CMP_SC 3050: Heaps

Rohit Chadha

September 4, 2014

Heaps

A heap is a nearly complete binary tree:

- The tree is completely filled except at the lowest level
- At the lowest level, the tree must be filled from the left upto a point

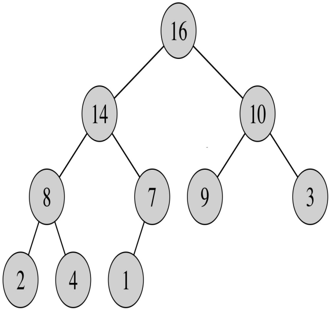
In addition, a heap must satisfy one of the following properties

1. Max-heap property: The value stored at every node must be greater than the value stored in its children

OR

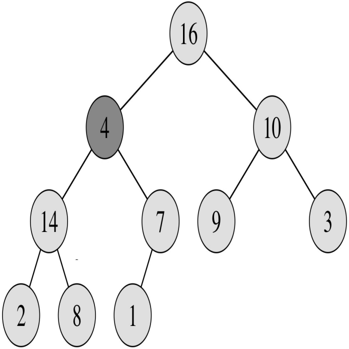
2. Min-heap property: The value stored at every node must be less than the value stored in its children

Example



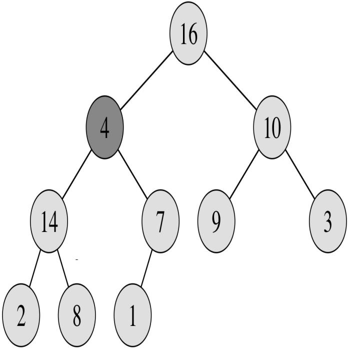
Convert a nearly complete binary tree to a max-heap

- First consider the case that at only one node the value stored is less than its children



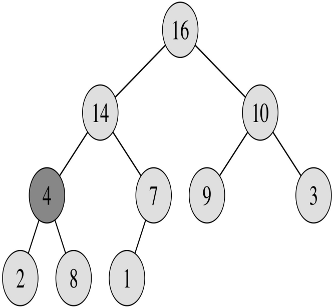
Convert a nearly complete binary tree to a max-heap

- First consider the case that at only one node the value stored is less than its children
- Swap the value at this node with the larger value of its children



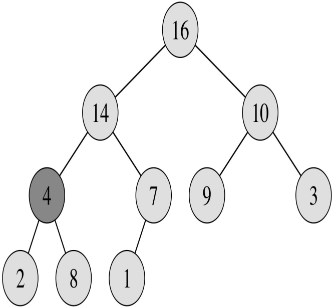
Convert a nearly complete binary tree to a max-heap

- First consider the case that at only one node the value stored is less than its children
- Swap the value at this node with the larger value of its children



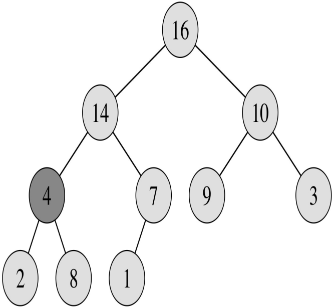
Convert a nearly complete binary tree to a max-heap

- First consider the case that at only one node the value stored is less than its children
- Swap the value at this node with the larger value of its children
- The resulting tree may not be a heap, but the error occurs at a lower level



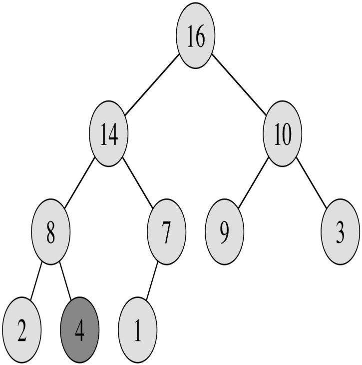
Convert a nearly complete binary tree to a max-heap

- First consider the case that at only one node the value stored is less than its children
- Swap the value at this node with the larger value of its children
- The resulting tree may not be a heap, but the error occurs at a lower level
- Continue until you get a proper heap



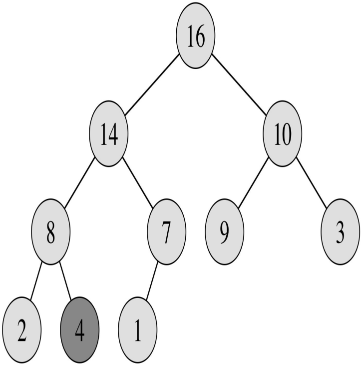
Convert a nearly complete binary tree to a max-heap

- First consider the case that at only one node the value stored is less than its children
- Swap the value at this node with the larger value of its children
- The resulting tree may not be a heap, but the error occurs at a lower level
- Continue until you get a proper heap



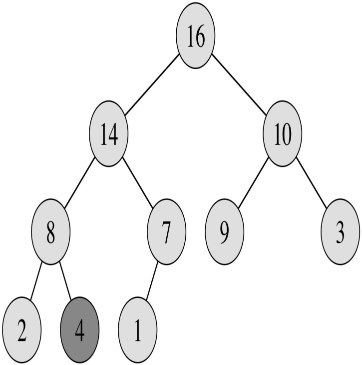
Convert a nearly complete binary tree to a max-heap

- First consider the case that at only one node the value stored is less than its children
- Swap the value at this node with the larger value of its children
- The resulting tree may not be a heap, but the error occurs at a lower level
- Continue until you get a proper heap
- This procedure is called MAX-HEAPIFY



Convert a nearly complete binary tree to a max-heap

- First consider the case that at only one node the value stored is less than its children
- Swap the value at this node with the larger value of its children
- The resulting tree may not be a heap, but the error occurs at a lower level
- Continue until you get a proper heap
- This procedure is called MAX-HEAPIFY
- What is the time complexity of Max-Heapify?
  - At each level the procedure spends a constant amount of time
  - Hence the running time is $O(h)$ if h is the height of the error



Convert a nearly complete binary tree to a max-heap continued

- Now, there might be more than one errors

Convert a nearly complete binary tree to a max-heap continued

- Now, there might be more than one errors
- Start at the bottom of the tree

Convert a nearly complete binary tree to a max-heap continued

- Now, there might be more than one errors
- Start at the bottom of the tree
- There are no errors there

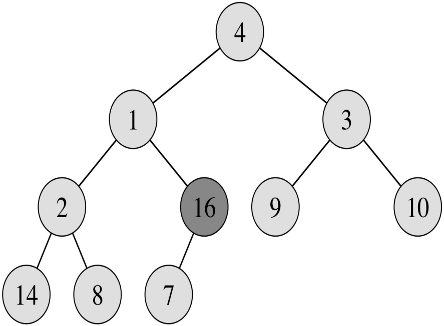
Convert a nearly complete binary tree to a max-heap continued

- Now, there might be more than one errors
- Start at the bottom of the tree
- There are no errors there
- Go one level up, fix the errors at this height starting from right to left

Convert a nearly complete binary tree to a max-heap continued

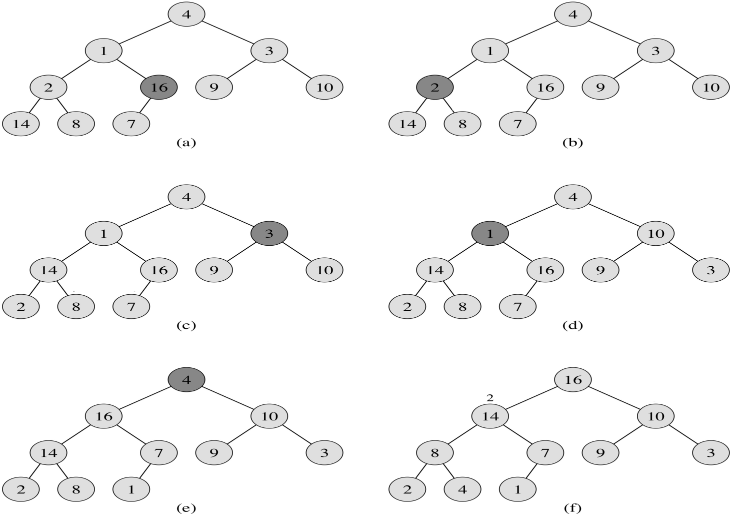
- Now, there might be more than one errors
- Start at the bottom of the tree
- There are no errors there
- Go one level up, fix the errors at this height starting from right to left
- Keep going up..
- This procedure is called BUILD-MAX-HEAP

Example: Build-Max-Heap



(a)

Example: Build-Max-Heap



*Time complexity of BUILD-MAX-HEAP

- MAX-HEAPIFY takes $O(h)$ times on nodes at height h

*Time complexity of BUILD-MAX-HEAP

- MAX-HEAPIFY takes $O(h)$ times on nodes at height h
- There are $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h

*Time complexity of Build-Max-Heap

- Max-Heapify takes $O(h)$ times on nodes at height h
- There are $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h
- Therefore, the total cost of Build-Max-Heap is

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

*Time complexity of BUILD-MAX-HEAP

- MAX-HEAPIFY takes $O(h)$ times on nodes at height h
- There are $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h
- Therefore, the total cost of BUILD-MAX-HEAP is

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

- But

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} < 2$$

*Time complexity of BUILD-MAX-HEAP

- MAX-HEAPIFY takes $O(h)$ times on nodes at height h
- There are $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h
- Therefore, the total cost of BUILD-MAX-HEAP is

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

- But

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} < 2$$

- Thus MAX-HEAPIFY takes $O(n)$ time

*Time complexity of BUILD-MAX-HEAP

- MAX-HEAPIFY takes $O(h)$ times on nodes at height h
- There are $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h
- Therefore, the total cost of BUILD-MAX-HEAP is

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

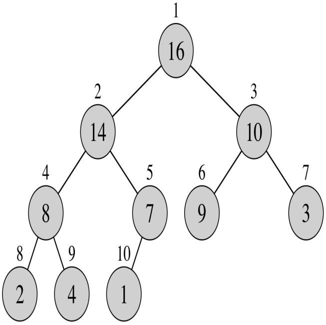
- But

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} < 2$$

- Thus MAX-HEAPIFY takes $O(n)$ time
- You should know the result

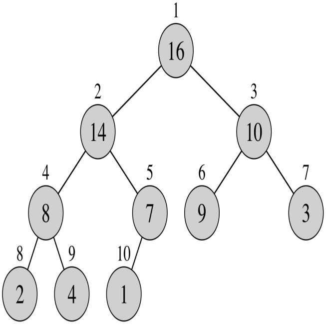
But how do we implement heaps?

- A heap of size n can be implemented as an array of size n



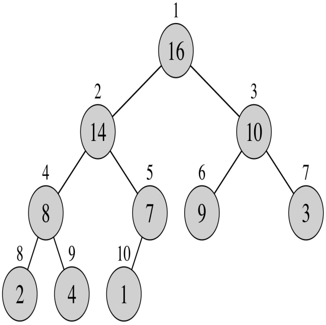
But how do we implement heaps?

- A heap of size n can be implemented as an array of size n
- We will traverse the whole heap from top level to bottom level, going from left to right at each individual level



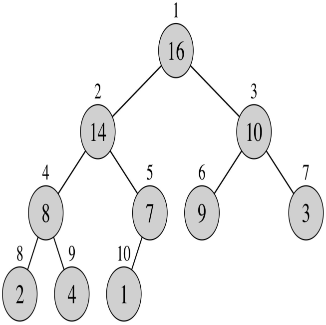
But how do we implement heaps?

- A heap of size n can be implemented as an array of size n
- We will traverse the whole heap from top level to bottom level, going from left to right at each individual level
- The node visited at the i-th step will be stored at the i-th location



But how do we implement heaps?

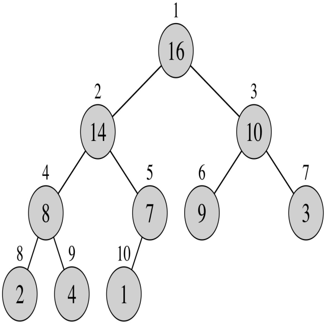
- A heap of size n can be implemented as an array of size n
- We will traverse the whole heap from top level to bottom level, going from left to right at each individual level
- The node visited at the i-th step will be stored at the i-th location

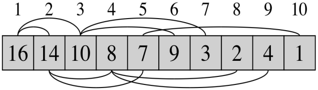


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

But how do we implement heaps?

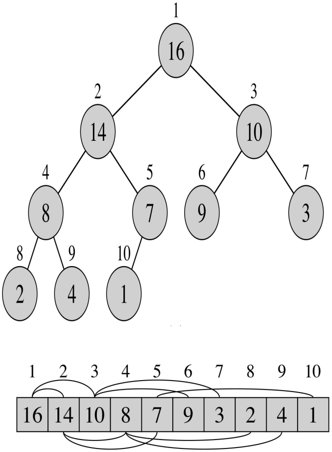
- A heap of size n can be implemented as an array of size n
- We will traverse the whole heap from top level to bottom level, going from left to right at each individual level
- The node visited at the i-th step will be stored at the i-th location
- We did this for heaps, but we can do the same thing for every nearly complete binary tree



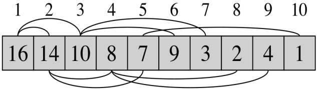


But how do we implement heaps?

- A heap of size n can be implemented as an array of size n
- We will traverse the whole heap from top level to bottom level, going from left to right at each individual level
- The node visited at the i-th step will be stored at the i-th location
- We did this for heaps, but we can do the same thing for every nearly complete binary tree
  - Infact, we can consider any array as a nearly complete binary tree!

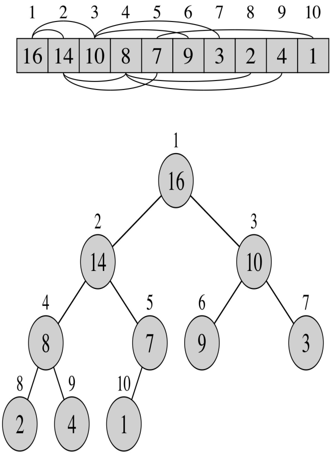


Arrays as nearly complete binary trees



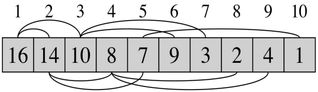
$$
\begin{array}{lcl}
\text{Root of the tree} & = & A[1] \\
\text{Left child of } A[i] & = & A[2i] \\
\text{Right child of } A[i] & = & A[2i+1] \\
\text{Parent of } A[i] & = & A[\lfloor \frac{i}{2} \rfloor]
\end{array}
$$

Arrays as nearly complete binary trees



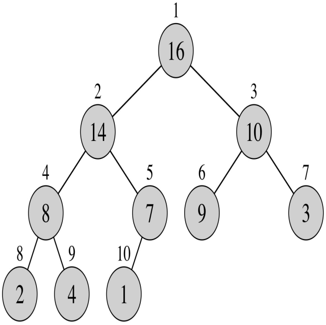
$$\begin{array}{lcl} \text{Root of the tree} & = & A[1] \\ \text{Left child of } A[i] & = & A[2i] \\ \text{Right child of } A[i] & = & A[2i+1] \\ \text{Parent of } A[i] & = & A[\lfloor \frac{i}{2} \rfloor] \end{array}$$

Arrays as nearly complete binary trees



$$\begin{aligned}
\text{Root of the tree} &= A[1] \\
\text{Left child of } A[i] &= A[2i] \\
\text{Right child of } A[i] &= A[2i+1] \\
\text{Parent of } A[i] &= A[\lfloor \tfrac{i}{2} \rfloor]
\end{aligned}$$

$$\begin{aligned}
\textsc{Left}[i] &= 2i \\
\textsc{Right}[i] &= 2i+1 \\
\textsc{Parent}[i] &= \lfloor \tfrac{i}{2} \rfloor
\end{aligned}$$



Pseudocode for MAX-HEAPIFY

```
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)
```

Makes the subtree rooted at $A[i]$ a heap if the subtrees rooted at $A[\text{LEFT}(i)]$ and $A[\text{RIGHT}(i)]$ are heaps

Pseudocode for Build-Max-Heap

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

Makes the nearly complete binary tree stored in $A[1 \,..\, n]$ a heap

An application of Heaps: Sorting

Given an array, the Heapsort algorithm on an array A of size n acts as follows:

- Convert array A into a heap A (use $\textsc{Build-Max-Heapify}$)

An application of Heaps: Sorting

Given an array, the Heapsort algorithm on an array A of size n acts as follows:

- Convert array A into a heap A (use Build-Max-Heapify)
- The root is the maximum element. Swap it with the element in the last position in the array.

An application of Heaps: Sorting

Given an array, the Heapsort algorithm on an array A of size n acts as follows:

- Convert array A into a heap A (use BUILD-MAX-HEAPIFY)
- The root is the maximum element. Swap it with the element in the last position in the array.
- The maximum is in the correct place

An application of Heaps: Sorting

Given an array, the Heapsort algorithm on an array A of size n acts as follows:

- Convert array A into a heap A (use Build-Max-Heapify)
- The root is the maximum element. Swap it with the element in the last position in the array.
- The maximum is in the correct place
- The new $A[1 \,..\, n-1]$ is a nearly complete binary tree

An application of Heaps: Sorting

Given an array, the Heapsort algorithm on an array A of size n acts as follows:

- Convert array A into a heap A (use $\textsc{Build-Max-Heapify}$)
- The root is the maximum element. Swap it with the element in the last position in the array.
- The maximum is in the correct place
- The new $A[1\,..\,n-1]$ is a nearly complete binary tree
- The $A[1\,..\,n-1]$ is also almost a heap except that the root may be smaller than its children

An application of Heaps: Sorting

Given an array, the Heapsort algorithm on an array A of size n acts as follows:

- Convert array A into a heap A (use $\textsc{Build-Max-Heapify}$)
- The root is the maximum element. Swap it with the element in the last position in the array.
- The maximum is in the correct place
- The new $A[1\,..\,n-1]$ is a nearly complete binary tree
- The $A[1\,..\,n-1]$ is also almost a heap except that the root may be smaller than its children
- Call $\textsc{Max-Heapify}$ to make $A[1\,..\,n-1]$ a heap

An application of Heaps: Sorting

Given an array, the Heapsort algorithm on an array A of size n acts as follows:

- Convert array A into a heap A (use BUILD-MAX-HEAPIFY)
- The root is the maximum element. Swap it with the element in the last position in the array.
- The maximum is in the correct place
- The new $A[1..n-1]$ is a nearly complete binary tree
- The $A[1..n-1]$ is also almost a heap except that the root may be smaller than its children
- Call MAX-HEAPIFY to make $A[1..n-1]$ a heap
- Repeat until only one node remains remains

Pseudocode and analysis of heap sort

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

Pseudocode and analysis of heap sort

HeapSort(A, n)	**cost**	**times**
Build-Max-Heap(A, n)	$O(n)$	1
for $i = n$ **downto** 2		
exchange $A[1]$ with $A[i]$		
Max-Heapify$(A, 1, i - 1)$		

Pseudocode and analysis of heap sort

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

Pseudocode and analysis of heap sort

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

Pseudocode and analysis of heap sort

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

Pseudocode and analysis of heap sort

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

Heapsort takes $O(n \log n)$ time

A new data structure: Max Priority Queues

- Like stacks, queues and lists, maintains a dynamic set
- Each element has a *key*. May have other data

Operations

$\textsc{Insert}(A, x)$:	inserts element x into priority queue A
$\textsc{Maximum}(A)$:	returns element of A with largest key
$\textsc{Extract-Max}(A)$:	removes and returns element of A with largest key
$\textsc{Increase-Key}(A, x, k)$:	increases value of element x's key to k

A new data structure: Max Priority Queues

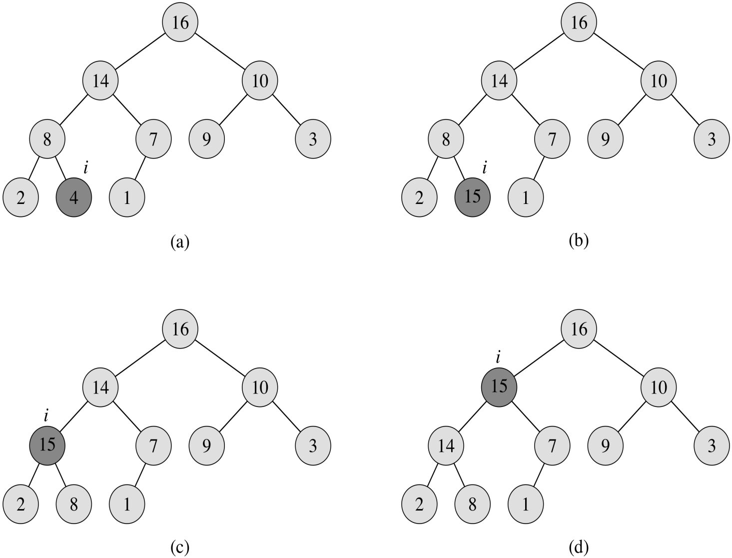
- Like stacks, queues and lists, maintains a dynamic set
- Each element has a *key*. May have other data

Operations

INSERT(A, x):	inserts element x into priority queue A
MAXIMUM(A):	returns element of A with largest key
EXTRACT-MAX(A):	removes and returns element of A with largest key
INCREASE-KEY(A, x, k):	increases value of element x's key to k

Can be implemented using heaps

Example INCREASE-KEY(A, x, k): Increase key 4 to 15



INCREASE-KEY(S, x, k)

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 $A[\text{PARENT}(i)]$
  $i = \text{PARENT}(i)$

- Make sure k is bigger than x's current key
- Update x's key value to k
- Traverse the heap upward comparing x to its parent and swapping keys until x's key is smaller than its parent's key

INCREASE-KEY(S, x, k)

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 $A[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

- Make sure k is bigger than x's current key
- Update x's key value to k
- Traverse the heap upward comparing x to its parent and swapping keys until x's key is smaller than its parent's key
- Time complexity is $O(\log n)$

Conclusion

- We learnt a new useful data structure: Heaps
- Heaps can be used to sort in $O(n \log n)$ time
- Heaps can implement Priority Queues with insertion, deletion and changing keys operations all taking $O(\log n)$ time
- We just finished Chapter 6 of the book

Next class: We start with graph algorithms (Chapter 22)