Design and Analysis of Algorithms

Heap Sort

Lecture 9-10

Instructor: Dr. G P Gupta

Heapsort

- Combines the better attributes of merge sort and insertion sort.
  - » Like merge sort, but unlike insertion sort, running time is $O(n \lg n)$.
  - » Like insertion sort, but unlike merge sort, ***sorts in place.***
- Introduces an algorithm design technique
  - » Create data structure (*heap*) to manage information during the execution of an algorithm.
- The *heap* has other applications beside sorting.
  - » Priority Queues

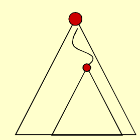
Binary heap

- 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

Heap Property (Max and Min)



- **Max-Heap**
  - » For every node excluding the root,

    $A[parent[i]] \geq A[i]$
- **Largest element is stored at the root.**
- In any subtree, no values are larger than the value stored at subtree root.
- **Min-Heap**
  - » For every node excluding the root,

    $A[parent[i]] \leq A[i]$
- **Smallest element is stored at the root.**
- In any subtree, no values are smaller than the value stored at subtree root

Binary Heap

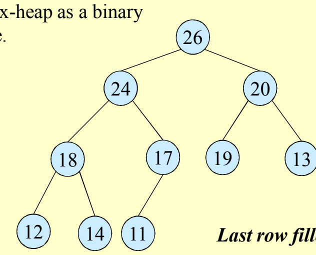
- **Array viewed as a nearly complete binary tree.**
  - » Physically – linear array.
  - » Logically – binary tree, filled on all levels (except lowest.)
- **Map from array elements to tree nodes and vice versa**
  - » Root – $A[1]$
  - » Left[i] – $A[2i]$
  - » Right[i] – $A[2i+1]$
  - » Parent[i] – $A[\lfloor i/2 \rfloor]$
- length[A] – number of elements in array A.
- heap-size[A] – number of elements in heap stored in A.
  - » heap-size[A] $\leq$ length[A]

Heaps – Example

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

Max-heap as an array.

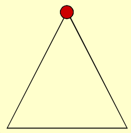
Max-heap as a binary tree.



Last row filled from left to right.

Height

- ***Height of a node in a tree*:** the number of edges on the longest simple downward path from the node to a leaf.
- ***Height of a tree*:** the height of the root.
- Height of a heap: $\lfloor \lg n \rfloor$
  - » Basic operations on a heap run in $O(\lg n)$ time



Binary heap - operations

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

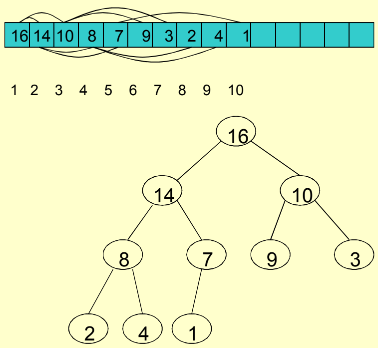
ExtractMax(S) – Return and remove the largest element in the set

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 ← 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)
```

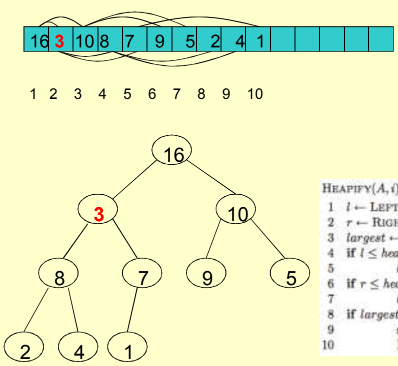

Heapify

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)
```

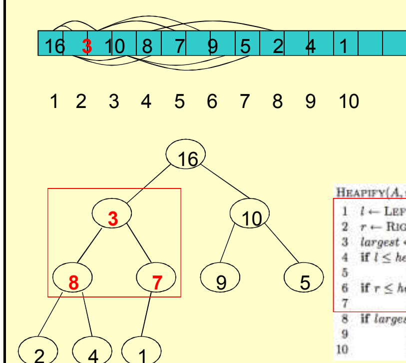

if a child is larger, swap and recurse

Heapify



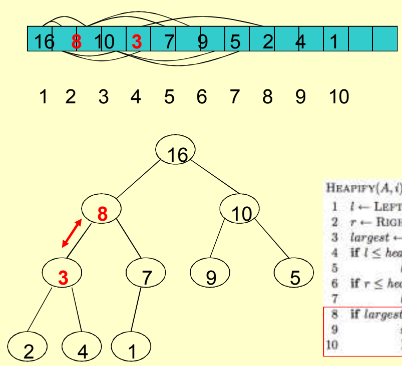

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```


Heapify



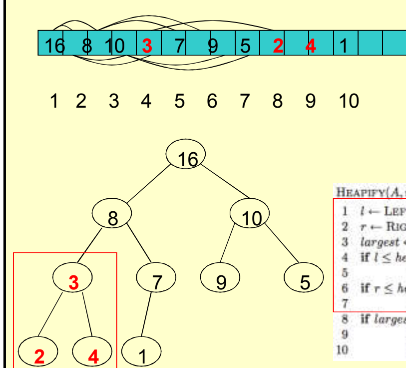

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```


Heapify



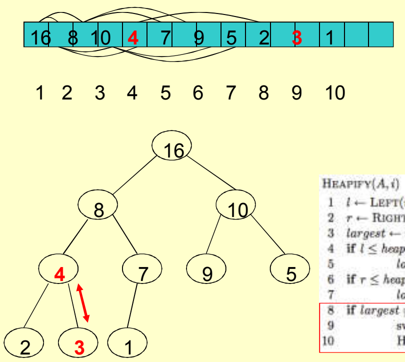

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```


Heapify



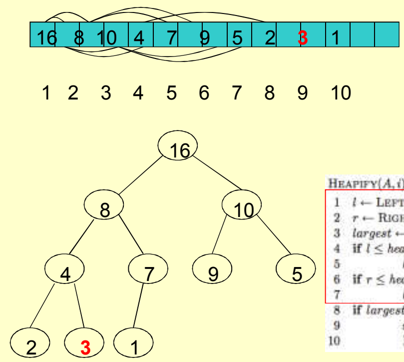

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
largest ← i
if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

Heapify



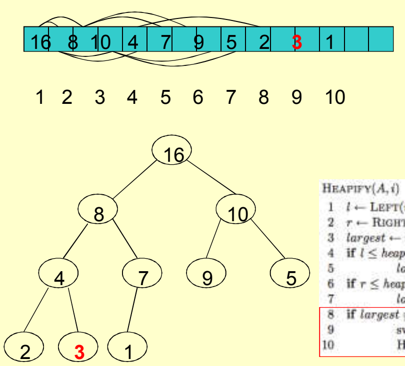

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  largest ← i
  if l ≤ heap-size[A] and A[l] > A[i]
        largest ← l
  if r ≤ heap-size[A] and A[r] > A[largest]
        largest ← r
  if largest ≠ i
        swap A[i] and A[largest]
        HEAPIFY(A, largest)
```

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)?

» $\Theta(1)$

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)
 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

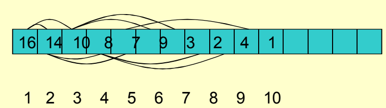
IncreaseElement(S, x, val) – increase the value of element x to val

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

Maximum

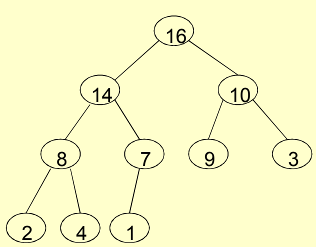
Return the largest element from the set

Return A[1]



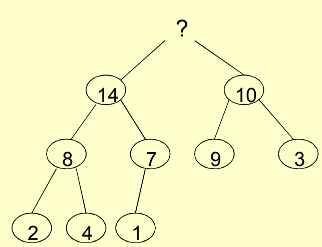
ExtractMax

Return and remove the largest element in the set



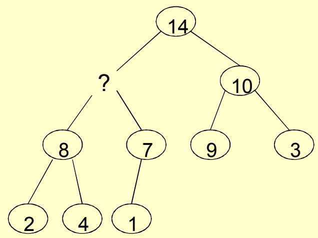
ExtractMax

Return and remove the largest element in the set



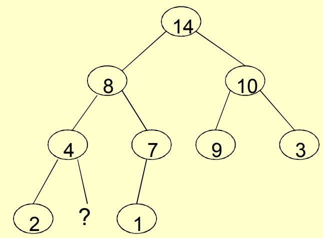
ExtractMax

Return and remove the largest element in the set



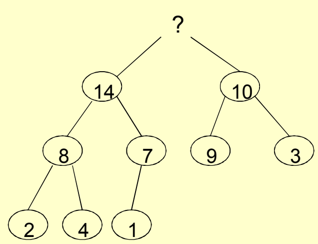
ExtractMax

Return and remove the largest element in the set



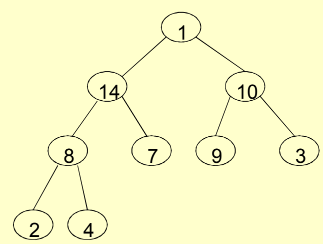
ExtractMax

Return and remove the largest element in the set



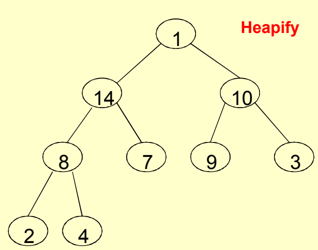
ExtractMax

Return and remove the largest element in the set



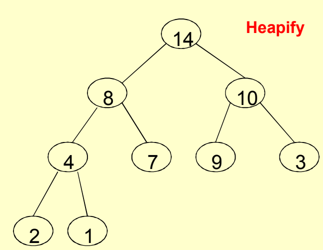
ExtractMax

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

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

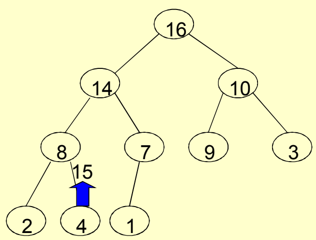
ExtractMax running time

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

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

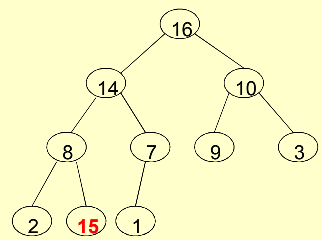
IncreaseElement

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



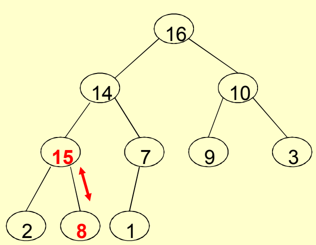
IncreaseElement

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



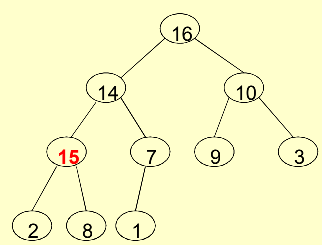
IncreaseElement

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



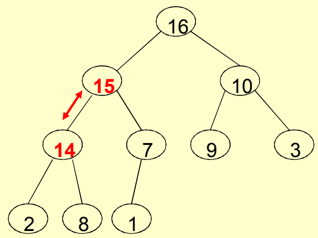
IncreaseElement

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

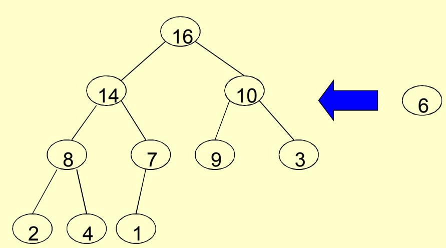
Worst case O(height of the tree)

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)
```

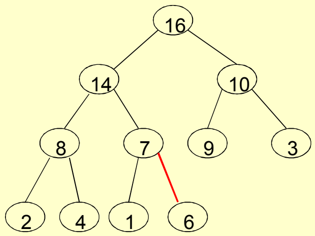
Insert

Insert *val* into the set



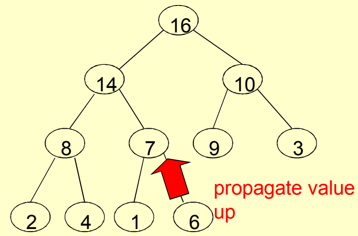
Insert

Insert *val* into the set



Insert

Insert *val* into the set



Insert

```
INSERT(A, val)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← −∞
INCREASE-ELEMENT(A, heap-size[A], val)
```

Running time of Insert

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

```
INSERT(A, val)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← −∞
INCREASE-ELEMENT(A, heap-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)

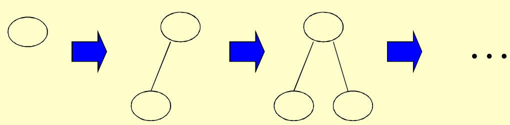
Building a heap

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

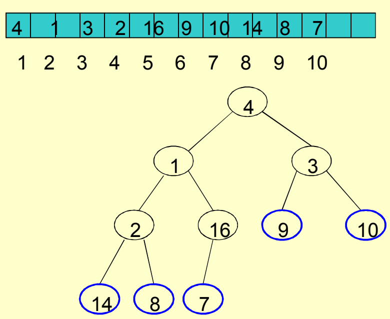
Start with n/2 "simple" heaps

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

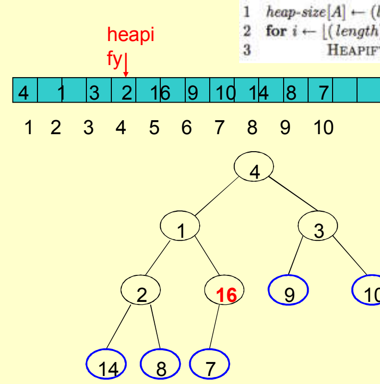
all children have smaller indices

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

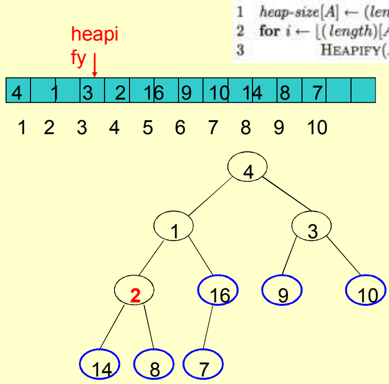
```
BUILD-HEAP2(A)
1  heap-size[A] ← (length)[A]
2  for i ← ⌊(length)[A]/2⌋ to 1
3        HEAPIFY(A, i)
```



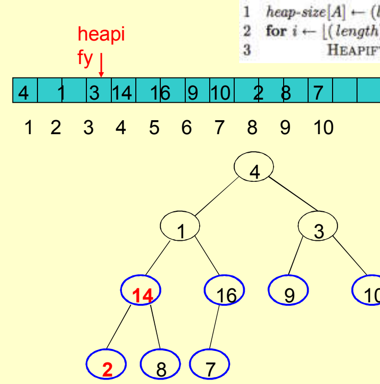

```
BUILD-HEAP2(A)
1  heap-size[A] ← (length)[A]
2  for i ← ⌊(length)[A]/2⌋ to 1
3        HEAPIFY(A, i)
```

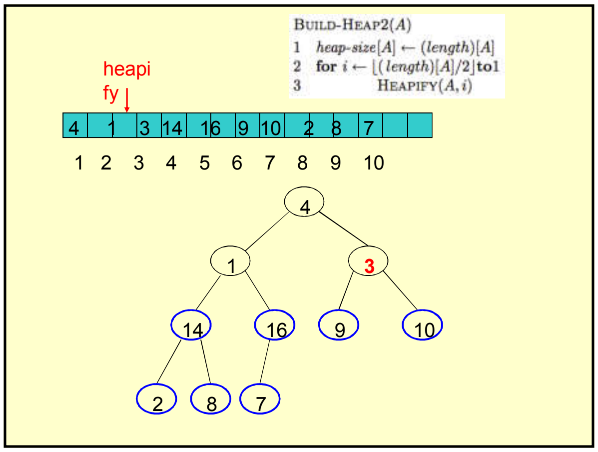


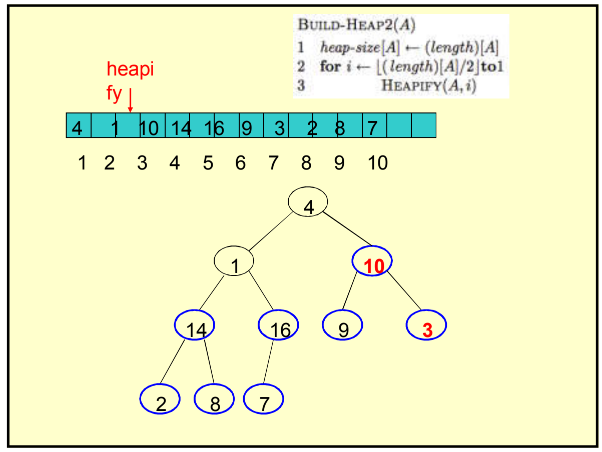

```
BUILD-HEAP2(A)
1  heap-size[A] ← (length)[A]
2  for i ← ⌊(length)[A]/2⌋ to 1
3        HEAPIFY(A, i)
```

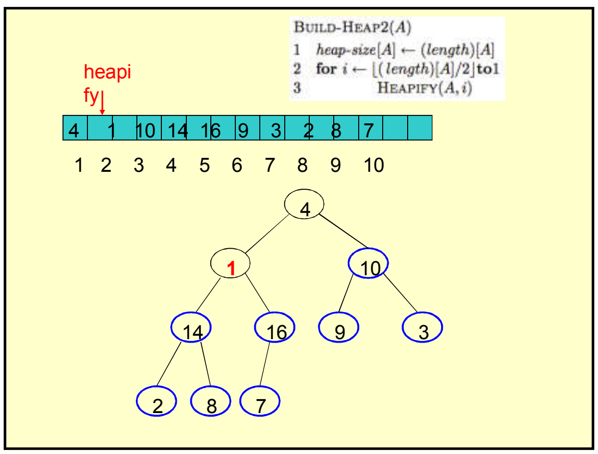


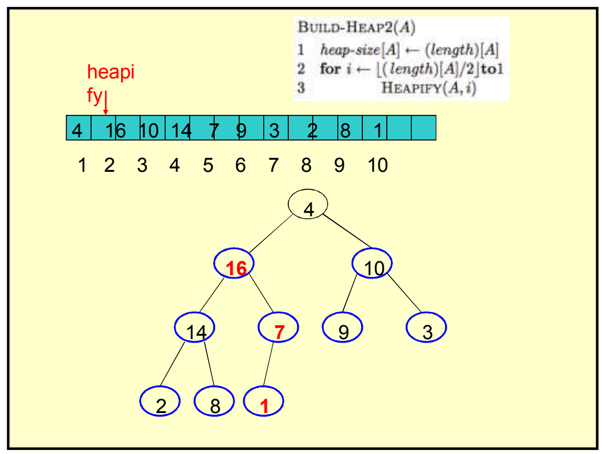

```
BUILD-HEAP2(A)
1  heap-size[A] ← (length)[A]
2  for i ← ⌊(length)[A]/2⌋ to 1
3        HEAPIFY(A, i)
```

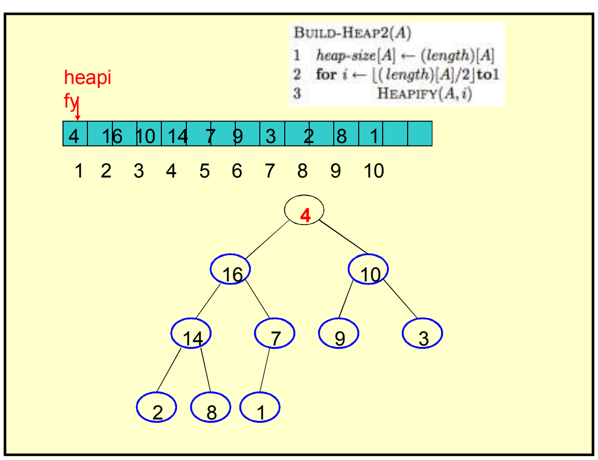


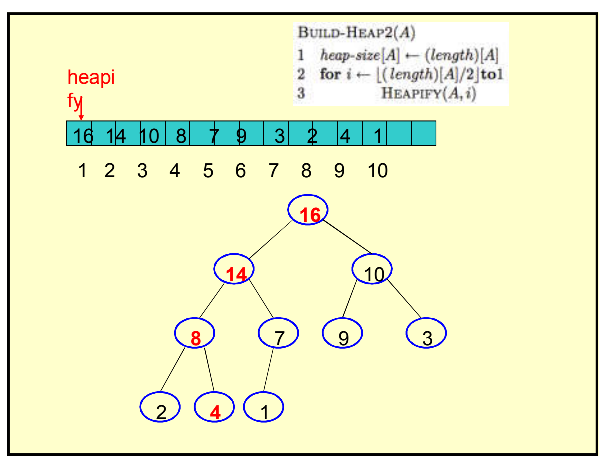

BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋to1
3 HEAPIFY(A, i)
heapi
fy
4 1 3 14 16 9 10 2 8 7
1 2 3 4 5 6 7 8 9 10


BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋to1
3 HEAPIFY(A, i)
heapi
fy
4 1 10 14 16 9 3 2 8 7
1 2 3 4 5 6 7 8 9 10


BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋to1
3 HEAPIFY(A, i)
heapi
fy
4 1 10 14 16 9 3 2 8 7
1 2 3 4 5 6 7 8 9 10


BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋to1
3 HEAPIFY(A, i)
heapi
fy
4 16 10 14 7 9 3 2 8 1
1 2 3 4 5 6 7 8 9 10


BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋to1
3 HEAPIFY(A, i)
heapi
fy
4 16 10 14 7 9 3 2 8 1
1 2 3 4 5 6 7 8 9 10


BUILD-HEAP2(A)
1 heap-size[A] ← (length)[A]
2 for i ← ⌊(length)[A]/2⌋to1
3 HEAPIFY(A, i)
heapi
fy
16 14 10 8 7 9 3 2 4 1
1 2 3 4 5 6 7 8 9 10

Correctness of BuildHeap2

Invariant:

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

Correctness of BuildHeap2

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

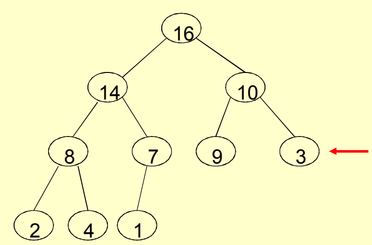
Running time of BuildHeap2

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



all nodes at the same level will have the same cost

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

Running time of BuildHeap2

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

?

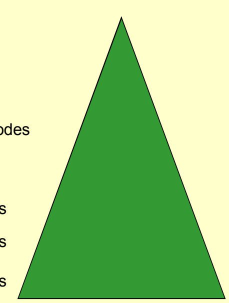
Nodes at height *h*

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

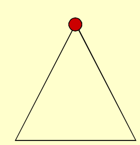
- scheduling tasks: jobs, processes, network traffic
- A* search algorithm

Heaps in Sorting

- **Use max-heaps for sorting.**
- **The array representation of max-heap is not sorted.**
- **Steps in sorting**
  - » Convert the given array of size n to a max-heap (*BuildMaxHeap*)
  - » Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves $n - 1$ elements to be placed in their appropriate locations.
    - However, the array of first $n - 1$ elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (MaxHeapify)
    - Repeat step 2 until the array is sorted.

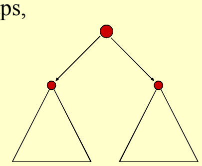
Heap Characteristics

- *Height* $= \lfloor \lg n \rfloor$
- No. of *leaves* $= \lceil n/2 \rceil$
- No. of nodes of height h $\leq \lceil n/2^{h+1} \rceil$



Maintaining the heap property

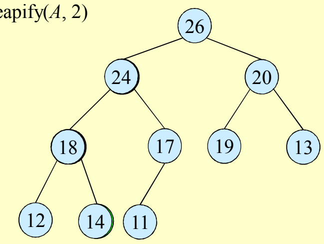
- Suppose two subtrees are max-heaps, but the root violates the max-heap property.



- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - » May lead to the subtree at the child not being a heap.
- Recursively fix the children until all of them satisfy the max-heap property.

MaxHeapify – Example

MaxHeapify(A, 2)



Procedure MaxHeapify

```
MaxHeapify(A, i)
l ← left(i)
r ← right(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest≠ i
   then exchange A[i] ↔ A[largest]
        MaxHeapify(A, largest)
```

Assumption:
Left(i) and Right(i) are max-heaps.

Running Time for MaxHeapify

```
MaxHeapify(A, i)
l ← left(i)
r ← right(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest≠ i
   then exchange A[i] ↔ A[largest]
        MaxHeapify(A, largest)
```

Time to fix node i and its children = $\Theta(1)$

PLUS

Time to fix the subtree rooted at one of i's children = T(size of subree at *largest*)

Running Time for MaxHeapify(A, n)

- $T(n) = T(largest) + \Theta(1)$
- $largest \leq 2n/3$ (worst case occurs when the last row of tree is exactly half full)
- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$
- Alternately, MaxHeapify takes $O(h)$ where h is the height of the node where MaxHeapify is applied

Building a heap

- Use *MaxHeapify* to convert an array A into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

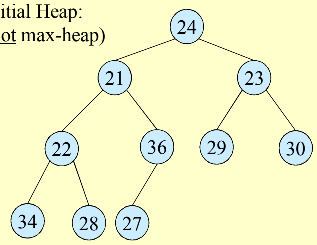
```
BuildMaxHeap(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    do MaxHeapify(A, i)
```

BuildMaxHeap – Example

Input Array:

24	21	23	22	36	29	30	34	28	27

Initial Heap:
(<u>not</u> max-heap)



BuildMaxHeap – Example

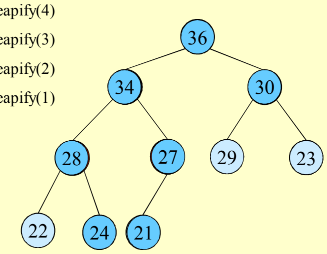
MaxHeapify(⌊10/2⌋ = 5)

MaxHeapify(4)

MaxHeapify(3)

MaxHeapify(2)

MaxHeapify(1)



Correctness of *BuildMaxHeap*

- <u>**Loop Invariant:**</u> At the start of each iteration of the **for** loop, each node $i+1$, $i+2$, …, n is the root of a max-heap.
- <u>**Initialization:**</u>
  - » Before first iteration $i = \lfloor n/2 \rfloor$
  - » Nodes $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, …, n are leaves and hence roots of max-heaps.
- <u>**Maintenance:**</u>
  - » By LI, subtrees at children of node i are max heaps.
  - » Hence, MaxHeapify(i) renders node i a max heap root (while preserving the max heap root property of higher-numbered nodes).
  - » Decrementing i reestablishes the loop invariant for the next iteration.

Running Time of *BuildMaxHeap*

- Loose upper bound:
  - » Cost of a *MaxHeapify* call × No. of calls to *MaxHeapify*
  - » $O(\lg n) \times O(n) = O(n \lg n)$
- Tighter bound:
  - » Cost of a call to *MaxHeapify* at a node depends on the height, h, of the node – $O(h)$.
  - » Height of most nodes smaller than n.
  - » Height of nodes h ranges from 0 to $\lfloor \lg n \rfloor$.
  - » No. of nodes of height h is $\lceil n/2^{h+1} \rceil$

Running Time of *BuildMaxHeap*

Tighter Bound for $T(BuildMaxHeap)$

$T(BuildMaxHeap)$

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

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \le \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 \qquad , x = 1/2 \text{ in (A.8)}$$

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

Can build a heap from an unordered array in linear time

Heapsort

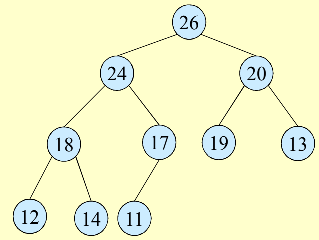
- Sort by maintaining the as yet unsorted elements as a max-heap.
- Start by building a max-heap on all elements in A.
  - » Maximum element is in the root, $A[1]$.
- Move the maximum element to its correct final position.
  - » Exchange $A[1]$ with $A[n]$.
- Discard $A[n]$ – it is now sorted.
  - » Decrement heap-size[A].
- Restore the max-heap property on $A[1..n–1]$.
  - » Call *MaxHeapify*(A, 1).
- Repeat until heap-size[A] is reduced to 2.

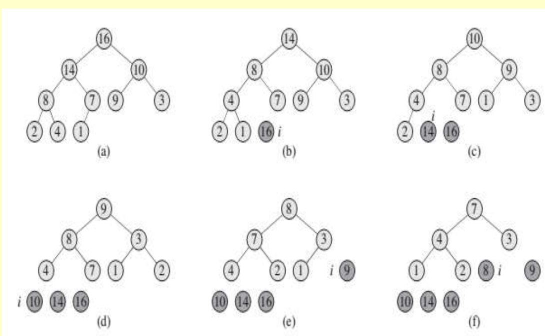
Heapsort(A)

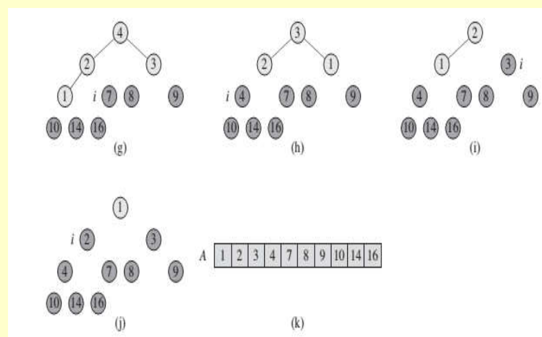
HeapSort(A)
1. Build-Max-Heap(A)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5. $MaxHeapify(A, 1)$

Heapsort – Example

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10







Algorithm Analysis

HeapSort(A)
1. Build-Max-Heap(A)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1$
5. $MaxHeapify(A, 1)$

- In-place
- Not Stable
- Build-Max-Heap takes $O(n)$ and each of the n-1 calls to Max-Heapify takes time $O(\lg n)$.
- Therefore, $T(n) = O(n \lg n)$

Heap Procedures for Sorting

- MaxHeapify $O(\lg n)$
- BuildMaxHeap $O(n)$
- HeapSort $O(n \lg n)$

Priority Queue

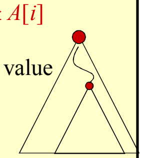
- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set *S* of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficiently.
- Applications:
  - » Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - » In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

Basic Operations

- Operations on a max-priority queue:
  - » Insert(S, x) - inserts the element x into the set S
    - $S \leftarrow S \cup \{x\}$.
  - » Maximum(S) - returns the element of S with the largest key.
  - » Extract-Max(S) - removes and returns the element of S with the largest key.
  - » Increase-Key(S, x, k) – increases the value of element x's key to the new value k.
- Min-priority queue supports Insert, Minimum, Extract-Min, and Decrease-Key.
- Heap gives a good compromise between fast insertion but slow extraction and vice versa.

Heap Property (Max and Min)

- Max-Heap
  - » For every node excluding the root, value is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.



- Min-Heap
  - » For every node excluding the root, value is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root

Heap-Extract-Max(A)

Implements the Extract-Max operation.

```
Heap-Extract-Max(A)
if heap-size[A] < 1
   then error "heap underflow"
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
MaxHeapify(A, 1)
return max
```

Running time : Dominated by the running time of MaxHeapify
$= O(\lg n)$

Heap-Insert(A, key)

```
Heap-Insert(A, key)
heap-size[A] ← heap-size[A] + 1
   i ← heap-size[A]
while i > 1 and A[Parent(i)] < key
   do A[i] ← A[Parent(i)]
        i ← Parent(i)
A[i] ← key
```

Running time is $O(\lg n)$

The path traced from the new leaf to the root has length $O(\lg n)$

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

```
Heap-Increase-Key(A, i, key)
If key < A[i]
   then error "new key is smaller than the current key"
A[i] ← key
while i > 1 and A[Parent[i]] < A[i]
      do exchange A[i] ↔ A[Parent[i]]
           i ← Parent[i]
```

```
Heap-Insert(A, key)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← –∞
Heap-Increase-Key(A, heap-size[A], key)
```

Examples

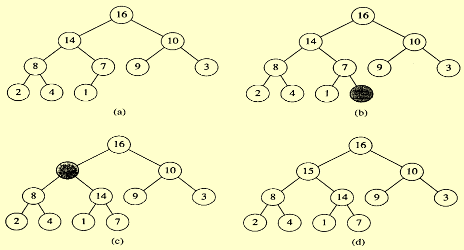


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.