Sorting

Review: Insertion Sort
    Merge Sort
Selection Sort
Heaps

Insertion Sort

key
5 4 6 1 3
1 2 3 4 5 6
2 4 3 6 5 1

θ(n²) algorithm

Readings: 2.1, 2.2, 2.3
6.1, 6.2, 6.3, 6.4
Thursday
Merge Sort

Divide n-element array into two subarrays of \( \frac{n}{2} \) elements each. Recursively sort sub-arrays using mergesort. Merge two sorted subarrays.

\[ \begin{array}{c}
L \quad 2 \quad 4 \quad 5 \quad 7 \\
R \quad 1 \quad 2 \quad 3 \quad 6 \\
\end{array} \]

\( \Theta(n) \) time
\( \Theta(n) \) auxiliary space

A

In-Place Sorting

Numbers re-arranged in the array A with at most a constant number of items stored outside the array at any time.

Insertion Sort: stores key outside array \( \Theta(n^2) \) in-place.

Merge Sort: Need \( \Theta(n) \) auxiliary space \( \Theta(n \log n) \) during merging.

Q: can we have \( \Theta(n \log n) \) in-place sorting?
Selection Sort

1. \( i = 1 \)
2. Find minimum value in list beginning with \( i \)
3. Swap it with the value in the \( i \)th position
4. \( i = i+1 \), stop if \( i = n \)

\( \Theta(n^2) \) time
in-place

\( O(n) \) time
(can we improve to \( O(\log n) \)?)

Heaps (not garbage collected storage!)

A heap is an array object that is viewed as a nearly complete binary tree
**DATA STRUCTURE**

root : A[i]

Node with index i

\[
\text{PARENT}(i) = \left\lfloor \frac{i}{2} \right\rfloor
\]

\[
\text{LEFT}(i) = 2i
\]

\[
\text{RIGHT}(i) = 2i + 1
\]

No pointers!

length[A] : number of elements in the array

heap-size[A] : number of elements in the heap stored within array A

\[
\text{heap-size}[A] \leq \text{length}[A]
\]

**MAX-HEAPS (and MIN-HEAPS)**

max-heap property: For every node i other than the root \( A[\text{PARENT}(i)] \geq A[i] \)

height of a binary heap \( O(lgn) \)

**MAX-HEAPIFY** : \( O(lgn) \) maintains max-heap property

**BUILD-MAX-HEAP** : \( O(n) \) produces max-heap from unordered input array

**HEAP-SORT** : \( O(n \cdot lgn) \) Heap operations insert, extract-max etc \( O(lgn) \)
**MAX-HEAPIFY** \((A, i)\)

\[
\begin{align*}
 & l \leftarrow \text{left}(i) \\
 & r \leftarrow \text{right}(i) \\
 & \text{if } l \leq \text{heap-size}(A) \text{ and } A[l] > A[i] \\
 & \quad \text{then largest} \leftarrow l \\
 & \quad \text{else largest} \leftarrow i \\
 & \text{if } r \leq \text{heap-size}(A) \text{ and } A[r] > \text{largest} \\
 & \quad \text{then largest} \leftarrow r \\
 & \text{if } \text{largest} \neq i \\
 & \quad \text{then exchange } A[i] \text{ and } A[\text{largest}] \\
 & \quad \text{MAX-HEAPIFY}(A, \text{largest})
\end{align*}
\]

Assumes trees rooted at left(i) and right(i) are max-heaps.

A[i] may be smaller than children violating max-heap property.

Let the A[i] value "float down" so subtree rooted at index i becomes a max-heap.
MAX-HEAPIFY (A, 2)
heap-size [A] = 10

Call MAX-HEAPIFY (A, 4)
because max-heap property is violated

no more calls