Sorting

Data Structures

Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

Sequential Search

- The efficiency of a searching strategy depends on the arrangement of records in the list.
  - Very efficient if the records are ordered
- What is “sequential search”?
  - The search examines the list of records in left-to-right or right-to-left order.
  - p. 334, Program 7.1

```
int seqSearch(element a[], int k, int n)
{/* search a[1:n]; return the least i such that
    a[i].key = k; return 0, if k is not in the array */
   int i;
   for (i = 1; i <= n && a[i].key != k; i++)
       ;
   if (i > n) return 0;
   return i;
}
```

Program 7.1 Sequential search

Sequential Search (contd.)

- An unsuccessful search requires n key comparisons.
  - The worst case time complexity: $O(n)$
- The # of comparisons made in a successful search depends on the position in the array.
  - The average case: $O(n)$

$$\left(\sum_{1 \le i \le n} i\right) / n = (n+1)/2$$

Binary Search

- After a comparison, either the search ends successfully or the size of the unsearched portion of the list is reduced by about one half.
  - After j key comparisons, the unsearched part is at most $\lceil n/2^j \rceil$.
    - $O(\log n)$ comparisons are required in the worst case.

Definitions

- Two important uses of sorting
  - As an aid to searching
  - As a means for matching entries in lists
  - Applications in areas such as optimization, graph theory, and job scheduling as well
- What is "sorting"?
  - Givens
    - A list of records $(R_1, R_2, \ldots, R_n)$, in which each record, R_i, has key value K_i.
  - Finding a permutation σ, such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, 1 < $i \leq n$ -1. The desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \ldots, R_{\sigma(n)})$.

Definitions (contd.)

- The permutation may not be unique since a list could have several identical key values.
- A sorting method is stable if the generated permutation σ_s is unique and has the following properties
  - $K_{\sigma_s(i)} \leq K_{\sigma_s(i+1)}$ for 0 < $i \leq n$ -1
  - If $i < j$ and $K_i == K_j$ in the input list, then R_i precedes R_j in the sorted list.
- We characterize sorting methods into two broad categories.
  - Internal methods
    - Used when the list to be sorted is small enough so that the entire sort can be carried out in main memory
  - External methods
    - Used on larger lists

Definitions (contd.)

- An internal sort -- the list is small enough to sort entirely in main memory
- An external sort is used when there is too much information to fit into main memory.
  - The file must be brought into the main memory in pieces until the entire file is sorted.

Insertion Sort

- The basic step
  - Inserting a new record into a sorted sequence of i records in such a way that the resulting sequence of size i+1 is also ordered.
  - p. 338, Program 7.4
- Begin with the ordered sequence $a[1]$ and successively insert the records $a[2]$, $a[3]$, …, $a[n]$.
  - Complete by making n-1 insertions for a n-record list
  - p. 338, Program 7.5

Insertion Sort (contd.)

- Analysis
  - In the worst case, $insert\,(e, a, i)$ makes i comparisons before making the insertion.
    - The computing time for inserting one record into the ordered list is $O(i)$.
  - The total worst case time is $O(\sum_{i=1}^{n-1}(i+1)) = O(n^2)$
  - Left out of order (LOO)
    - R_i is LOO iff $R_i < \max_{1 \le j < i}\{R_j\}$
    - The insertion step is executed only for those records LOO.
  - Stable
    - Very desirable when only a very few records are LOO (i.e., $k << n$)

```
void insert(element e, element a[], int i)
{/* insert e into the ordered list a[1:i] such that the
    resulting list a[1:i+1] is also ordered, the array a
    must have space allocated for at least i+2 elements */
   a[0] = e;
   while (e.key < a[i].key)
   {
      a[i+1] = a[i];
      i--;
   }
   a[i+1] = e;
}
```

Program 7.4: Insertion into a sorted list

```
void insertionSort(element a[], int n)
{/* sort a[1:n] into nondecreasing order */
    int j;
    for (j = 2; j <= n; j++) {
        element temp = a[j];
        insert(temp, a, j-1);
    }
}
```

Program 7.5: Insertion sort

Insertion Sort (contd.)

- Variations
  - Binary Insertion Sort
    - Reduce the number of comparisons by replacing the sequential searching technique with binary search
    - The number of record moves remains unchanged.
  - Linked Insertion Sort
    - Using linked list representation rather than an array
    - No record moves
    - Retain sequential search

Quick Sort

- The best in average behavior among all the sorting methods we shall be studying
- The pivot key
  - The key currently controlling the insertion
- Step 1: Select a pivot record from among the records to be sorted.
- Step 2: Reorder the records to be sorted.
- Step 3: The records to the left of the pivot and those to its right are sorted independently.
  - Recursion!

Quick Sort (contd.)

pivot

$R_0, \ldots, R_{s(i)-1}$ $\quad$ $R_{s(i)}$ $\quad$ $R_{s(i)+1}, \ldots, R_{n-1}$

May be sorted independently

$\Rightarrow$ Recursion!

- p. 341, Program 7.6
  - Ex. p. 340, Example 7.3
- Analysis
  - The time to position a record in a file of size n is $O(n)$.

Quick Sort (contd.)

- Let $T(n)$ be the time taken to sort a file of n records. Also assume that the file splits roughly into two equal parts each time a record is positioned correctly.
  - $T(n) \le cn + 2T(n/2)$, for some constant c

$$
\begin{aligned}
T(n) &\le cn + 2T(n/2) \\
&\le cn + 2\,(cn/2 + 2T(n/4)) \\
&\le 2cn + 4T(n/4) \\
&\;\;\vdots \\
&\le cn \log_2 n + nT(1) = O(n \log_2 n)
\end{aligned}
$$

- The worst-case behavior is $O(n^2)$.

Quick Sort (contd.)

- The best of the internal sorting methods as far as average computing time is concerned
- **Lemma 7.1** (The average computing time for quick sort)
  - Let $T_{avg}(n)$ be the expected time for quicksort to sort a file with n records. Then there exists a constant k such that $T_{avg}(n) \le kn \log_e n$ for $n \ge 2$.
  - $T_{avg}(n)$
    $$\le cn + \frac{1}{n}\sum_{j=0}^{n-1}(T_{avg}(j) + T_{avg}(n-j-1)) = cn + \frac{2}{n}\sum_{j=0}^{n-1}T_{avg}(j), n \ge 2$$
  - By induction on n

Quick Sort (contd.)

- Variation
  - Quick sort using a median-of-three
    - A better choice for this pivot is the median of the first, middle, and last keys in the current list.

Merge Sort

- How to merge two sorted lists to get a single sorted list?
  - p. 346, Program 7.7
  - $O(n)$ additional space
  - Time complexity: $O(n)$
- Iterative merge sort
  - n sorted lists, each of length 1
  - Merge sublists pairwise to obtain $n/2$ lists of size 2
  - Then merge the $n/2$ lists pairwise, and so on, until a we are left with only one sublist.

```
void merge(element initList[], element mergedList[],
           int i, int m, int n)
{/*  the sorted lists initList[i:m] and initList[m+1:n] are
     merged to obtain the sorted list mergedList[i:n] */
  int j,k,t;
  j = m+1;          /* index for the second sublist */
  k = i;            /* index for the merged list */

  while (i <= m && j <= n) {
    if (initList[i].key <= initList[j].key)
      mergedList[k++] = initList[i++];
    else
      mergedList[k++] = initList[j++];
  }
  if (i > m)
  /* mergedList[k:n] = initList[j:n] */
    for (t = j; t <= n; t++)
      mergedList[t] = initList[t];
    else
    /* mergedList[k:n] = initList[i:m] */
      for (t = i; t <= m; t++)
        mergedList[k+t-i] = initList[t];
}
```

Program 7.7: Merging two sorted lists

Merge Sort (contd.)

- Based on a single merge pass
  - Merge adjacent pairs of sorted segments
  - p. 348, Program 7.8
- p. 348, Program 7.9
- p. 349, Fig. 7.5
- Analysis
  - Several passes over the input
  - The ith pass merges segments of size 2^{i-1}
  - The total # of passes: $\lceil \log_2 n \rceil$
    - Each pass takes $O(n)$ time.
    - Total computing time: $O(n \log n)$
  - Stable

Merge Sort (contd.)

- Recursive merge sort
  - Associate an integer pointer with each record to eliminate the record copying that takes place when Program 7.7 is used
    - $link[1{:}n]$; $link[i]$ gives the record that follows record I in the sorted sublist
  - p. 350, Program 7.10
    - Based on *listMerge* (p. 351, Program 7.11)
    - Return the first position of the resulting chain
  - Analysis
    - Stable
    - Time complexity: $O(n \log n)$

Merge Sort (contd.)

- Summary
  - $O(n \log n)$ computing time both in the worst case and the average case
  - Additional storage requirement
- Variation
  - Natural Merge Sort
    - Make an initial pass over the data to determine the sequences of records that are in order
    - Ex. p. 351, Fig. 7.6

Heap Sort

- Only a fixed amount of additional storage requirement
- $O(n \log n)$ computing time both in the worst case and the average case
- Slightly slower than merge sort
- Utilize the max heap structure
  - Step 1: Insert the n records into an initially empty max heap
  - Step 2: Extract records from the max heap one at a time

Heap Sort (contd.)

- How to adjust a binary tree to establish the heap?
  - p. 353, Program 7.12
  - Time complexity: $O(d)$ if the tree depth is d
- The swap, decrement heap size, readjust heap process is repeated n - 1 times to sort the entire array.
  - On each pass, swap the first an last records in the heap
  - Place the record with the ith highest key in position $n - i + 1$

Heap Sort (contd.)

- p. 354, Program 7.13
  - Suppose $2^{k\text{-}1} \le n < 2^k$ so that the tree has k levels
  - In the first **`for`** loop, `adjust` is called once for each node that has a child
    - The time required for this loop is the sum, over each level, of the # of nodes on a level times the maximum distance the node can move.

$$\sum_{i=1}^{k} 2^{i-1}(k-i) = \sum_{i=0}^{k-1} 2^{k-i-1} i \le n \sum_{i=0}^{k-1} \frac{i}{2^i} < 2n = O(n)$$

Heap Sort (contd.)

- In the second **`for`** loop, `adjust` is called n - 1 times with maximum depth: $\lceil \log_2(n+1) \rceil$
  - Time complexity: $O(n \log n)$
- The total computing time: $O(n \log n)$

- Ex. p. 352, Example 7.7
  - p. 354, Fig. 7.7(a)
  - p. 354, Fig. 7.7(b) (max heap following the first **`for`** loop of *heapsort*)
  - p. 355, Fig. 7.8

Sorting on Several Keys

- Sorting records that have several keys
  - Key labeling: K^1, K^2, $\cdots$, K^r, with K^1 being the most significant key and K^r the least
  - K_i^j : key K^j of record R_i
  - A list of records, R_1, $\cdots$, R_n, is lexically sorted with respect to the keys K^0, K^1, $\cdots$, $K^{r\text{-}1}$ iff for every pair of records i and j, i < j and$(K_i^1, K_i^2, \ldots, K_i^r) \le (K_j^1, K_j^2, \ldots, K_j^r)$
    - $(x_1, x_2, \cdots, x_r) \le (y_1, y_2, \cdots, y_r)$ iff
      - $x_i = y_i$, $1 \le i \le j$ and $x_{j+1} < y_{j+1}$ for some $j < r$, or
      - $x_i = y_i$, $1 \le i \le r$

Radix Sort (contd.)

- Ex. Sorting a deck of poker cards
  - Two keys: K^0 [Suit] and K^1 [Face value]
  - MSD (Most Significant Digit) sort vs. LSD (Least Significant Digit) sort
    - Ex. p. 356
    - MSD or LSD indicate only the order in which the keys are sorted instead of how each key is to be sorted.
- In a radix sort, the sort key is decomposed into digits using radix r.
  - r bins are needed to sort on each digit

Radix Sort (contd.)

- Ex. An LSD radix-r sort
  - n records $(R_1, \cdots, R_n)$
  - Each key has d digits in the range 0 through $r - 1$.
  - p. 358, Program 7.14
    - The bins are implemented as queues.
    - *front*$[i]$ *and rear*$[i]$, $0 \le i < r$
  - Ex. p. 359, Example 7.8 and Fig. 7.9
- Analysis
  - d passes over the data and each pass takes $O(n + r)$ time.
  - Time complexity: $O(d(n + r))$
  - The value of d depends on the choice of the radix r and the largest key.

Summary of Internal Sorting

- Insertion sort is the best sorting method for small n.
- Merge sort has the best worst case behavior.
  - More storage requirement than heap sort
- Quick sort has the best average behavior.
  - But its worst case behavior is $O(n^2)$
- P. 370, Fig. 7.15

External Sorting

- Assume that the file to be sorted resides on a disk.
- The applied overheads when reading/writing from/to a disk
  - Seek time: time taken to position the read/write head to the correct cylinder.
  - Latency time: time until the right sector of the track is under the read/write head.
  - Transmission time: time to transmit the data to/from the disk

External Sorting (contd.)

- A block is the unit of data that is read from or written to the disk at one time.
  - Will usually contain several records
- Runs -- the segments of the input file sorted using internal sort
- The most popular method for sorting on external storage devices is merge sort.
  - It requires only the leading records of the two runs being merged to be present in memory at one time, so it is possible to merge large runs together.

External Sorting (contd.)

- Phase 1: Segments of the input file are sorted using a good internal sort method and then written onto external storage as they are generated.
- Phase 2: The runs generated in phase 1 are merged together following the merge-tree pattern of Fig. 7.4 until only one run is left.

- Ex. p. 377
  - Assumptions
    - A block length of 250 records
    - The input file contains 4500 records (i.e., 18 blocks).

External Sorting (contd.)

- An internal memory capable of sorting at most 750 records (i.e., 3 blocks)
- Another available disk as a scratch pad

- Phase 1: Internally sort 3 blocks at a time
  - Six runs $R_1 \sim R_6$ are obtained and written out to the scratch disk. (p. 377, Fig. 7.19)
- Phase 2: Two blocks of memory are used as input buffers and the third as an output buffer.
  - Blocks of runs are merged from the input buffers into the output buffer. (p. 377, Fig. 7.20)
    - The output buffer is written out onto disk when getting full.
    - The input buffer is refilled with another block from the same run when getting empty.

External Sorting (contd.)

- The time required by the external sort
  - t_{IO} = time to input or output one block $= t_s + t_l + t_{rw}$
    - t_s = maximum seek time
    - t_l = maximum latency time
    - t_{rw} = time to read or write one block of 250 records
  - t_{IS} = time to internally sort 750 records
  - nt_m = time to merge n records from input buffers to the output buffer

External Sorting (contd.)

operation	time
read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36t_{IO} + 6t_{IS}$
merge runs 1-6 in pairs	$36t_{IO} + 4500t_m$
merge two runs of 1500 records each, 12 blocks	$24t_{IO} + 3000t_m$
merge one run of 3000 records with one run of 1500 records	$36t_{IO} + 4500t_m$
total time	$132t_{IO} + 12000t_m + 6t_{IS}$

External Sorting -- k-way Merging

- The # of passes over m runs can be reduced by using a higher-order merge, i.e., k-way merge for $k \geq 2$.
  - Simultaneously merge k runs together
  - The I/O time may be reduced by using a higher-order merge.
  - Ex. k = 4 and m = 16 (p. 380, Fig. 7.22)
  - At most $\lceil \log_k m \rceil$ passes

External Sorting -- *k*-way Merging (contd.)

- The most direct way to determine the next record to output in k-merge is making k-1 comparisons.
  - Time complexity: $O((k\text{-}1)\sum_{1}^{k} s_i)$, where s_i is the size of the i-th run, $1 \le i \le k$
  - With n being the # of records in the file, the total # of key comparisons is

    $$n(k\text{-}1)\log_k m = n(k\text{-}1)\log_2 m/\log_2 k$$

    - The factor $(k\text{-}1)/\log_2 k$