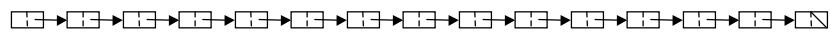




CSE 373: Selection & Simple Sorting

Selection: bits of Chapters 1, 4, 6, 7

Simple Sorting: Chapter 7



The Selection Problem



Goal: Given a list of n numbers, find the k^{th} smallest

Special Cases:

$k = 1$: **findMin()**

$k = n$: **findMax()**

$k = n/2$: the *median* of the list

Any ideas?

Selection Brainstorming



Which of the data structures that we've studied would be appropriate for selection?

- List
- Stack
- Queue
- Tree
- BST
- Hash Table
- Heap

- must be able to store data
- must maintain some sort of ordering information

List-Based Selection



Naive algorithm:

- Insert each element into a second list using `insertSorted()`
- Return the element in the k th position
- Running time?

Slightly improved algorithm:

- Store only the k smallest elements seen so far
- Running time?

Tree-Based Selection



Naive Algorithm:

- **insert()** all elements into a BST
- Traverse the tree using an in-order traversal
- Count off until we reach the k^{th} element
- Running time?

Improved Algorithm?

Heap-Based Selection



Naive Algorithm:

- **buildHeap()** all elements into a min-heap
- Perform **deleteMin()** $k-1$ times
- The next **deleteMin()** returns the target value
- Running time?

Improved Algorithm?

Relating Selection and Sorting



If we were to do selections for $k = 1, 2, \dots, n$, we would end up with a sorted list

– Running time?

Alternatively, if we were to sort our input list, we could do any selection in $O(1)$ time

– Running time?

Motivation for Sorting



- Sorted arrays allow us to do binary searches
- They also allow us to do fast selection
- The mode could be computed trivially in $O(n)$ time if the input was sorted

(but perhaps most importantly...)

- Humans tend to like things in sorted order

How could we use our data structures to sort?

Which would be appropriate? Efficient?

Introduction to Sorting



Sorting: One of the most fundamental algorithms

Input: An array $A[]$ of values and its size, n .

Output: The array stored in sorted order:

if $i < j$ then $A[i] \leq A[j]$, $\forall i, j \leq n$

Goals: sort as quickly as possible

ideally, use $O(1)$ memory (other than $A[]$)

handle pre-sorted lists quickly

Insertion Sort



Insertion Sort: One of the simplest sorting algorithms, based on List ADT `insert()`.

- $n-1$ passes
- after pass i , elements $0..i$ will be in sorted order
- in pass i , we ripple the i^{th} element down the array until it's sorted (with respect to elements $0..i-1$)

Insertion Sort Example



position: 0 1 2 3 4 5

input: 7 4 9 5 8 2

pass 1:

pass 2:

pass 3:

pass 4:

pass 5:

Insertion Sort Analysis



- Why ripple down rather than up?
- Best case input? Running time?
- Worst case input? Running time?

Adjacent Swap Algorithms



A class of algorithms that sort simply by comparing and swapping adjacent elements

- Insertion Sort
- Bubble Sort
- Selection Sort

Inversions



- Given $A[]$, an *inversion* is a pair (i, j) such that $i < j$, but $A[i] > A[j]$.

- How many inversions in our example?

7 4 9 5 8 2

- The number of inversions in $A[]$ equals the number of adjacent swaps required to sort it
 - Why?

Average Case Analysis



Q: What is the average number of inversions in a random input array?

A: Consider an arbitrary list L with n unique values

Consider the reversal of the list L_R

Every pair (i, j) represents an inversion in L or in L_R

The total number of distinct (i, j) pairs is $n(n-1)/2$

On average, half of these will be in L , half will be in L_R

Thus, the average array has $n(n-1)/4$ inversions

So, adjacent swap algorithms run in $\Theta(n^2)$ on average