Lecture 2

Insertion Sort

Chapter 6 (pp 253—300)

Overview

- **Sorting Background**
- **Insertion Sort**
- **Shell Sort**

- **Reference pages 253—300**

Sorting background

- **Why sorting at all?**
  - **Necessary if need to process items in sorted order (e.g., priority queue)**
  - **If set of items is sorted, one can**
    - **Find items faster**
    - **Find particular items in constant time (e.g., min, max, median)**
    - **Find identical items faster**
- **Trade off:**
  - **Faster searching after slow sorting**
  - **Slow searching without sorting**

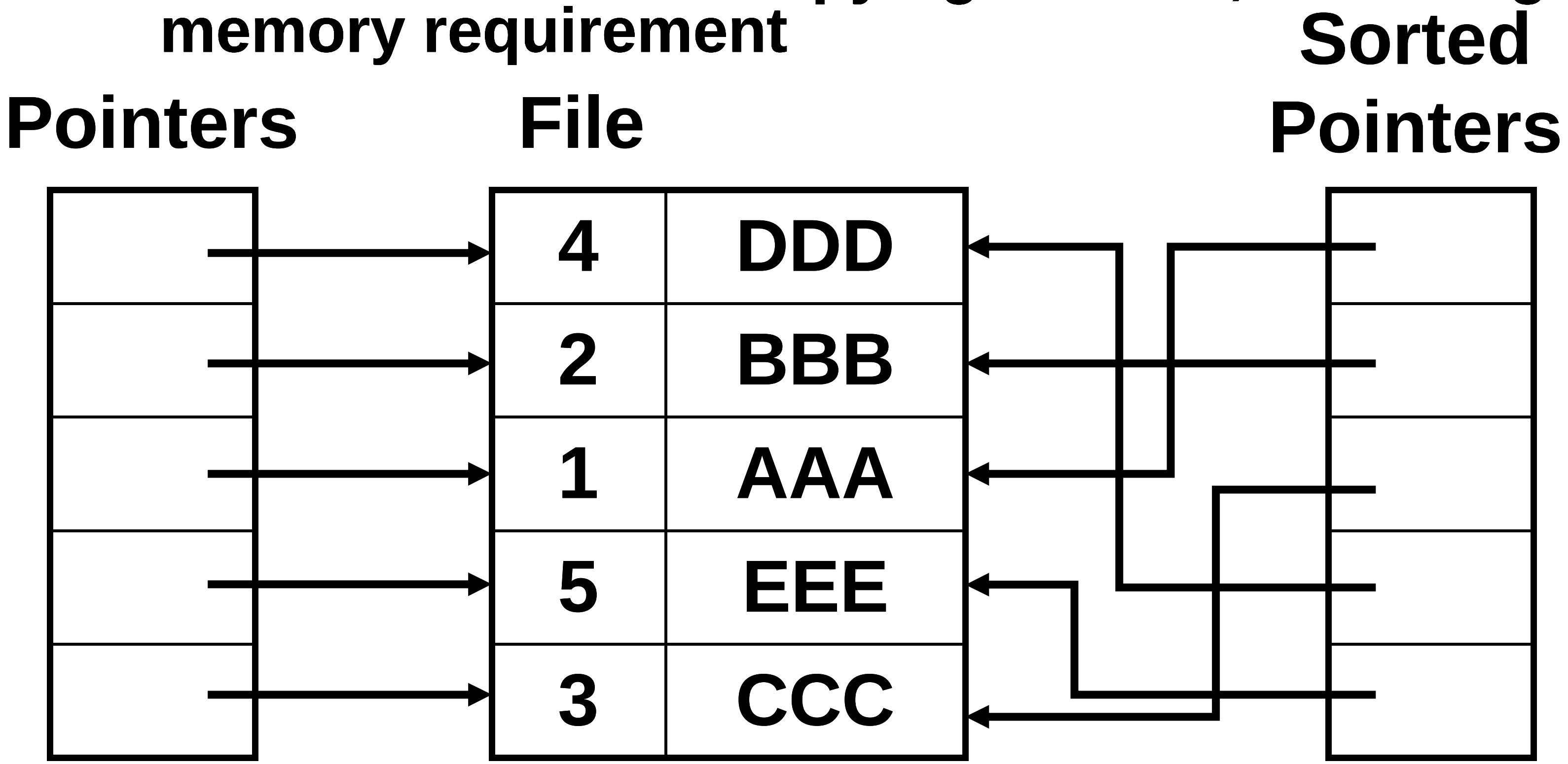
Sorting records

k[i]	r[i]
4	DDD
2	BBB
1	AAA
5	EEE
3	CCC

k[i]	r[i]
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Sorting pointers

- **Instead of records, pointers (or indices) to them are sorted (sorted by address)**
  - **More efficient than copying records, but at higher memory requirement**



Simple insertion sort

- Insert an item into an already sorted array
- Compare item with items in the sorted array from largest to smallest
  - If reverse order, swap items
- Continue until all items are sorted

Insertion sort: algorithm

Sort *n* integers r[0] to r[*n*-1] in ascending order

```
for j ← 1 to n-1
   for i ← j downto 1
      if r[i-1] > r[i]
         r[i-1] ↔ r[i]
      else
         break
```

Insertion sort: example

								j	i	swap
3	1	6	3	2	5	4	5	1	1	yes
1	3	6	3	2	5	4	5	2	2	no
1	3	6	3	2	5	4	5	3	3	yes
1	3	3	6	2	5	4	5	3	2	no
1	3	3	6	2	5	4	5	4	4	yes

Insertion sort: example

								j	i	swap
1	3	3	2	6	5	4	5	4	3	yes
1	3	2	3	6	5	4	5	4	2	yes
1	2	3	3	6	5	4	5	4	1	no
1	2	3	3	6	5	4	5	5	5	yes
1	2	3	3	5	6	4	5	5	4	no

Insertion sort: example

								j	i	swap
1	2	3	3	5	6	4	5	6	6	yes
1	2	3	3	5	4	6	5	6	5	yes
1	2	3	3	4	5	6	5	6	4	no
1	2	3	3	4	5	6	5	7	7	yes
1	2	3	3	4	5	5	6	7	6	no

→ **exit**

Insertion sort comments

- **Only neighbors are swapped and identical items are never exchanged**
  - **Thus, insertion sort is stable**
- **The algorithm builds up a sorted subarray at the start and successively inserts items into it**
- **Move up array to find proper position to insert**

Insertion sort: move vs. swap

Move larger items down the array to make room, insert new record into correct position

```
for j ← 1 to n-1
    temp_r ←  r[j]
    for i ← j downto 1
        if r[i-1] > temp_r
            r[i] ← r[i-1]
        else
            break
    r[i] ← temp_r
```

Making room

Insertion sort: while-loop vs. for-loop

Use `while` instead of `for` in inner loop

i > 0 often tested but seldom false

```
for j ← 1 to n-1
    temp_r ← r[j]
    i ← j
    while i > 0 and r[i-1] > temp_r
        r[i] ← r[i-1]
        i ← i-1
    r[i] ← temp_r
```

Insertion sort: using sentinel

```
for j ← n-1 downto 1
    if r[j] < r[j-1]
        r[j] ↔ r[j-1]
for j ← 2 to n-1
    temp_r ← r[j]
    i ← j
    while r[i-1] > temp_r
        r[i] ← r[i-1]
        i ← i-1
    r[i] ← temp_r
```

Find min item

r[0] ≤ temp_r guaranteed

Insertion sort: efficiency

- **Memory:**
  - **In-place sort, *O*(1)**
- **Time: examine critical operations**
  - **Swaps or moves**
  - **Comparisons**
  - **O(*n*) for finding the sentinel**

Insertion sort: time – best case

Best case: array already sorted

- **For the items 1 to *n*-1**
  - **O(1) copy operation or assignment operation**
  - **O(1) comparison**
- **Result: $O(n)$ (both copy and compare)**

Insertion sort: time – worst case

Worst case: array sorted descendingly

- **For the *i*-th item (*i* from 1 to *n*-1)**
  - **Dominated by *i* comparisons and *i* moves or copy operations**

$$(n\text{-}1) + (n\text{-}2) + \ldots + 1 = (n\text{-}1)*((n\text{-}1) + 1) / 2$$

$$= n^2/2 - n/2 \text{ comparisons or copy operations}$$

- **Result: $O(n^2)$ (both copy and compare)**

Insertion sort: time – average case

Average case: array in random order

- **For the i-th record (i from 1 to n-1)**
  - **On average i / 2 positions from the correct one**
  - **On average (1+i)/2 comparisons and copy operations**

$$((n) + (n-1) + \ldots + 3 + 2) / 2 = (n+1)*(n)/4 - 1/2$$

$$= n^2/4 - n/4 - 1/2 \text{ comparisons/copies}$$

- **Result: $O(n^2)$ (both copy and compare)**

Insertion sort: other improvements?

- **Use memmove() to move all items (to make room for insertion) “at the same time”**
- **Use binary search to find correct position**
- **Swap the minimum item with the first item**

Insertion sort: summary

- **Memory complexity: optimal, *O*(1)**
- **Time complexity:**
  - **Perfectly sorted: $O(n)$**
  - **Average input: very poor $O(n^2)$**
  - **On average $n^2/4$ comparisons and moves**
- **Advantages:**
  - **Easy to implement**
  - **One of the best sorts for sorted input and/or small input sizes**

Shortcomings of insertion sort

- **Perform comparisons between adjacent neighbors**
  - **$O(n^2)$ comparisons (worst-case and average-case)**
- **Perform swaps/moves between adjacent items**
  - **Take several swaps/moves to correct one inversion (pair of integers out of order)**
  - **$O(n^2)$ swaps/moves**

Shell sort: improving insertion sorts

- **Allow comparisons and swaps between non-adjacent neighbors**
- **Perform insertion sort on *k* subarrays of original array**
  - **r[0], r[k], r[2k], r[3k], …**
  - **r[1], r[1+k], r[1+2k], r[1+3k], …**
  - **…**
  - **r[k-1], r[2k-1], r[3k-1], r[4k-1], …**
- **Iterate the process, use a smaller *k* in each pass**
- **In the last pass, use *k* = 1**

Example

- **Given array**
  - **3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2**
- **Arrange into a 2-D array with k = 7 columns**

3 7 9 0 5 1 6		**3 3 2 0 5 1 5**
8 4 2 0 6 1 5	$\rightarrow$	**7 4 4 0 6 1 6**
7 3 4 9 8 2		**8 7 9 9 8 2**

Example

- **Arrange into a 2-D array with k = 3 columns**

– 3 3 2		0 0 1
– 0 5 1		1 2 2
– 5 7 4		3 3 4
– 4 0 6	$\rightarrow$	4 5 6
– 1 6 8		5 6 8
– 7 9 9		7 7 9
– 8 2		8 9

- **Arrange into a single column (k = 1) in the last step**

Shell sort: algorithm

```
for each k (in descending order)
  for j ← k to n-1
    temp_r ← r[j]
    i ← j
    while i ≥ k and r[i-k] > temp_r
       r[i] ← r[i-k]
       i ← i-k
    r[i] ← temp_r
```

Shell sort: example

Pass 1, *k* = 5

3	5	6	3	4	5	1	2

5 0 no

3	5	6	3	4	5	1	2

6 1 yes

3	1	6	3	4	5	5	2

7 2 yes

Shell sort: example

Pass 2, *k* = 1

3	1	2	3	4	5	5	6	1	0	yes
1	3	2	3	4	5	5	6	2	1	yes
1	2	3	3	4	5	5	6	1	0	no
1	2	3	3	4	5	5	6	3	2	no

No more swaps in the remainder of insertion sort

Shell sort: choosing k

- **Define recursively**
  - $h(1) = 1$
  - $h(i + 1) = 3 * h(i) + 1$
- **Let x be the smallest integer such that $h(x) \geq n$**
- **Set the number of passes to be $x - 1$**
- **For pass j, use $k = h(x - j)$**

Shell sort: complexity

- **Analysis is quite sophisticated, will skip that for this course**
- **Sequence of integers of the form 2^p3^q (< n):**

1

2 3 **Bottom to top**

4 6 9 **Right to left**

8 12 18 27

  - $O(n(\log n)^2)$
- **Sequence {1, 3, 7, 15, …, 2^k – 1, …}: $O(n^{1.5})$**
- **Sequence {1, 4, 13, …, 3*h(k-1)* + 1, …}: $O(n^{1.5})$**