

DATA STRUCTURE AND ALGORITHM

CHAPTER 8



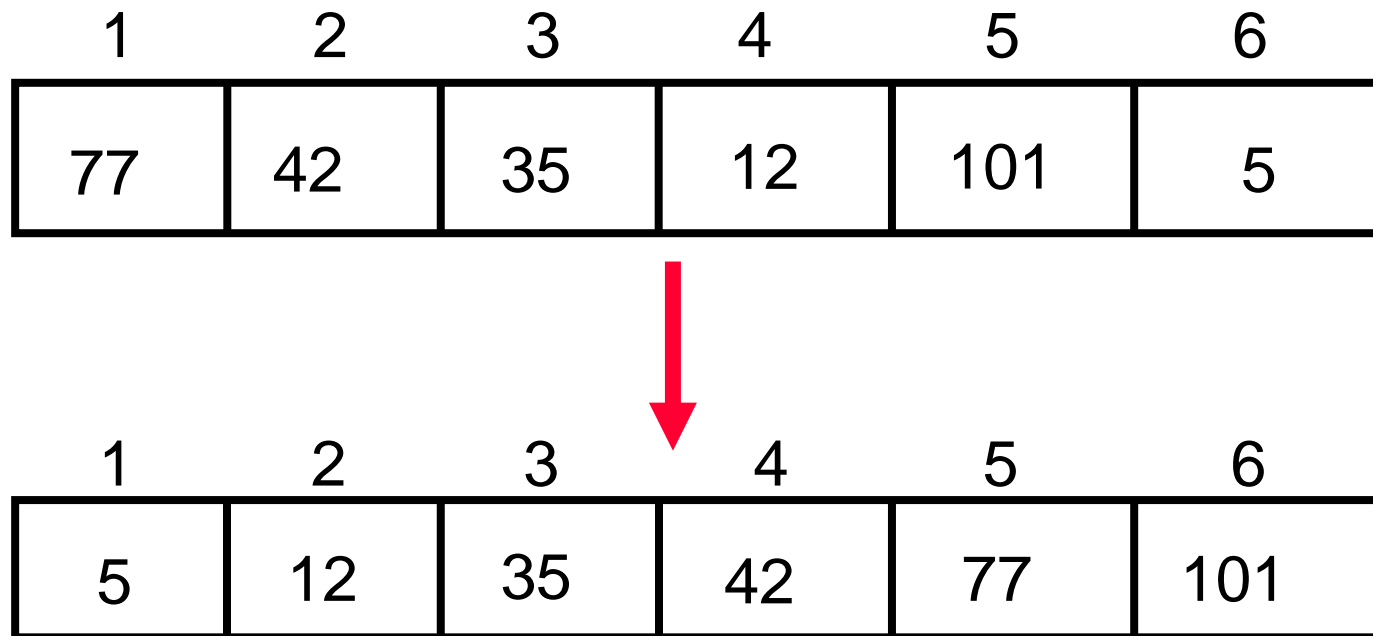
Advanced Sorting and Searching

CONTENTS

- Sorting
- Divide and Conquer
- Advanced Sorting Algorithms
 - Shell Sort
 - Merge Sort
 - Quick Sort
 - Heap Sort

SORTING

- **Sorting** takes an unordered collection and makes it an ordered one.



SHELL SORT

Shell sort algorithm:

- Insertion sort is an efficient algorithm only if the list is already partially sorted and results in an inefficient solution in an average case.
- To overcome this limitation, a computer scientist, D.L. Shell proposed an improvement over the insertion sort algorithm.
- The new algorithm was called shell sort after the name of its proposer.

IMPLEMENTING SHELL SORT ALGORITHM

Shell sort algorithm:

- Improves insertion sort by comparing the elements separated by a distance of several positions to form multiple sublists
- Applies insertion sort on each sublist to move the elements towards their correct positions
- Helps an element to take a bigger step towards its correct position, thereby reducing the number of comparisons

CONTD.

- To understand the implementation of shell sort algorithm, consider an unsorted list of numbers stored in an array.

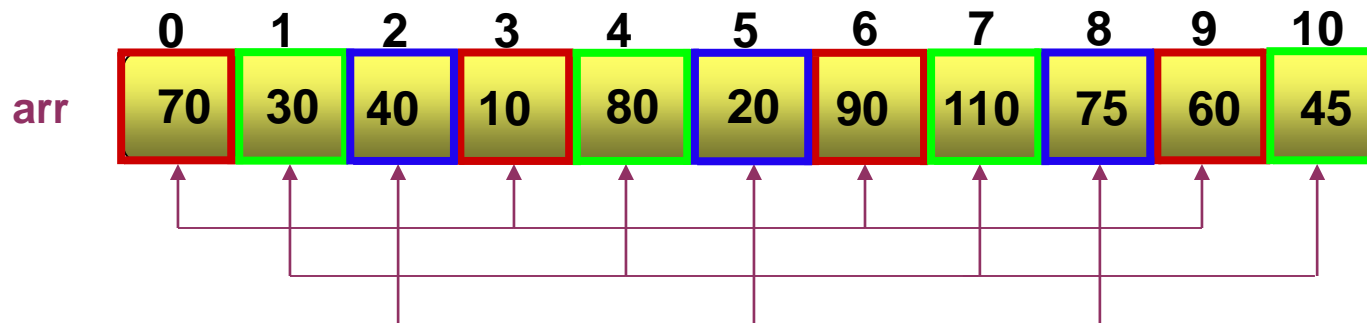
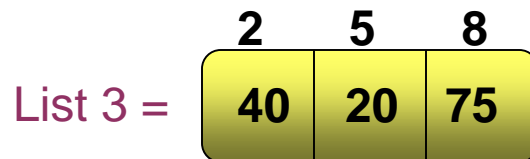
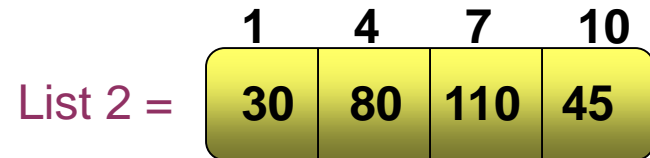
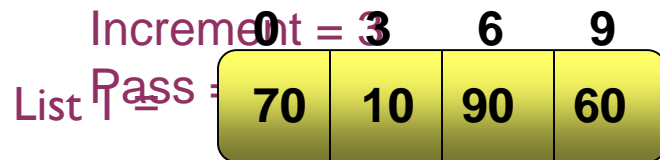
	0	1	2	3	4	5	6	7	8	9	10
arr	70	30	40	10	80	20	90	110	75	60	45

CONTD

- To apply shell sort on this array, you need to:
- Select the distance by which the elements in a group will be separated to form multiple sublists.
- Apply insertion sort on each sublist to move the elements towards their correct positions.

	0	1	2	3	4	5	6	7	8	9	10
arr	70	30	40	10	80	20	90	110	75	60	45

Implementing Shell Sort Algorithm (Contd.)



Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	3	6	9
70	60	90	80

List 2 =

1	4	7	10
30	85	100	450

List 3 =

2	5	8
20	20	75

Apply insertion sort to sort the
The lists are sorted
three lists

Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	3	6	9
10	60	70	90

List 2 =

1	4	7	10
30	45	80	110

List 3 =

2	5	8
20	40	75

arr

0	1	2	3	4	5	6	7	8	9	10
10	30	20	60	45	40	70	80	75	90	110

Implementing Shell Sort Algorithm (Contd.)

Inc
List 1 = Pa


0	2	4	6	8	10
10	20	45	70	75	110

List 2 =

1	3	5	7	9
30	60	40	80	90

arr

0	1	2	3	4	5	6	7	8	9	10
10	30	20	60	45	40	70	80	75	90	110



Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	2	4	6	8	10
10	20	45	70	75	110

List 2 =

1	3	5	7	9
30	60	40	80	90

Apply insertion sort on each sublist

Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	2	4	6	8	10
10	20	45	70	75	110

List 2 =

1	3	5	7	9
30	40	60	80	90

The lists are now sorted

Implementing Shell Sort Algorithm (Contd.)

List 1 =

0	2	4	6	8	10
10	20	45	70	75	110

List 2 =

1	3	5	7	9
30	40	60	80	90

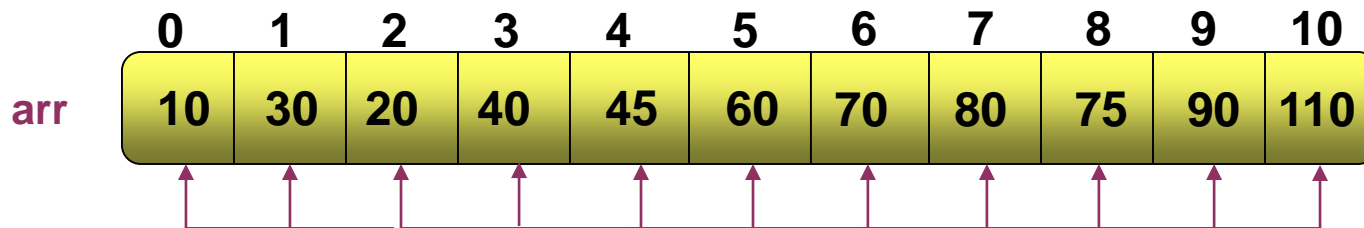
arr

0	1	2	3	4	5	6	7	8	9	10
10	30	20	40	45	60	70	80	75	90	110

Implementing Shell Sort Algorithm (Contd.)

Increment = 1

Pass = 3

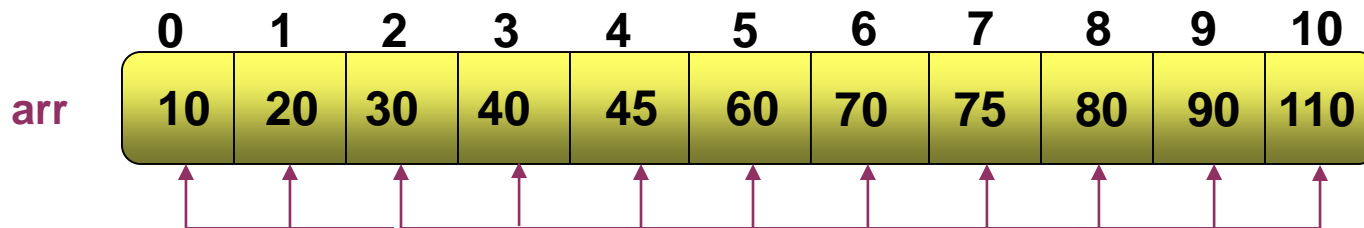


Apply insertion sort to sort the list

Implementing Shell Sort Algorithm (Contd.)

Increment = 1

Pass = 3



The list is now sorted

Just a minute

- Which of the following sorting algorithms compares the elements separated by a distance of several positions to sort the data? The options are:
 1. Insertion sort
 2. Selection sort
 3. Bubble sort
 4. Shell sort

Answer:

4. Shell sort

DIVIDE & CONQUER

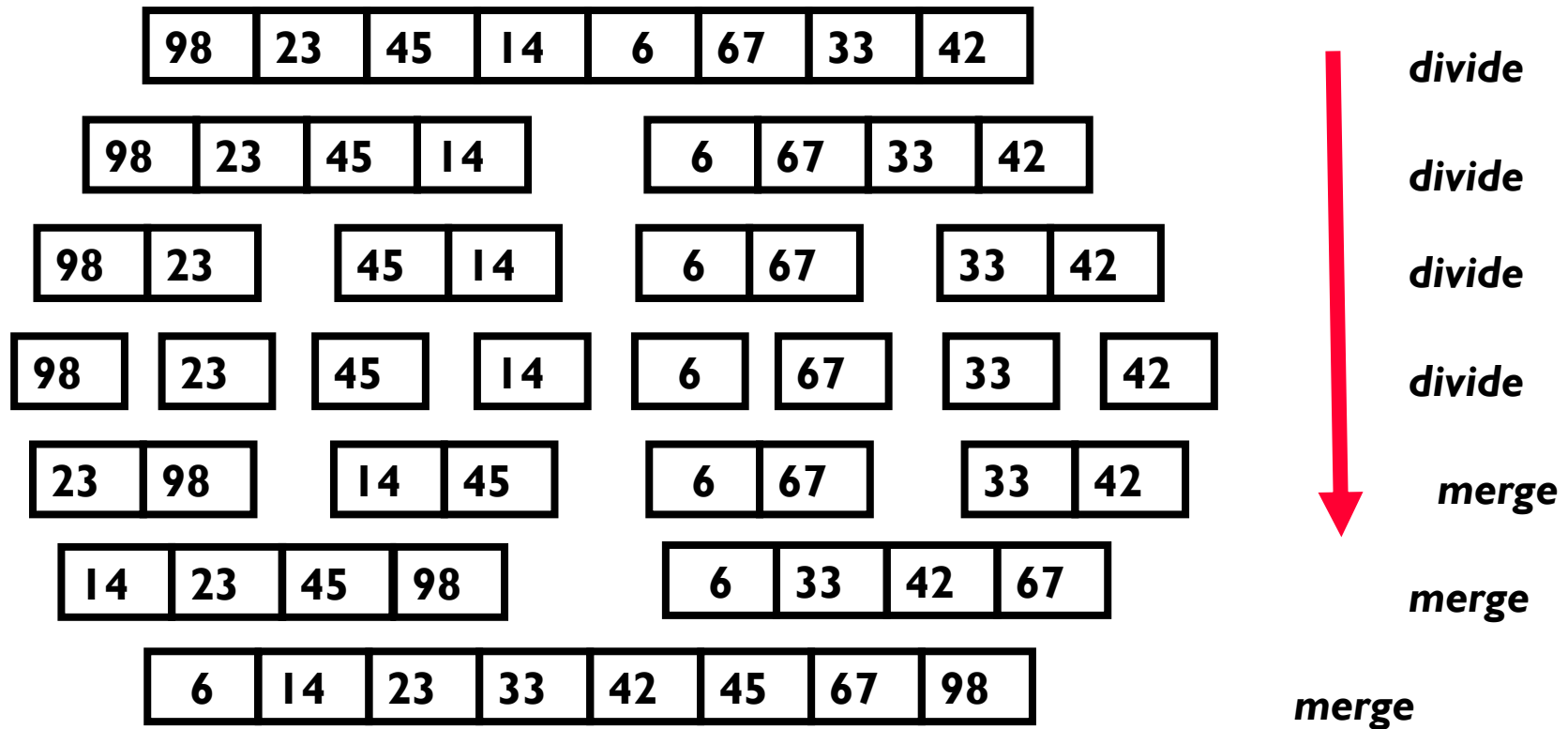
- Recursive in structure
 - **Divide** the problem into sub-problems that are similar to the original but smaller in size
 - **Conquer** the sub-problems by solving them *recursively*. If they are small enough, just solve them in a straightforward manner.
 - **Combine** the solutions to create a solution to the original problem

MERGE SORT

Based on divide-and-conquer strategy

- Divide the list into two smaller lists of about equal sizes
- Sort each smaller list *recursively*
- Merge the two sorted lists to get one sorted list

MERGE SORT - EXAMPLE



Implementing Merge Sort Algorithm (Contd.)

Write an algorithm to implement merge sort:

MergeSort(low,high)

1. If (low \geq high):
 - a. Return
2. Set mid = (low + high)/2
3. Divide the list into two sublists of nearly equal lengths, and sort each sublist by using merge sort. The steps to do this are as follows:
 - a. MergeSort(low, mid
 - b. MergeSort(mid + 1, high)
4. Merge the two sorted sublists:
 - a. Set i = low
 - b. Set j = mid + 1
 - c. Set k = low
 - d. Repeat until i > mid or j > high: **// This loop will terminate when
// you reach the end of one of the
// two sublists.**

Implementing Merge Sort Algorithm (Contd.)

- i. If ($\text{arr}[i] \leq \text{arr}[j]$)
 - Store $\text{arr}[i]$ at index k in array B
 - Increment i by 1
 - Else
 - Store $\text{arr}[j]$ at index k in array
 - Increment j by 1
 - ii. Increment k by 1
 - e. Repeat until $j > \text{high}$: **// If there are still some elements in the
// second sublist append them to the new list**
 - i. Store $\text{arr}[j]$ at index k in array B
 - ii. Increment j by 1
 - iii. Increment k by 1
 - f. Repeat until $i > \text{mid}$: **// If there are still some elements in the
// first sublist append them to the new list**
 - i. Store $\text{arr}[i]$ at index k in array B
 - ii. Increment i by 1
 - iii. Increment k by 1
5. Copy all elements from the sorted array B into the original array arr

Determining the Efficiency of Merge Sort Algorithm

- To sort the list by using merge sort algorithm, you need to recursively divide the list into two nearly equal sublists until each sublist contains only one element.
- To divide the list into sublists of size one requires $\log n$ passes.
- In each pass, a maximum of n comparisons are performed.
- Therefore, the total number of comparisons will be a maximum of $n \times \log n$.
- The efficiency of merge sort is equal to $O(n \log n)$
- There is no distinction between best, average, and worst case efficiencies of merge sort because all of them require the same amount of time.

ANALYSIS OF MERGE SORT

- The amount of extra memory used is $O(n)$
- Let $T(N)$ denote the worst-case running time of merge sort to sort N numbers.

Assume that N is a power of 2.

- Divide step: $O(1)$ time
- Conquer step: $2 T(N/2)$ time
- Combine step: $O(N)$ time
- Recurrence equation:

$$T(1) = 1$$

$$T(N) = 2T(N/2) + N$$

ANALYSIS: SOLVING RECURRENCE

$$\begin{aligned}T(N) &= 2T\left(\frac{N}{2}\right) + N \\&= 2\left(2T\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N \\&= 4T\left(\frac{N}{4}\right) + 2N \\&= 4\left(2T\left(\frac{N}{8}\right) + \frac{N}{4}\right) + 2N \\&= 8T\left(\frac{N}{8}\right) + 3N = \dots \\&= 2^k T\left(\frac{N}{2^k}\right) + kN\end{aligned}$$

Since $N=2^k$, we have $k=\log_2 n$

$$\begin{aligned}T(N) &= 2^k T\left(\frac{N}{2^k}\right) + kN \\&= N + N \log N \\&= O(N \log N)\end{aligned}$$

QUICK SORT

- Efficient sorting algorithm
 - Discovered by **C.A.R. Hoare** in 1962.
- Example of **Divide and Conquer** algorithm
- Two phases
 - Partition phase
 - **Divides** the work into half
 - Sort phase
 - **Conquers** the halves!

CONTD.

■ Partition

- Choose a **pivot**
- Find the position for the pivot so that
 - all elements to the left are less
 - all elements to the right are greater



■ Conquer

- Apply the same algorithm to each half



QUICKSORT EXAMPLE

- Recursive implementation with the left most array entry selected as the pivot element.

0	15	12	3	21	25	3	9	8	18	28	5
1	9	12	3	5	8	3	15	25	18	28	21
2	8	3	3	5	9	12	15	21	18	25	28
3	5	3	3	8	9	12	15	18	21	25	28
4	3	3	5	8	9	12	15	18	21	25	28
5	3	3	5	8	9	12	15	18	21	25	28
6	3	3	5	8	9	12	15	18	21	25	28

Implementing Quick Sort Algorithm (Contd.)

Write an algorithm to implement quick sort:

QuickSort(low,high)

1. If (low > high):
 - a. Return
2. Set pivot = arr[low]
3. Set i = low + 1
4. Set j = high
5. Repeat step 6 until i > high or arr[i] > pivot // **Search for an element greater than pivot**
6. Increment i by 1
7. Repeat step 8 until j < low or arr[j] < pivot // **Search for an element smaller than pivot**
8. Decrement j by 1
9. If i < j: // **If greater element is on the left of smaller element**
 - a. Swap arr[i] with arr[j]

Implementing Quick Sort Algorithm (Contd.)

10. If $i \leq j$:

a. Go to step 5 // **Continue the search**

11. If $low < j$:

a. Swap $arr[low]$ with $arr[j]$ // **Swap pivot with last element in first part of the list**

12. $QuickSort(low, j - 1)$ // **Apply quicksort on list left to pivot**

13. $QuickSort(j + 1, high)$ // **Apply quicksort on list right to pivot**

Determining the Efficiency of Quick Sort Algorithm

- The total time taken by this sorting algorithm depends on the position of the pivot value.
- The worst case occurs when the list is already sorted.
- If the first element is chosen as the pivot, it leads to a worst case efficiency of $O(n^2)$.
- If you select the median of all values as the pivot, the efficiency would be $O(n \log n)$.

QUICK SORT ANALYSIS

- Running time
 - pivot selection: constant time, i.e. $O(1)$
 - partitioning: linear time, i.e. $O(N)$
 - running time of the two recursive calls
- $T(N) = T(i) + T(N-i-1) + cN$ where c is a constant
 - i : number of elements in S_1

QUICK SORT – WORST CASE ANALYSIS

- The pivot is the smallest element, all the time
- Partition is always unbalanced

$$T(N) = T(N - 1) + cN$$

$$T(N - 1) = T(N - 2) + c(N - 1)$$

$$T(N - 2) = T(N - 3) + c(N - 2)$$

$$\vdots$$

$$T(2) = T(1) + c(2)$$

$$T(N) = T(1) + c \sum_{i=2}^N i = O(N^2)$$

QUICK SORT – BEST & CASE AVERAGE ANALYSIS

- Partition is perfectly balanced.
- Pivot is always in the middle (median of the array)

$$T(N) = 2T(N/2) + cN$$

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + c$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + c$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + c$$

\vdots

$$\frac{T(2)}{2} = \frac{T(1)}{1} + c$$

$$\frac{T(N)}{N} = \frac{T(1)}{1} + c \log N$$

$$T(N) = cN \log N + N = O(N \log N)$$

- On average, the running time is $O(N \log N)$

Activity

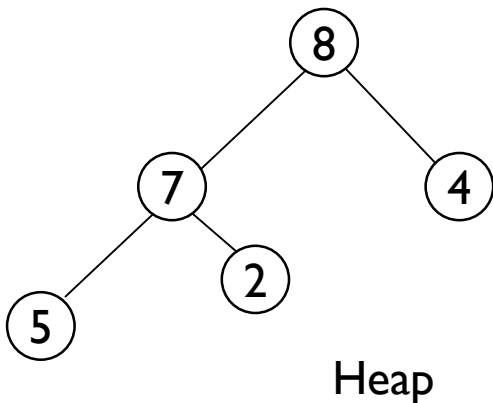
Which algorithm uses the following procedure to sort a given list of elements?

1. Select an element from the list called a pivot.
2. Partition the list into two parts such that one part contains elements lesser than the pivot, and the other part contains elements greater than the pivot.
3. Place the pivot at its correct position between the two lists.
4. Sort the two parts of the list using the same algorithm.

On which algorithm design technique are quick sort and merge sort based?

THE HEAP DATA STRUCTURE

- **Def:** A **heap** is a 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 , $\text{Parent}(x) \geq x$



From the heap property, it follows that:

“The root is the maximum element of the heap!”

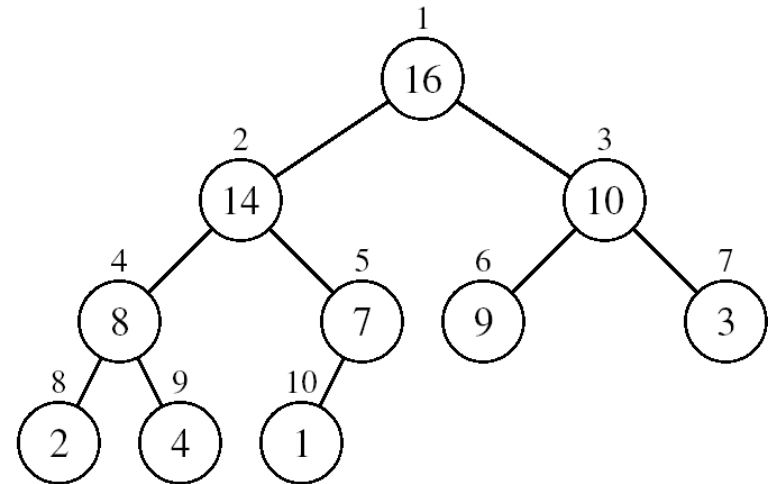
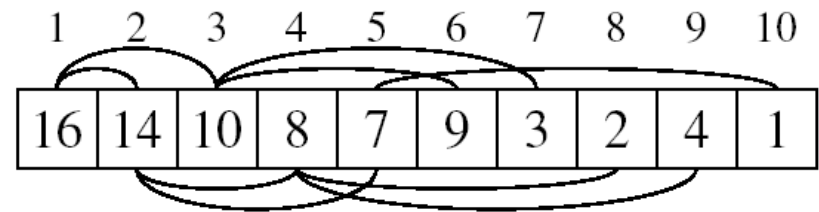
A heap is a binary tree that is filled in order

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 $A[i] = A[\lfloor i/2 \rfloor]$
- $\text{Heapsize}[A] \leq \text{length}[A]$

- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves



HEAP TYPES

- **Max-heaps** (largest element at root), have the *max-heap property*:

- for all nodes i , excluding the root:

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

- **Min-heaps** (smallest element at root), have the *min-heap property*:

- for all nodes i , excluding the root:

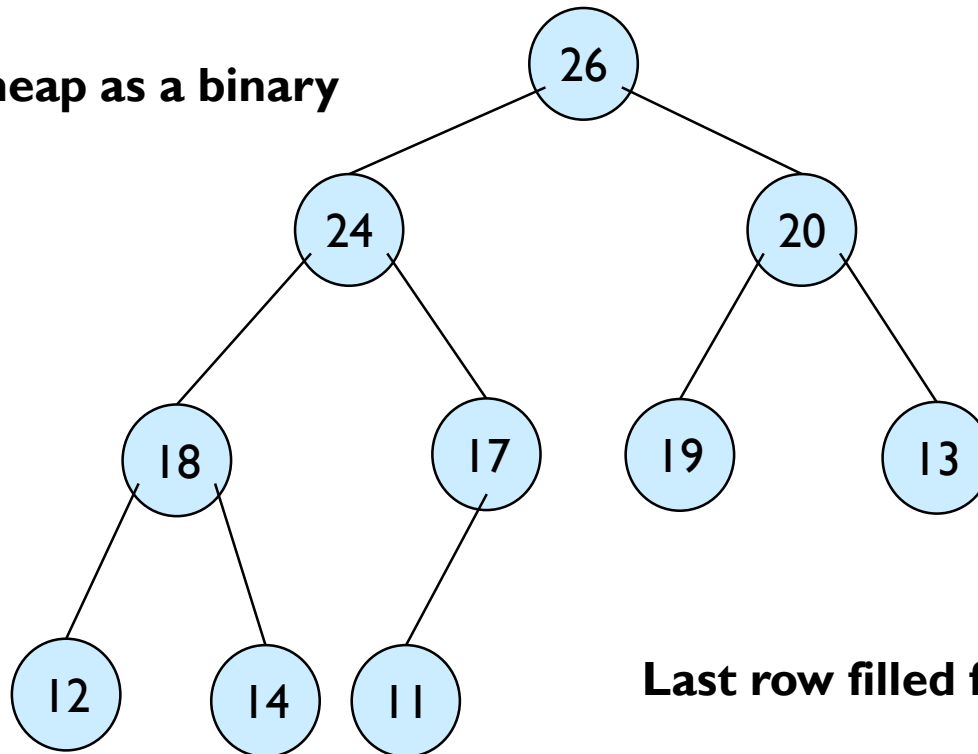
$$A[\text{PARENT}(i)] \leq A[i]$$

MAP HEAP – 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.

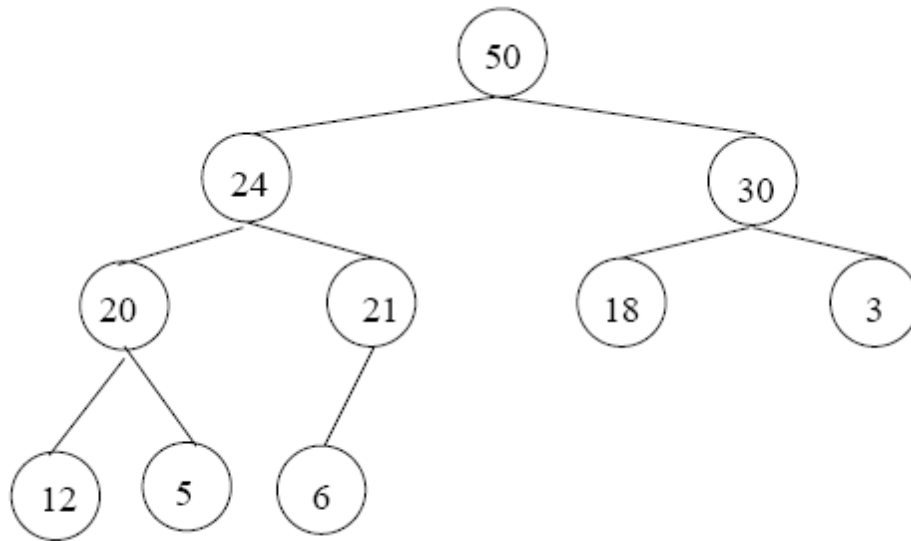
Max-heap as a binary tree.



Last row filled from left to right.

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)

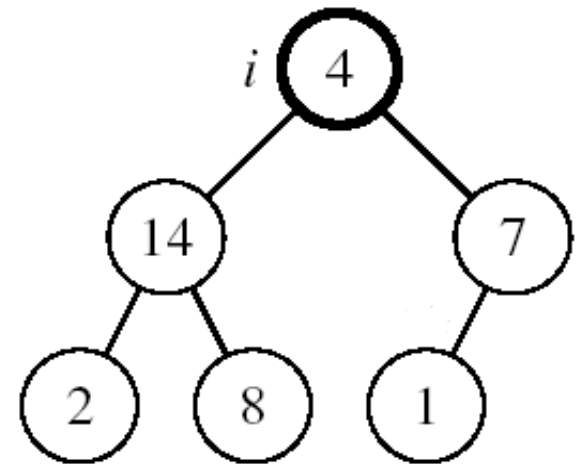


OPERATIONS ON HEAPS

- 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

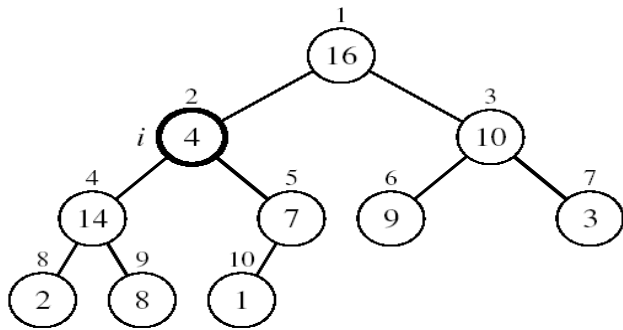
MAINTAINING THE HEAP PROPERTY

- 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



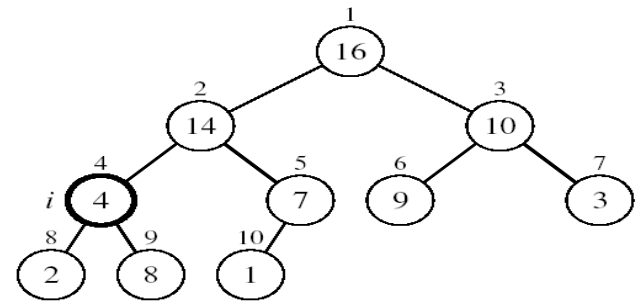
EXAMPLE

MAX-HEAPIFY(A, 2, 10)

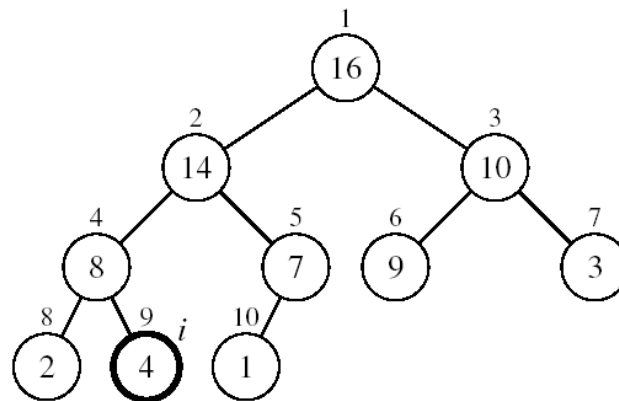


A[2] violates the heap property

$A[2] \leftrightarrow A[4]$



A[4] violates the heap property

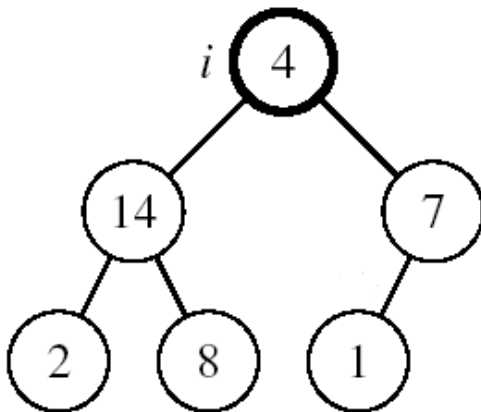


$A[4] \leftrightarrow A[9]$

Heap property restored

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)

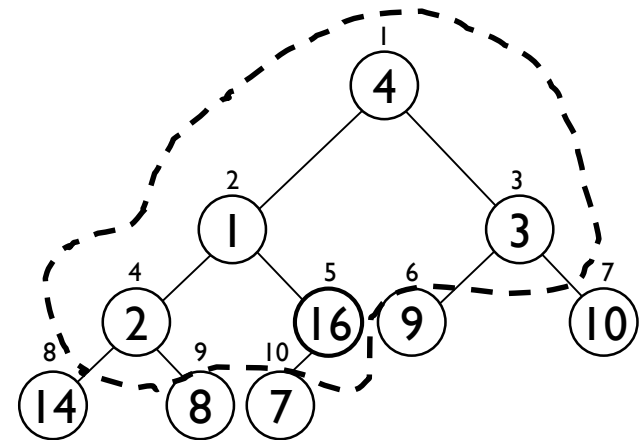
1. $l \leftarrow \text{LEFT}(i)$
2. $r \leftarrow \text{RIGHT}(i)$
3. **if** $l \leq n$ and $A[l] > A[i]$
4. **then** $\text{largest} \leftarrow l$
5. **else** $\text{largest} \leftarrow i$
6. **if** $r \leq n$ and $A[r] > A[\text{largest}]$
7. **then** $\text{largest} \leftarrow r$
8. **if** $\text{largest} \neq i$
9. **then** exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MAX-HEAPIFY($A, \text{largest}, n$)

BUILDING A HEAP

- Convert an array $A[1 \dots n]$ into a max-heap ($n = \text{length}[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \dots n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(A)

1. $n = \text{length}[A]$
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY(A, i, n)



A:

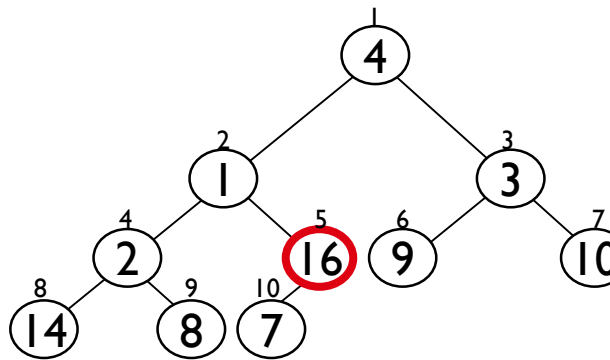
4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

EXAMPLE

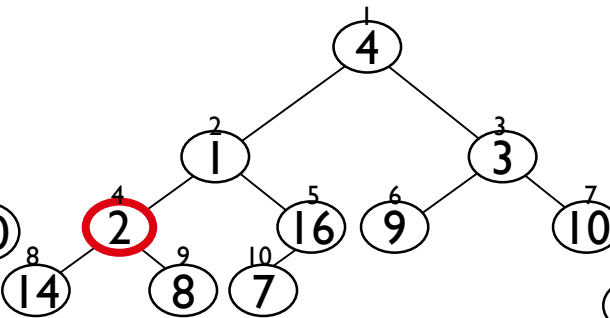
A

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

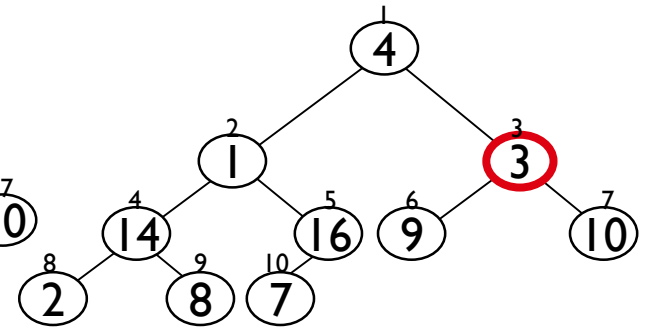
$i = 5$



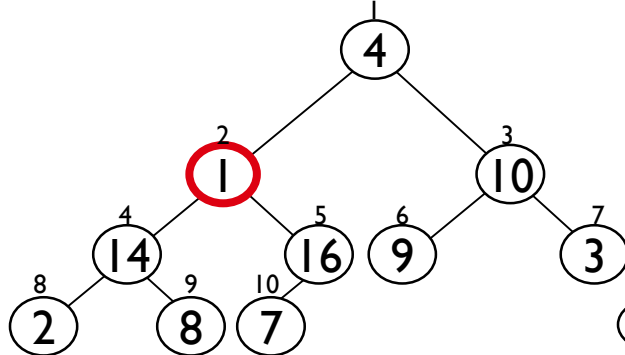
$i = 4$



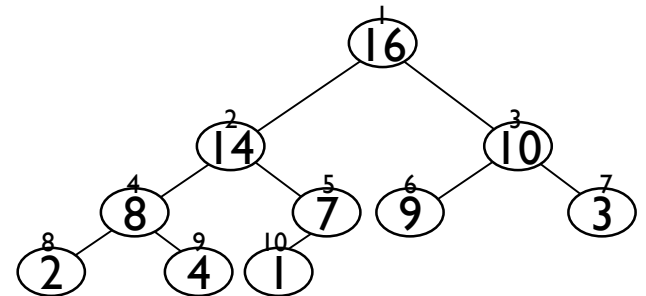
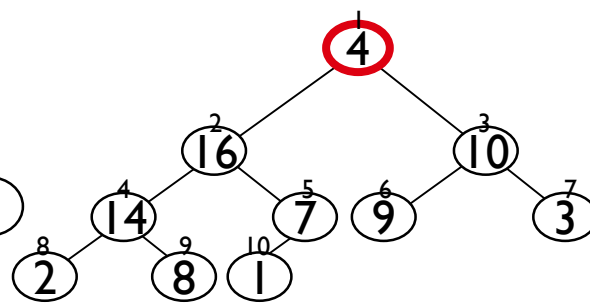
$i = 3$



$i = 2$

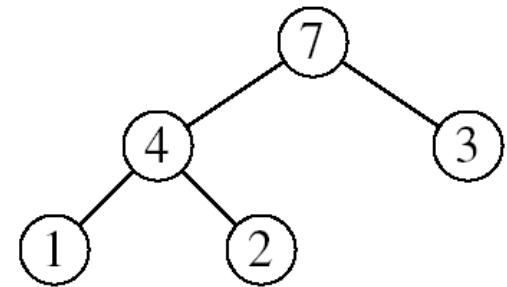


$i = 1$

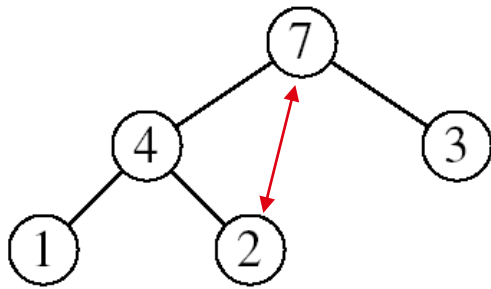


HEAP SORT

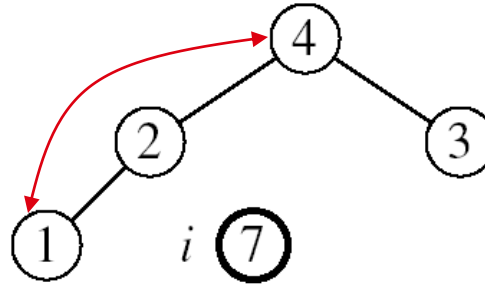
- Goal:
 - Sort an array using heap representations
- Idea:
 - Build a **max-heap** from the array
 - Swap the root (the maximum element) with the last element in the array
 - “Discard” this last node by decreasing the heap size
 - Call MAX-HEAPIFY on the new root
 - Repeat this process until only one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



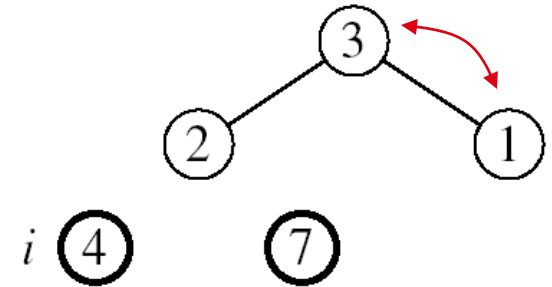
EXAMPLE: $A=[7, 4, 3, 1, 2]$



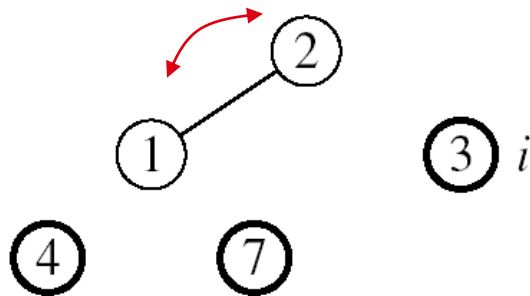
MAX-HEAPIFY(A, 1, 4)



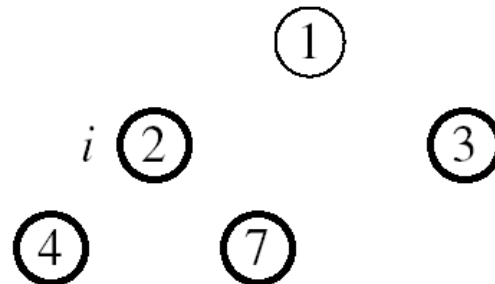
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)



A

1	2	3	4	7
---	---	---	---	---

ALG: HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. **for** $i \leftarrow \text{length}[A]$ **downto** 2
3. **do** exchange $A[1] \leftrightarrow A[i]$
4. MAX-HEAPIFY(A, 1, $i - 1$)

Heap sort: Analysis

- Running time
 - *worst case* is $\Theta(N \log N)$
 - *Average case* is also $O(N \log N)$

SELF-REVIEW QUESTIONS

1. What is a heap? Differentiate between min heap and max heap?
2. Given a queue of elements with priorities: 21, 13, 17, 10, 7, 11 do the following:
 - a. Build the binary heap (draw the tree at each step) and show the corresponding array
 - b. delete the element with the highest priority, draw the tree at each step of the deleting procedure
 - c. insert the new element with the priority 15 and draw the tree at each step of the insertion procedure