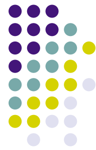
# Mergeable Heaps

David Kauchak
cs302
Spring 2012



## Admin

- go/codegolf
- CS/linguistics talks tomorrow starting at 12:35
  - 3 talks
  - Axinn 229

## 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 - array

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

1 2 3 4 5 6 7 8 9 10



## Heapify

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

```
HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  largest ← i
4  if l ≤ heap-size[A] and A[l] > A[i]
5        largest ← l
6  if r ≤ heap-size[A] and A[r] > A[largest]
7        largest ← r
8  if largest ≠ i
9        swap A[i] and A[largest]
10       HEAPIFY(A, largest)
```

## Heapify

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

```
HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  largest ← i
4  if l ≤ heap-size[A] and A[l] > A[i]
5        largest ← l
6  if r ≤ heap-size[A] and A[r] > A[largest]
7        largest ← r
8  if largest ≠ i
9        swap A[i] and A[largest]
10       HEAPIFY(A, largest)
```

## Heapify

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

```
HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  largest ← i
4  if l ≤ heap-size[A] and A[l] > A[i]
5        largest ← l
6  if r ≤ heap-size[A] and A[r] > A[largest]
7        largest ← r
8  if largest ≠ i
9        swap A[i] and A[largest]
10       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$(A, i)$
1 $l \leftarrow$ LEFT$(i)$
2 $r \leftarrow$ RIGHT$(i)$
3 $largest \leftarrow i$
4 if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
5 $largest \leftarrow l$
6 if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
7 $largest \leftarrow r$
8 if $largest \neq i$
9 swap $A[i]$ and $A[largest]$
10 HEAPIFY$(A, largest)$

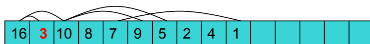
## Heapify

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

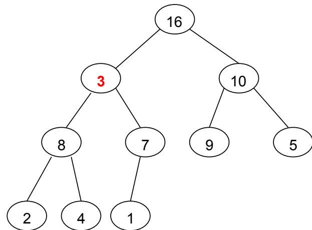
HEAPIFY$(A, i)$
1 $l \leftarrow$ LEFT$(i)$
2 $r \leftarrow$ RIGHT$(i)$
3 $largest \leftarrow i$
4 if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
5 $largest \leftarrow l$
6 if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
7 $largest \leftarrow r$
8 if $largest \neq i$
9 swap $A[i]$ and $A[largest]$
10 HEAPIFY$(A, largest)$

if a child is larger, swap and recurse

## Heapify

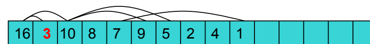



1 2 3 4 5 6 7 8 9 10

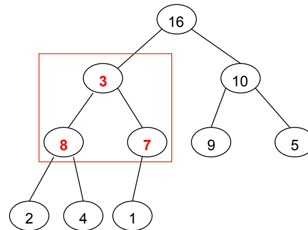



HEAPIFY$(A, i)$
1 $l \leftarrow$ LEFT$(i)$
2 $r \leftarrow$ RIGHT$(i)$
3 $largest \leftarrow i$
4 if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
5 $largest \leftarrow l$
6 if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
7 $largest \leftarrow r$
8 if $largest \neq i$
9 swap $A[i]$ and $A[largest]$
10 HEAPIFY$(A, largest)$

## Heapify




1 2 3 4 5 6 7 8 9 10




HEAPIFY$(A, i)$
1 $l \leftarrow$ LEFT$(i)$
2 $r \leftarrow$ RIGHT$(i)$
3 $largest \leftarrow i$
4 if $l \leq heap\text{-}size[A]$ and $A[l] > A[i]$
5 $largest \leftarrow l$
6 if $r \leq heap\text{-}size[A]$ and $A[r] > A[largest]$
7 $largest \leftarrow r$
8 if $largest \neq i$
9 swap $A[i]$ and $A[largest]$
10 HEAPIFY$(A, largest)$

# Heapify

| 16 | 8 | 10 | 3 | 7 | 9 | 5 | 2 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

16
8 10
3 7 9 5
2 4 1

```
HEAPIFY(A, i)
1   l ← LEFT(i)
2   r ← RIGHT(i)
3   largest ← i
4   if l ≤ heap-size[A] and A[l] > A[i]
5         largest ← l
6   if r ≤ heap-size[A] and A[r] > A[largest]
7         largest ← r
8   if largest ≠ i
9         swap A[i] and A[largest]
10        HEAPIFY(A, largest)
```

# Heapify

| 16 | 8 | 10 | 3 | 7 | 9 | 5 | 2 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

16
8 10
3 7 9 5
2 4 1

```
HEAPIFY(A, i)
1   l ← LEFT(i)
2   r ← RIGHT(i)
3   largest ← i
4   if l ≤ heap-size[A] and A[l] > A[i]
5         largest ← l
6   if r ≤ heap-size[A] and A[r] > A[largest]
7         largest ← r
8   if largest ≠ i
9         swap A[i] and A[largest]
10        HEAPIFY(A, largest)
```

# Heapify

| 16 | 8 | 10 | 4 | 7 | 9 | 5 | 2 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

16
8 10
4 7 9 5
2 3 1

```
HEAPIFY(A, i)
1   l ← LEFT(i)
2   r ← RIGHT(i)
3   largest ← i
4   if l ≤ heap-size[A] and A[l] > A[i]
5         largest ← l
6   if r ≤ heap-size[A] and A[r] > A[largest]
7         largest ← r
8   if largest ≠ i
9         swap A[i] and A[largest]
10        HEAPIFY(A, largest)
```

# Heapify

| 16 | 8 | 10 | 4 | 7 | 9 | 5 | 2 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

16
8 10
4 7 9 5
2 3 1

```
HEAPIFY(A, i)
1   l ← LEFT(i)
2   r ← RIGHT(i)
3   largest ← i
4   if l ≤ heap-size[A] and A[l] > A[i]
5         largest ← l
6   if r ≤ heap-size[A] and A[r] > A[largest]
7         largest ← r
8   if largest ≠ i
9         swap A[i] and A[largest]
10        HEAPIFY(A, largest)
```

## Heapify

| 16 | 8 | 10 | 4 | 7 | 9 | 5 | 2 | 3 | 1 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

1 2 3 4 5 6 7 8 9 10

16
8 10
4 7 9 5
2 3 1

```
HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  largest ← i
4  if l ≤ heap-size[A] and A[l] > A[i]
5         largest ← l
6  if r ≤ heap-size[A] and A[r] > A[largest]
7         largest ← r
8  if largest ≠ i
9         swap A[i] and A[largest]
10        HEAPIFY(A, largest)
```

## Correctness of Heapify

```
HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  largest ← i
4  if l ≤ heap-size[A] and A[l] > A[i]
5         largest ← l
6  if r ≤ heap-size[A] and A[r] > A[largest]
7         largest ← r
8  if largest ≠ i
9         swap A[i] and A[largest]
10        HEAPIFY(A, largest)
```

## Correctness of Heapify

- Base case:
  - Heap with a single element
  - Trivially a heap

## Correctness of Heapify

- Both children are valid heaps
- Three cases:
- Case 1: A[i] (current node) is the largest

```
8  if largest ≠ i
9         swap A[i] and A[largest]
10        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

```
 8  if largest ≠ i
 9        swap A[i] and A[largest]
10        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)?
  - $\Theta(1)$
- 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

$$2^h \leq n$$

$$h \leq \log_2 n$$

O(log n)

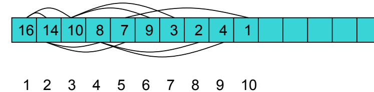
```
HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  largest ← i
 4  if l ≤ heap-size[A] and A[l] > A[i]
 5        largest ← l
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7        largest ← r
 8  if largest ≠ i
 9        swap A[i] and A[largest]
10        HEAPIFY(A, largest)
```

## 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

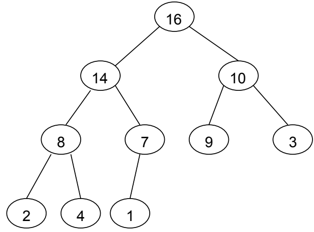
- Return the largest element from the set
- **Return** A[1]




1 2 3 4 5 6 7 8 9 10

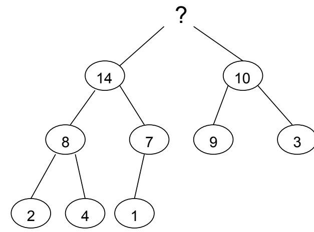
## ExtractMax

- Return and remove the largest element in the set



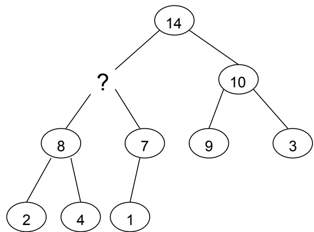

## ExtractMax

- Return and remove the largest element in the set



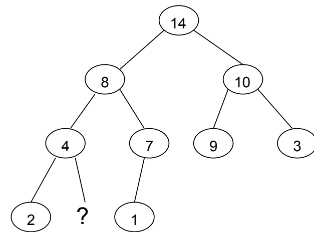

## ExtractMax

- Return and remove the largest element in the set



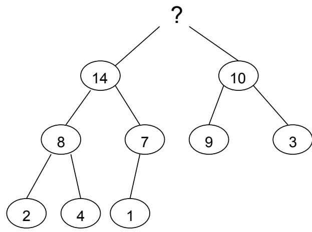

## ExtractMax

- Return and remove the largest element in the set



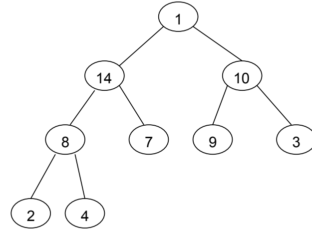
## ExtractMax

- Return and remove the largest element in the set




## ExtractMax

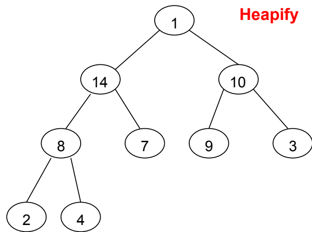
- Return and remove the largest element in the set




## ExtractMax

- Return and remove the largest element in the set

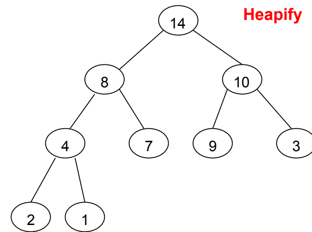
Heapify




## ExtractMax

- Return and remove the largest element in the set

Heapify



## ExtractMax

- Return and remove the largest element in the set

```
EXTRACT-MAX(A)
1  if heap-size[A] < 1
2          error
3  max ← A[1]
4  A[1] ← A[HEAP-SIZE[A]] − 1
5  heap-size[A] ← heap-size[A] − 1
6  HEAPIFY(A, 1)
7  return max
```

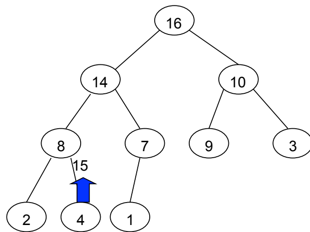
## ExtractMax running time

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

```
EXTRACT-MAX(A)
1  if heap-size[A] < 1
2          error
3  max ← A[1]
4  A[1] ← A[HEAP-SIZE[A]] − 1
5  heap-size[A] ← heap-size[A] − 1
6  HEAPIFY(A, 1)
7  return max
```

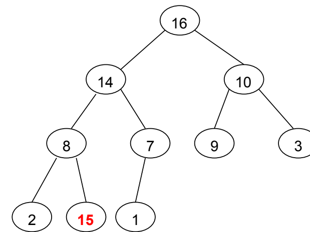
## IncreaseElement

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



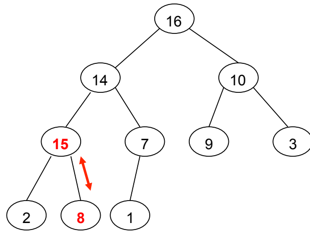

## IncreaseElement

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



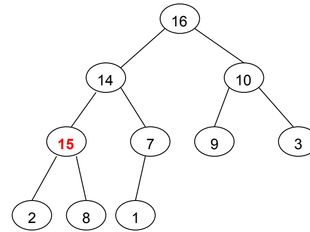
## IncreaseElement

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



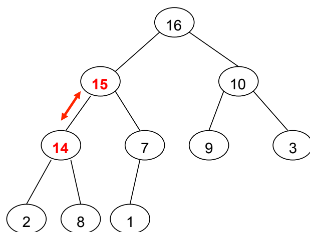

## IncreaseElement

- 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)
1  if val < A[i]
2          error
3  A[i] ← val
4  while i > 1 and A[PARENT(i)] < A[i]
5          swap A[i] and A[PARENT(i)]
6          i ← PARENT(i)
```

## Correctness of IncreaseElement

- Why is it ok to swap values with parent?

INCREASE-ELEMENT$(A, i, val)$
1 **if** $val < A[i]$
2     **error**
3 $A[i] \leftarrow val$
4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5     swap $A[i]$ and $A[\text{PARENT}(i)]$
6     $i \leftarrow \text{PARENT}(i)$

## Correctness of IncreaseElement

- Stop when heap property is satisfied

INCREASE-ELEMENT$(A, i, val)$
1 **if** $val < A[i]$
2     **error**
3 $A[i] \leftarrow val$
4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5     swap $A[i]$ and $A[\text{PARENT}(i)]$
6     $i \leftarrow \text{PARENT}(i)$

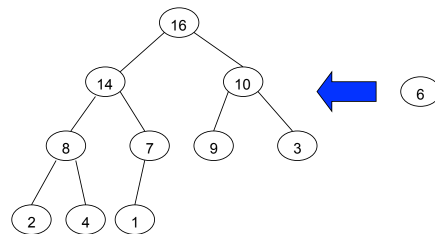
## Running time of IncreaseElement

- Follows a path from a node to the root
- Worst case O(height of the tree)
- O(log n)

INCREASE-ELEMENT$(A, i, val)$
1 **if** $val < A[i]$
2     **error**
3 $A[i] \leftarrow val$
4 **while** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5     swap $A[i]$ and $A[\text{PARENT}(i)]$
6     $i \leftarrow \text{PARENT}(i)$

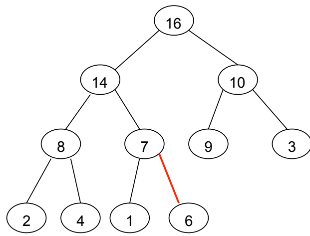
## Insert

- Insert *val* into the set



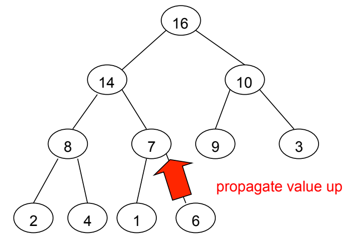
## Insert

- Insert *val* into the set




## Insert

- Insert *val* into the set




## Insert

INSERT$(A, val)$
1 $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2 $A[heap\text{-}size[A]] \leftarrow -\infty$
3 INCREASE-ELEMENT$(A, heap\text{-}size[A], val)$

## Running time of Insert

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

INSERT$(A, val)$
1 $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2 $A[heap\text{-}size[A]] \leftarrow -\infty$
3 INCREASE-ELEMENT$(A, heap\text{-}size[A], val)$

## Building a heap

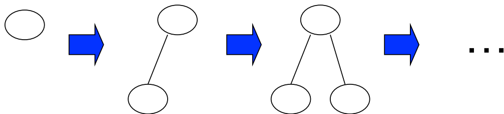
- Can we build a heap using the functions we have so far?
  - Maximum(S)
  - ExtractMax(S)
  - Insert(S, val)|
  - IncreaseElement(S, x, val)

## Building a heap

BUILD-HEAP1($A$)
1 copy $A$ to $B$
2 $heap\text{-}size[A] \leftarrow 0$
3 **for** $i \leftarrow 1$ **to** $length[B]$
4     INSERT($A, B[i]$)

## Running time of BuildHeap1

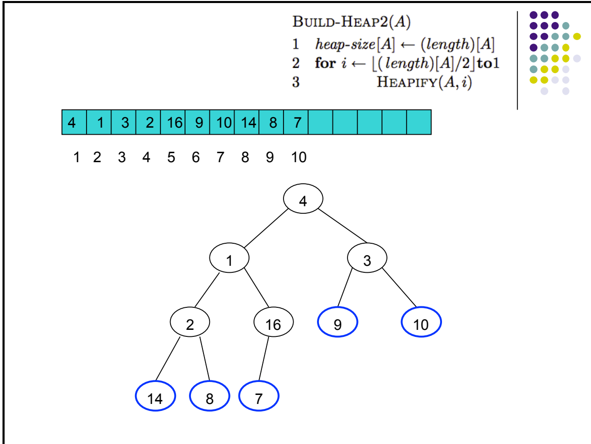
- *n* calls to Insert – O(n log n)
- Can we do better?

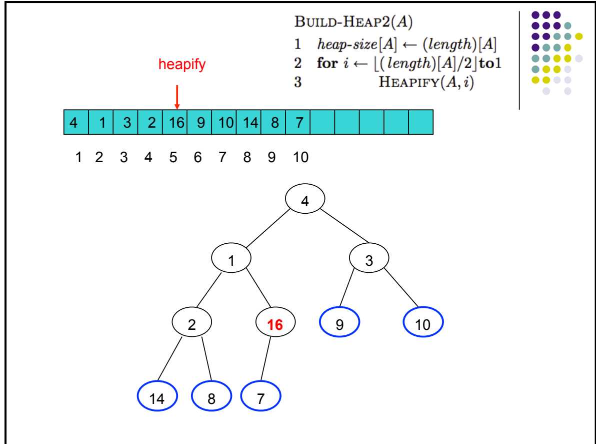


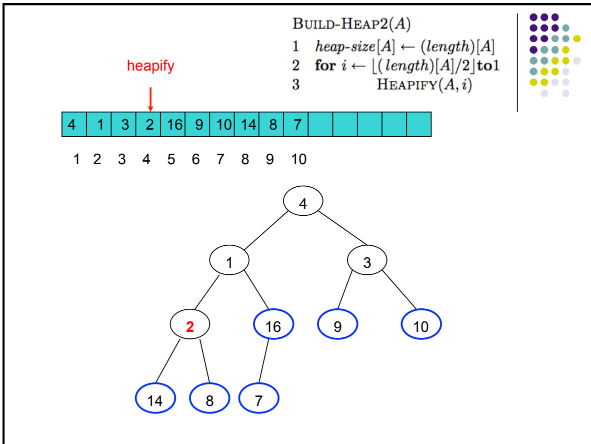
## Building a heap: take 2

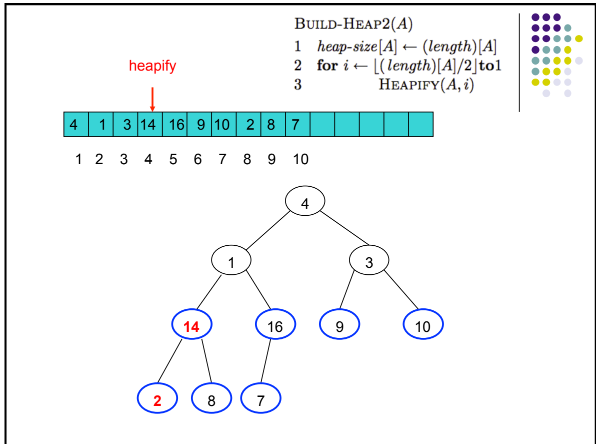
BUILD-HEAP2($A$)
1 $heap\text{-}size[A] \leftarrow (length)[A]$
2 **for** $i \leftarrow \lfloor (length)[A]/2 \rfloor$**to**1
3     HEAPIFY($A, i$)

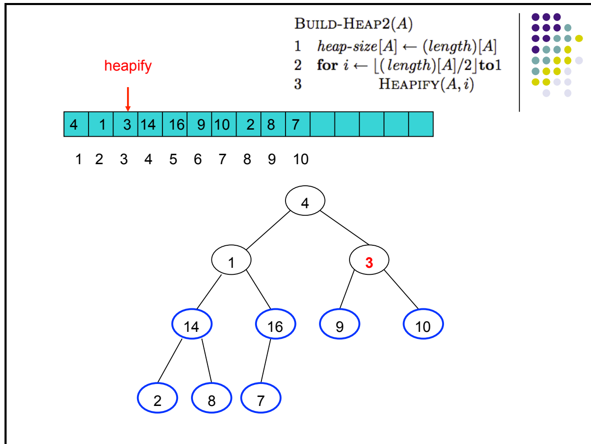
- Start with n/2 “simple” heaps
- call Heapify on element n/2-1, n/2-2, n/2-3 …
- all children have smaller indices
- building from the bottom up, makes sure that all the children are heaps

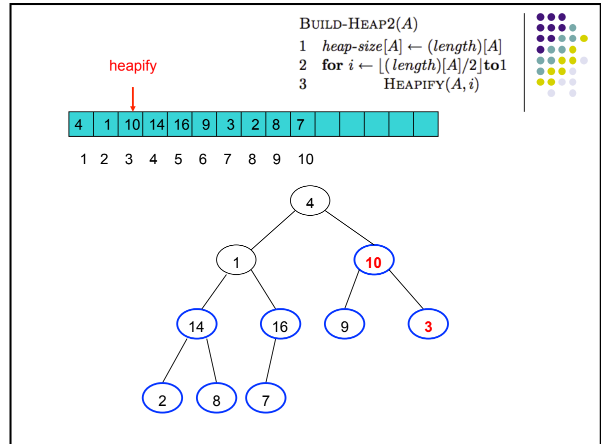

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
4 1 3 2 16 9 10 14 8 7
1 2 3 4 5 6 7 8 9 10
4
1
3
2
16
9
10
14
8
7

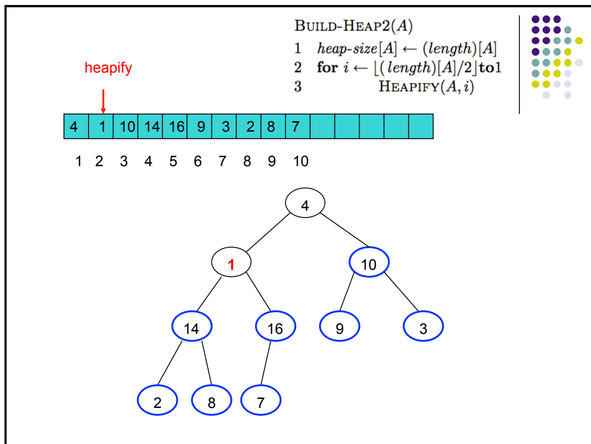

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 3 2 16 9 10 14 8 7
1 2 3 4 5 6 7 8 9 10
4
1
3
2
16
9
10
14
8
7

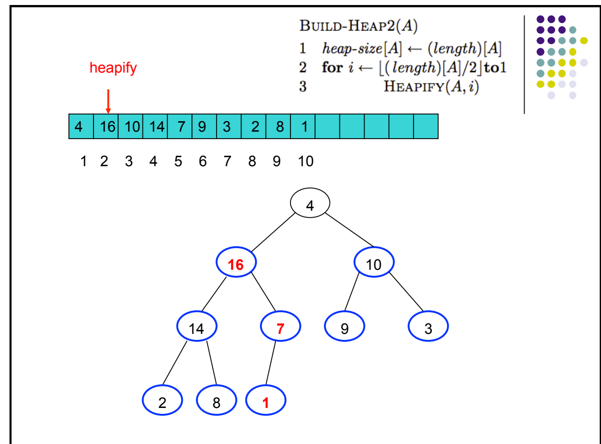

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 3 2 16 9 10 14 8 7
1 2 3 4 5 6 7 8 9 10
4
1
3
2
16
9
10
14
8
7

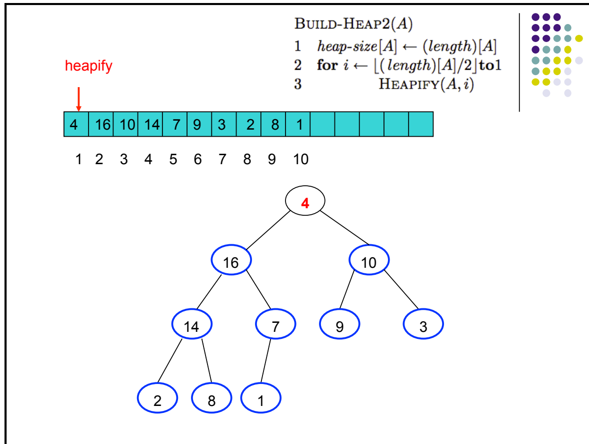

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 HEAPIFY(A, i)
heapify
4 1 3 14 16 9 10 2 8 7
1 2 3 4 5 6 7 8 9 10
4
1
3
14
16
9
10
2
8
7

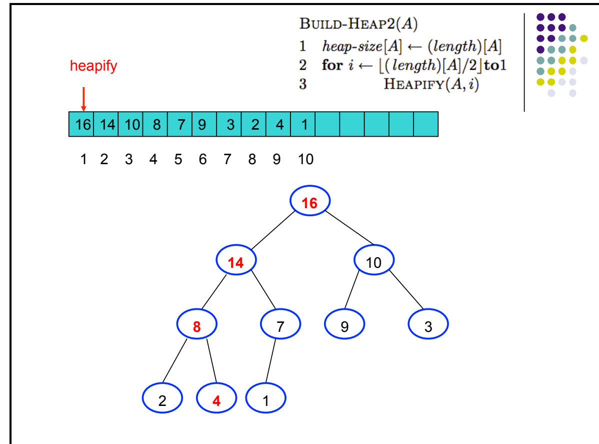

Build-Heap2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋ to 1
3 Heapify(A, i)
heapify
4 1 3 14 16 9 10 2 8 7
1 2 3 4 5 6 7 8 9 10
4
1
3
14
16
9
10
2
8
7


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




## Correctness of BuildHeap2

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

- Invariant:

## Correctness of BuildHeap2

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

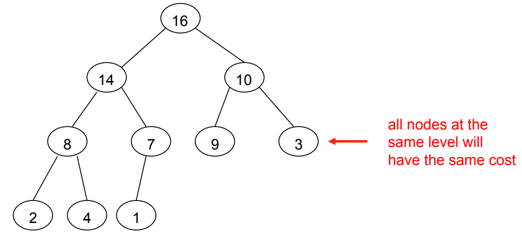
- Invariant: elements A[i+1…n] are all heaps
- Base case: i = floor(n/2). All elements i+1, i +2, …, n are “simple” heaps
- Inductive case: We know i+1, i+2, .., n are all heaps, therefore the call to Heapify(A,i) generates a heap at node i
- Termination?

## Running time of BuildHeap2

- *n*/2 calls to Heapify – O(n log n)
- Can we get a tighter bound?

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

## Running time of BuildHeap2




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

## Running time of BuildHeap2

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

?

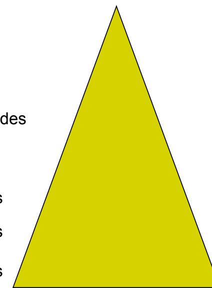
## Nodes at height *h*

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

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

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

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

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

$$= O(n)$$

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

## BuildHeap1 vs. BuildHeap2

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

```
BUILD-HEAP2(A)
1  heap-size[A] ← (length)[A]
2  for i ← ⌊(length)[A]/2⌋ to 1
3          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])

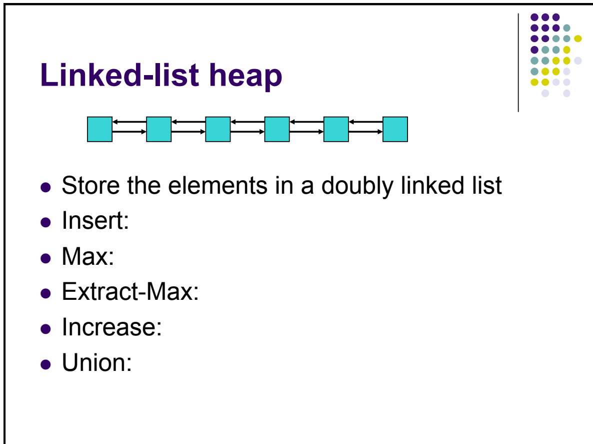
- Mergeable heaps support the union operation
- Allows us to combine two heaps to get a single heap
- Union runtime for binary heaps?

## 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 | $\Theta(n)$ |
| INCREASE-ELEMENT | $\Theta(\log n)$ |
| DELETE | $\Theta(\log n)$ |

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

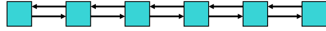
concatenate the arrays and then call Build-Heap



## 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 | $\Theta(n)$ | $\Theta(n)$ |
| INSERT | $\Theta(\log n)$ | $\Theta(1)$ |
| MAXIMUM | $\Theta(1)$ | $\Theta(n)$ |
| EXTRAC-MAX | $\Theta(\log n)$ | $\Theta(n)$ |
| UNION | $\Theta(n)$ | $\Theta(1)$ |
| INCREASE-ELEMENT | $\Theta(\log n)$ | $\Theta(1)$ |
| DELETE | $\Theta(\log n)$ | $\Theta(1)$ |

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

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

## Binomial Tree