Unit-V Heap Data Structure

1

What is a "heap"?

- Definitions of heap:
  1. A large area of memory from which the programmer can allocate blocks as needed, and deallocate them (or allow them to be garbage collected) when no longer needed
  2. A balanced, left-justified binary tree in which no node has a value greater than the value in its parent
- These two definitions have little in common
- Heapsort uses the second definition

2

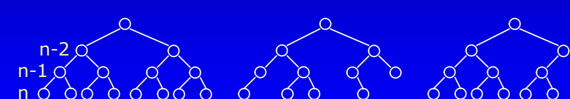
Why study Heapsort?

- It is a well-known, traditional sorting algorithm you will be expected to know
- Heapsort is *always* O(n log n)
  - Quicksort is usually O(n log n) but in the worst case slows to $O(n^2)$
  - Quicksort is generally faster, but Heapsort is better in time-critical applications
- Heapsort is a *really cool* algorithm!

3

Balanced binary trees

- Recall:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
- A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children

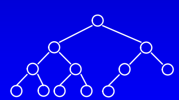


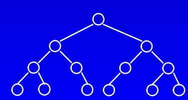
Balanced Balanced Not balanced

4

Left-justified binary trees

- A balanced binary tree is left-justified if:
  - all the leaves are at the same depth, or
  - all the leaves at depth `n+1` are to the left of all the nodes at depth `n`


Left-justified


Not left-justified

Heaps

A **heap** is a certain kind of complete binary tree.

Heaps

Root

A **heap** is a certain kind of complete binary tree.

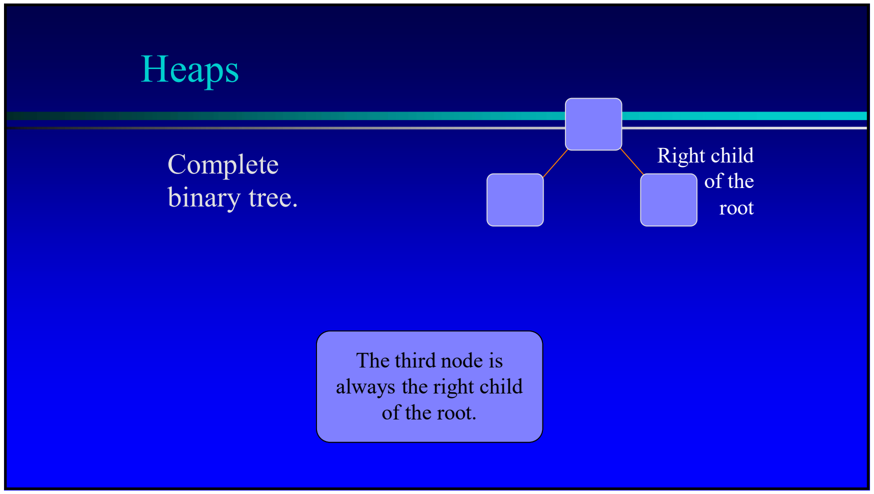
When a complete binary tree is built, its first node must be the root.

Heaps

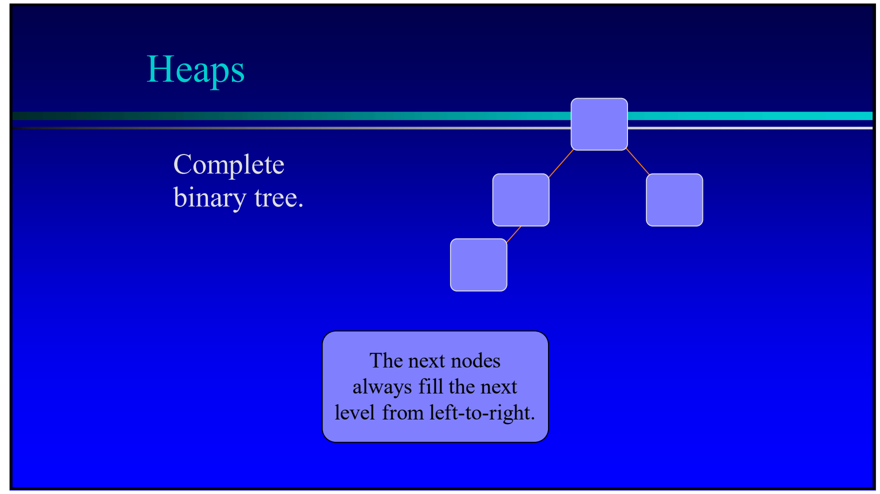
Complete binary tree.

Left child of the root

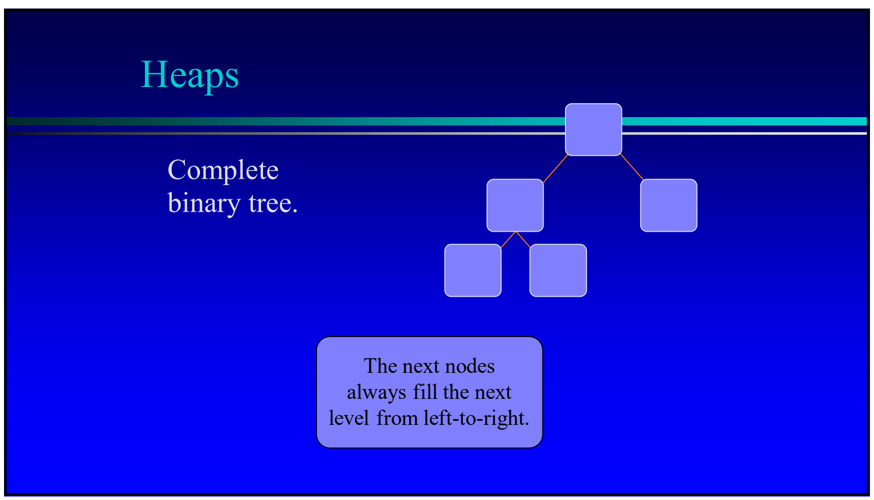
The second node is always the left child of the root.


Heaps
Complete
binary tree.
Right child
of the
root
The third node is
always the right child
of the root.

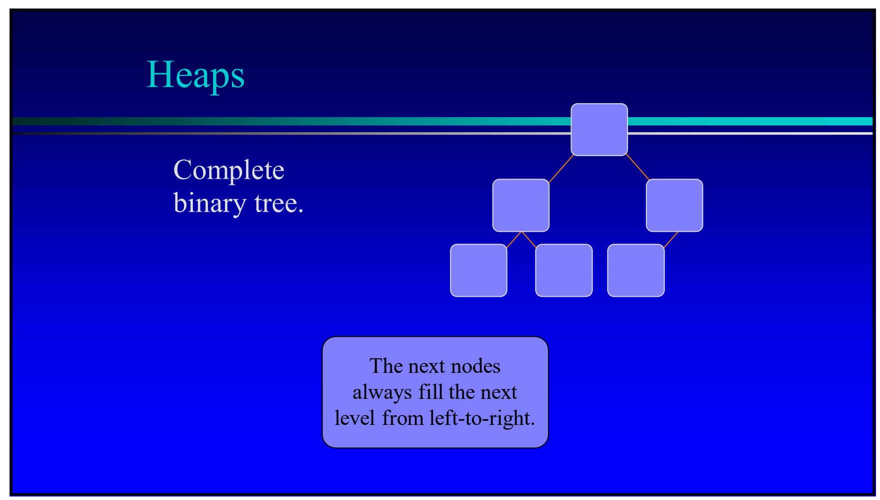
9


Heaps
Complete
binary tree.
The next nodes
always fill the next
level from left-to-right.

10


Heaps
Complete
binary tree.
The next nodes
always fill the next
level from left-to-right.

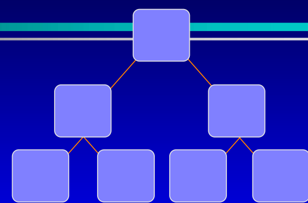
11


Heaps
Complete
binary tree.
The next nodes
always fill the next
level from left-to-right.

12

Heaps

Complete
binary tree.

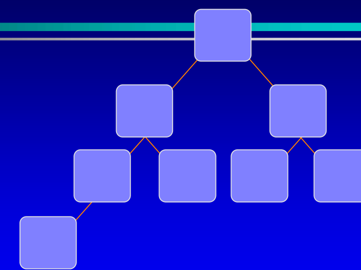


The next nodes always fill the next level from left-to-right.

13

Heaps

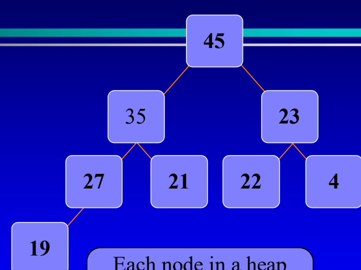
Complete
binary tree.



14

Heaps

A heap is a **certain** kind of complete binary tree.

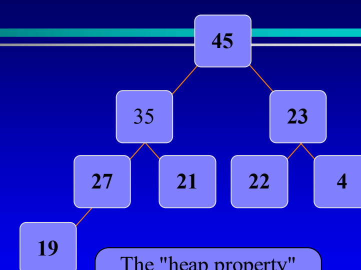


Each node in a heap contains a key that can be compared to other nodes' keys.

15

Heaps

A heap is a **certain** kind of complete binary tree.

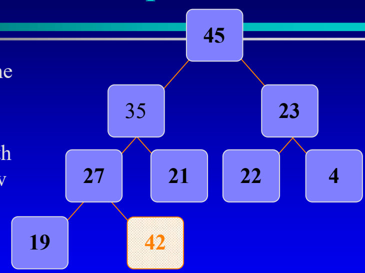


The "heap property" requires that each node's key is >= the keys of its children

16

Adding a Node to a Heap

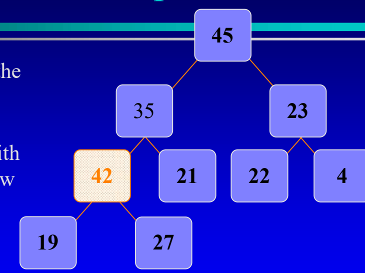
- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



17

Adding a Node to a Heap

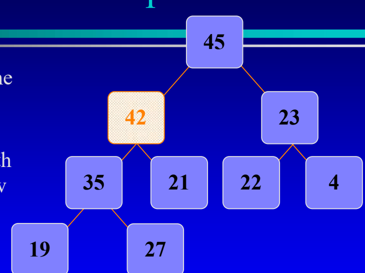
- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



18

Adding a Node to a Heap

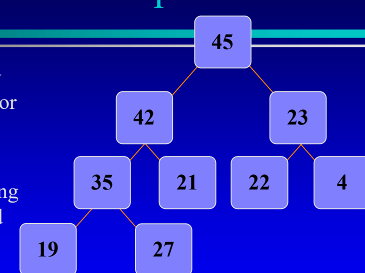
- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



19

Adding a Node to a Heap

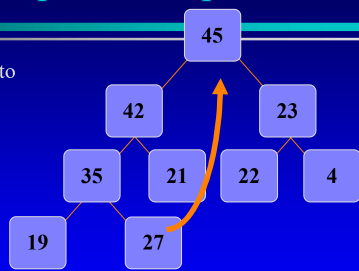
- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



20

Removing the Top of a Heap

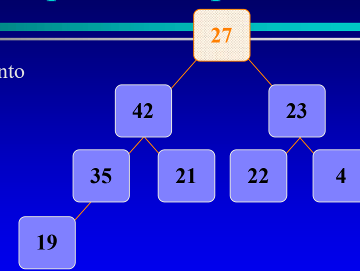
- Move the last node onto the root.



21

Removing the Top of a Heap

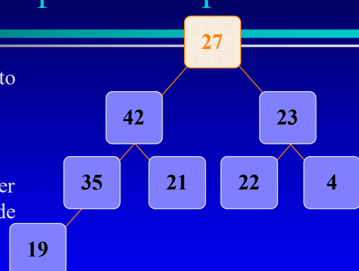
- Move the last node onto the root.



22

Removing the Top of a Heap

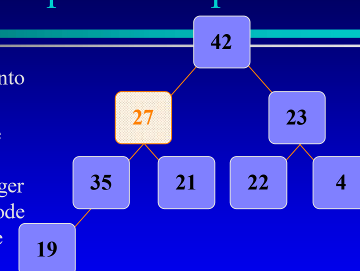
- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



23

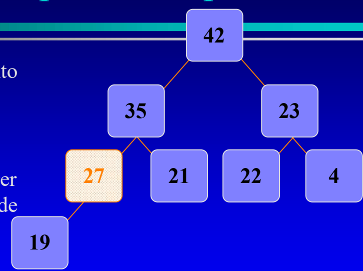
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



24

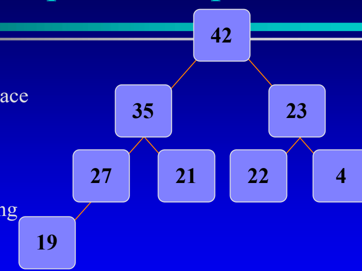
Removing the Top of a Heap



- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.

25

Removing the Top of a Heap

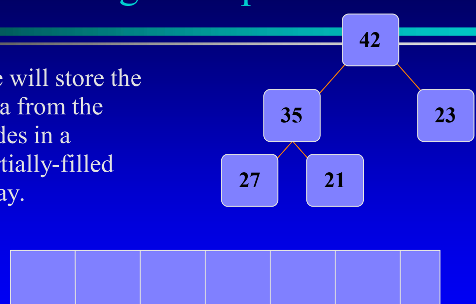


- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.

26

Implementing a Heap

- We will store the data from the nodes in a partially-filled array.

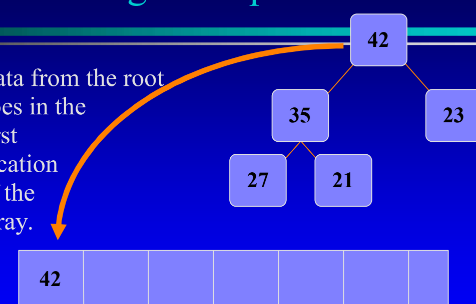


An array of data

27

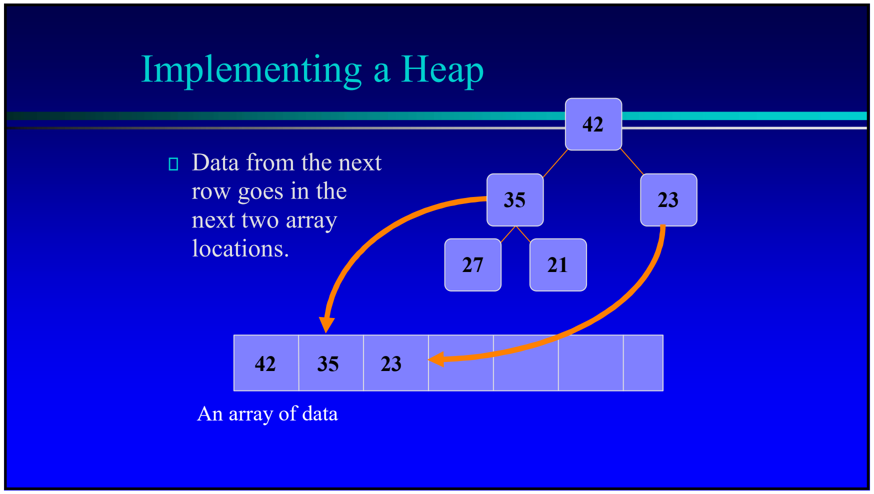
Implementing a Heap

- Data from the root goes in the first location of the array.

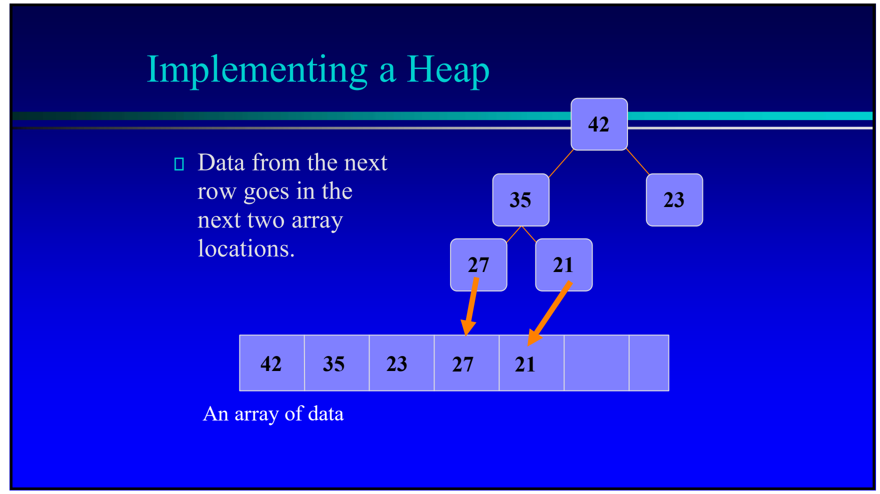


An array of data

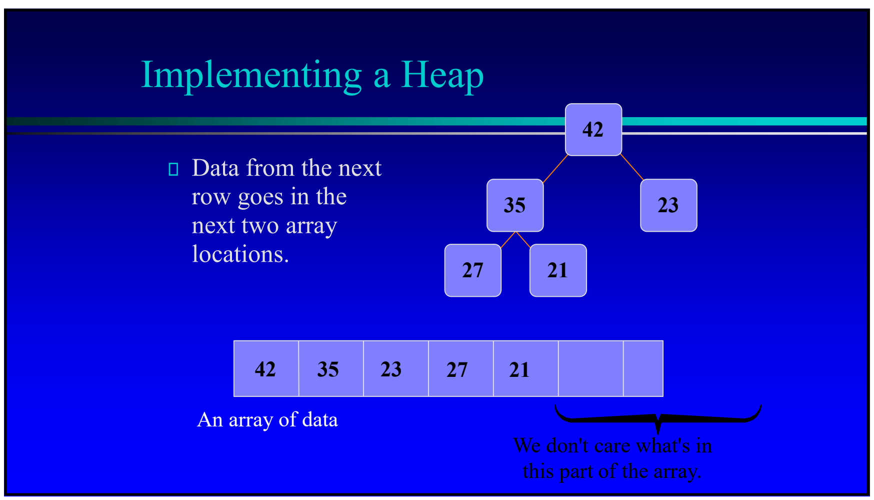
28


Implementing a Heap
Data from the next row goes in the next two array locations.
42
35
23
27
21
42 35 23
An array of data

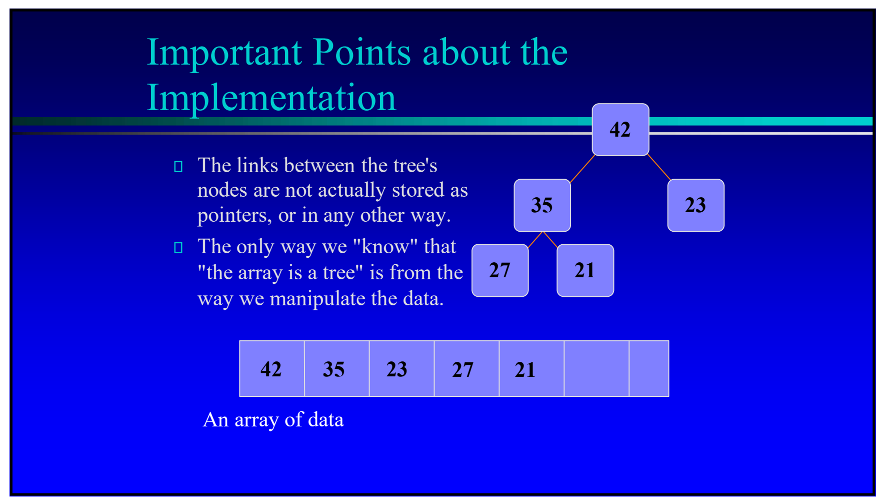
29


Implementing a Heap
Data from the next row goes in the next two array locations.
42
35
23
27
21
42 35 23 27 21
An array of data

30


Implementing a Heap
Data from the next row goes in the next two array locations.
42
35
23
27
21
42 35 23 27 21
An array of data
We don't care what's in this part of the array.

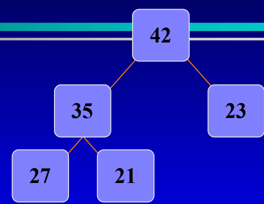
31


Important Points about the Implementation
The links between the tree's nodes are not actually stored as pointers, or in any other way.
The only way we "know" that "the array is a tree" is from the way we manipulate the data.
42
35
23
27
21
42 35 23 27 21
An array of data

32

Important Points about the Implementation

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children. Formulas are given in the book.



42	35	23	27	21		
[1]	[2]	[3]	[4]	[5]		

33

Summary

- A heap is a complete binary tree, where the entry at each node is greater than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.

34

The heap property

- A node has the heap property if the value in the node is as large as or larger than the values in its children

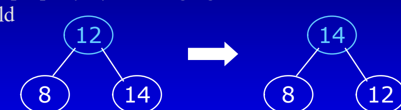
Blue node has heap property | Blue node has heap property | Blue node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a heap if *all* nodes in it have the heap property

35

siftUp

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



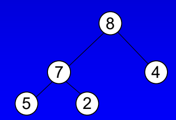
Blue node does not have heap property | Blue node has heap property

- This is sometimes called sifting up
- Notice that the child may have *lost* the heap property

36

The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x
    Parent(x) ≥ x

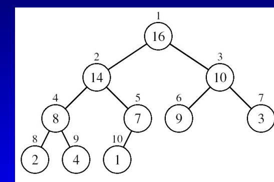


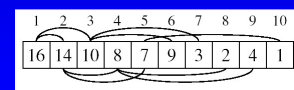
Heap

From the heap property, it follows that:
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

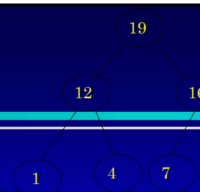
Array Representation of Heaps





- A heap can be stored as an array *A*.
  - Root of tree is A[1]
  - Left child of A[i] = A[2i]
  - Right child of A[i] = A[2i + 1]
  - Parent of A[i] = A[⌊i/2⌋]
  - Heapsize[A] ≤ length[A]
- The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves

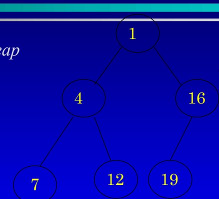
Heap Types



19	12	16	1	4	7

Array A

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:
    A[PARENT(i)] ≥ A[i]



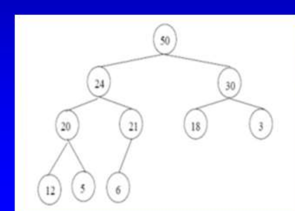
- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:
    A[PARENT(i)] ≤ A[i]

1	4	16	7	12	19

Array A

Adding/Deleting Nodes

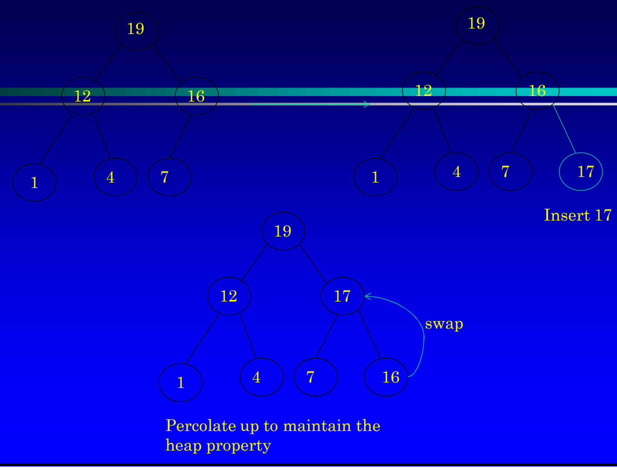
- **New nodes are always inserted at the bottom level (left to right)**
- **Nodes are removed from the bottom level (right to left)**



- Algorithm- Insertion
  1. Add the new element to the next available position at the lowest level
  2. Restore the max-heap property if violated
     - General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child.

     OR

     Restore the min-heap property if violated
     - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.



41

Deletion

- Delete max
  - Copy the last number to the root (overwrite the maximum element stored there).
  - Restore the max heap property by percolate down.
- Delete min
  - Copy the last number to the root (overwrite the minimum element stored there).
  - Restore the min heap property by percolate down.

42

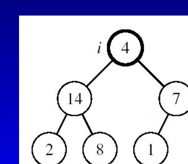
Operations on Heaps

- **Maintain/Restore the max-heap property**
  - MAX-HEAPIFY
- **Create a max-heap from an unordered array**
  - BUILD-MAX-HEAP
- **Sort an array in place**
  - HEAPSORT
- **Priority queues**

43

Maintaining the Heap Property

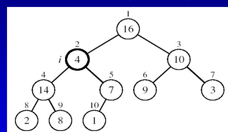
- Suppose a node is smaller than a child
  - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



44

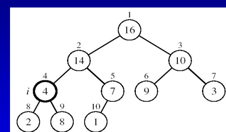
Example

MAX-HEAPIFY(A, 2, 10)



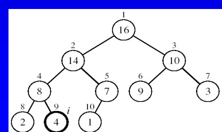
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

A[4] ↔ A[9]

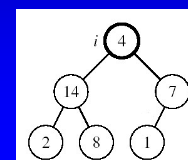


Heap property restored

45

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - A[i] may be smaller than its children



Alg: MAX-HEAPIFY(A, i, n)

1. l ← LEFT(i)
2. r ← RIGHT(i)
3. **if** l ≤ n and A[l] > A[i]
4. **then** largest ←l
5. **else** largest ←i
6. **if** r ≤ n and A[r] > A[largest]
7. **then** largest ←r
8. **if** largest ≠ i
9. **then** exchange A[i] ↔ A[largest]
10. MAX-HEAPIFY(A, largest, n)

46

MAX-HEAPIFY Running Time

- Intuitively:
  - It traces a path from the root to a leaf (longest path length h)
  - At each level, it makes exactly 2 comparisons
  - Total number of comparisons is 2h
  - Running time is O(h) or $O(lgn)$
- Running time of MAX-HEAPIFY is $O(lgn)$
- Can be written in terms of the height of the heap, as being $O(h)$
  - Since the height of the heap is $\lfloor lgn \rfloor$

47

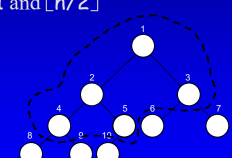
47

Building a Heap

- Convert an array A[1 … n] into a max-heap (n = length[A])
- The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves
- Apply MAX-HEAPIFY on elements between 1 and ⌊n/2⌋

Alg: BUILD-MAX-HEAP(A)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)



A:

4	1	3	2	16	9	10	14	8	7

48

48

Example: A

4	1	3	2	16	9	10	14	8	7

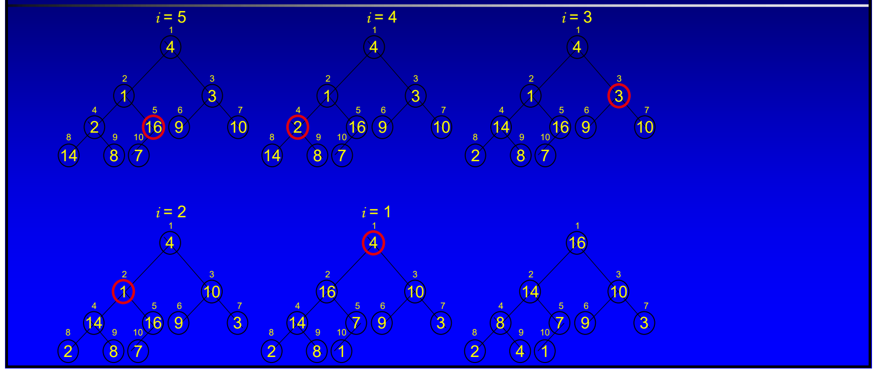
Alg: MAX-HEAPIFY(*A*, i, n)

1. l ← LEFT(i)
2. r ← RIGHT(i)
3. **if** l ≤ n and *A*[l] > *A*[i]
4. **then** *largest* ←l
5. **else** *largest* ←i
6. **if** r ≤ n and *A*[r] > *A*[largest]
7. **then** *largest* ←r
8. **if** *largest* ≠ i
9. **then** exchange *A*[i] ↔ *A*[largest]
10. MAX-HEAPIFY(*A*, largest, n)

49

Example: A

4	1	3	2	16	9	10	14	8	7



50

Running Time of BUILD MAX HEAP

Alg: BUILD-MAX-HEAP(*A*)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)

$O(\lg n)$ $O(n)$

⇒ Running time: $O(n\lg n)$

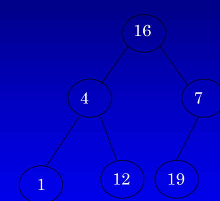
- This is not an asymptotically tight upper bound

51

Example: Convert the following array to a heap

16	4	7	1	12	19

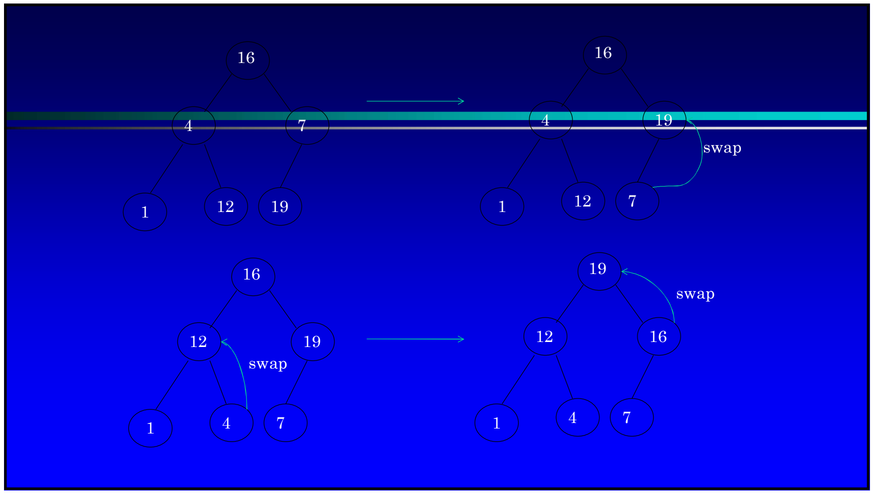
Picture **the array as a complete binary tree:**



Alg: MAX-HEAPIFY(*A*, i, n)

1. l ← LEFT(i)
2. r ← RIGHT(i)
3. **if** l ≤ n and *A*[l] > *A*[i]
4. **then** *largest* ←l
5. **else** *largest* ←i
6. **if** r ≤ n and *A*[r] > *A*[largest]
7. **then** *largest* ←r
8. **if** *largest* ≠ i
9. **then** exchange *A*[i] ↔ *A*[largest]
10. MAX-HEAPIFY(*A*, largest, n)

52

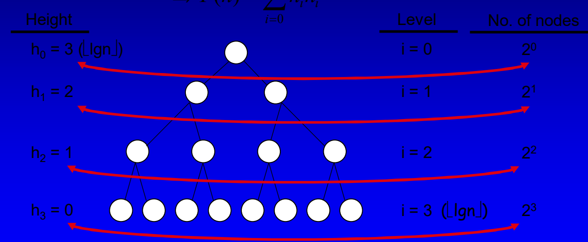


53

Running Time of BUILD MAX HEAP

- HEAPIFY takes $O(h)$ $\Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i \qquad = \sum_{i=0}^{h} 2^i h_i = O(n)$$



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

54

Running Time of BUILD MAX HEAP

$T(n) = \sum_{i=0}^{h} n_i h_i$ Cost of HEAPIFY at level i $*$ number of nodes at that level

$= \sum_{i=0}^{h} 2^i (h-i)$ Replace the values of n_i and h_i computed before

$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$ Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$ Change variables: k = h - i

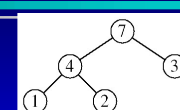
$\le n \sum_{k=0}^{\infty} \frac{k}{2^k}$ The sum above is smaller than the sum of all elements to ∞ and h = lgn

$= O(n)$ The sum above is smaller than 2

Running time of BUILD-MAX-HEAP: $T(n) = O(n)$

55

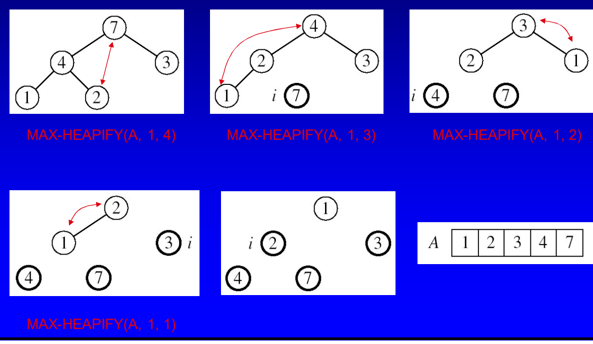
Heapsort



- Goal:
  - Sort an array using heap representations
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

56

Example: A=[7, 4, 3, 1, 2]



57

Alg: HEAPSORT*(A)*

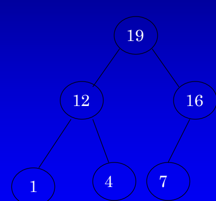
1. BUILD-MAX-HEAP(*A*) O(n)
2. **for** i ← length[*A*] **downto** 2
3. **do** exchange *A*[1] ↔ *A*[i] n-1 times
4. MAX-HEAPIFY(*A*, 1, i - 1) O(lgn)

- Running time: O(nlgn) --- Can be shown to be Θ(nlgn)

58

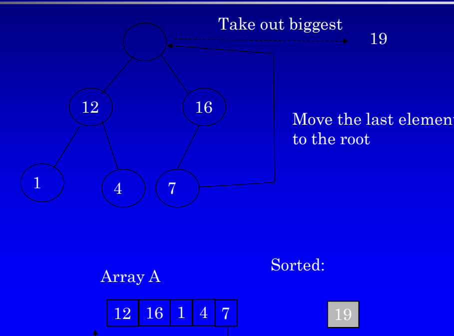
Heap Sort

- The heapsort algorithm consists of two phases:
  - build a heap from an arbitrary array
  - use the heap to sort the data
- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap

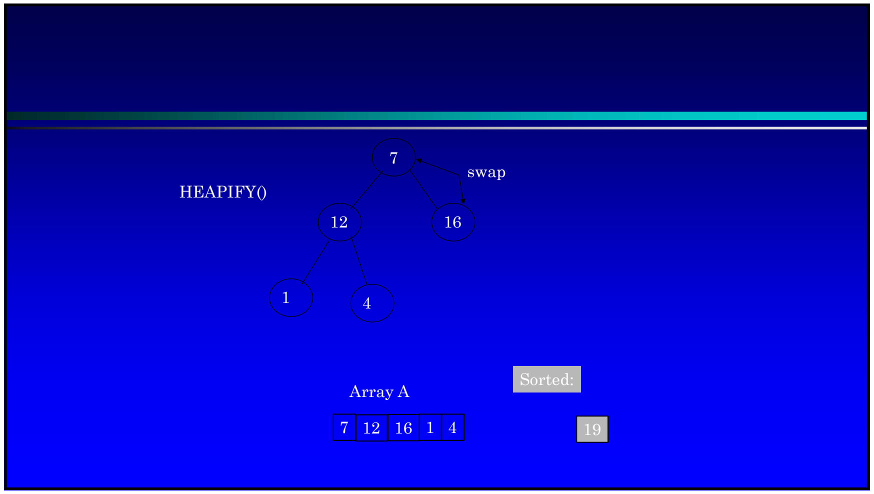


59

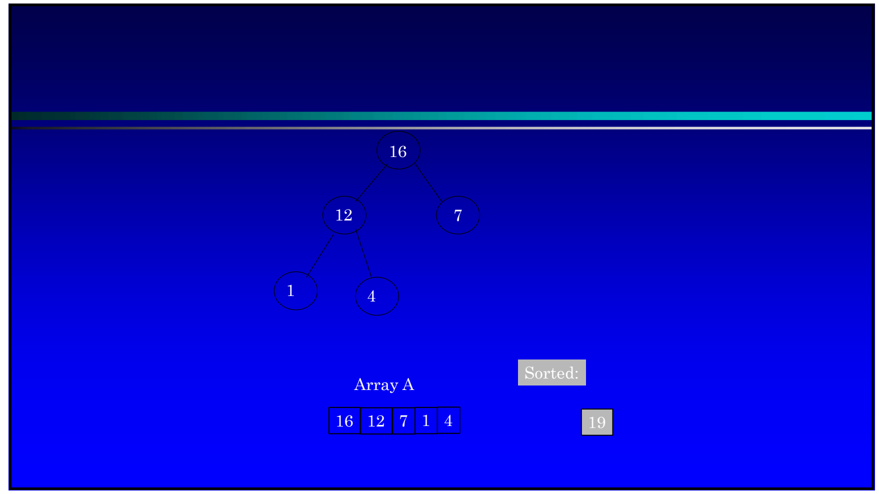
Example of Heap Sort



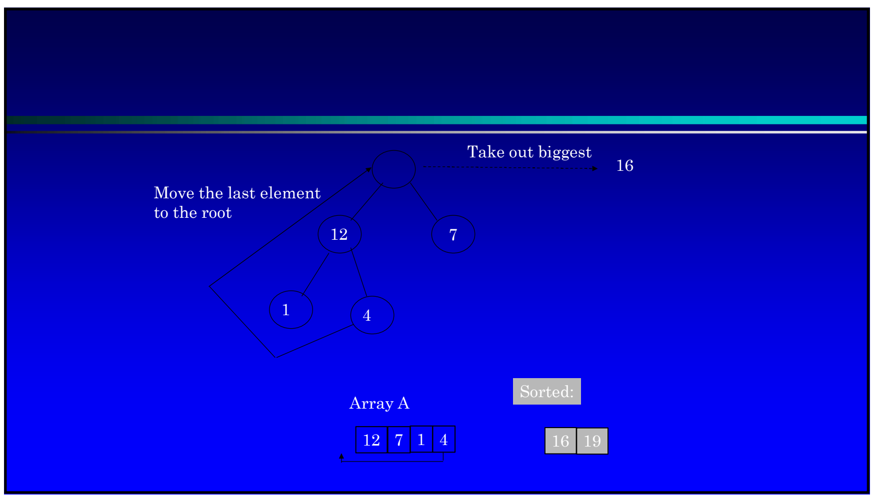
60


HEAPIFY()
7
swap
12
16
1
4
Array A
7 12 16 1 4
Sorted:
19

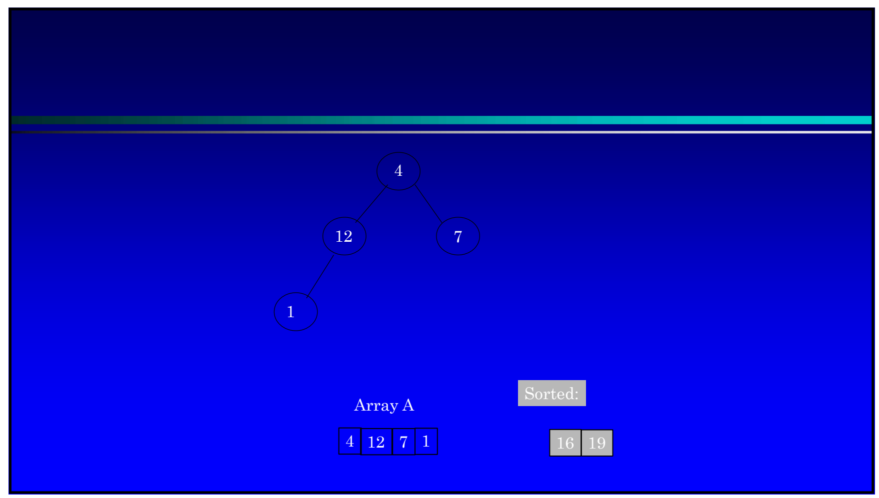
61


16
12
7
1
4
Array A
16 12 7 1 4
Sorted:
19

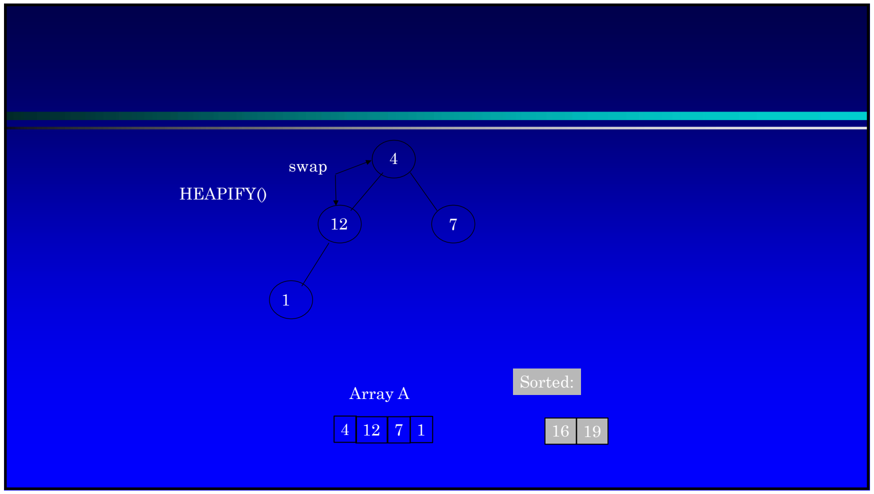
62


Take out biggest
16
Move the last element
to the root
12
7
1
4
Array A
12 7 1 4
Sorted:
16 19

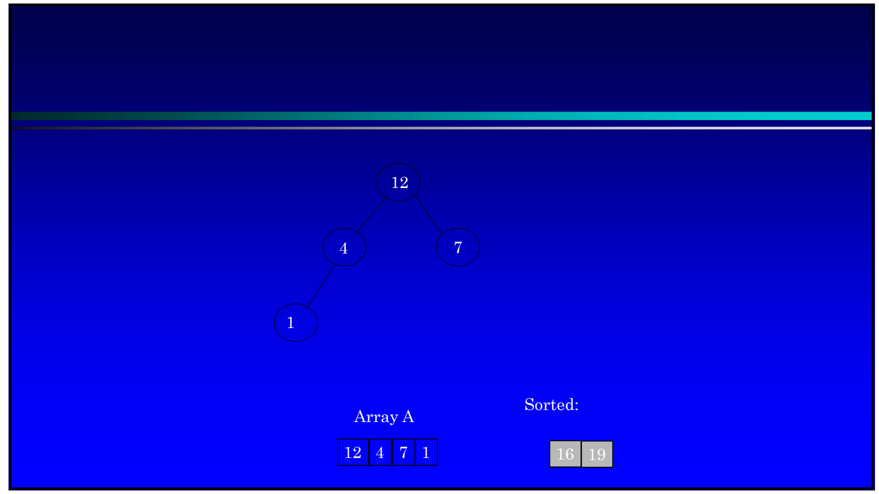
63


4
12
7
1
Array A
4 12 7 1
Sorted:
16 19

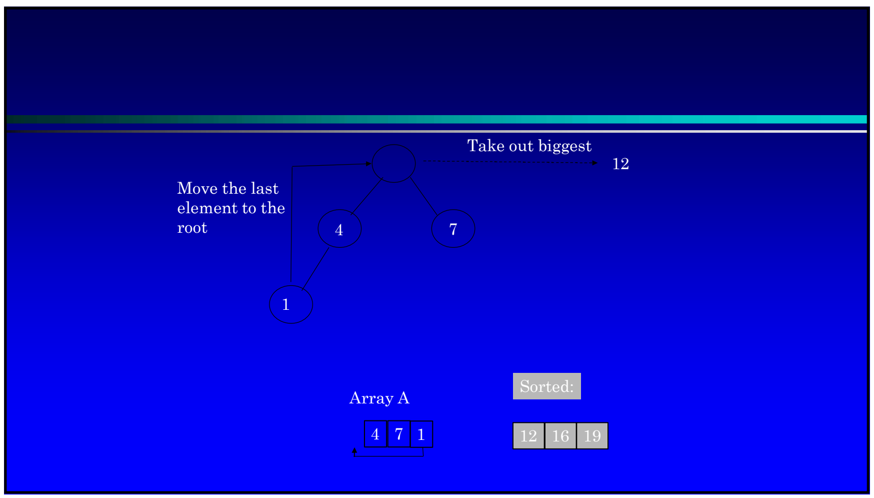
64


HEAPIFY()
swap
4
12
7
1
Array A
4 12 7 1
Sorted:
16 19

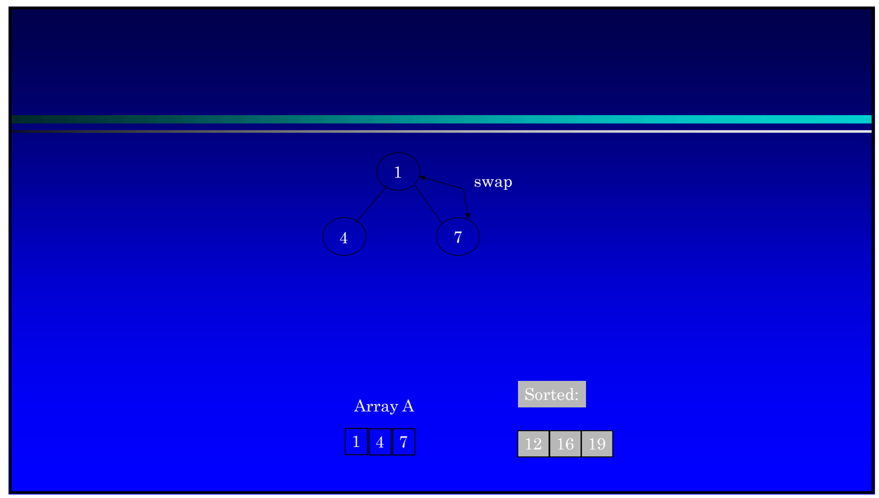
65


12
4
7
1
Array A
12 4 7 1
Sorted:
16 19

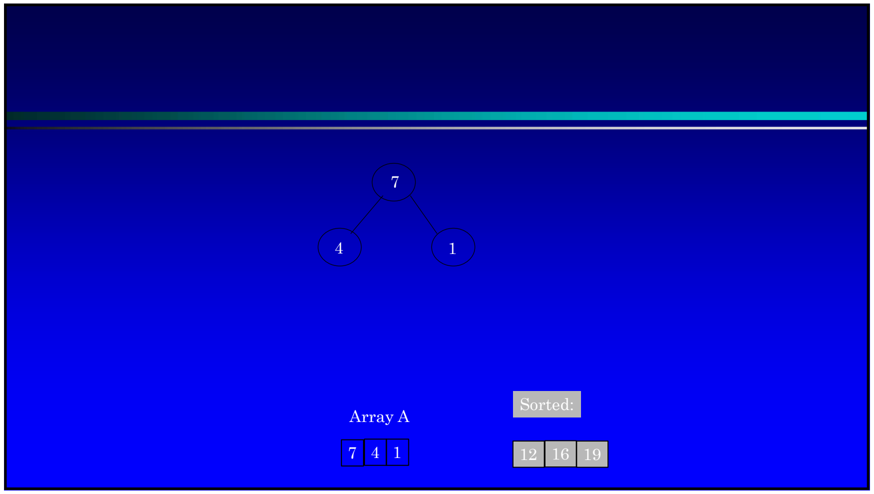
66


Take out biggest
12
Move the last element to the root
4
7
1
Array A
4 7 1
Sorted:
12 16 19

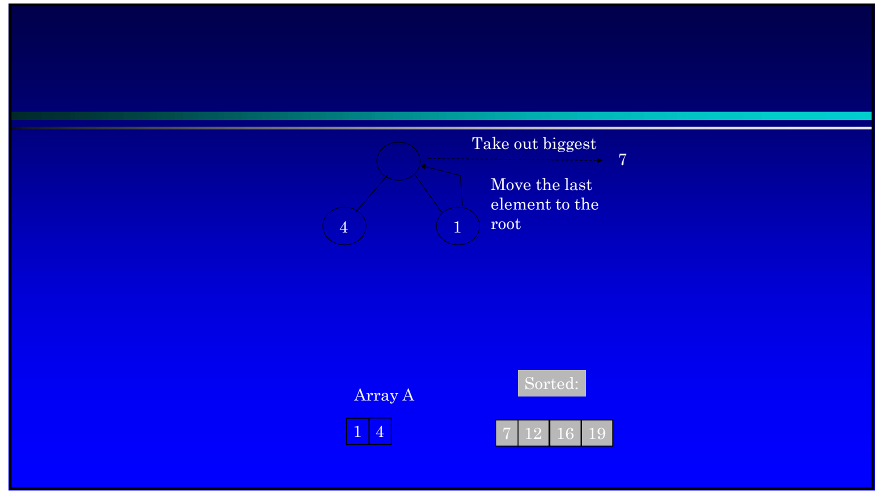
67


1
swap
4
7
Array A
1 4 7
Sorted:
12 16 19

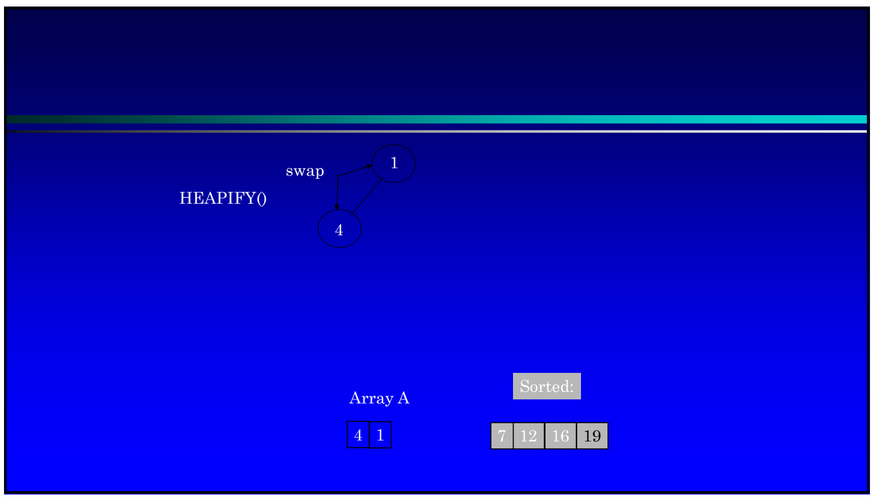
68


7
4
1
Array A
7 4 1
Sorted:
12 16 19

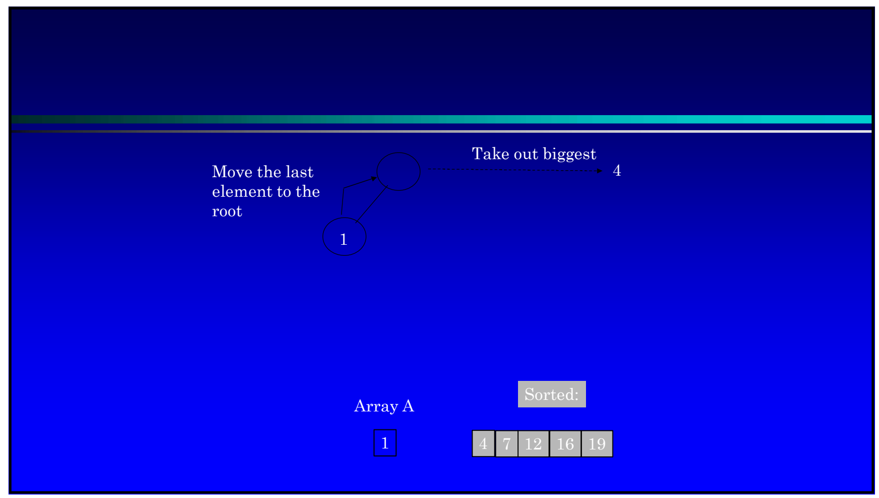
69


Take out biggest
7
Move the last element to the root
4
1
Array A
1 4
Sorted:
7 12 16 19

70


swap
HEAPIFY()
1
4
Array A
4 1
Sorted:
7 12 16 19

71


Take out biggest
4
Move the last element to the root
1
Array A
1
Sorted:
4 7 12 16 19

72

Take out biggest

1

Array A

Sorted:

1	4	7	12	16	19

73

Sorted:

1	4	7	12	16	19

74

Time Analysis

- Build Heap Algorithm will run in O(n) time
- There are *n*-1 calls to Heapify each call requires O(log *n*) time
- Heap sort program combine Build Heap program and Heapify, therefore it has the running time of O(n log n) time
- Total time complexity: O(n log n)

75

Heapify– Using location 0 in Array

Heapify function to construct a heap

```
void heapify( arr[], n, root)
{   // largest = root // Initialize largest as root
     // left_child = 2*root + 1 // left = 2*i + 1
     // right_child = 2*root + 2 // right = 2*i + 2
   // If left child is larger than root
   if (left_child < n && arr[left_child] > arr[largest])
      largest = left_child
    // If right child is larger than largest so far
   if (right_child < n && arr[right_child] > arr[largest])
      largest = right_child
   // If largest is not root
     if (largest != root)
     {
       swap(arr[root], arr[largest])
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest)
     }
}
```

76

Heap Sort– Using location 0 in Array

Heap sort algo

```
void heapSort(arr[], n)
{
   // Build heap (rearrange array)
   for ( i = n / 2 - 1  to i >= 0 )
      heapify(arr, n, i)
   // One by one extract an element from heap
   for (i=n-1 to 0)
   {
      // Move current root to end
      swap(arr[0], arr[i])
      // call max heapify on the reduced heap
      heapify(arr, i, 0)
   }
}
```

77

Comparison with Quick Sort and Merge Sort

- Quick sort is typically somewhat faster, due to better cache behavior and other factors, but the worst-case running time for quick sort is O (n^2), which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk.
- The quick sort algorithm also requires Ω (log n) extra storage space, making it not a strictly in-place algorithm. This typically does not pose a problem except on the smallest embedded systems, or on systems where memory allocation is highly restricted. Constant space (in-place) variants of quick sort are possible to construct, but are rarely used in practice due to their extra complexity.

78

Comparison with Quick Sort and Merge Sort (cont)

- Thus, because of the O(n log n) upper bound on heap sort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heap sort.
- Heap sort also competes with merge sort, which has the same time bounds, but requires Ω(n) auxiliary space, whereas heap sort requires only a constant amount. Heap sort also typically runs more quickly in practice. However, merge sort is simpler to understand than heap sort, is a stable sort, parallelizes better, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Heap sort shares none of these benefits; in particular, it relies strongly on random access.

79

Possible Application

- When we want to know the task that carry the highest priority given a large number of things to do
- Interval scheduling, when we have a lists of certain task with start and finish times and we want to do as many tasks as possible
- Sorting a list of elements that needs and efficient sorting algorithm

80

Conclusion

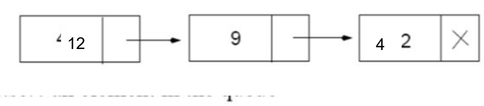
- The primary advantage of the heap sort is its efficiency. The execution time efficiency of the heap sort is O(n log n). The memory efficiency of the heap sort, unlike the other n log n sorts, is constant, O(1), because the heap sort algorithm is not recursive.
- The heap sort algorithm has two major steps. The first major step involves transforming the complete tree into a heap. The second major step is to perform the actual sort by extracting the largest element from the root and transforming the remaining tree into a heap.

81

Priority Queues

Properties

- Each element is associated with a value (priority)
- The key with the highest (or lowest) priority is extracted first



82

Operations on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(S, x): inserts element x into set S
  - EXTRACT-MAX(S): removes and returns element of S with largest key
  - MAXIMUM(S): returns element of S with largest key
  - INCREASE-KEY(S, x, k): increases value of element x's key to k (Assume $k \geq x$'s current key value)

83

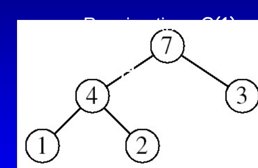
HEAP-MAXIMUM

Goal:

- Return the largest element of the heap

Alg: HEAP-MAXIMUM(A)

1. **return** $A[1]$



Heap-Maximum(A) returns 7

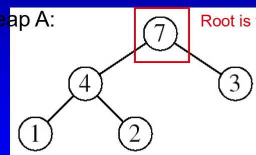
84

HEAP-EXTRACT-MAX

Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

Heap A:

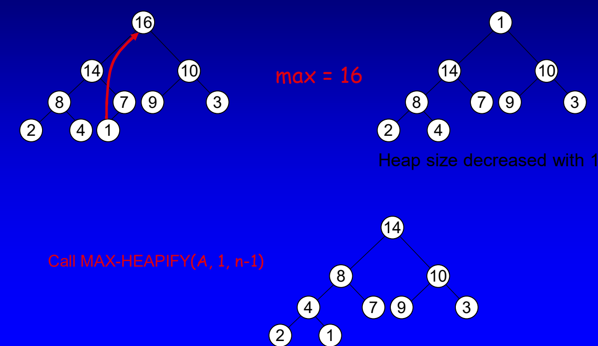


Root is the largest element

Idea:

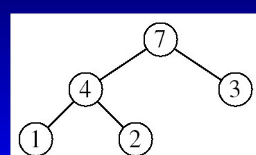
- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

Example: HEAP-EXTRACT-MAX



HEAP-EXTRACT-MAX

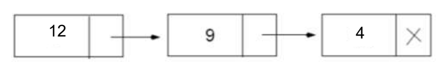
Alg: HEAP-EXTRACT-MAX(A, n)



if n < 1
   then error "heap underflow"
max ← A[1]
A[1] ← A[n]
MAX-HEAPIFY(A, 1, n-1) remakes heap
return max

Running time: O(lgn)

Priority Queue Using Linked List



Remove a key: O(1)

Insert a key: O(n)

Increase key: O(n)

Extract max key: O(1)

Average: O(n)

Problems

(a) What is the maximum number of nodes in a max heap of height h?

(b) What is the maximum number of leaves?

(c) What is the maximum number of internal nodes?

Problems

- Demonstrate, step by step, the operation of Build-Heap on the array

  A=[5, 3, 17, 10, 84, 19, 6, 22, 9]

Problems

- Let A be a heap of size n. Give the most efficient algorithm for the following tasks:

(a) Find the sum of all elements

(b) Find the sum of the largest lgn elements