CSIEB0100 Data Structures

Lecture 08 Sorting I : Internal Sorting

Shiow-yang Wu 吳秀陽

Department of Computer Science and Information Engineering

National Dong Hwa University

Lecture material is mostly home-grown, partly taken from slides came with the textbook originally prepared by Professor Jiun-Long Huang of NCTU.

Sorting

- Sorting is one of the most commonly used operations in computer systems.
  - Arrange things in order
  - Ranking
  - Search for things (?)
- Sorting of n elements is to rearrange elements into ascending or descending order.
  - 7, 3, 6, 2, 1 ➔ 1, 2, 3, 6, 7

Sorting (more specifically)

- A list is a collection of records.
- Each record has one or more fields.
- The fields used to distinguish among records are known as keys.
- Sorting is to rearrange records in order based on key values.
- **Example**: A telephone directory is a list or records with three fields: **name**, **address** and **phone number**. Any one of them can be used as key, depending on the application or need.

Why Sorting

- To search for a record with the specified key, we may examine the record one by one until the one with the matching key is found. (*SeqSeach*)
- Sequential search is costly and slow when the list is large. (average # comparisons = (n+1)/2 = O(n))
- We can do much better (O(log$n$)) using binary search (Charter 1) if the list is sorted.
- It is beneficial to sort and store the list if it is to be searched repeatedly.

Elements of a List

- Normally, we define element class to represent records of a list.

```
class Element
{
  public:
    int getKey() const {return key;};
    void setKey(int k) {key = k;};
    …
  private:
    int key;
    // other records
    …
}
```


Sequential Search

- Searching records one by one is known as sequential search.

```
int SeqSearch (Element *f, const int n, const int k)
/* Search a list f with key values f[1].key, …, f[n].key.
Return i such that f[i].key == k. If there is no such record,
return 0 */
{
  int i = n;
  f[0].setKey(k);
  while (f[i].getKey() != k)
    i--;
  return i;
}
```

Sequential Search (contd.)

- The number of comparisons for a record key *i* is *n–i+1*.
- The average number of comparisons for a successful search is

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

which is *O(n)*.
- Binary search is much better than sequential search.

Binary Search on Ordered List

- A binary search takes *O(log n)* time to search an ordered list with *n* records. (see Chapter 1)
- However, humans do not search a phone directory in either sequential or binary way.
- To search for “Wu”, we will look directly at near the end of the directory.
- This is actually an interpolation scheme.
- Both binary search and interpolation scheme rely on the target list to be in order.
- Sorting is therefore a very important operation.

Sorting in Applications

- Another example use of order lists is to compare elements in different lists.
- Sorting is used in many other applications. It is estimated that 25% of all computing time is spent on sorting.
- There is no ideal sorting method for all initial orderings of the target list.
- We therefore need to study different sorting algorithms and know when to use them.

The Sorting Problem

- Given a list of n records (R_1, R_2, …, R_n), each R_i has key value K_i.
- The sorting problem is to find a permutation, σ, such that $K_{\sigma(i)} \leq K_{\sigma(i+1)}$. $1 \leq i \leq n-1$.
- The original list is rearranged into a sorted list ($R_{\sigma(1)}$, $R_{\sigma(2)}$, …, $R_{\sigma(n)}$).
- When the list has several key values that are identical, the permutation, σ, is not unique.

Stable Sort

- We distinguish the permutation, σ_s, from the other as the one with the following properties:
- $K_{\sigma(i)} \leq K_{\sigma(i+1)}$, $1 \leq i \leq n-1$
- If $i<j$ and $K_i==K_j$ in the input list, then R_i precedes R_j in the sorted list. (i.e. keep the order of records in the original list)
- A sorting method that generates the permutation σ_s is *stable*.
- In most cases, we prefer (or even require) stable sorting methods.

In-Place Sorting

- A sorting algorithm is said to be *in-place* if it requires very little additional space besides the initial space holding the records that are to be sorted.
- Normally "very little" is taken to mean that for sorting *n* elements, no more than $O(\log n)$ extra space is required.
- If memory space is rare (eg. embedded systems), then in-place sorting algorithms may be required.

Categories of Sorting Methods

- Internal sorting: Methods used when the list to be sorted is small enough so that the entire list and sorting can be carried out in the main memory.
  - Insertion sort, quick sort, merge sort, heap sort and radix sort.
- External sorting: Methods used on larger lists that don't fit into main memory.
  - Only a portion of data can be loaded into main memory at a time.
  - The entire list must still be sorted.

Insert into a Sorted List

- A basic operation of sorting is to insert an element into a sorted list.

```
void insert(const Element e, Element* list, int i)
{
  while (e.getKey() < list[i].getKey())
  {
    list[i+1] = list[i];  // Shift larger elements
    i--;
  }
  list[i+1] = e;  // put e into the right place
}
```

O(i)

Insertion Sort

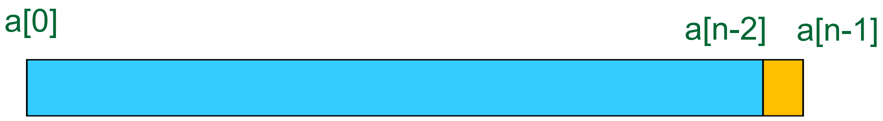
- Simply insert elements one by one into a sorted list (initially empty) of already inserted elements.

```
void InsertSort(Element* list, const int n)
/* Sort list in nondecreasing order of key */
{
  list[0].setKey(MININT);
  for (int j = 2; j <= n; j++)
    insert(list[j], list, j-1);
}
```

$$O(\sum_{i=1}^{n-1}(i+1) = O(n^2))$$

- In the worst case, *insert(e, a, i)* takes *i+1* comparisons. Hence the complexity above.

Insertion Sort Illustrated



- n <= 1 ➔ already sorted. So, assume n > 1.
- a[0:n-2] is sorted recursively.
- a[n-1] is inserted into the sorted a[0:n-2].
- Complexity is $O(n^2)$.
- Usually implemented nonrecursively (see text).

Insert Sort Example 1

- Record R_i is left out of order (LOO) iff

$$R_i < \max_{1 \le j < i}\{R_j\}$$

- The insert step is only needed for LOO records.
- Example 7.1: Assume n = 5 and the input key sequence is 5, 4, 3, 2, 1

j	[1]	[2]	[3]	[4]	[5]
-	5	4	3	2	1
2	4	5	3	2	1
3	3	4	5	2	1
4	2	3	4	5	1
5	1	2	3	4	5

Insert Sort Example 2

- Example 7.2: Assume n = 5 and the input key sequence is 2, 3, 4, 5, 1

j	[1]	[2]	[3]	[4]	[5]	
-	2	3	4	5	1	
2	2	3	4	5	1	O(1)
3	2	3	4	5	1	O(1)
4	2	3	4	5	1	O(1)
5	1	2	3	4	5	O(n)

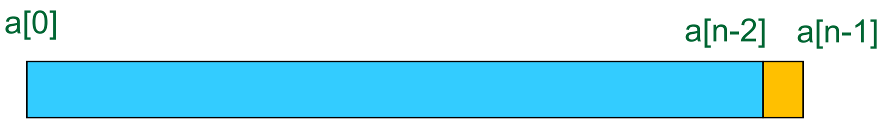
Insert Sort Variations

- Binary insertion sort:
  - Using binary search to reduce the number of comparisons in an insertion sort.
  - The number of records moves remains the same.
- Linked insertion sort:
  - Use a linked list rather than array to represent the list of elements.
  - The number of record moves becomes zero because only the link fields require adjustment.
- Do both at home.

Other $O(n^2)$ Sorting Algorithms

- Selection sort (next slide)
  - Chapter 1
- Bubble sort
- Do both at home.

Selection Sort Illustrated



- n <= 1 ➔ already sorted. So, assume n > 1.
- Move the largest element to the right end of the list.
- Recursively sort the remaining n-1 elements a[0:n-2].
- Complexity is $O(n^2)$.
- Usually implemented nonrecursively.

Quick Sort

- Developed by C. A. R. Hoare.
- Has the best average case performance.
- The basic idea:
  1. Select a pivot record *p* from the list.
  2. Reorder the list so that for all records to the left of the *p*, say *l*, $l.key \leq p.key$; and for all records to the right of the *p*, say *r*, $r.key > p.key$.
  3. Recursively Quick Sort the left and right sublists independently.
- Easily parallelizable due to the independent sort of the sublists.

Quick Sort Illustrated

6	2	8	5	11	10	4	1	9	7	3

Use 6 as the pivot.

2	5	4	1	3	6	7	9	10	11	8

Sort left and right segments recursively.

Quick Sort Function

```
void QuickSort(Element *list, const int left, const int right)
{
  if (left < right) {
    int i = left, j = right + 1, pivot = list[left].getKey();
    do {
      do i++; while (list[i].getKey() <= pivot);
      do j--; while (list[j].getKey() > pivot);
      if (i < j) InterChange(list, i, j);
    } while (i < j);
    InterChange(list, left, j);

    QuickSort(list, left, j-1);
    QuickSort(list, j+1, right);
  }
}
```

Quick Sort Example

- Example 7.3: The input list has 10 records with keys (26, 5, 37, 1, 61, 11, 59, 15, 48, 19).

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	Left	Right
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Choice of Pivot

- Pivot is leftmost element in list that is to be sorted.
  - When sorting a[6:20], use a[6] as the pivot.
  - Textbook implementation does this.
- Randomly select one of the elements to be sorted as the pivot.
- When sorting a[6:20], generate a random number r in the range [6, 20]. Use a[r] as the pivot.

Choice of Pivot: Median-of-Three

- From the leftmost, middle and rightmost elements, select the one with median key as pivot, i.e. pivot = median{K_l, $K_{(l+r)/2}$, K_r}.
  - When sorting a[6:20], examine a[6], a[13] ((6+20)/2), and a[20]. Select the element with median (i.e., middle) key.
  - If a[6].key = 30, a[13].key = 2, and a[20].key = 10, a[20] becomes the pivot.
  - If a[6].key = 3, a[13].key = 2, and a[20].key = 10, a[6] becomes the pivot.
  - If a[6].key = 30, a[13].key = 25, and a[20].key = 10, a[13] becomes the pivot.

Analysis of Quick Sort

- In QuickSort(), list[n+1] has been set to have a key at least as large as the remaining keys.
- QuickSort complexity
  - The worst case is $O(n^2)$
  - If each time a record is correctly positioned, the left and right sublists are of the same size. Assume T(n) is the time taken to sort a list of size n:

    $T(n) \le cn + 2T(n/2)$, for some constant c

    $\le cn + 2(cn/2 + 2T(n/4))$

    $\le 2cn + 4T(n/4)$

    …

    $\le cn \log_2 n + T(1) = O(n \log n)$
- Quick sort is unstable. (why?)

Average Complexity of Quick Sort

- **Lemma 7.1**: Let $T_{avg}(n)$ be the expected time for function QuickSort to sort a list with *n* records. Then there exists a constant *k* such that $T_{avg}(n) \leq kn \log_e n$ for $n \geq 2$.
- This means that the average time complexity of Quick Sort is $O(n \log n)$.
- The lemma can be proved by induction. (read details in the textbook)

Space Complexity of Quick Sort

- While insertion sort only needs additional space for a record, quick sort needs stack space for recursion.
- If the lists split evenly (best case), the maximum recursion depth would be $\log n$ and the stack space is of $O(\log n)$.
- The worst case is when the lists split into a left sublist of size $n–1$ and a right sublist of size 0 at each level of recursion. In this case, the recursion depth is *n*, the stack space of $O(n)$.
- The worst case stack space can be reduced by a factor of 4 since right sublists of size less than 2 need not be stacked.
- Asymptotic reduction in stack space can be achieved by sorting smaller sublists first. In this case the additional stack space is at most $O(\log n)$.

C++ STL sort Function

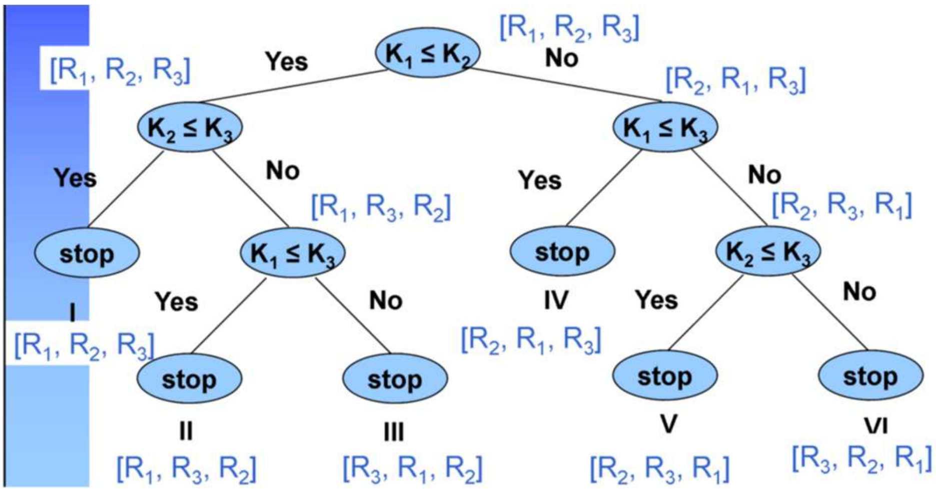
- The performance of Quick Sort can be improved by stopping recursion when segment size is small (say <= 15) and sort these small segments using insertion sort.
- The C++ STL sort function uses Quick Sort but
  - Switch to heap sort when number of subdivisions exceeds some constant times $\log_2 n$.
  - Switch to insertion sort when segment size becomes small.

How Fast can We Sort ?

- So far both insertion sorting and quick sorting have worst-case complexity of $O(n^2)$.
- If we restrict the question to sorting algorithms in which the only operations permitted on keys are comparisons and interchanges, then $O(n \log n)$ is the best possible time.
- This can be shown by using a tree that describes the sorting process. Each vertex of the tree represents a key comparison, and the branches indicate the result.
- Such a tree is called **decision tree**.

Decision Tree for Insertion Sort

- Apply insertion sort on R1, R2, R3



Decision Tree Analysis of Sorting

- **Theorem 7.1**: Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!)+1$
  - When sorting n elements, there are $n!$ different possible results (permutations).
  - Thus, every decision tree for sorting must have at least $n!$ leaves.
  - A decision tree is also a binary tree, which can have at most 2^{k-1} leaves if its height is $k = \log_2(2^{k-1})+1$.
  - The height must be at least $\log_2(n!)+1$.

Decision Tree Analysis of Sorting

- **Corollary**: Any algorithm that sorts only by comparisons must have a worst-case computing time of $\Omega(n \log n)$.
  - By Theorem 7.1, there is a path of length $\log_2(n!)$
  - $n!=n(n-1)\ldots(2)(1) \geq (n/2)^{n/2}$
  - $\log_2(n!) \geq (n/2)\log_2(n/2) = \Omega(n \log n)$

Merge Sort

- Partition the n > 1 elements into two smaller instances.
- First ceil(n/2) elements define one of the smaller instances; remaining floor(n/2) elements define the second smaller instance.
- Each of the two smaller instances is sorted recursively.
- The sorted smaller instances are combined using a process called merge.
- Complexity is *O(n log n)*.
- Usually implemented nonrecursively.

Merging Two Sorted Lists

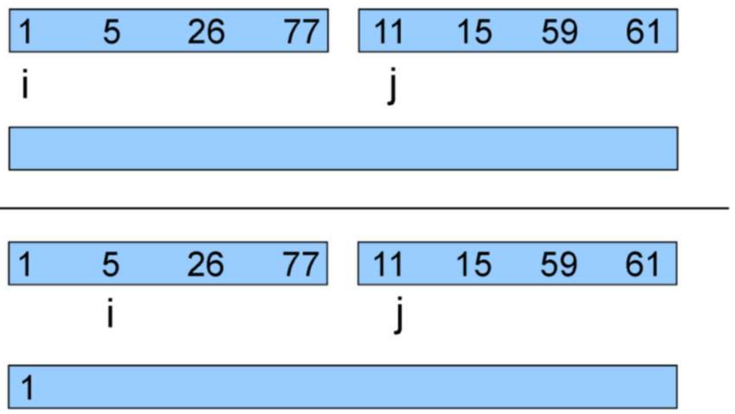
- Merge two lists stored in initList[l:m] and iniList[m+1:n] and produce the result in mergedList[l:n].

```
void merge(Element *initList, Element *mergedList,
           const int l, const int m, const int n)
{
  for (int i1 = l, i2 = m+1, iResult = l;  i1 <= m && i2 <= n;  iResult++)
    if (initList[i1].getKey() <= initList[i2].getKey()) {
      mergedList[iResult] = initList[i1];
      i1++;
    }
    else {
      mergedList[iResult] = initList[i2];
      i2++;
    }
    if (i1 > m)  // copy remaining elements of the second list
      for (int t = i2; t <= n; t++) mergedList[iResult+t-i2] = initList[t];
    else // copy remaining elements of the first list
      for (int t = i1; t <= m; t++) mergedList[iResult+t-i1] = initList[t];
}
```

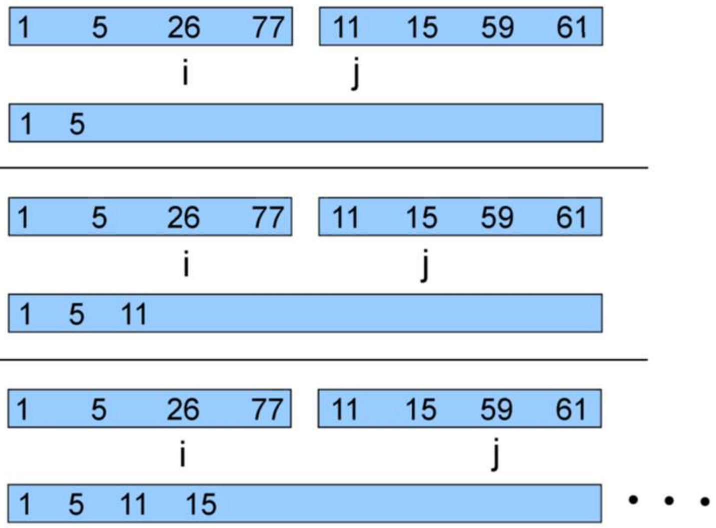
$O(n - l + 1)$

Merge Example

- Merge two sorted lists (1, 5, 26, 77) and (11, 15, 59, 61)



Merge Example



Analysis of Simple Merging

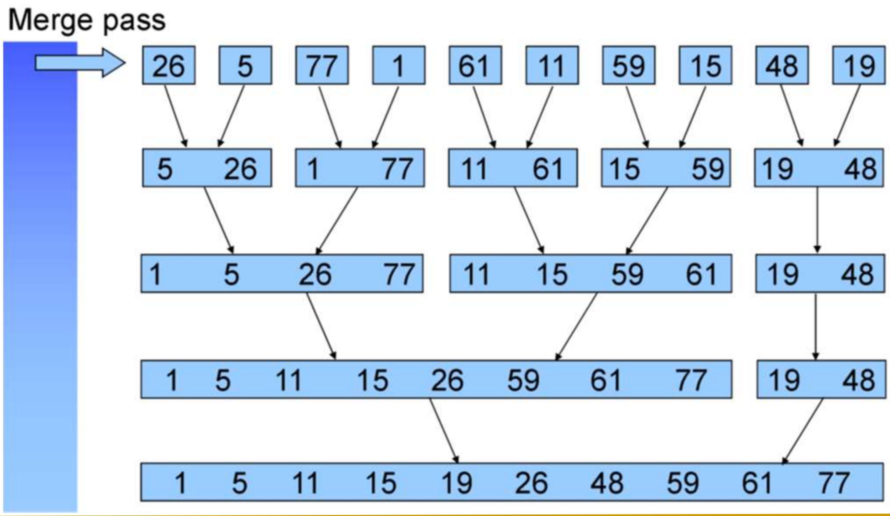
- If an array is used, additional space for *n–l+1* records is needed.
  - *n-l+1* = the number of elements to be merged
  - Time complexity of SimpleMerge is linear.
- If linked list is used instead, then additional space for *n–l+1* links is needed.

Iterative Merge Sort

- Treat the input as n sorted lists, each of length 1.
- Lists are merged by pairs to obtain n/2 lists, each of size 2 (if n is odd, the one list is of length 1).
- The n/2 lists are then merged by pairs, and so on until we are left with only one list.

Merge Tree

- Input list = (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

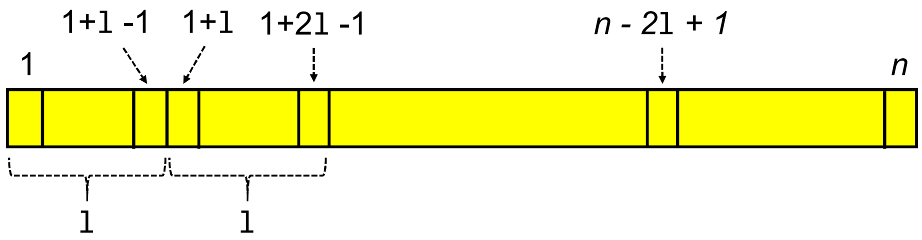


Merge Pass Function

```
void MergePass(Element *initList, Element *resultList,
const int n, const int l)
/* One pass of merge sort. Adjacent pairs of sublists of
length l are merged from initList to resultList. n is the
number of records in initList */
{
  int i;
  for (i = 1; i <= n - 2*l + 1; i += 2*l)
    merge(initList, resultList, i, i+l-1, i+2*l-1);
  // merge remaining list of length < 2*l
  if ((i+l-1) < n)  // merge remaining two sublists
    merge(initList, resultList, i, i+l-1, n);
  else  // copy the remaining one sublist
    for (int t = i; t <= n; t++)
      resultList[t] = initList[t];
}
```


Merge Pass Illustrated

- Each pass merges successive pairs of lists of length l from start (1) to end (n).



- It's easy to understand the loop index setting by starting from the lowest index(*1*), extending it to *i* and taking care of the end(*n*). (figure it out!!)

Iterative Merge Sort

```
void MergeSort(Element *list, const int n)
{
  Element *tempList = new Element[n+1];
  // l is the length of the sublist
  for (int l = 1; l < n; l *= 2) {
    MergePass(list, tempList, n, l);
    l *= 2;
    MergePass(tempList, list, n, l);
  }
  delete [] tempList;
}
```

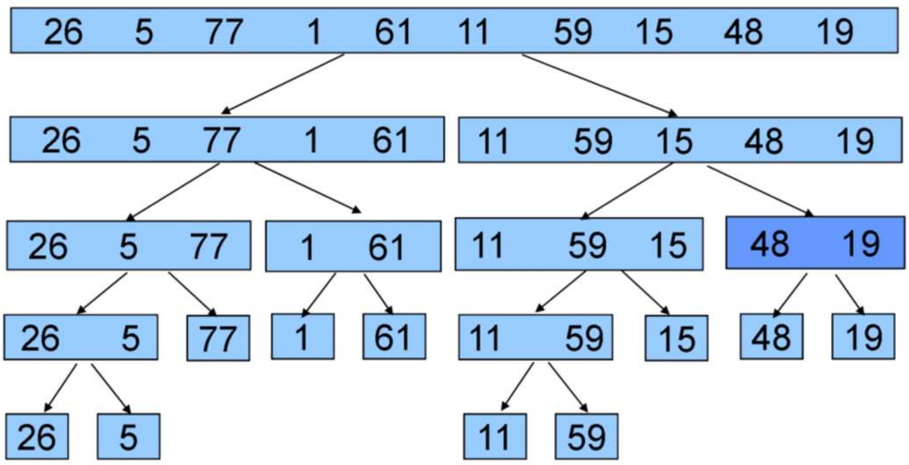

Analysis of Iterative MergeSort

- Total of $\lceil \log_2 n \rceil$ passes are made over the data. Each pass of merge sort takes *O(n)* time.
- The total of computing time is *O(n log n)*.
- MergeSort is stable. (Why ?)

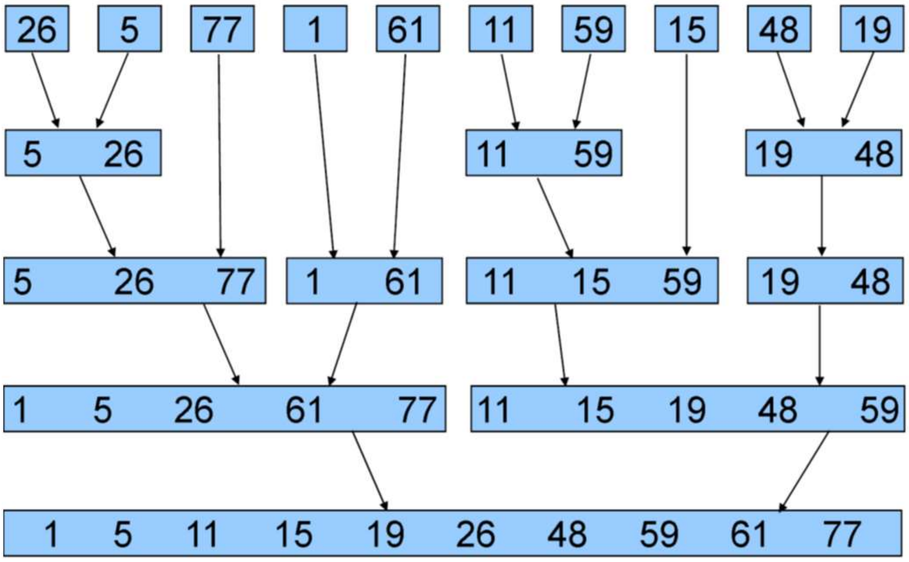
Recursive Merge Sort

- Recursive merge sort divides the list to be sorted into two roughly equal parts:
  - the left sublist [left : (left+right)/2]
  - the right sublist [(left+right)/2 +1 : right]
- These sublists are sorted recursively, and the sorted sublists are merged.
- To avoid copying, the use of a linked list (integer instead of real link) for sublist is desirable.
- The recursive merge sort is stable.

Sublist Partitioning for Recursive Merge Sort



Merging for Recursive Merge Sort



Merging Two Linked Lists

```
int ListMerge(Element *list, const int start1, const int start2)
// Sorted linked lists indexed by start1 and start2 are merged. The index of
// the sorted list is returned. Integer links are used.
{
  int i1, i2, iResult = 0;
  for (i1 = start1, i2 = start2; i1 && i2; )
    if (list[i1].key <= list[i2].key) {
      list[iResult].link = i1;
      iResult = i1; i1 = list[i1].link;
    }
    else {
      list[iResult].link = i2;
      iResult = i2; i2 = list[i2].link;
  }
  // chain the remaining list of elements
  if (i1 == 0) list[iResult].link = i2;
  else list[iResult].link = i1;
  return list[0].link;
}
```

Recursive Merge Sort

```
int rMergeSort(Element *list, const int left, const
int right)
// List (list[left],... ,list[right]) is to be sorted.
// The link field in each record that is initially 0.
// Return the index of the 1st element of sorted list.
// list[0] is for intermediate results in ListMerge.
{
  if (left >= right) return left;
  int mid = (left + right)/2;
  return ListMerge(list, rMergeSort(list, left, mid),
                      rMergeSort(list, mid+1, right));
}
```

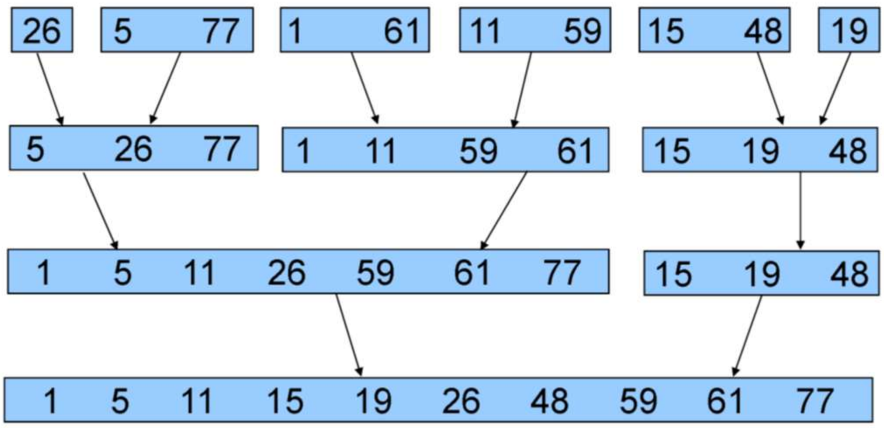
O(n log n)

Natural Merge Sort

- Natural merge sort takes advantage of the prevailing order within the list before performing merge sort.
- It runs an initial pass over the data to determine the sublists of records that are in order.
- Then it uses the sublists for the merge sort.
- It is natural because we do not artificially break the sublists that are already in order.

Natural Merge Sort Example

- With input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

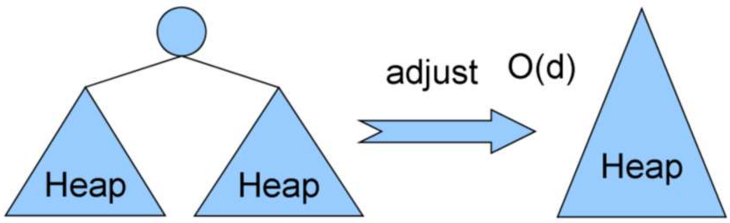


Heap Sort

- Merge sort needs additional storage space proportional to the number of records in the file being sorted, even though its computing time is O(n log n).
- O(1) space merge only needs O(1) space but the sorting algorithm is much slower.
- We will see that heap sort only requires a fixed amount of additional storage and achieves worst case and average computing time *O(n log n).*
- Heap sort uses the max-heap structure.
- Heap sort is unstable.

Heap Sort (contd.)

- The n records are first inserted into an initially empty max heap.
- Next, the records are extracted from the max heap one at a time to form the sorted list.
- A special function **adjust()** is used to create the initial heap faster than from an empty max heap.



Adjusting Max Heap

```
void adjust(Element *tree, const int root, const int n)
// Adjust the binary tree with root root into heap.
// The left and right subtrees already satisfy the heap property.
// No node has index greater than n.
{
   Element e = tree[root];
   int j, k = e.getKey();
   for (j = 2*root; j <= n; j *= 2)
   { // first find max of left and right child
     if (j < n) if (tree[j].getKey() < tree[j+1].getKey()) j++;
     // compare max child with k. If k is max, then done
     if (k >= tree[j].getKey()) break;
     tree[j/2] = tree[j]; // move jth record up the tree
   }
   tree[j/2] = e;
}
```

Heap Sort

```
void HeapSort(Element *list, const int n)
/* The list list = (list[1], ... , list[n]) is sorted
into nondecreasing order of the field key. */
{
  for (int i = n/2; i >= 1; i--) // heapify, n/2 is the
    adjust(list, i, n);          // parent of the last

  for (int i = n-1; i >= 1; i--)  // sort
  {
    Element t = list[i+1]; // swap the first and last
    list[i+1] = list[1];
    list[1] = t;
    adjust(list, 1, i);  // recreate heap
  }
}
```


Analysis of Heap Sort

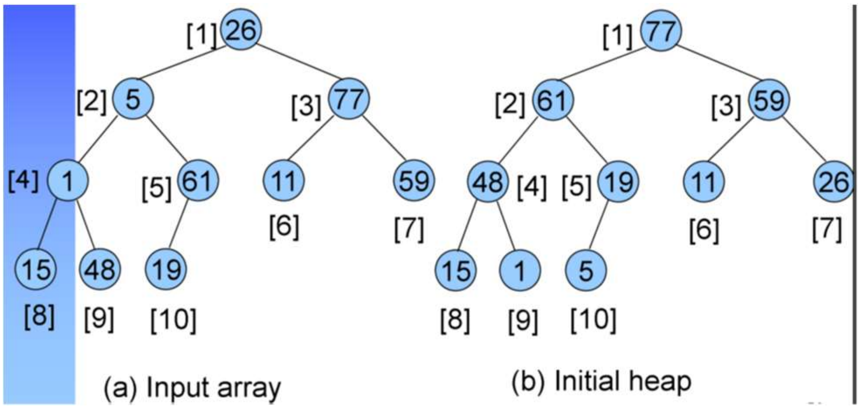
- Time complexity of the first for loop

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

- In the next for loop, adjust is called n - 1 times with max tree depth $\lceil \log_2(n+1) \rceil$. The swap is done n - 1 times.
- Therefore, the total complexity is *O(n log n)*.
- The only additional space needed is the one element for swapping.

Heap Sort Example 1

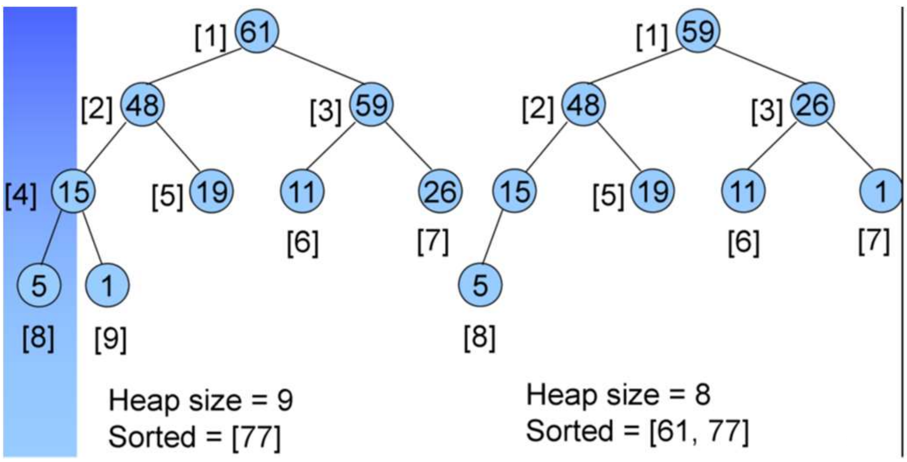
- We represent the input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19) as a binary tree. The first for loop turns it into the initial max heap.



(a) Input array (b) Initial heap

Heap Sort Example 2

- The first two passes of the sort phase



Summary of Comparison Sort

Name	Average Case	Worst Case	Extra Memory	Stable
Selection sort	$O(n^2)$	$O(n^2)$	O(1)	Yes
Insertion sort	$O(n^2)$	$O(n^2)$	O(1)	No
Merge sort	O(n log n)	O(n log n)	O(n) O(1)*	Yes
Heapsort	O(n log n)	O(n log n)	O(1)	No
Quicksort	O(n log n)	$O(n^2)$	O(log n)	No

From Wikipedia

Sorting on Several Keys

- A list of records are said to be sorted with respect to the keys $K^1, K^2, \ldots, K^r$ iff for every pair of records i and j, $i < j$ and $(K^1_i, K^2_i, \ldots, K^r_i) \le (K^1_j, K^2_j, \ldots, K^r_j)$.
- The r-tuple $(x_1, x_2, \ldots, x_r)$ is less than or equal to the r-tuple $(y_1, y_2, \ldots, y_r)$ iff either $x_i = y_i$, $1 \le i \le j$, and $x_{j+1} < y_{j+1}$ for some $j < r$ or $x_i = y_i$, $1 \le i \le r$.
- **Example**: sorting a deck of cards: suite and face value (i.e. two keys).
  - K^1 : Club < Diamond < Heart < Spade
  - K^2 : 2 < 3 < … < 10 < J < Q < K < A

Sorting Several Keys

- Two popular ways to sort on multiple keys.
  - Most-Significant-Digit-first (MSD) sort: Sort on the most significant key K^1 into multiple piles (each having the same value for K^1). For each pile, sort on the second significant key K^2, and so on. Then piles are combined.
  - Least-Significant-Digit-first (LSD) sort: The other way is to sort on the least significant digit first, and so on. (Not exactly the same way as MSD. Figure it out!)
- LSD is simpler since the piles and subpiles do not need to be sorted independently. (Why?)

Sorting Several Keys (contd.)

- LSD and MSD only define the order in which the keys are to be sorted.
- They do not specify how each key is sorted.
- LSD and MSD can be used even when there is only one key.
  - E.g., if the keys are numeric, then each decimal digit may be regarded as a subkey. => Radix sort.

Radix Sort

- In Radix Sort, we decompose the sort key using some radix *r*.
  - In a Radix-*r* Sort, the number of bins needed is *r*.
- Assume the records $R_1, R_2, \ldots, R_n$ to be sorted based on a radix of *r*. Each key has *d* digits in the range of *0* to *r-1*. (Thus, *r* bins.)

Radix Sort (contd.)

- Assume each record has a link field.
- Then the records in the same bin are linked together into a chain:
  - f[i], $0 \le i < r$ (the pointer to the first record in bin i)
  - e[i], (the pointer to the end record in bin i)
  - The chain will operate as a queue.
  - Each record object is assumed to have a public attribute array key[d], 0 ≤ key[i] < r, 0 ≤ i < d. (the keys)

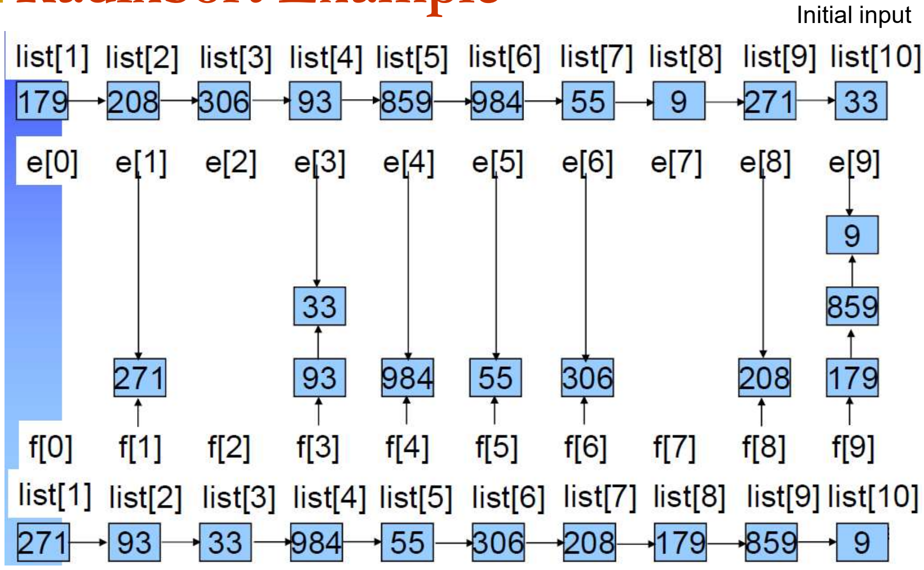
RadixSort Function

```
void RadixSort(Element *list, const int d, const int n)
// Sort list=(list[1],...,list[n]) on the keys key[0],... ,key[d-1] (d digits)
// The range of each key is 0<=key[i]<radix.  radix is a constant.
// Sorting within a key is done using a bin sort.
{
   int i, j, e[radix], f[radix]; // queue pointers
   for (i = 1; i < n; i++) list[i].link = i+1; // link into a chain
   list[n].link = 0; int current = 1; // starting at current element
   for (i = d-1; i >= 0; i--) // sort on key key[i]
   {
     for (j = 0; j < radix; j++) f[j] = 0; // initialize bins to empty
     for (; current; current = list[current].link) {
       // put all records into queues
       int k = list[current].key[i]; // ith key of the current element
       if (f[k] == 0) f[k] = current; // empty queue, current is the first
       else list[e[k]].link = current; // otherwise, link current to the end
       e[k] = current; // adjust the end pointer
     }
```


RadixSort Function (contd.)

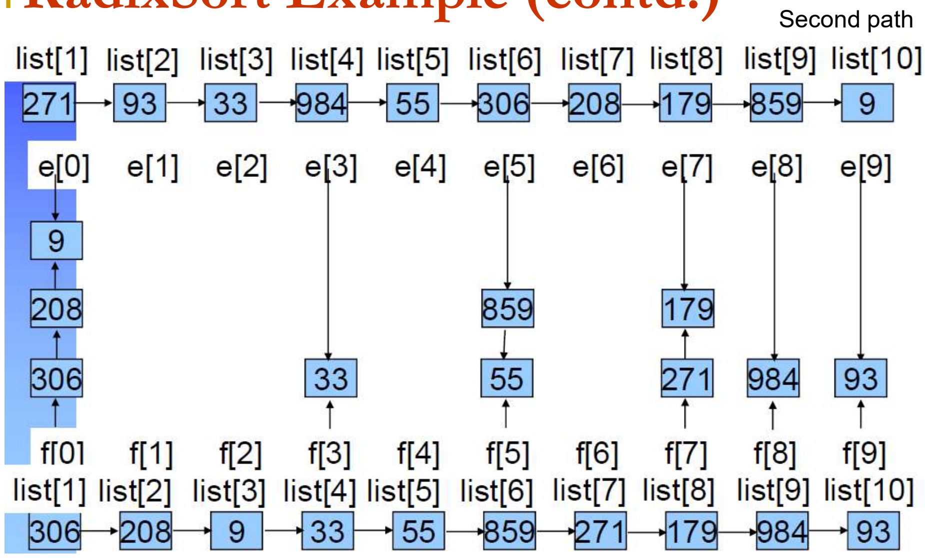
```
     for (j = 0; f[j] == 0; j++); // find first nonempty queue
     current = f[j]; int last = e[j];

     for (int k = j+1; k < radix; k++) {
       // concatenate remaining nonempty queues into new list
       if (f[k]) {
         list[last].link = f[k];
         last = e[k];
       }
     }
     list[last].link = 0;
     // print the sorted keys after each pass
     for (int q = current; q; q= list[q].link) {
       for (int p = 0; p < d; p++)
         cout << list[q].key[p] << " , ";
       cout << endl;
     }
   }
}
```

RadixSort Example



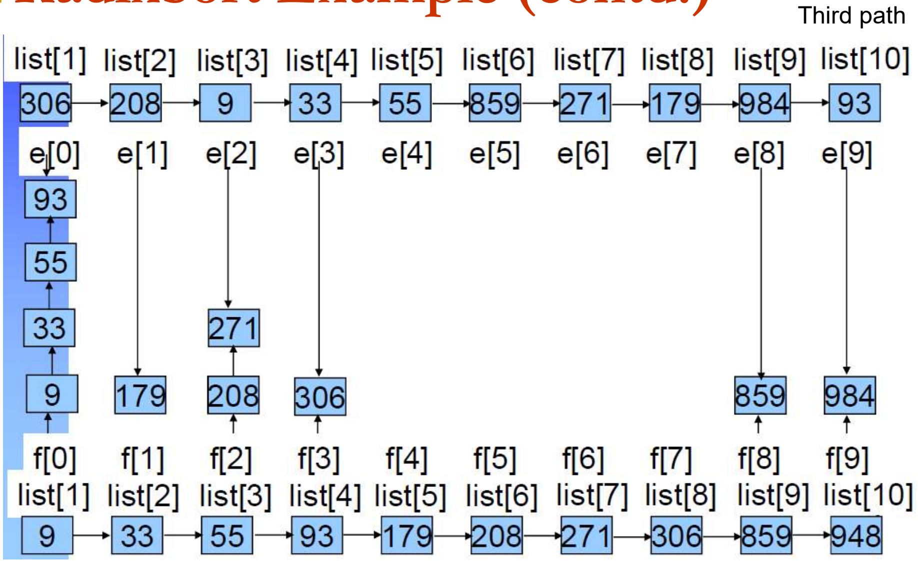
CSIEB0100 Data Structures Sorting I – Internal Sorting 69

RadixSort Example (contd.)



CSIEB0100 Data Structures Sorting I – Internal Sorting 70

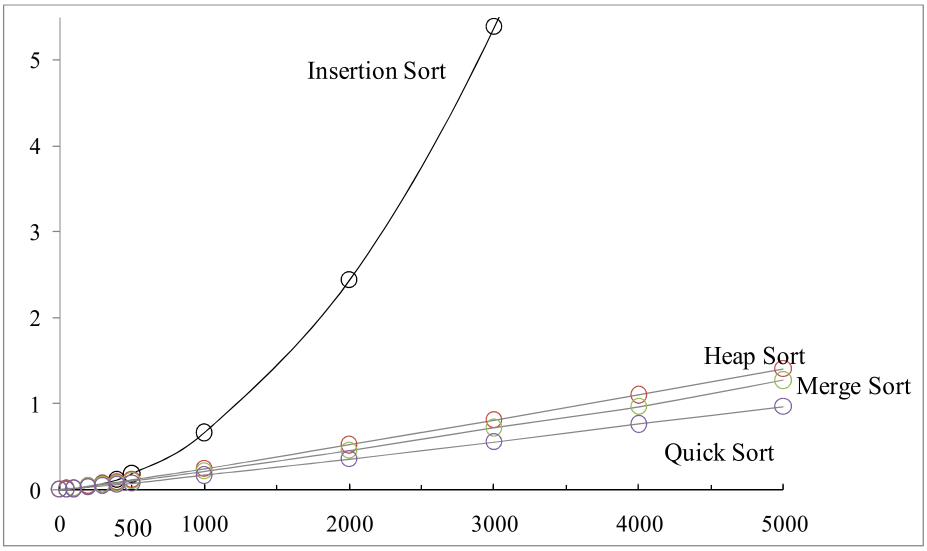
RadixSort Example (contd.)



Summary of Internal Sorting

- No one method is best for all conditions.
  - Insertion sort is good when the list is already partially ordered. And it is best for small n (number of records).
  - Merge sort has the best worst-case behavior but needs more storage than heap sort.
  - Quick sort has the best average behavior, but its worst-case behavior is $O(n^2)$.
  - The behavior of radix sort depends on the size of the keys and the choice of r.

Average Time Comparison



Remarks (Wiki)

- Sorting in-place is possible but is very complicated, and will offer little performance gains in practice, even if the algorithm runs in $O(n \log n)$ time.
- In these cases, algorithms like heapsort usually offer comparable speed, and are far less complex.

Sort with Linked Lists

- Apart from radix and recursive merge sort, all sorting methods above require excessive data movement.
- When the amount of data is large, data movement tends to slow down the process.
- It is desirable to minimize the data movement.
- Methods such as insertion sort or merge sort can work with linked lists rather than sequential lists. Instead of movement, link field is used to reflect the change in the position of records in the list.
- Perform linked-list sort and then physically rearrange the records according to the order specified in the list.

Table Sort

- The list-sort technique is not well suited for quick sort and heap sort.
- One can maintain an auxiliary table, t, with one entry per record. The entries serve as an indirect reference to the records.
- Initially, t[i] = i. When interchanges are required, only the table entries are exchanged.
- It may be necessary to physically rearrange the records according to the permutation specified by t sometimes.

Table Sort (contd.)

- The function to rearrange records corresponding to the permutation t[1], t[2], …, t[n] can be considered as an application of a theorem from mathematics:
  - Every permutation is made up of disjoint cycles.
  - The cycle for any element i is made up of i, t[i], t^2[i], …, t^k[i], where t^j[i]=t[t^{j-1}[i]], t^0[i]=i, t^k[i]=i.
- Details in the textbook.