CompSci 161
Spring 2021 Lecture 3:
InsertionSort and HeapSort

2

InsertionSort

⌇ ← is sorted and contains only elements that were there previously

Idea:

85 ⌇	24	63	45	17	31	96	50
24	85 ⌇	63	45	17	31	96	50
24	63	85 ⌇	45	17	31	96	50

InsertionSort

if always false, we get best-case behavior

for $j \leftarrow 2$ to n **do**
  key $\leftarrow A[j]$
  $i \leftarrow j - 1$
  while $i > 0$ and $A[i] >$key **do**
    $A[i+1] \leftarrow A[i]$
    $i = i - 1$
  $A[i+1] \leftarrow$ key

How many?

always true? worst case behavior

- What is the running time of InsertionSort?

best: $\sum_{j=2}^{n} c$ is $c \cdot n$ is $O(n)$

worst $\sum_{j=2}^{n} j$ is $O(n^2)$

InsertionSort

$A[1 \ldots j-1]$ sorted, values that started there.

for $j \leftarrow 2$ to n **do**
  key $\leftarrow A[j]$
  $i \leftarrow j - 1$
  while $i > 0$ and $A[i] >$key **do**
    $A[i+1] \leftarrow A[i]$
    $i = i - 1$ } removes one inverted pair
  $A[i+1] \leftarrow$ key

- Why is InsertionSort correct?
- What is true *every time* we check the **for** loop? (including the time we find $j > n$ and stop)

4

About that running time ...

- Why are we so concerned with worst case?
- Why not examine average case?

Inverted Pair: $i < j$ but $A[i] > A[j]$

Insertion Sort: $O(n + \mathcal{I})$ # inverted pairs

worst? $\binom{n}{2} \longrightarrow \frac{n(n-1)}{2}$

If all $n!$ permutations equally likely?

Expected # inverted? $\frac{n(n-1)}{2} \times \frac{1}{2} = \frac{n(n-1)}{4}$

is still $O(n^2)$

5

Exp. # inversions in an array

- Ordered pair (i, j) is called an *inversion* if :
  - $i < j$ but
  - j preceeds i in the permutation.
- Six inversions in the permutation $3, 5, 1, 4, 2$.
- If all permutations are equally likely, what is expected number of inversions in a permutation of the first n positive integers?

6 ## Exp. # comparisons by InsertionSort

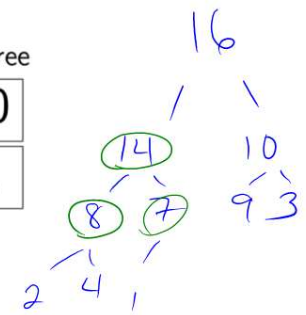
- An expected $n(n-1)/4$ inversions in a permutation of the first n positive integers.
- Average number of comparisons used by InsertionSort to sort n distinct elements?

- $X = \#$ of comparisons used by the algorithm.
- X_i # of comparisons used to insert a_i
- $X = X_2 + X_3 + \ldots + X_n$
- $E(X) = E(X_2) + E(X_3) + \ldots + E(X_n)$
- We now need only to determine each $E(X_i)$.

7 ## What is a heap?

- Any array can be interpreted as a binary tree

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



- Where is the parent of node i? $i/2$
- Where is left child of node i? $2i$
- Where is right child of node i? $2i+1$
- What is a **complete binary tree**?

All levels full except maybe last, which fills left to right

- What is the **max heap property**?

Each element is no smaller than any of its children

8

How tall is a heap?

Claim: A heap with n entries has height $h = \lfloor \log n \rfloor$

- How many nodes in level $i < h$? 2^i
- How many nodes prior to level h?
  $2^0 + 2^1 + 2^2 + \ldots 2^{h-1} = 2^h - 1$
- Level h has at most how many?
  $[1, 2^h]$

$n \geq 2^h - 1 + 1$

$n \leq 2^h - 1 + 2^h$

$= 2^{h+1} - 1$

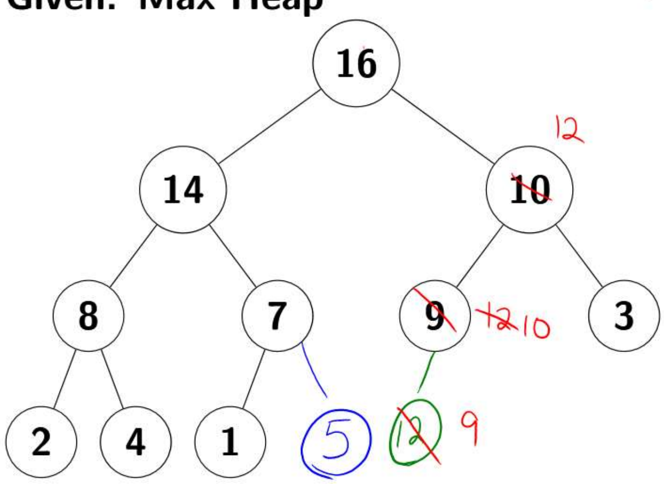
9

How do we maintain a heap?

Keep the tree complete

Given: Max Heap

insert 12?

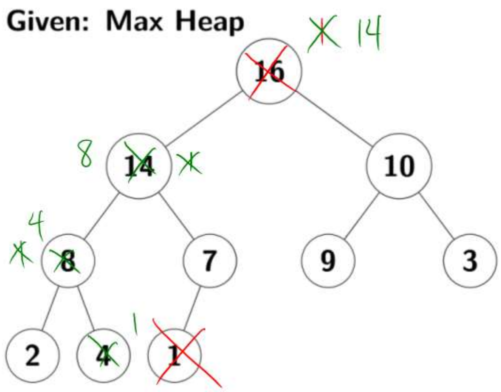


Insert value 5

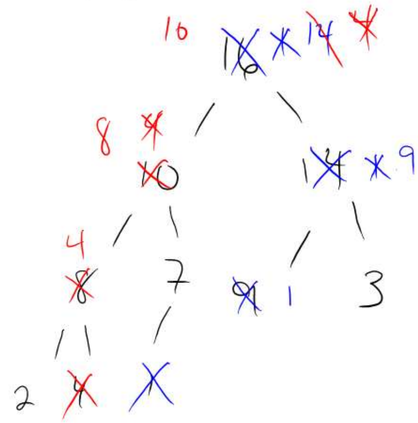
9 ## How do we maintain a heap?

What if...?

Given: Max Heap



Remove max



10 ## HeapSort

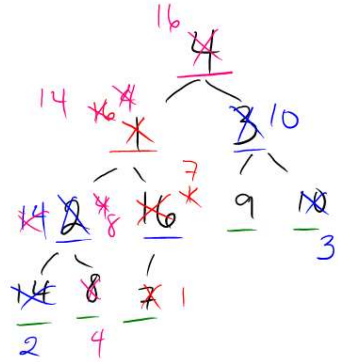
Idea: Use a heap.

- Find max, put max at end
- Then second-max, etc.
- Use the yet-to-be-sorted array as max heap

Heapify: make array into max heap

- Idea 1: insert each into growing heap

11
Heapify: Better way



4	1	3	2	16	9	10	14	8	7

- Treat array as complete tree.
- Where are leaf nodes?

  last $n/2$

- What should we do with non-leaf nodes?
- In which order?

$$\frac{n}{2}\cdot 0 + \frac{n}{4}\cdot 1 + \frac{n}{8}\cdot 2 + \dots$$

12
How long to heapify?

- The cost to insert varies by height.
- Node at height h costs $\mathcal{O}(h)$.
- Cost for total is:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \text{ is } O\left(n \underbrace{\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}}_{2}\right)$$

$$\Theta(n)$$

13 And how, HeapSort