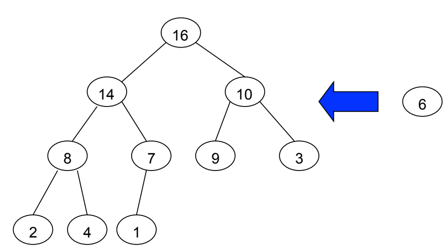
Worst case O(height of the tree)

O(log n)

```
INCREASE-ELEMENT(A, i, val)
if val < A[i]
        error
A[i] ← val
while i > 1 and A[PARENT(i)] < A[i]
        swap A[i] and A[PARENT(i)]
        i ← PARENT(i)
```

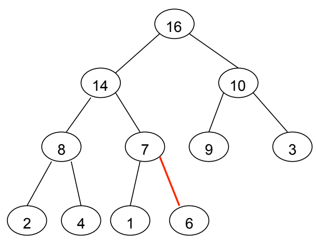
Insert

Insert *val* into the set



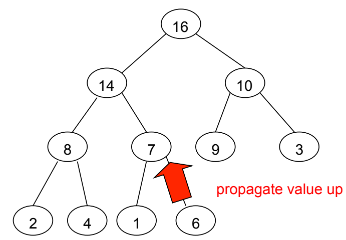
Insert

Insert *val* into the set



Insert

Insert *val* into the set



Insert

$\text{INSERT}(A, val)$

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$A[heap\text{-}size[A]] \leftarrow -\infty$
$\text{INCREASE-ELEMENT}(A, heap\text{-}size[A], val)$

Running time of Insert

Constant amount of work plus one call to IncreaseElement – O(log n)

$\text{INSERT}(A, val)$

$heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
$A[heap\text{-}size[A]] \leftarrow -\infty$
$\text{INCREASE-ELEMENT}(A, heap\text{-}size[A], val)$

Building a heap

Can we build a heap using the functions we have so far?

- Maximum(S)
- ExtractMax(S)
- Insert(S, val)|
- IncreaseElement(S, x, val)

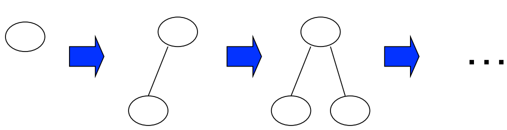
Building a heap

```
BUILD-HEAP1(A)
  copy A to B
  heap-size[A] ← 0
  for i ← 1 to length[B]
          INSERT(A, B[i])
```

Running time of BuildHeap1

n calls to Insert – O(n log n)

Can we do better?



Building a heap: take 2

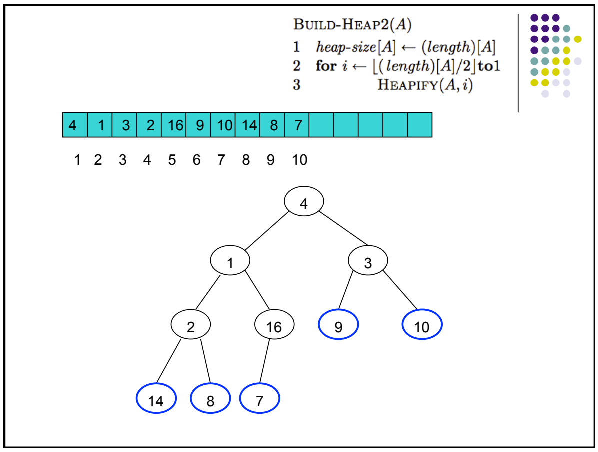
```
BUILD-HEAP2(A)
  heap-size[A] ← (length)[A]
  for i ← ⌊(length)[A]/2⌋ to 1
          HEAPIFY(A, i)
```

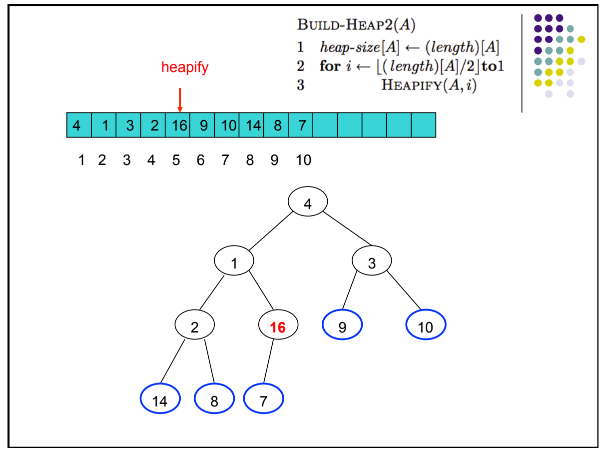
Start with n/2 “simple” heaps

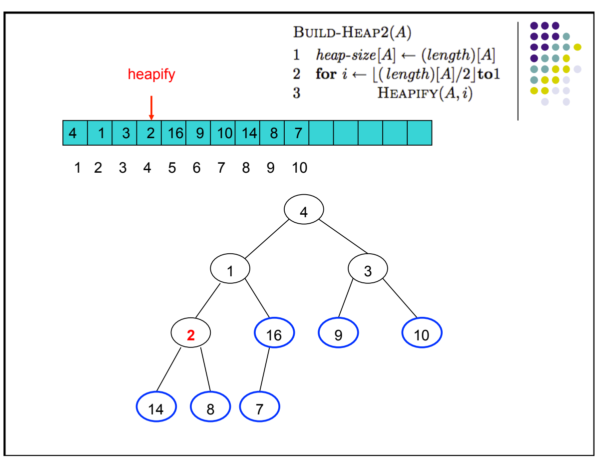
call Heapify on element n/2-1, n/2-2, n/2-3 …

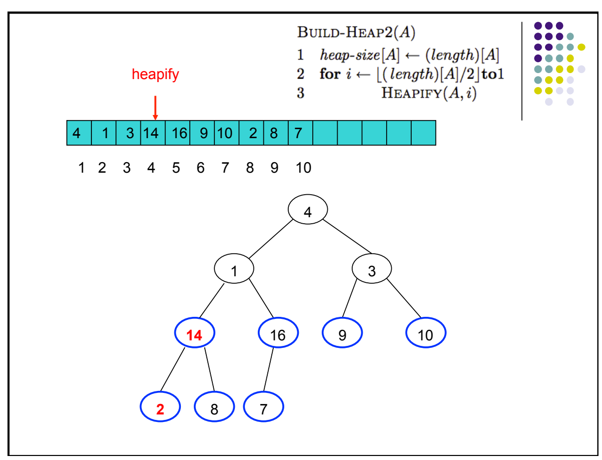
all children have smaller indices

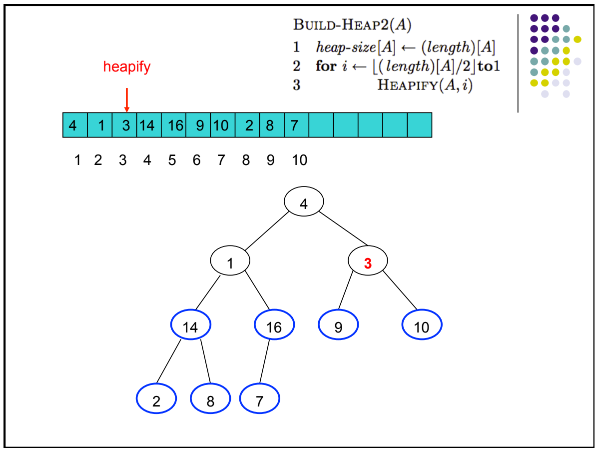
building from the bottom up, makes sure that all the children are heaps

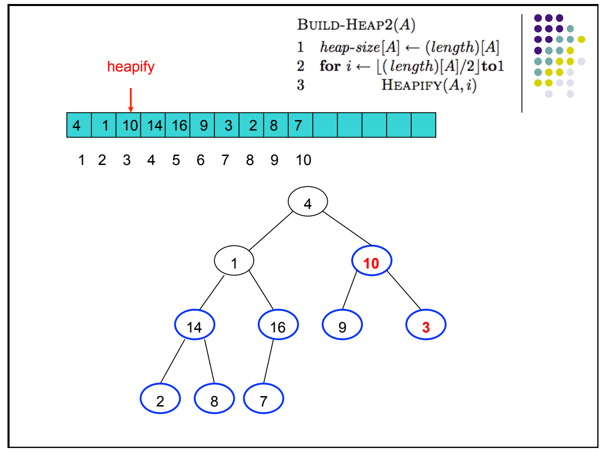

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
4 1 3 2 16 9 10 14 8 7
1 2 3 4 5 6 7 8 9 10
4
1 3
2 16 9 10
14 8 7

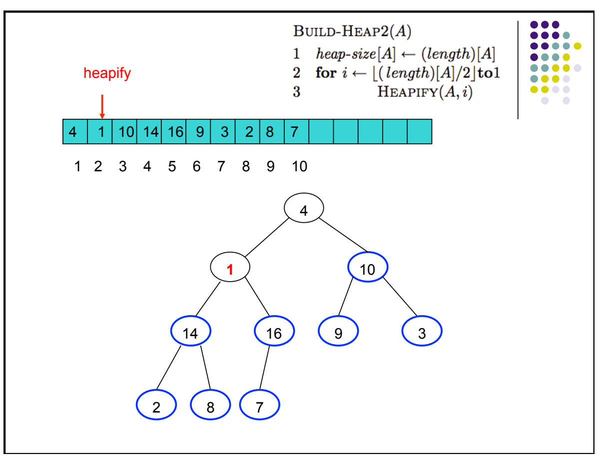

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 3 2 16 9 10 14 8 7
1 2 3 4 5 6 7 8 9 10
4
1 3
2 16 9 10
14 8 7

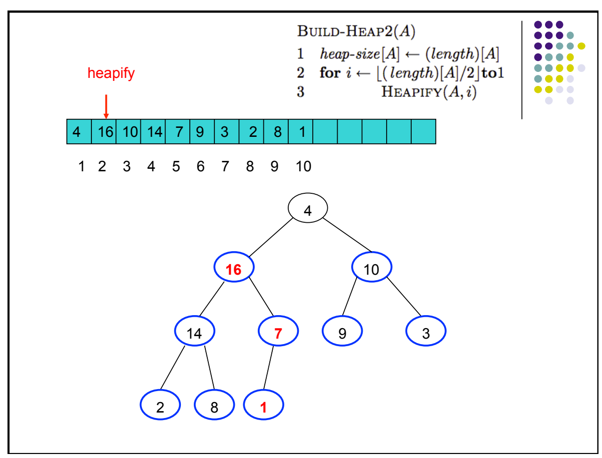

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 3 2 16 9 10 14 8 7
1 2 3 4 5 6 7 8 9 10
4
1 3
2 16 9 10
14 8 7


BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 3 14 16 9 10 2 8 7
1 2 3 4 5 6 7 8 9 10
4
1 3
14 16 9 10
2 8 7

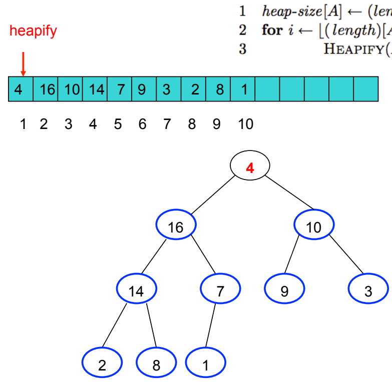

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 3 14 16 9 10 2 8 7
1 2 3 4 5 6 7 8 9 10
4
1 3
14 16 9 10
2 8 7


BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 10 14 16 9 3 2 8 7
1 2 3 4 5 6 7 8 9 10
4
1 10
14 16 9 3
2 8 7

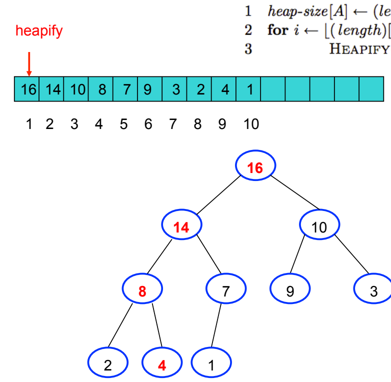

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 10 14 16 9 3 2 8 7
1 2 3 4 5 6 7 8 9 10
4
1 10
14 16 9 3
2 8 7


BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 16 10 14 7 9 3 2 8 1
1 2 3 4 5 6 7 8 9 10
4
16 10
14 7 9 3
2 8 1

BUILD-HEAP2(A)
$heap\text{-}size[A] \leftarrow length[A]$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **to** 1
  HEAPIFY(A, i)



BUILD-HEAP2(A)
$heap\text{-}size[A] \leftarrow length[A]$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **to** 1
  HEAPIFY(A, i)



Correctness of BuildHeap2

BUILD-HEAP2(A)
$heap\text{-}size[A] \leftarrow length[A]$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **to** 1
  HEAPIFY(A, i)

Invariant:

Correctness of BuildHeap2

BUILD-HEAP2(A)
$heap\text{-}size[A] \leftarrow length[A]$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ **to** 1
  HEAPIFY(A, i)

Invariant: elements A[i+1…n] are all heaps

Base case: i = floor(n/2). All elements i+1, i+2, …, n are "simple" heaps

Inductive case: We know i+1, i+2, .., n are all heaps, therefore the call to Heapify(A,i) generates a heap at node i

Termination?

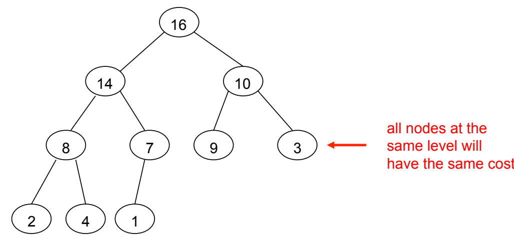
Running time of BuildHeap2

n/2 calls to Heapify – O(n log n)

Can we get a tighter bound?

```
BUILD-HEAP2(A)
heap-size[A] ← (length)[A]
for i ← ⌊(length)[A]/2⌋ to 1
        HEAPIFY(A, i)
```

Running time of BuildHeap2



How many nodes are at level *d*? 2^d

Running time of BuildHeap2

$$T(n) = \sum_{d=0}^{\log n} 2^d O(d)$$

?

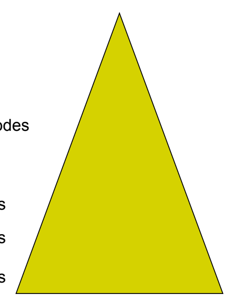
Nodes at height *h*

h < ceil($n/2^{h+1}$) nodes

h=2 < ceil(n/8) nodes

h=1 < ceil(n/4) nodes

h=0 < ceil(n/2) nodes



Running time of BuildHeap2

$$T(n) = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

$$= O\left(n \sum_{h=0}^{\log n} \left\lceil \frac{1}{2^{h+1}} \right\rceil h \right)$$

$$= O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h} \right)$$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(n)$$

$$\boxed{\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2}$$

BuildHeap1 vs. BuildHeap2

```
BUILD-HEAP1(A)
copy A to B
heap-size[A] ← 0
for i ← 1 to length[B]
        INSERT(A, B[i])
```

```
BUILD-HEAP2(A)
heap-size[A] ← (length)[A]
for i ← ⌊(length)[A]/2⌋ to 1
        HEAPIFY(A, i)
```

Runtime

- O(n) vs. O(n log n)

Memory

- Both O(n)
- BuildHeap1 requires an additional array, i.e. 2n memory

Complexity/Ease of implementation

Heap uses

Could we use a heap to sort?

Heap uses

Heapsort

- Build a heap
- Call ExtractMax for all the elements
- O(n log n) running time

Priority queues

- scheduling tasks: jobs, processes, network traffic
- A* search algorithm

Binary heaps

Procedure	Binary heap (worst-case)
BUILD-HEAP	$\Theta(n)$
INSERT	$\Theta(\log n)$
MAXIMUM	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$
UNION	
INCREASE-ELEMENT	$\Theta(\log n)$
DELETE	$\Theta(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

Mergeable heaps

Procedure	Binary heap (worst-case)
BUILD-HEAP	$\Theta(n)$
INSERT	$\Theta(\log n)$
MAXIMUM	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$
UNION	
INCREASE-ELEMENT	$\Theta(\log n)$
DELETE	$\Theta(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

- Mergeable heaps support the union operation
- Allows us to combine two heaps to get a single heap
- Union runtime for binary heaps?

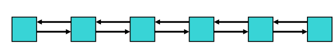
Union for binary heaps

Procedure	Binary heap (worst-case)
BUILD-HEAP	$\Theta(n)$
INSERT	$\Theta(\log n)$
MAXIMUM	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$
UNION	$\boxed{\Theta(n)}$
INCREASE-ELEMENT	$\Theta(\log n)$
DELETE	$\Theta(\log n)$

(adapted from Figure 19.1, pg. 456 [1])

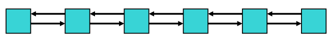
concatenate the arrays and then call Build-Heap

Linked-list heap



- Store the elements in a doubly linked list
- Insert:
- Max:
- Extract-Max:
- Increase:
- Union:

Linked-list heap



- Store the elements in a doubly linked list
- Insert: add to the end/beginning
- Max: search through the linked list
- Extract-Max: search and delete
- Increase: increase value
- Union: concatenate linked lists

Linked-list heap

Procedure	Binary heap (worst-case)	Linked-list
BUILD-HEAP	$\Theta(n)$	Θ(n)
INSERT	$\Theta(\log n)$	Θ(1)
MAXIMUM	$\Theta(1)$	Θ(n)
EXTRAC-MAX	$\Theta(\log n)$	Θ(n)
UNION	$\Theta(n)$	Θ(1)
INCREASE-ELEMENT	$\Theta(\log n)$	Θ(1)
DELETE	$\Theta(\log n)$	Θ(1)

(adapted from Figure 19.1, pg. 456 [1])

Faster Union, Increase, Insert and Delete… but slower Max operations

Binomial Tree