APA: Advanced Programming, Algorithms and Data Structures

Emre Güney, PhD

Master in Bioinformatics for Health Sciences
Universitat Pompeu Fabra

Lectures 3-4

October 1-3, 2019

IMIM Parc de Salut MAR Barcelona
Institut Hospital del Mar d'Investigacions Mèdiques
GRIB
RESEARCH PROGRAMME ON BIOMEDICAL INFORMATICS
UPF UNIVERSITAT POMPEU FABRA

Previously on APA

- Stable matching problem
  - Correctness of an algorithm
  - Efficiency of an algorithm
- Asymptotic bounds of computational complexity
  - (Worst case) Time complexity in terms of input size
  - Big-Oh, Omega and Theta notations (O, Ω, Θ)
- Searching & sorting
  - Buble sort, selection sort, insertion sort, merge sort
  - Exhaustive search (brute force) vs divide & conquer
- Plotting and big-Oh assignments

More on stable matching algorithm

- Symmetry of the problem vs algorithm
  - Stable matching problem is symmetric w.r.t. to men and women
  - The G-S algorithm is asymmetric: If all men put different women as their first choice, they will end up with their first choice
  - The women's preferences disregarded, introducing unfairness to the algorithm

- Deterministic vs undeterministic algorithm

```
while (some man is free and hasn't proposed to every woman) {
        Choose such a man m
```

  - The algorithm does not specify which man should be choosen

COMPLEXITY OF COMMON PYTHON FUNCTIONS

- Lists: `n` is `len(L)`
  - index O(1)
  - store O(1)
  - length O(1)
  - append O(1)
  - == O(n)
  - remove O(n)
  - copy O(n)
  - reverse O(n)
  - iteration O(n)
  - in list O(n)

- Dictionaries: `n` is `len(d)`
- worst case
  - index O(n)
  - store O(n)
  - length O(n)
  - delete O(n)
  - iteration O(n)
- average case
  - index O(1)
  - store O(1)
  - delete O(1)
  - iteration O(n)

Source: MIT OpenCourseWare

See wiki.python.org/moin/TimeComplexity for more info on the (amortized) worst case performance

Binary Insertion sort

BINARY-INSERTION-SORT (A, n) ▷ $A[1 \,..\, n]$

for $j \leftarrow 2$ **to** n

insert key A[j] **into the (already sorted) sub-array** A[1 .. j-1]**.**
Use binary search to find the right position

Binary search with take $\Theta(\log n)$ time.
However, shifting the elements after insertion will still take $\Theta(n)$ time.

Complexity: $\Theta(n \log n)$ comparisons
(n^2) swaps

Source: MIT OpenCourseWare

Merge sort: Merging step

13 7 2 20 12 9 1 11

7 13 | 2 20 | 9 12 | 1 11

2 7 13 20 | 1 9 11 12

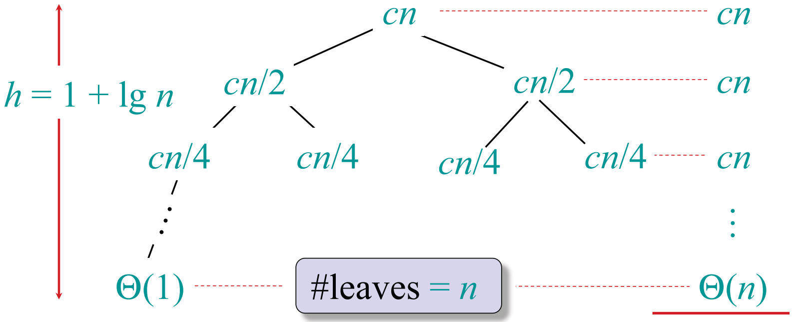
Merging two sorted arrays

20	12
13	11
7	9
2	1

Source: MIT OpenCourseWare

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

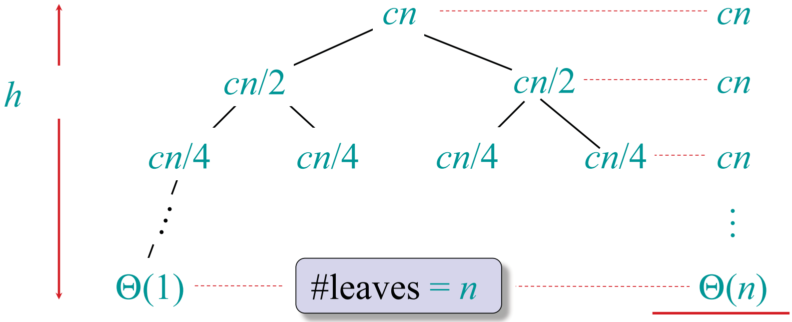


Total ?

Source: MIT OpenCourseWare

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Total $= \Theta(n \lg n)$

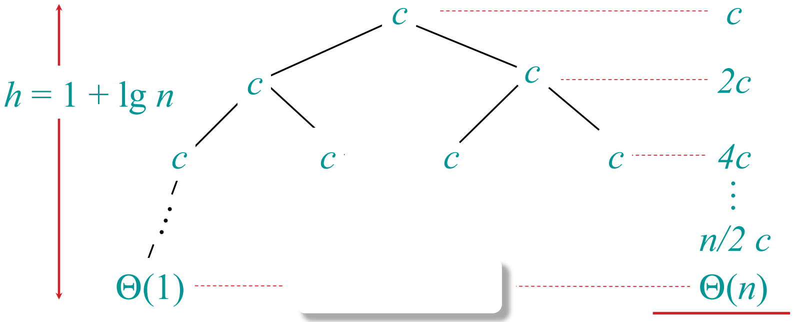
Equal amount of work done at each level

Source: MIT OpenCourseWare

44

Tree for different recurrence

Solve $T(n) = 2T(n/2) + c$, where $c > 0$ is constant.



Note that $1 + ½ + ¼ + \ldots < 2$

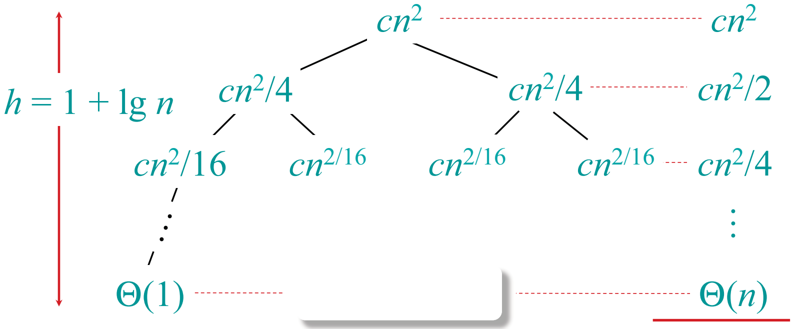
All the work done at the leaves

Total $= \Theta(n)$

Source: MIT OpenCourseWare

Tree for yet another recurrence

Solve $T(n) = 2T(n/2) + cn^2$, $c > 0$ is constant.



Note that $1 + ½ + ¼ + \ldots < 2$

All the work done at the root

Total $= \Theta(n^2)$

Source: MIT OpenCourseWare

Master theorem for recurrence relations

Applicable when the problem can be divided into a subproblems of size n/b (recursively) and the solutions to the subproblems can be combined in $\Theta(n^d)$ time, that is

$$T(n) = a \cdot T(n/b) + \Theta(n^d)$$

where $T(n)$ is a monotonically increasing function, $a \geq 1$, $b \geq 2$, and $d \geq 0$.

$$T(n) = \begin{cases} \Theta(n^d) & \text{, if } (a < b^d) \\ \Theta(n^d \log n) & \text{, if } (a = b^d) \\ \Theta(n^{\log_b^a}) & \text{, if } (a > b^d) \end{cases}$$

The landscape of sorting algorithms

V • T • E	Sorting algorithms [hide]
Theory	Computational complexity theory • Big O notation • Total order • Lists • Inplacement • Stability • Comparison sort • Adaptive sort • Sorting network • Integer sorting • X + Y sorting • Transdichotomous model • Quantum sort
Exchange sorts	Bubble sort • Cocktail shaker sort • Odd–even sort • Comb sort • Gnome sort • Quicksort • Slowsort • Stooge sort • Bogosort
Selection sorts	Selection sort • Heapsort • Smoothsort • Cartesian tree sort • Tournament sort • Cycle sort • Weak-heap sort
Insertion sorts	Insertion sort • Shellsort • Splaysort • Tree sort • Library sort • Patience sorting
Merge sorts	Merge sort • Cascade merge sort • Oscillating merge sort • Polyphase merge sort
Distribution sorts	American flag sort • Bead sort • Bucket sort • Burstsort • Counting sort • Interpolation sort • Pigeonhole sort • Proxmap sort • Radix sort • Flashsort
Concurrent sorts	Bitonic sorter • Batcher odd–even mergesort • Pairwise sorting network
Hybrid sorts	Block merge sort • Timsort • Introsort • Spreadsort • Merge-insertion sort
Other	Topological sorting (Pre-topological order) • Pancake sorting • Spaghetti sort

Source: Wikipedia

The landscape of sorting algorithms

Sorting algorithms	
Theory	Computational complexity theory · Big O notation · Total order · Lists · Inplacement · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting · X + Y sorting · Transdichotomous model · Quantum sort
Exchange sorts	Bubble sort · Cocktail shaker sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Slowsort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort · Weak-heap sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsort · Counting sort · Interpolation sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network
Hybrid sorts	Block merge sort · Timsort · Introsort · Spreadsort · Merge-insertion sort
Other	Topological sorting (Pre-topological order) · Pancake sorting · Spaghetti sort

Source: Wikipedia

- Merge sort in python: $2.2\, n \lg(n)$
- Insertion sort in python: $0.2\, n^2$
- Insertion sort in C: $0.01\, n^2$

$n >\sim 10,000$ merge sort in python better than insertion sort in C

Source: MIT OpenCourseWare

Computational complexity in practice

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Source: www.bigocheatsheet.com

Quick sort (Tony Hoare, 1959)

Suppose that we know a number x such that one-half of the elements of a vector are greater than or equal to x and one-half of the elements are smaller than x.

- Partition the vector into two equal parts ($n - 1$ comparisons)
- Sort each part recursively

- Problem: we do not know x.

- The algorithm also works no matter which x we pick for the partition. We call this number the **pivot**.

- **Observation:** the partition may be unbalanced.

Quick sort: example

pivot

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

Partition

1	2	4	5	3	6	12	9	15	7	10	13	8	11	16	14

Qsort Qsort

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The key step of quick sort is the partitioning algorithm.

Question: how to find a good pivot?

Quick sort: partition

```
def partition(A, left, right) # Pseudo-Python!!
    # A[left..right]: segment to be sorted
    # Returns the middle of the partition with
    #   A[middle] = pivot
    #   A[left..middle-1] ≤ pivot
    #   A[middle+1..right] > pivot
    x = A[left]  # the pivot
    i = left
    j = right
    while i < j:
        while A[i] ≤ x and i ≤ right:
            i = i+1
        while A[j] > x and j ≥ left:
            j = j-1
        if i < j:
            swap(A[i], A[j])
    swap(A[left], A[j])
    return j
```


Source: UPC - Jordi Cortadella

Quick sort partition: example

pivot

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14

6	2	4	5	3	9	12	1	15	7	10	13	8	11	16	14

6	2	4	5	3	1	12	9	15	7	10	13	8	11	16	14

1	2	4	5	3	6	12	9	15	7	10	13	8	11	16	14

1	2	4	5	3	6	12	9	15	7	10	13	8	11	16	14

middle

Quick sort: algorithm

```python
def qsort(A, left, right):

    # A[left..right]: segment to be sorted

    if left < right:
        mid = partition(A, left, right)
        qsort(A, left, mid-1)
        qsort(A, mid+1, right)
```

Quick sort: Hoare's partition

```
def hoare_partition(A, left, right) # Pseudo-Python!!
    # A[left..right]: segment to be sorted
    # Output: The left part has elements ≤ than the pivot
    # The right part has elements ≥ than the pivot
    # Returns the index of the last element of the left part
    x = A[left]  # the pivot
    i = left-1
    j = right+1
    while True:
        i += 1
        while A[i] <  x:
            i += 1
        j -= 1
        while A[j] > x:
            j -= 1
        if i ≥ j:
            return j
        swap(A[i],A[j])
```

Admire a unique piece of art by Hoare:
The first swap creates two sentinels.
After that, the algorithm flies …

Quick sort partition: example

pivot

6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14

First swap: 4 is a sentinel for R; 6 is a sentinel for L → no need to check for boundaries

4	2	8	5	10	9	12	1	15	7	3	13	6	11	16	14
i												j			

4	2	3	5	10	9	12	1	15	7	8	13	6	11	16	14

4	2	3	5	1	9	12	10	15	7	8	13	6	11	16	14

4	2	3	5	1	9	12	10	15	7	8	13	6	11	16	14

j (middle)

Quick sort with Hoare's partition

```
def qsort(A, left, right):

    # A[left..right]: segment to be sorted

    if left < right:
        mid = hoare_partition(A, left, right)
        qsort(A, left, mid)
        qsort(A, mid+1, right)
```

Quick sort: hybrid approach

```
def qsort(A, left, right):     # Pseudo-Python!!
    # A[left..right]: segment to be sorted.
    # K is a break-even size in which insertion sort
    # is more efficient than quick sort.
    if right – left ≥  K:
        mid = hoare_partition(A, left, right)
        qsort(A, left, mid)
        qsort(A, mid+1, right)

def sort(A):
    qsort(A, 0, A.size()-1)
    insertion_sort(A)
```

Quick sort: complexity analysis

- The partition algorithm is $O(n)$.

- Assume that the partition is balanced:

$$T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$$

- Worst case runtime: the pivot is always the smallest element in the vector → $O(n^2)$

- Selecting a good pivot is essential. There are different strategies, e.g.,
  – Take the median of the first, last and middle elements
  – Take the pivot at random

Quick sort: complexity analysis

- Let us assume that x_i is the ith smallest element in the vector.

- Let us assume that each element has the same probability of being selected as pivot.

- The runtime if x_i is selected as pivot is:

$$T(n) = n - 1 + T(i - 1) + T(n - i)$$

Quick sort: complexity analysis

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i))$$

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}T(i-1) + \frac{1}{n}\sum_{i=1}^{n}T(n-i)$$

$$T(n) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1}T(i) \leq 2(n+1)(H(n+1) - 1.5)$$

$H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$ is the Harmonic series, that has a simple approximation: $H(n) = \ln n + \gamma + O(1/n)$.
$\gamma = 0.577\ldots$ is Euler's constant.

$$T(n) \leq 2(n+1)(\ln n + \gamma - 1.5) + O(1) = O(n \log n)$$

Source: UPC - Jordi Cortadella

Quick sort: complexity analysis summary

- Runtime of quicksort:

$$T(n) = O(n^2)$$
$$T(n) = \Omega(n \log n)$$
$$T_{\text{avg}}(n) = O(n \log n)$$

- Be careful: some malicious patterns may increase the probability of the worst case runtime, e.g., when the vector is sorted or almost sorted.

- Possible solution: use random pivots.

Source: UPC - Jordi Cortadella

‘Quicksort’ in practice

- Widely used in practice
- Insertion sort is typically faster for sorting very small arrays
- Quicksort implementations use insertion sort for arrays smaller than a certain threshold and for arrays arising in subproblems
- Exact threshold must be determined experimentally and depends on the machine (commonly around $n < 10$)

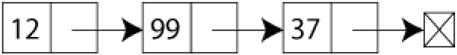
Source: Wikipedia

Data structures recap

- Linked lists
- Stacks and queues
- Heaps & priority queues

Linked list

A collection of items (of any data type) that allows access, insertion and removal of an item at an arbitrary position

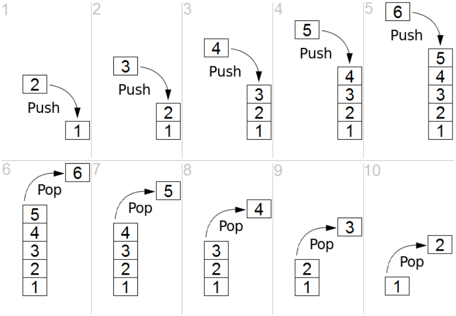


- Each node stores info on the data stored and the next node
- Need to store the info on the head node
- Operations:
  - Insert
  - Remove
  - Find
  - Traverse
  - Size

Source: Wikipedia

Stack

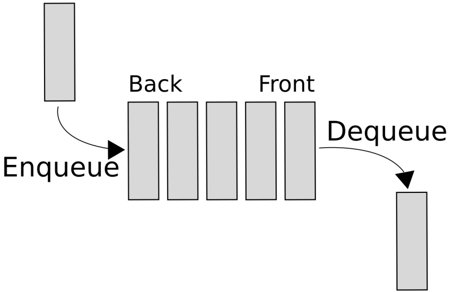
A collection of items where Last In, First Out (LIFO)



- Operations:
  - Push
  - Pop
  - Test empty
  - Test full
  - Peek
  - Size

Queue

A collection of items where First In, First Out (FIFO)

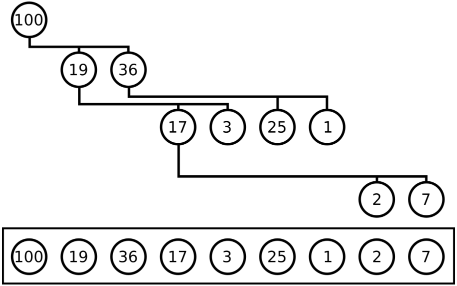


- Operations:
  - Insert
  - Remove
  - Test empty
  - Test full
  - Peek
  - Size

Source: Wikipedia

Priority queue

A collection of items where each item has a 'priority' associated with it and this priority is used in deciding which item will be served next

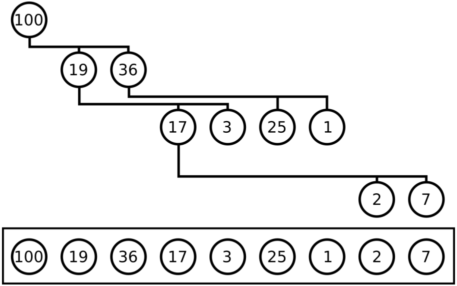


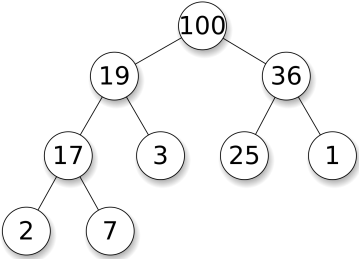
- Operations:
  - Insert with priority
  - Pop (pull highest priority element)
  - Test empty
  - Peek (find the highest priority element)
  - Delete element (with the highest priority)
  - Size

Source: Wikipedia

Priority queue

A collection of items where each item has a 'priority' associated with it and this priority is used in deciding which item will be served next





- Operations:
  - Insert with priority
  - Pop (pull highest priority element)
  - Test empty
  - Peek (find the highest priority element)
  - Delete element (with the highest priority)
  - Size

Source: Wikipedia

Priority Queue

A data structure implementing a set S of elements, each associated with a key, supporting the following operations:

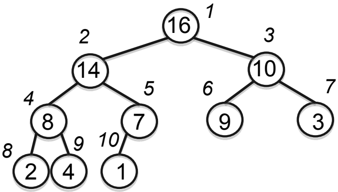
insert(S, x) :	insert element x into set S
max(S) :	return element of S with largest key
extract_max(S) :	return element of S with largest key and remove it from S
increase_key(S, x, k) :	increase the value of element x' s key to new value k (assumed to be as large as current value)

Source: MIT OpenCourseWare

Heap

- Implementation of a priority queue
- An array, visualized as a nearly complete binary tree
- Max Heap Property: The key of a node is $\geq$ than the keys of its children

  (Min Heap defined analogously)



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

4

Source: MIT OpenCourseWare

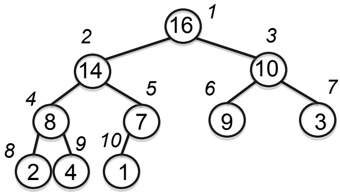
Heap as a Tree

root of tree: first element in the array, corresponding to $i = 1$

parent(i) =i/2: returns index of node's parent

left(i)=2i: returns index of node's left child

right(i)=2i+1: returns index of node's right child



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

No pointers required! Height of a binary heap is O(lg n)

Source: MIT OpenCourseWare

Heap Operations

build_max_heap : produce a max-heap from an unordered array

max_heapify : correct a single violation of the heap property in a subtree at its root

insert, extract_max, heapsort

Source: MIT OpenCourseWare

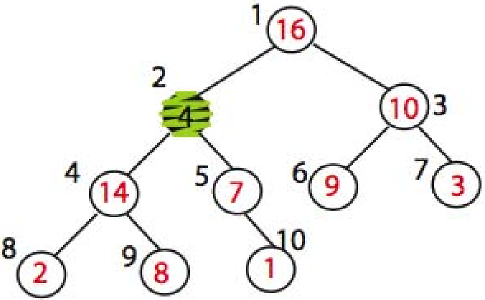
Max_heapify

- **Assume that the trees rooted at left(i) and right(i) are max-heaps**

- If element A[i] violates the max-heap property, correct violation by “trickling” element A[i] down the tree, making the subtree rooted at index i a max-heap

Source: MIT OpenCourseWare

Max_heapify (Example)

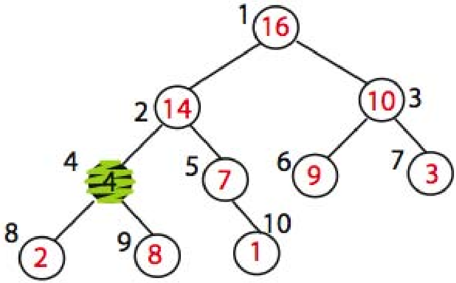


MAX_HEAPIFY (A,2)
heap_size[A] = 10

Node 10 is the left child of node 5 but is drawn to the right for convenience

Source: MIT OpenCourseWare

Max_heapify (Example)

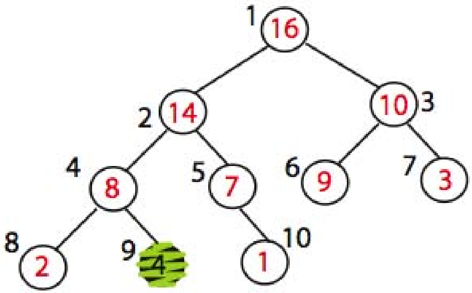


Exchange A[2] with A[4]
Call MAX_HEAPIFY(A,4)
because max_heap property
is violated

9

Source: MIT OpenCourseWare

Max_heapify (Example)



Exchange A[4] with A[9]
No more calls

Time=? O(log n)

10

Source: MIT OpenCourseWare

Max_Heapify Pseudocode

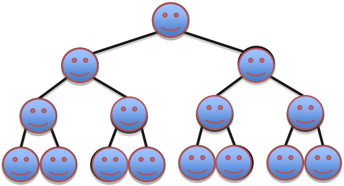
```
l = left(i)
r = right(i)
if (l <= heap-size(A) and A[l] > A[i])
    then largest = l     else largest = i
if (r <= heap-size(A) and A[r] > A[largest])
    then largest = r
if largest ≠ i
    then exchange A[i] and A[largest]
        Max_Heapify(A, largest)
```

11

Source: MIT OpenCourseWare

Build_Max_Heap(A)

Converts $A[1\ldots n]$ to a max heap



```
Build_Max_Heap(A):
    for i=n/2 downto 1
        do Max_Heapify(A, i)
```

Why start at n/2?

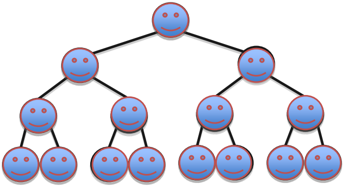
Because elements A[n/2 + 1 … n] are all leaves of the tree
$2i > n$, for $i > n/2 + 1$

Time=? O(n log n) via simple analysis

Source: MIT OpenCourseWare

Build_Max_Heap(A) Analysis

Converts A[1…n] to a max heap

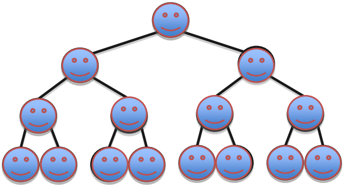


```
Build_Max_Heap(A):
    for i=n/2 downto 1
        do Max_Heapify(A, i)
```

Observe however that Max_Heapify takes O(1) for time for nodes that are one level above the leaves, and in general, O(l) for the nodes that are l levels above the leaves. We have n/4 nodes with level 1, n/8 with level 2, and so on till we have one root node that is lg n levels above the leaves.

Build_Max_Heap(A) Analysis

Converts A[1…*n*] to a max heap



Build_Max_Heap(A):
  for i=n/2 downto 1
    do Max_Heapify(A, i)

Total amount of work in the for loop can be summed as:

$n/4\ (1\ c) + n/8\ (2\ c) + n/16\ (3\ c) + \ldots + 1\ (\lg n\ c)$

Setting $n/4 = 2^k$ and simplifying we get:

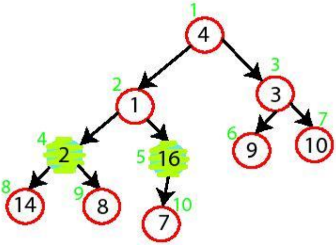
$c\ 2^k(1/2^0 + 2/2^1 + 3/2^2 + \ldots\ (k+1)/2^k\ )$

The term is brackets is bounded by a constant!

This means that Build_Max_Heap is O(n)

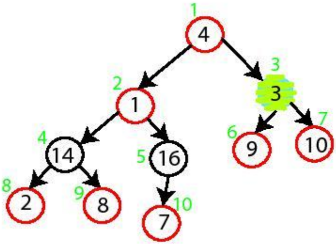
Source: MIT OpenCourseWare

Build-Max-Heap Demo



A									
4	1	3	2	16	9	10	14	8	7

MAX-HEAPIFY (A,5)
no change
MAX-HEAPIFY (A,4)
Swap A[4] and A[8]

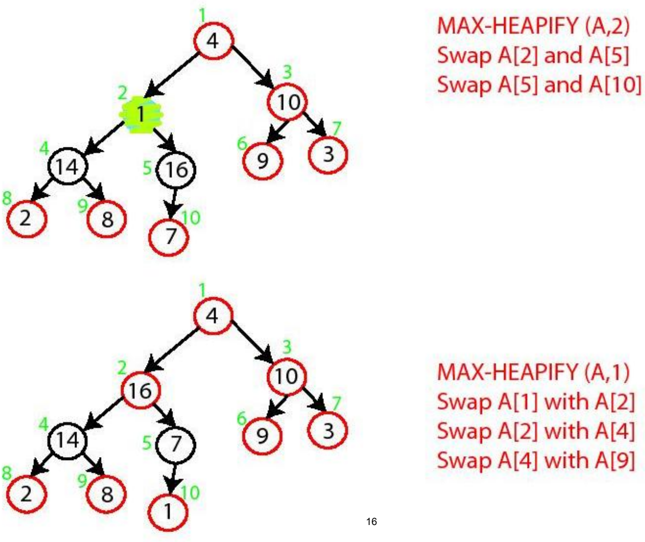


MAX-HEAPIFY (A,3)
Swap A[3] and A[7]

15

Source: MIT OpenCourseWare

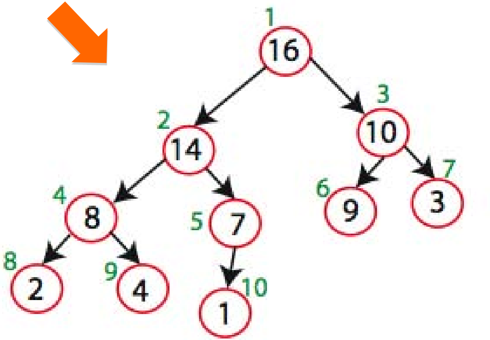
Build-Max-Heap Demo



Source: MIT OpenCourseWare

Build-Max-Heap

A									
4	1	3	2	16	9	10	14	8	7



17

Source: MIT OpenCourseWare

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!

Source: MIT OpenCourseWare

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

3. Swap elements A[*n*] and A[1]:
   now max element is at the end of the array!

4. Discard node *n* from heap
   (by decrementing heap-size variable)

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!

4. Discard node n from heap
   (by decrementing heap-size variable)

5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.

Source: MIT OpenCourseWare

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

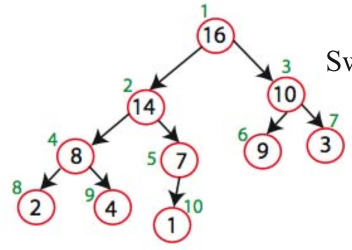
3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!

4. Discard node n from heap
   (by decrementing heap-size variable)

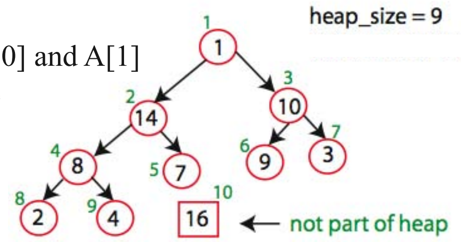
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.

6. Go to Step 2 unless heap is empty.

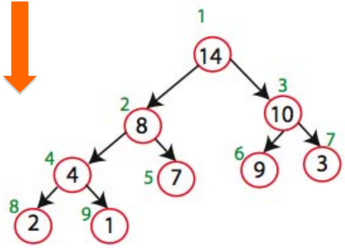
Heap-Sort Demo



Swap A[10] and A[1]

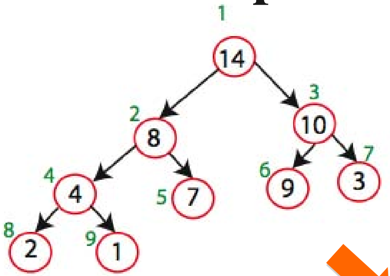


Max_heapify(A,1)

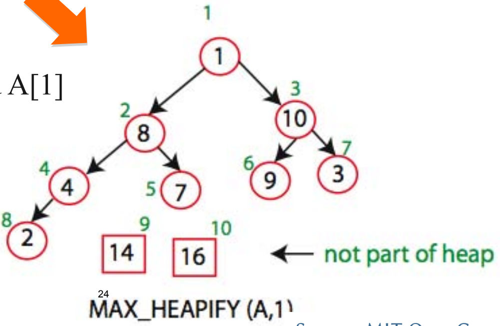


Source: MIT OpenCourseWare

Heap-Sort Demo

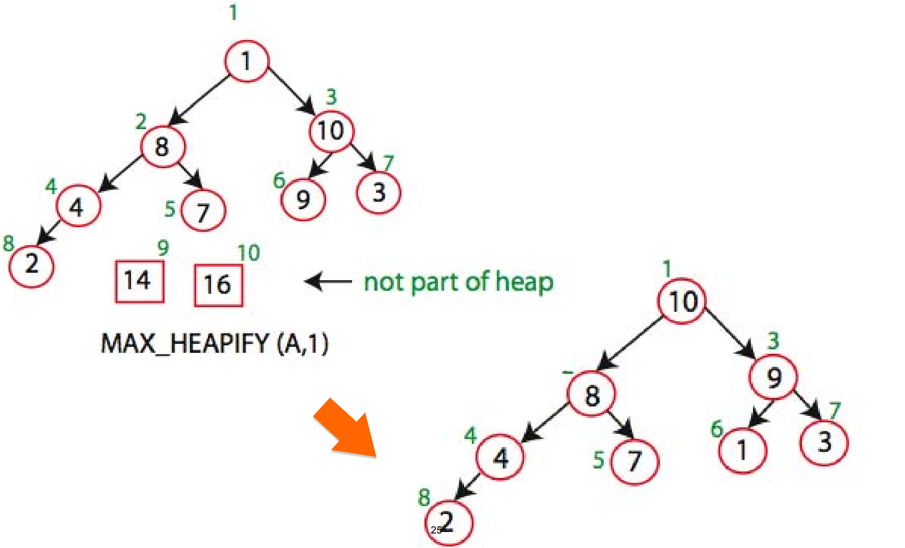


Swap A[9] and A[1]



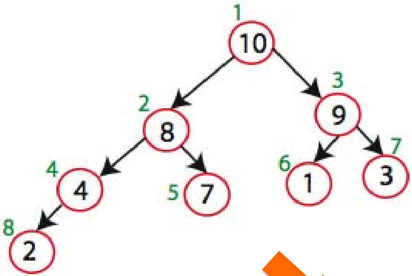
Source: MIT OpenCourseWare

Heap-Sort Demo

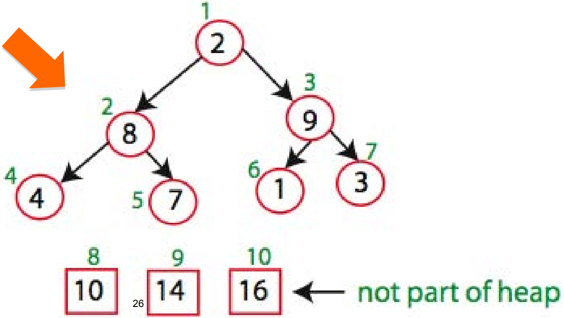


Source: MIT OpenCourseWare

Heap-Sort Demo



Swap A[8] and A[1]



Source: MIT OpenCourseWare

Heap-Sort

Running time:

after n iterations the Heap is empty

every iteration involves a swap and a max_heapify operation; hence it takes O(log n) time

Overall O(n log n)

Source: MIT OpenCourseWare