The 7th Course

SORTING

Requirements

- In an external sort, there are so many records to be sorted that they must be kept in external files on disks, tapes, or the like. In an internal sort the records can all be kept internally in high-speed memory. We consider only internal sorting.
- We use the notation and classes set up in Chapters 6. Thus we shall sort lists of records into the order determined by keys associated with the records.
- If two or more of the entries in a list have the same key, we must exercise special care.

Requirements

- To analyze sorting algorithms, we consider both the number of comparisons of keys and the number of times entries must be moved inside a list, particularly in a contiguous list.
- Both the worst-case and the average performance of a sorting algorithm are of interest. To and the average, we shall consider what would happen if the algorithm were run on all possible orderings of the list (with n entries, there are n! such orderings altogether) and take the average of the results.

Requirements

- To write efficient sorting programs, we must access the private data members of the lists being sorted. Therefore, we shall add sorting functions as methods of our basic List data structures. The augmented list structure forms a new ADT that we shall call a Sortable_list.

Requirements

- a Sortable_list is a List with extra sorting methods.
- The base list class can be any of the List implementations of Chapter 6.
- We use a template parameter class called Record to stand for entries of the Sortable_list.

Requirements

- Every Record has an associated key of type Key. A Record can be implicitly converted to the corresponding Key.
- The keys (hence also the records) can be compared under the operations `<,` `>,` `>=,` `<=,` `==,` and `!=.`

Ordered Insertion

- An *ordered list* is an abstract data type, defined as a list in which each entry has a key, and such that the keys are in order; that is, if entry i comes before entry j in the list, then the key of entry i is less than or equal to the key of entry j .

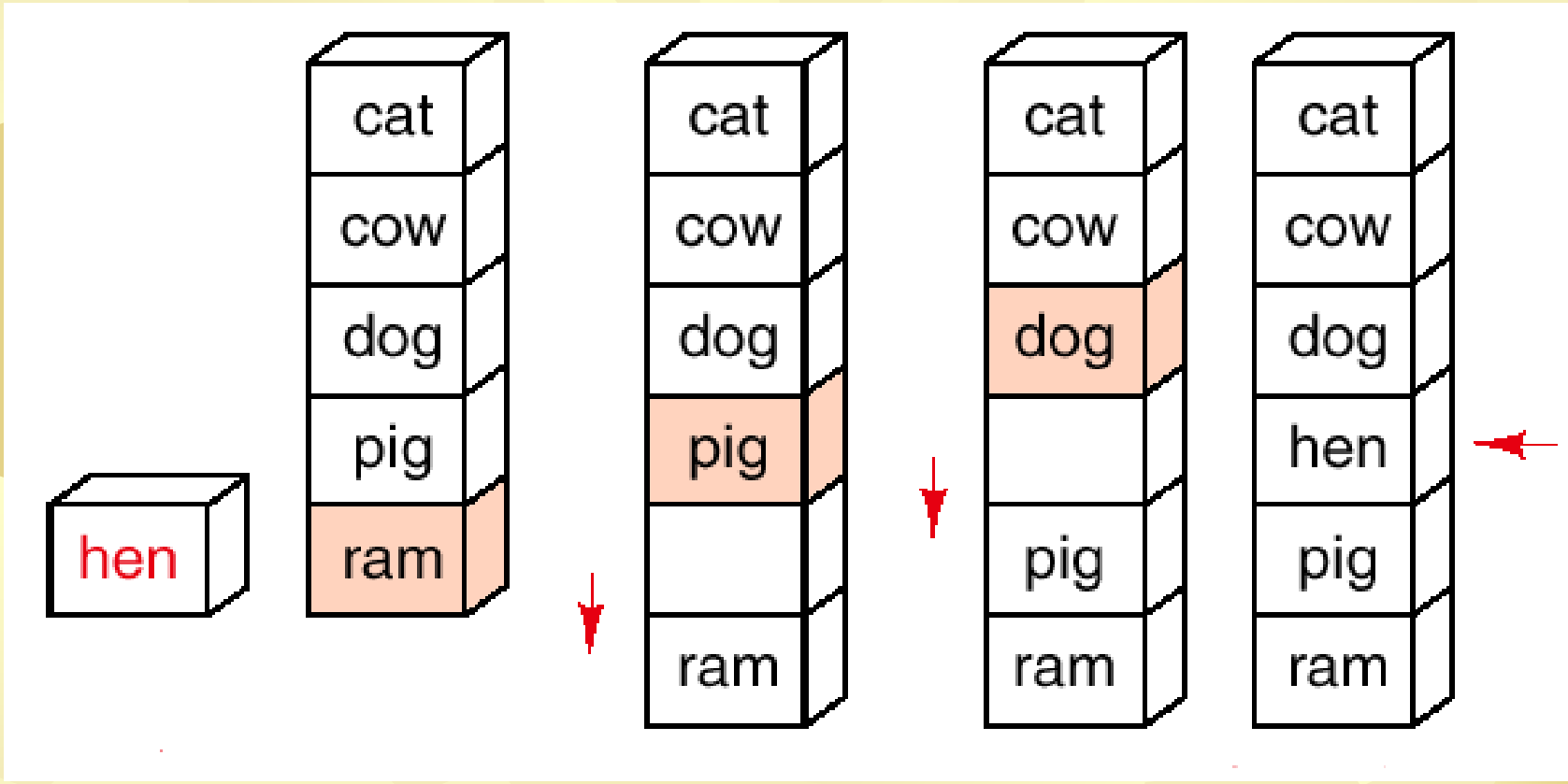
Ordered Insertion

- For ordered lists, we shall often use two new operations that have no counterparts for other lists, since they use keys rather than positions to locate the entry.
  - One operation retrieves an entry with a specified key from the ordered list. Retrieval by key from an ordered list is exactly the same as searching.
  - The second operation, ordered insertion, inserts a new entry into an ordered list by using the key in the new entry to determine where in the list to insert it.

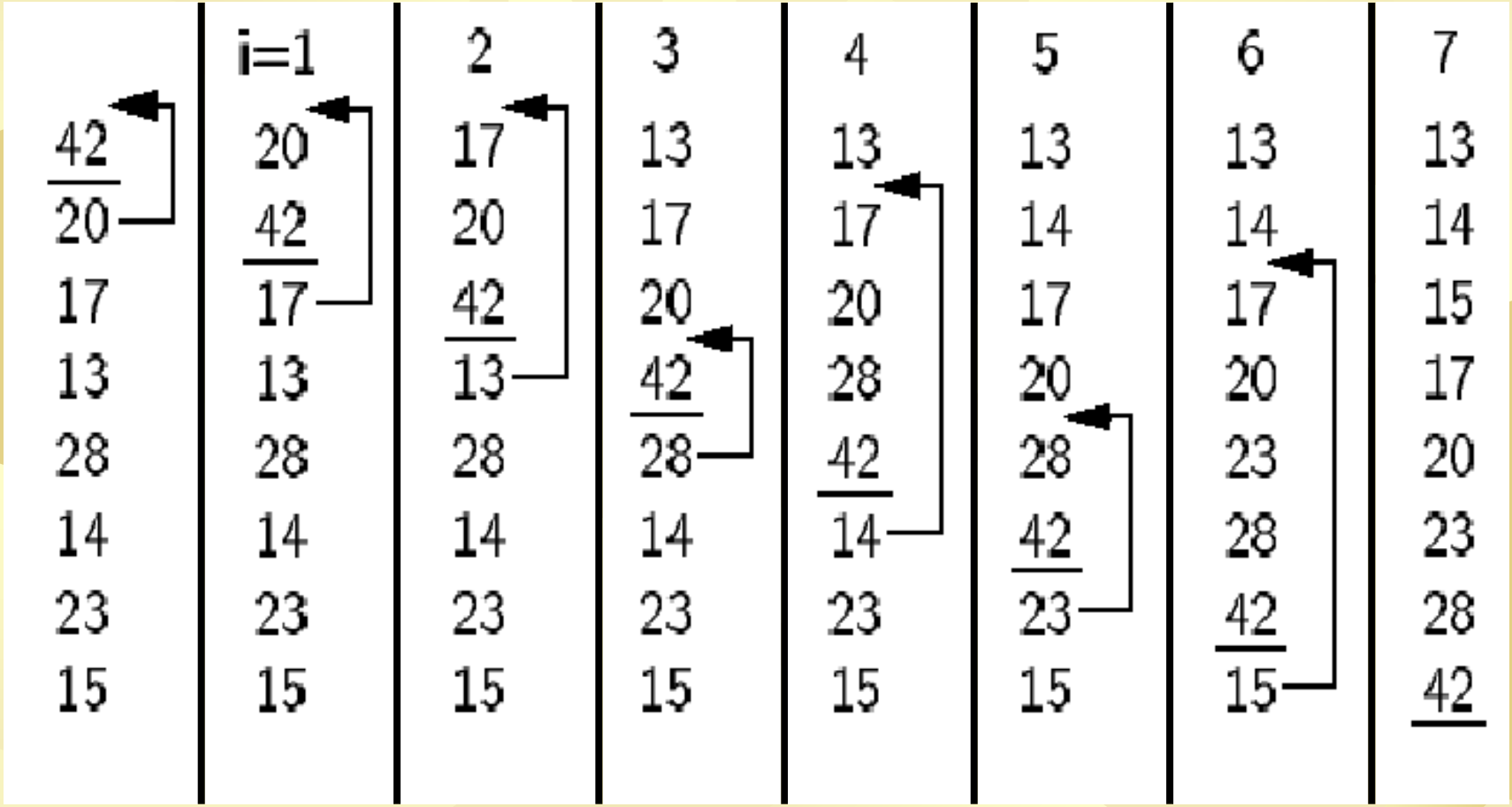
Ordered Insertion

- Note that ordered insertion is not uniquely specified if the list already contains an entry with the same key as the new entry, since the new entry could go into more than one position.

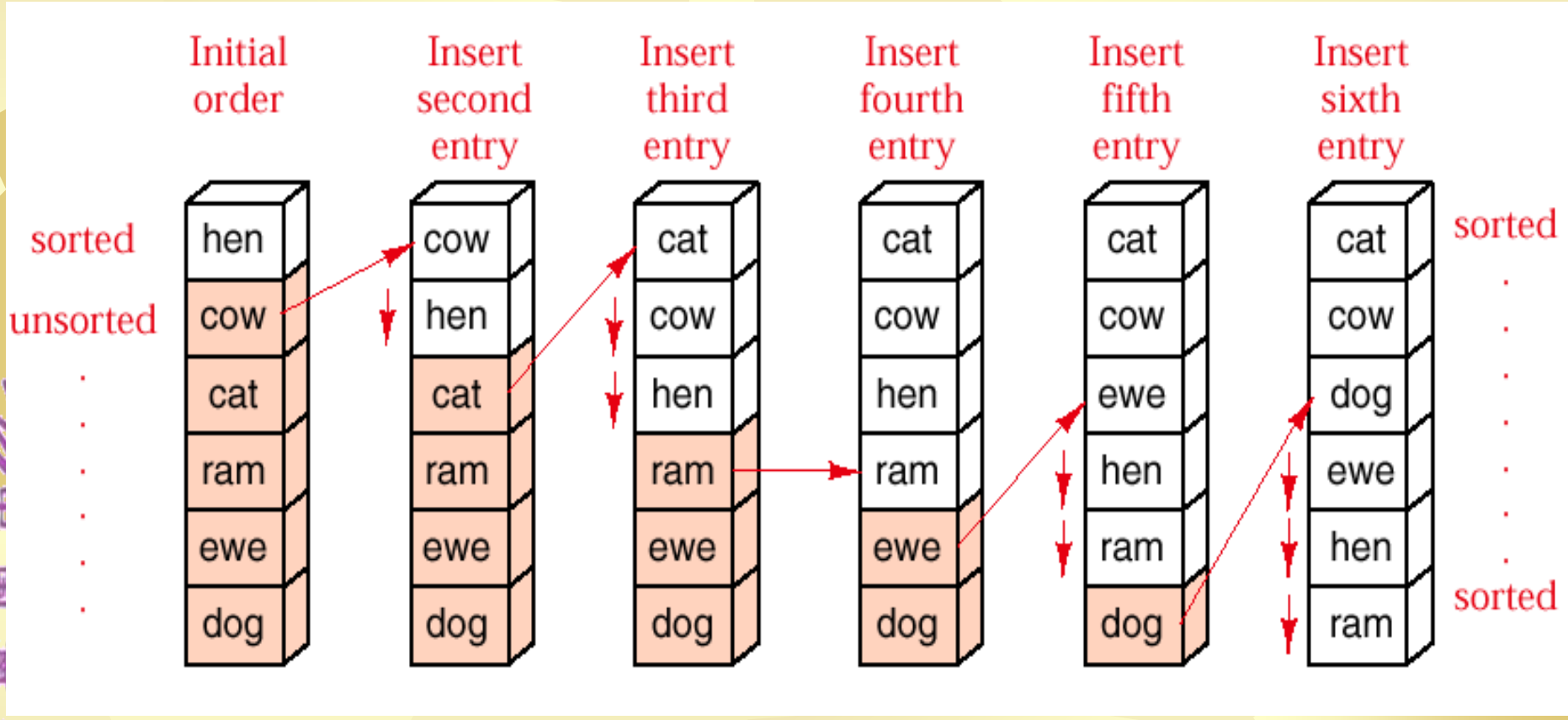
Ordered Insertion



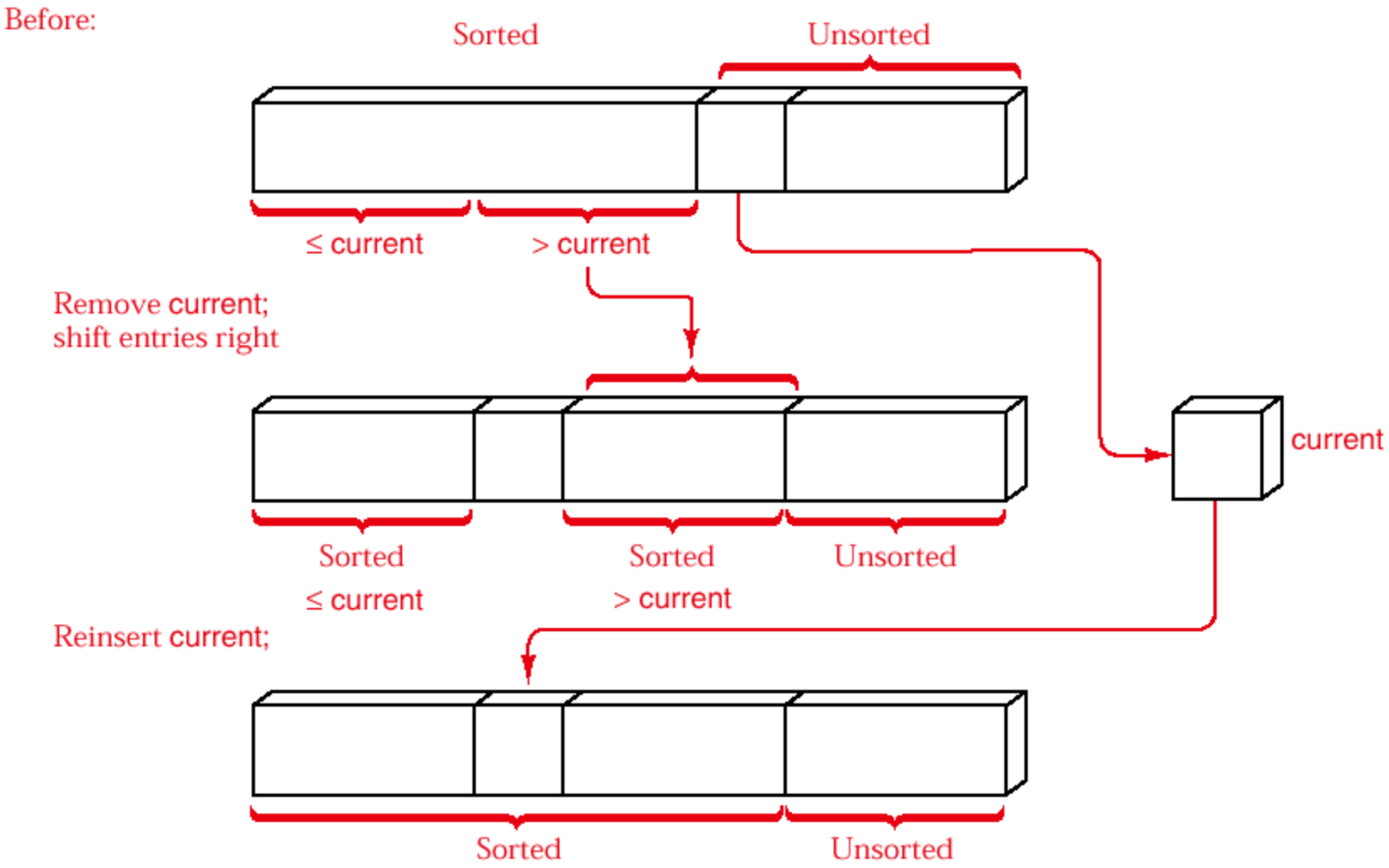
Insertion Sort



Insertion Sort



Ordered Insertion



Linked Insertion Sort

- With no movement of data, there is no need to search from the end of the sorted sublist, as for the contiguous case.
- Traverse the original list, taking one entry at a time and inserting it in the proper position in the sorted list.
- Pointer last_sorted references the end of the sorted part of the list.
- Pointer first_unsorted == last_sorted->next references the first entry that has not yet been inserted into the sorted sublist.

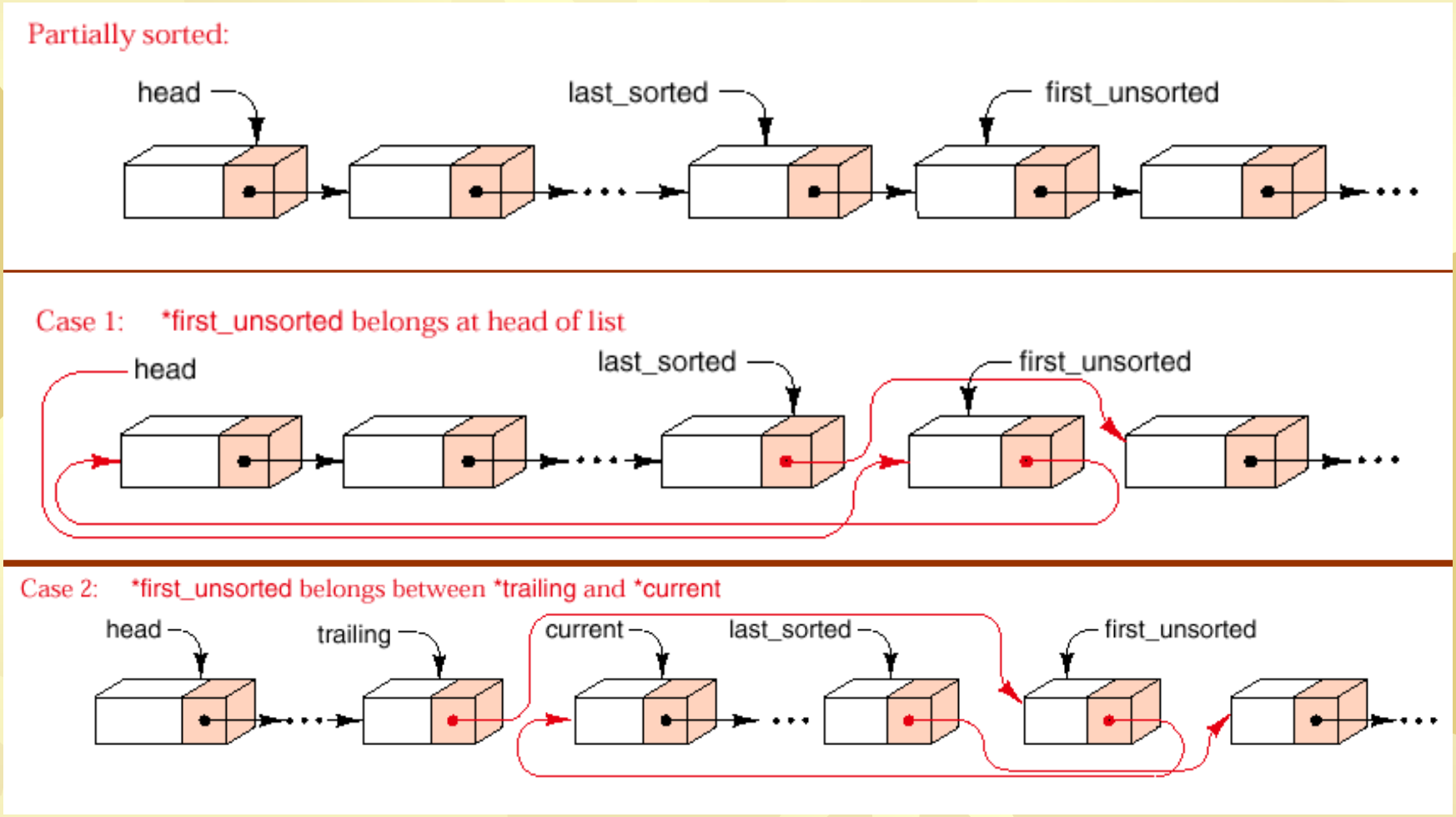
Linked Insertion Sort

- Pointer current searches the sorted part of the list to find where to insert *first_unsorted.
- If *first_unsorted belongs before the head of the list, then insert it there.
- Otherwise, move current down the list until first_unsorted->entry <= current->entry and then insert *first_unsorted before *current. To enable insertion before *current, keep a second pointer trailing in lock step one position closer to the head than current.

Linked Insertion Sort

- A *sentinel* is an extra entry added to one end of a list to ensure that a loop will terminate without having to include a separate check. Since last_sorted->next == first_unsorted, the node *first_unsorted is already in position to serve as a sentinel for the search, and the loop moving current is simplified.
- A list with 0 or 1 entry is already sorted, so by checking these cases separately we avoid trivialities elsewhere.

Linked Insertion Sort



Analysis of Insertion Sort

- The average number of comparisons for insertion sort applied to a list of n items in random order is $\frac{1}{4}n^2 + O(n)$
- The average number of assignments of items for insertion sort applied to a list of n items in random order is also $\frac{1}{4}n^2 + O(n)$
- The best case for contiguous insertion sort occurs when the list is already in order, when insertion sort does nothing except n - 1 comparisons of keys.

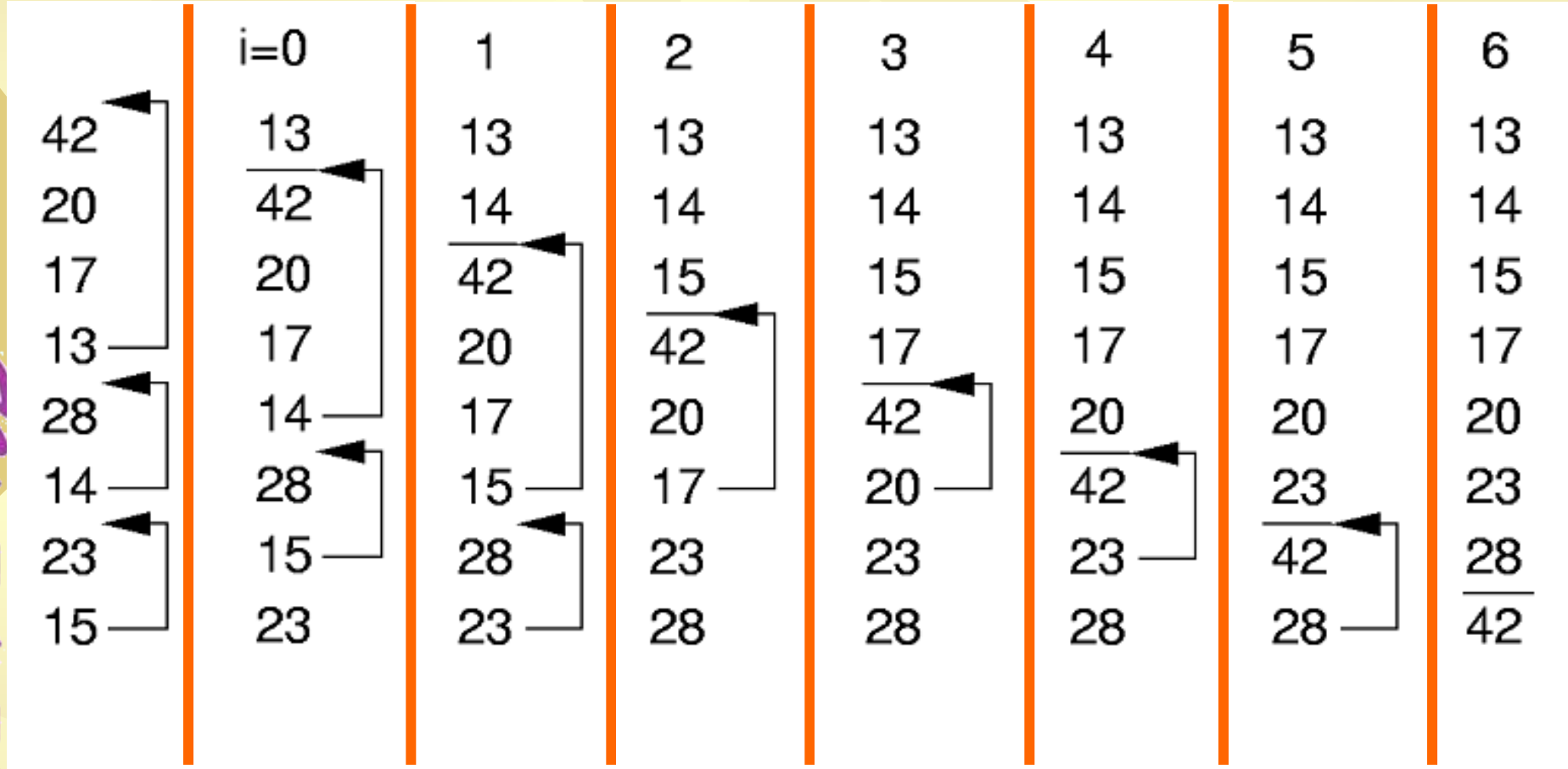
Measures of cost

- Best Case
  - 0 swaps, n - 1 comparisons
- Worst Case
  - $n^2/2$ swaps and compares
- Average Case
  - $n^2/4$ swaps and compares
- Good best case performance

Analysis of Insertion Sort

- The worst case for contiguous insertion sort occurs when the list is in reverse order.
- THEOREM Verifying that a list of n entries is in the correct order requires at least n - 1 comparisons of keys.
- Insertion sort verifies that a list is correctly sorted as quickly as any method can.

Bubble Sort

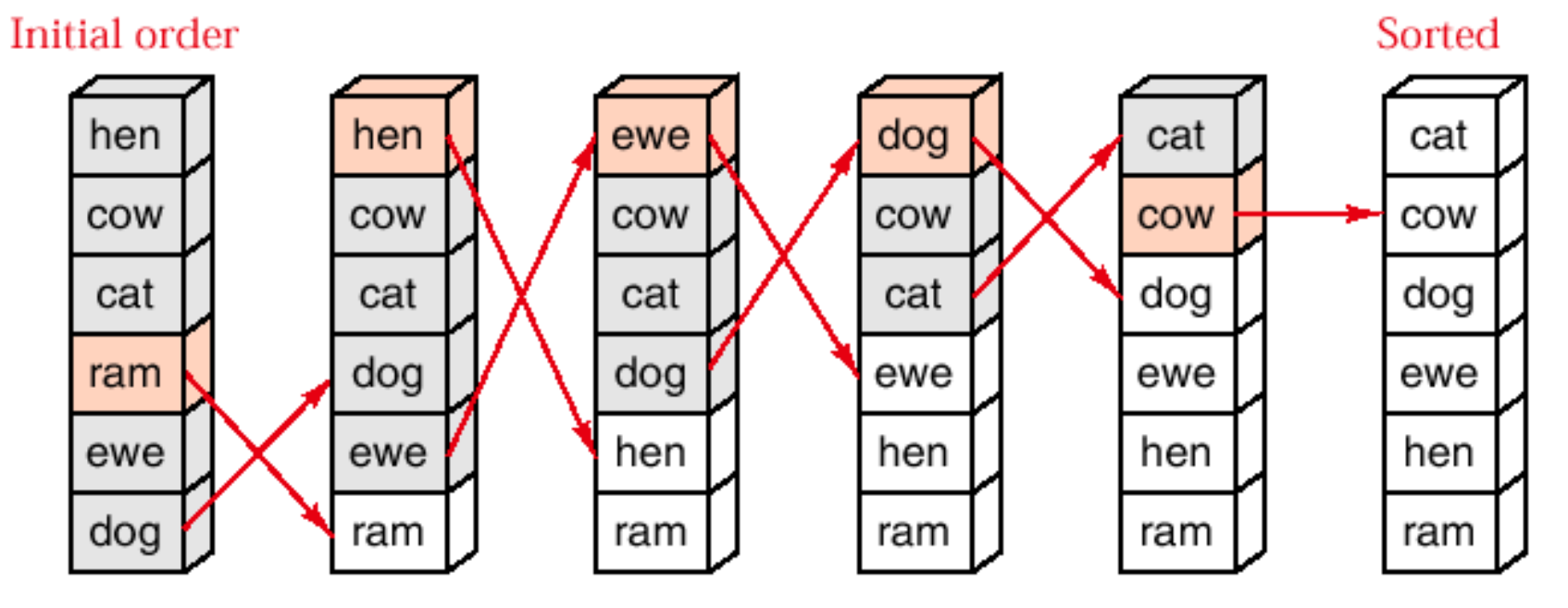

i=0
1
2
3
4
5
6
42 13 13 13 13 13 13
20 42 14 14 14 14 14
17 20 42 15 15 15 15
13 17 20 42 17 17 17
28 14 17 20 42 20 20
14 28 15 17 20 42 23
23 15 28 23 23 23 42
15 23 23 28 28 28 28

Measures of cost

- Best Case
  - $n^2/2$ compares, 0 swaps
- Worst Case
  - $n^2/2$ compares, $n^2/2$ swaps
- Average Case
  - $n^2/2$ compares, $n^2/4$ swaps
- No redeeming features to this sort

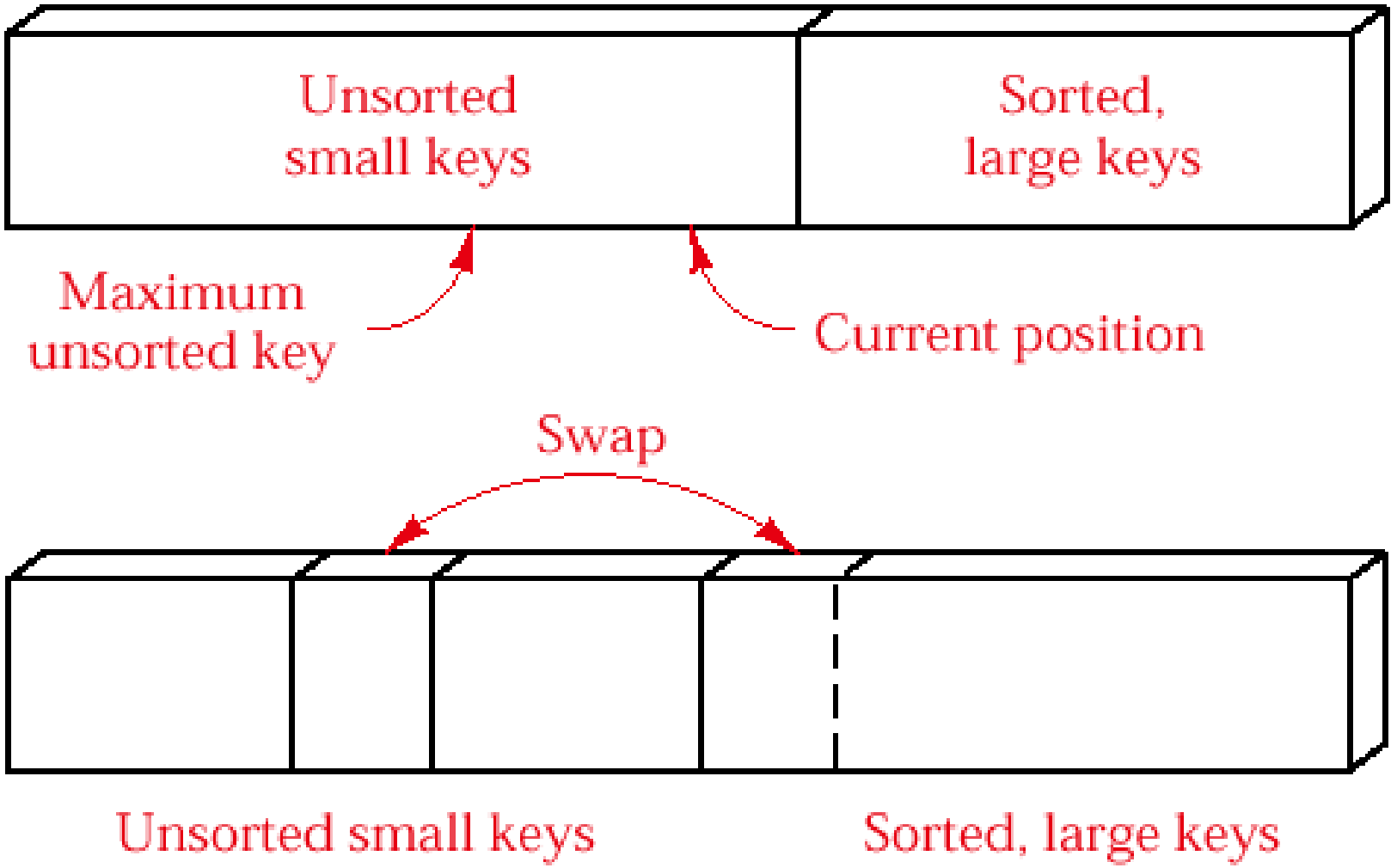
Selection Sort

- Example:

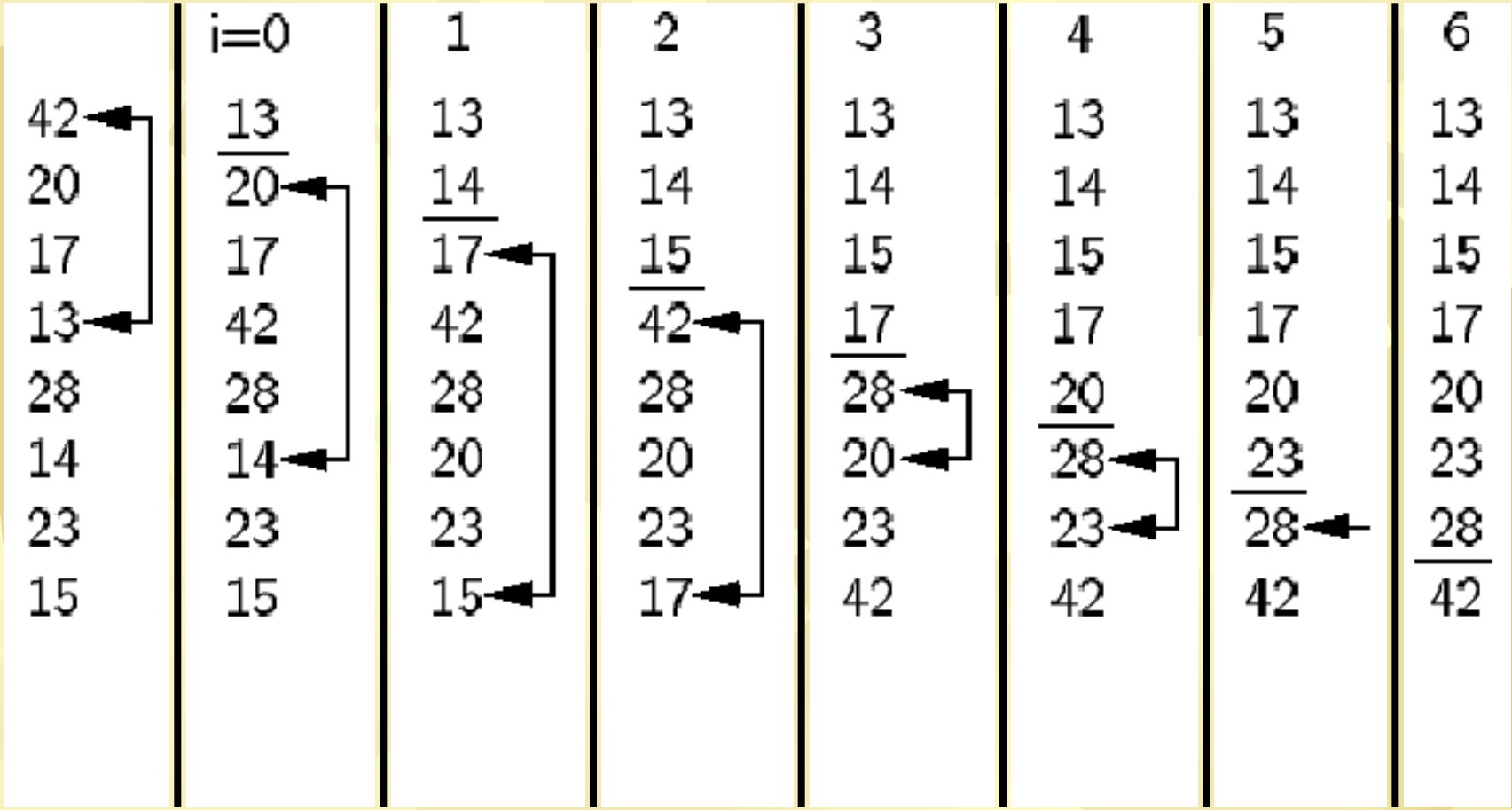


Colored box denotes largest unsorted key. Gray boxes denote other unsorted keys.

Selection Sort



Selection Sort



Selection Sort

- Analysis and comparison:

	Selection	*Insertion (average)*
Assignments of entries	$3.0n + O(1)$	$0.25n^2 + O(n)$
Comparisons of keys	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

Exchange Sorting--Summary

- Comparisons:

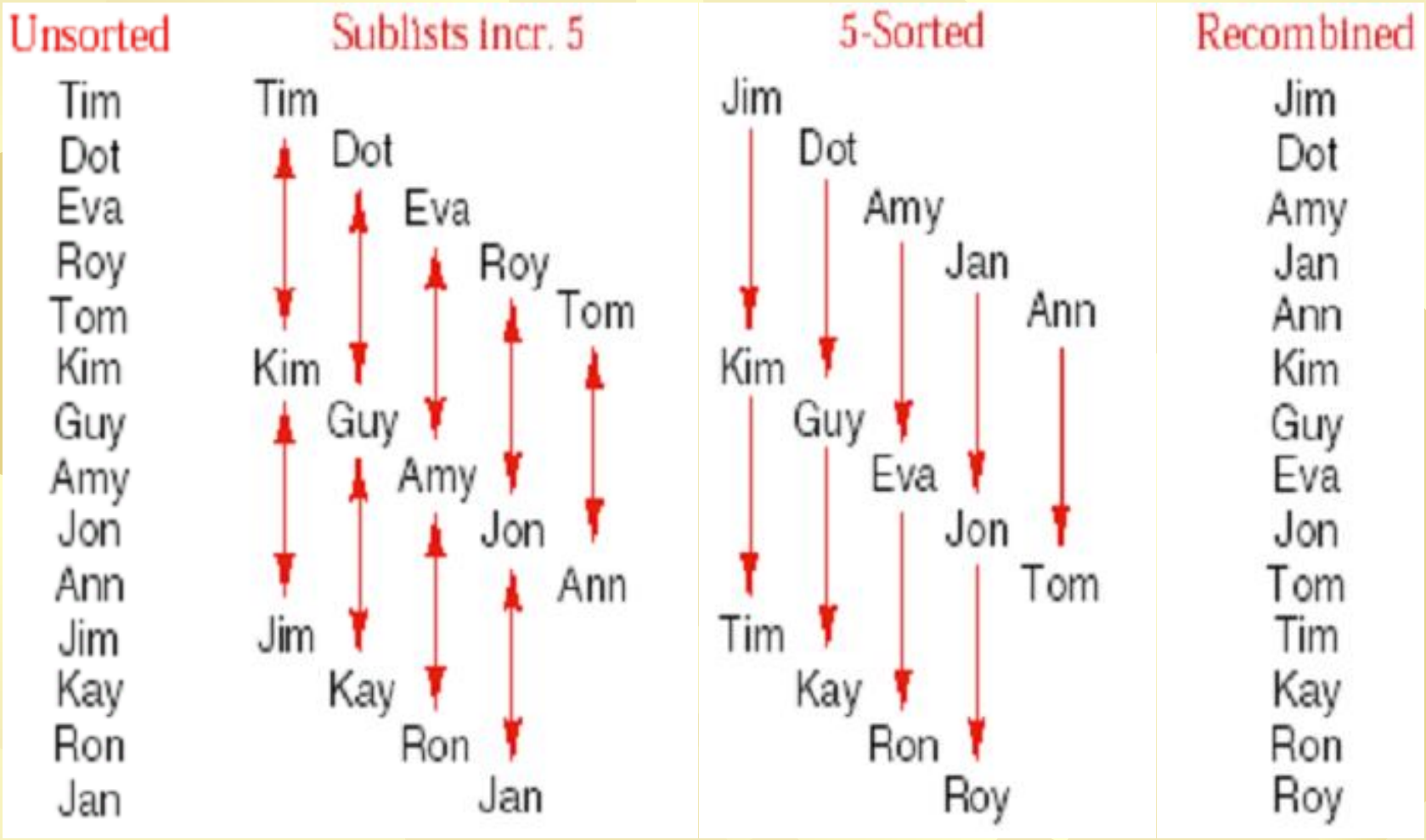
	Insertion	Bubble	Selection
Best Case	O (n)	O (n^2)	O (n^2)
Average Case	O (n^2)	O (n^2)	O (n^2)
Worst Case	O (n^2)	O (n^2)	O (n^2)

Exchange Sorting--Summary

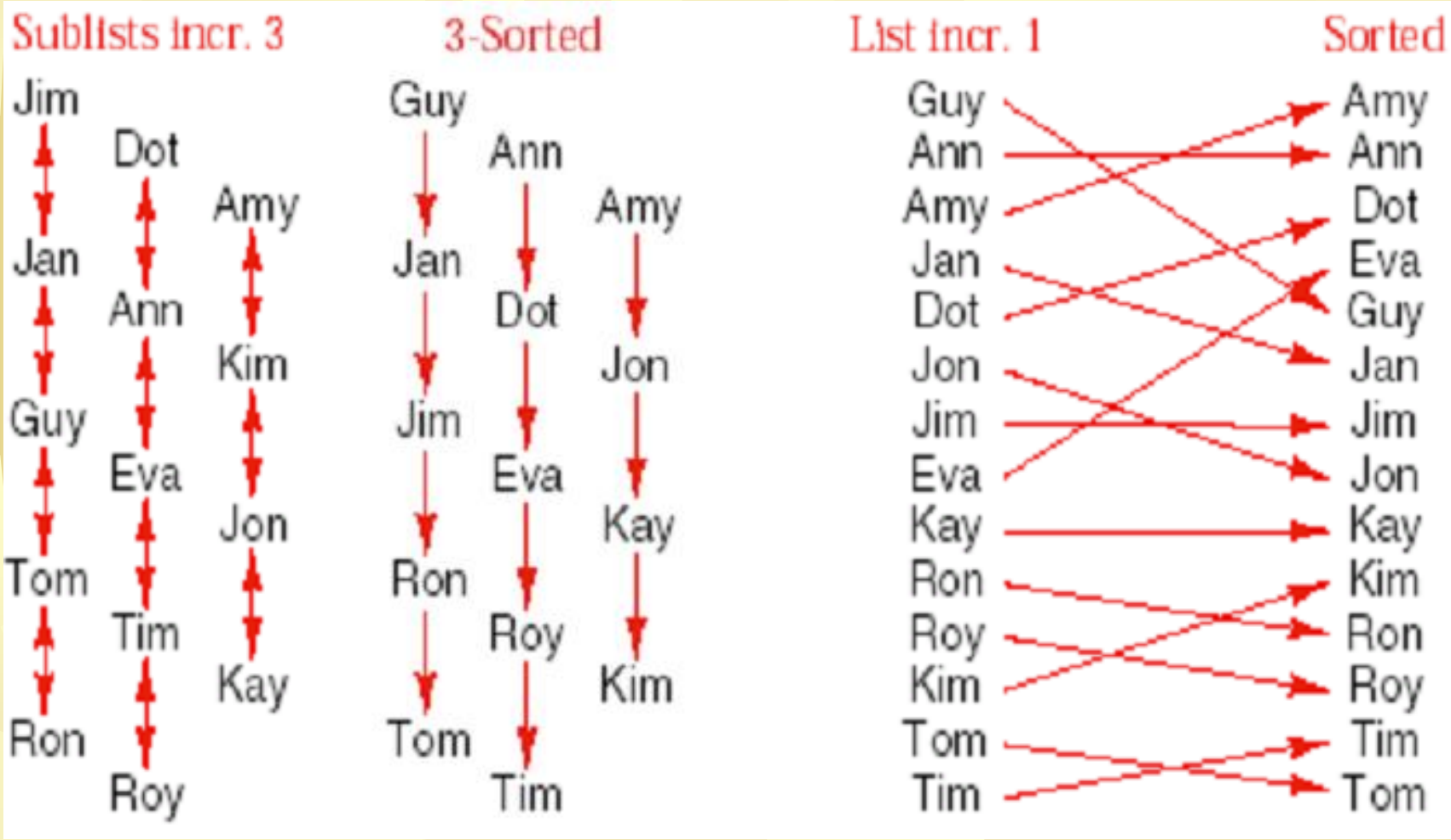
- Swap:

	Insertion	Bubble	Selection
Best Case	0	0	O (n)
Average Case	O (n^2)	O (n^2)	O (n)
Worst Case	O (n^2)	O (n^2)	O (n)

Example of Shell Sort

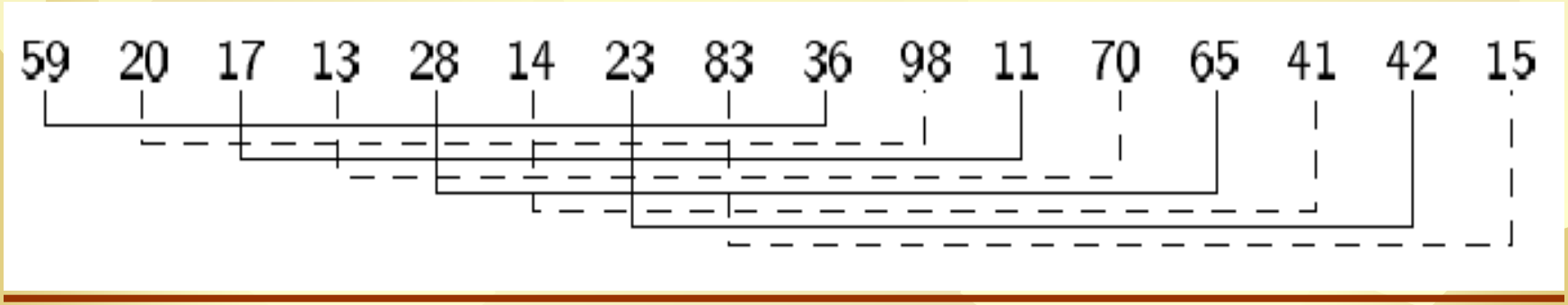


Example of Shell Sort

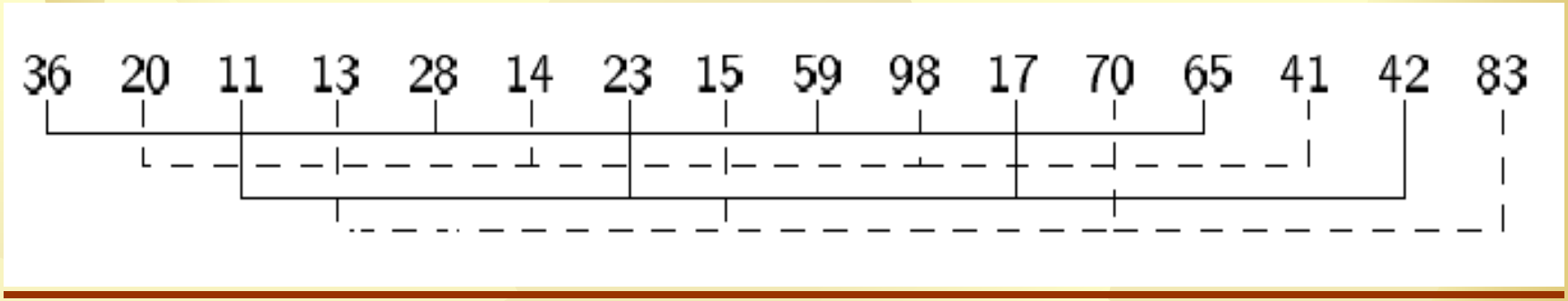


Shellsort

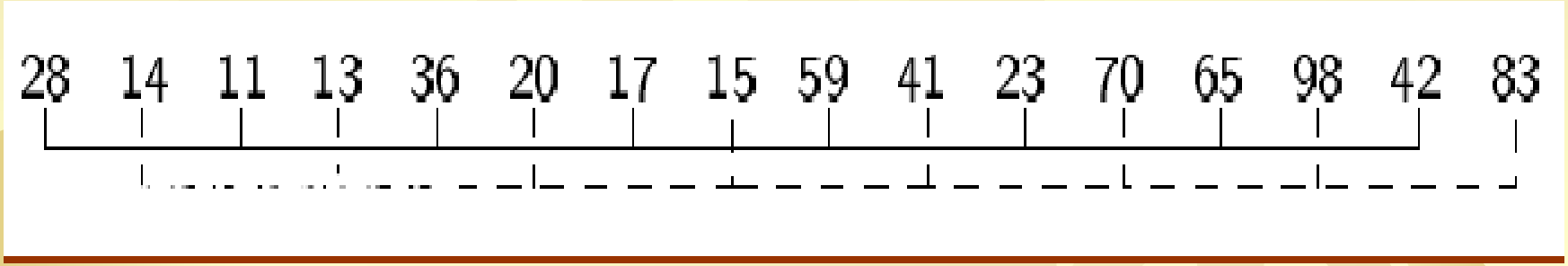
Incr = 8 ---- 8 lists of length 2



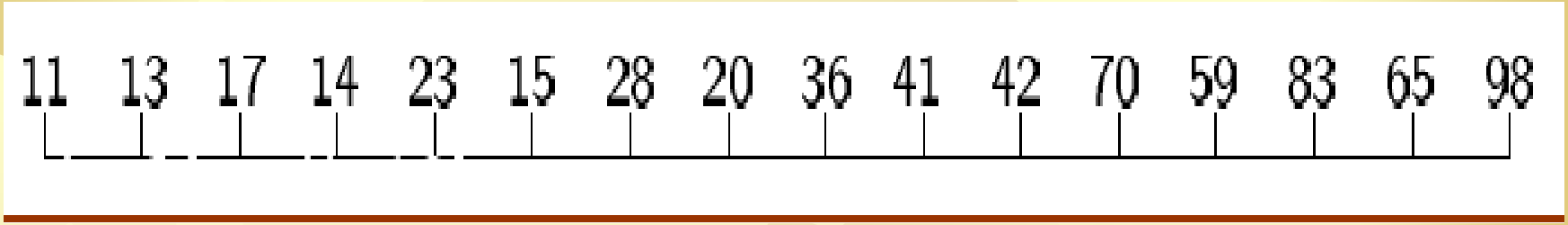
Incr = 4 ---- 4 lists of length 4



Incr = 2 ---- 2 lists of length 8



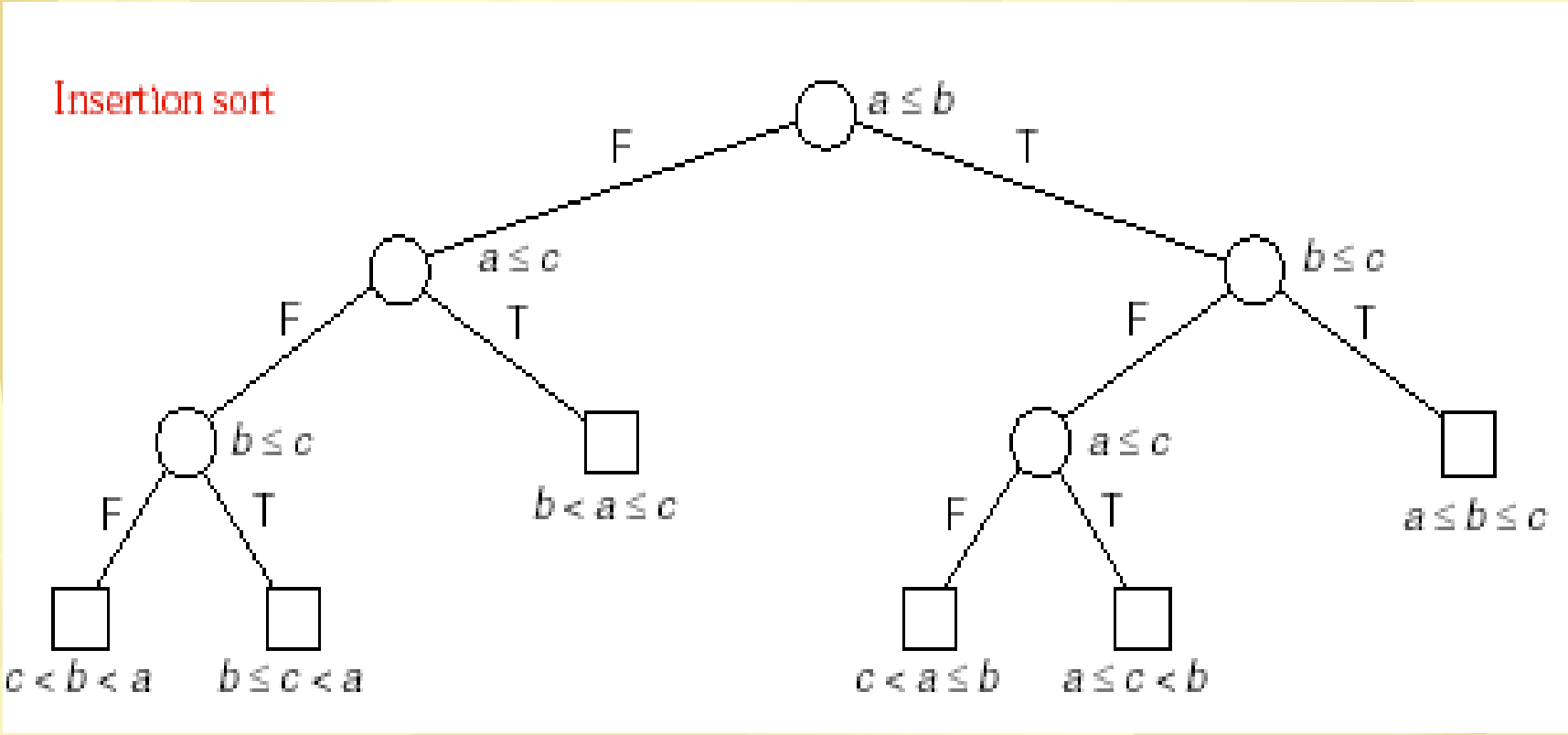
Incr = 1 ---- 1 list of length 16



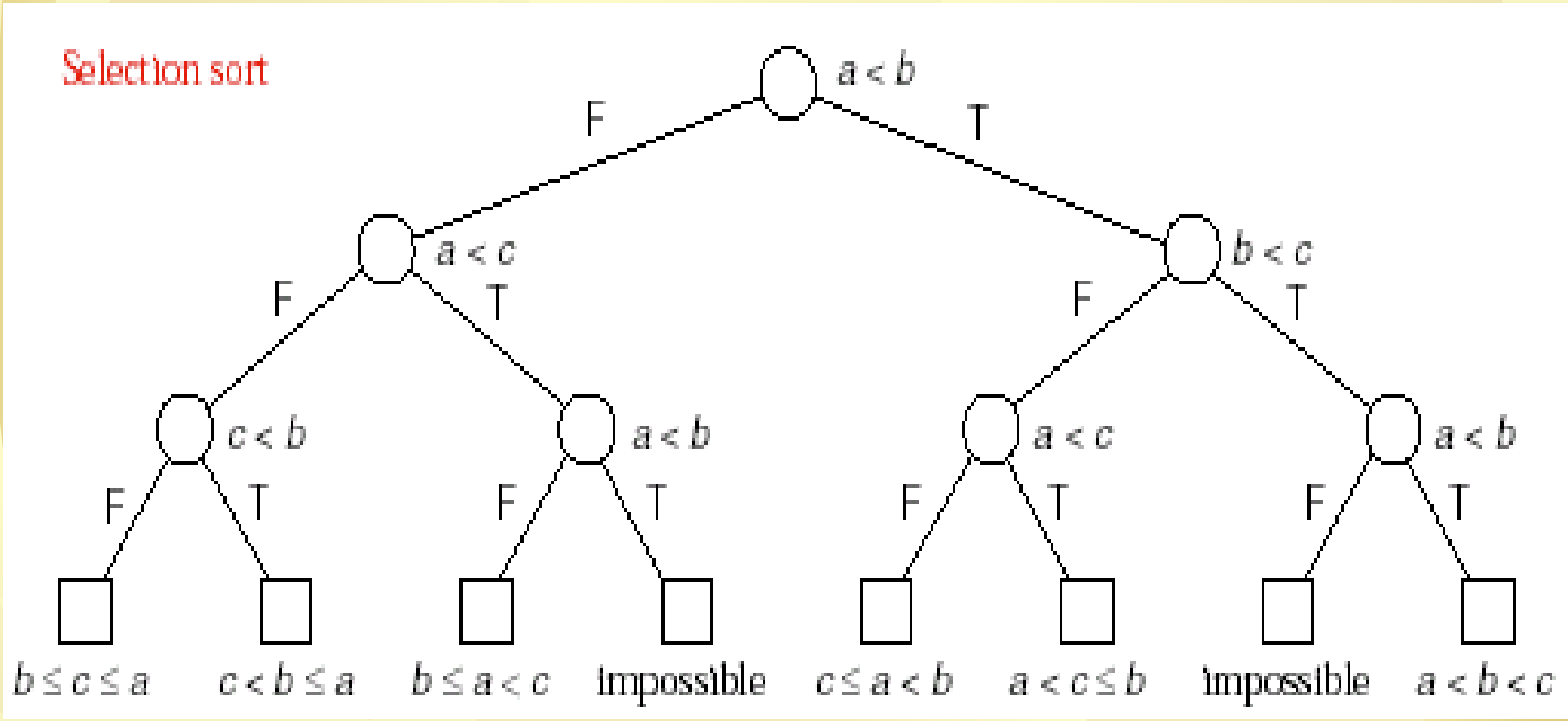
11 13 14 15 17 20 23 28 36 41 42 59 65 70 83 98

Lower Bounds for Sorting

- How fast is it possible to sort?



Lower Bounds for Sorting



Lower Bounds for Sorting

- THEOREM Any algorithm that sorts a list of n entries by use of key comparisons must, in its worst case, perform at least $\lceil \lg n! \rceil$ comparisons of keys, and, in the average case, it must perform at least $\lceil \lg n! \rceil$ comparisons of keys.

Divide and Conquer Sorting

Outline:

```
void Sortable_list :: sort( )
{
    if the list has length greater than 1 {
            partition the list into lowlist, highlist;
            lowlist.sort( );
            highlist.sort( );
            combine(lowlist, highlist);
    }
}
```

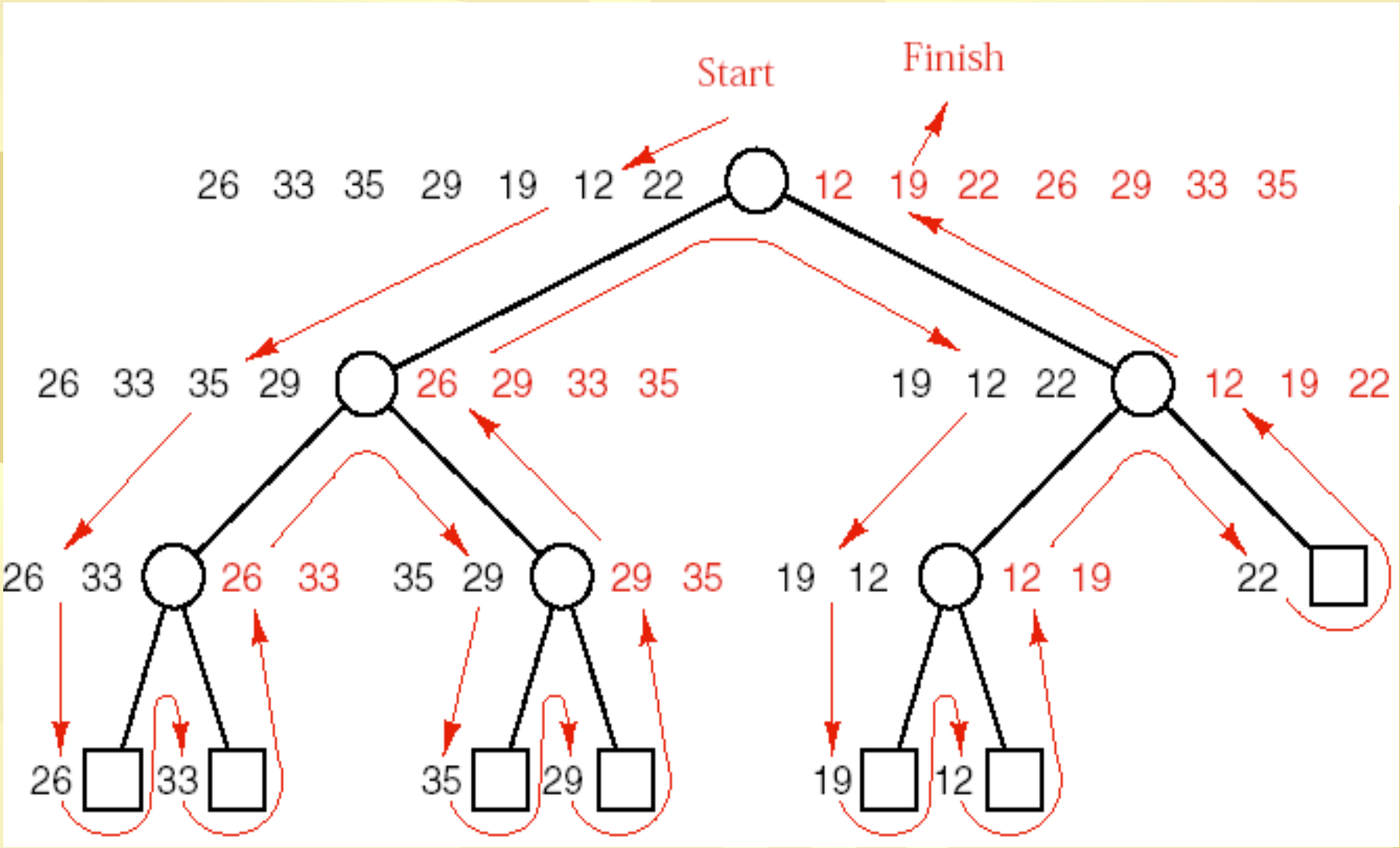
Divide and Conquer Sorting

- Mergesort
  - We chop the list into two sublists of sizes as nearly equal as possible and then sort them separately. Afterward, we carefully merge the two sorted sublists into a single sorted list.

Divide and Conquer Sorting

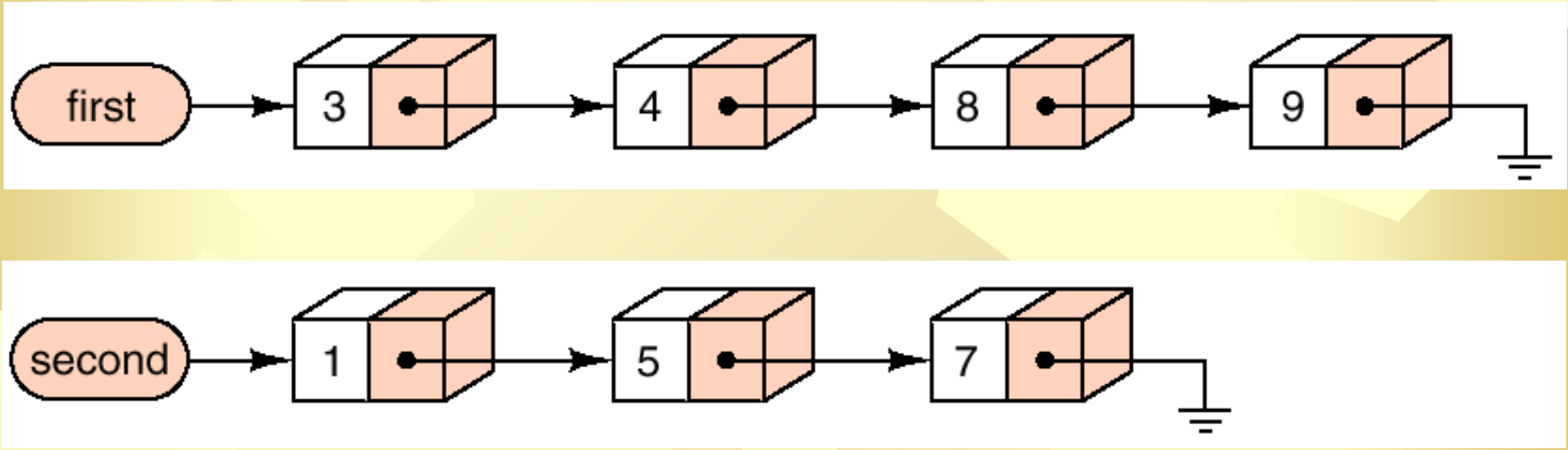
- Quicksort
  - We first choose some key from the list for which, we hope,about half the keys will come before and half after. Call this key the *pivot*. Then we partition the items so that all those with keys less than the pivot come in one sublist, and all those with greater keys come in another. Then we sort the two reduced lists separately, put the sublists together, and the whole list will be in order.

Divide and Conquer Sorting



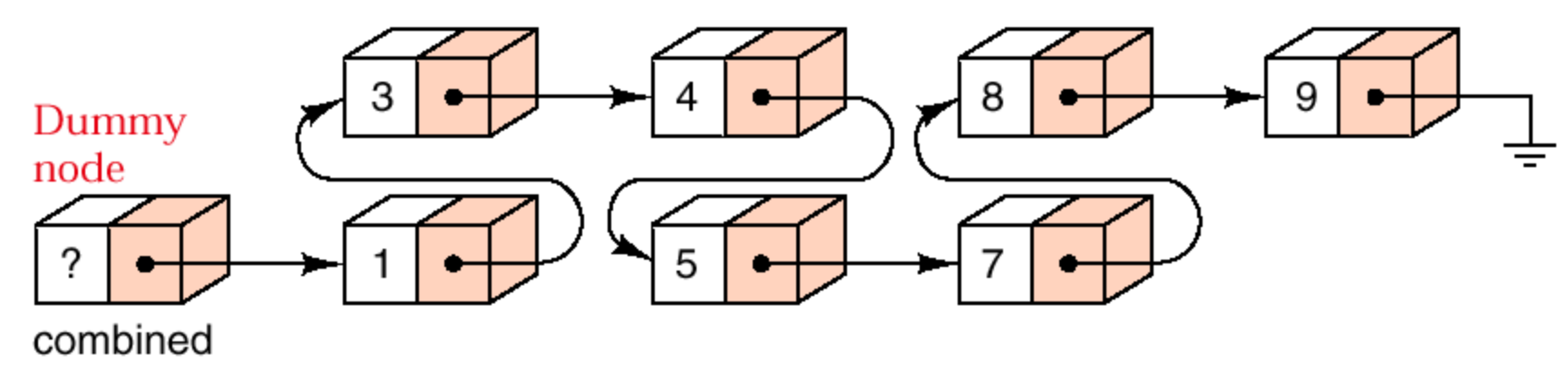
Divide Linked List in Half

- Initial situation

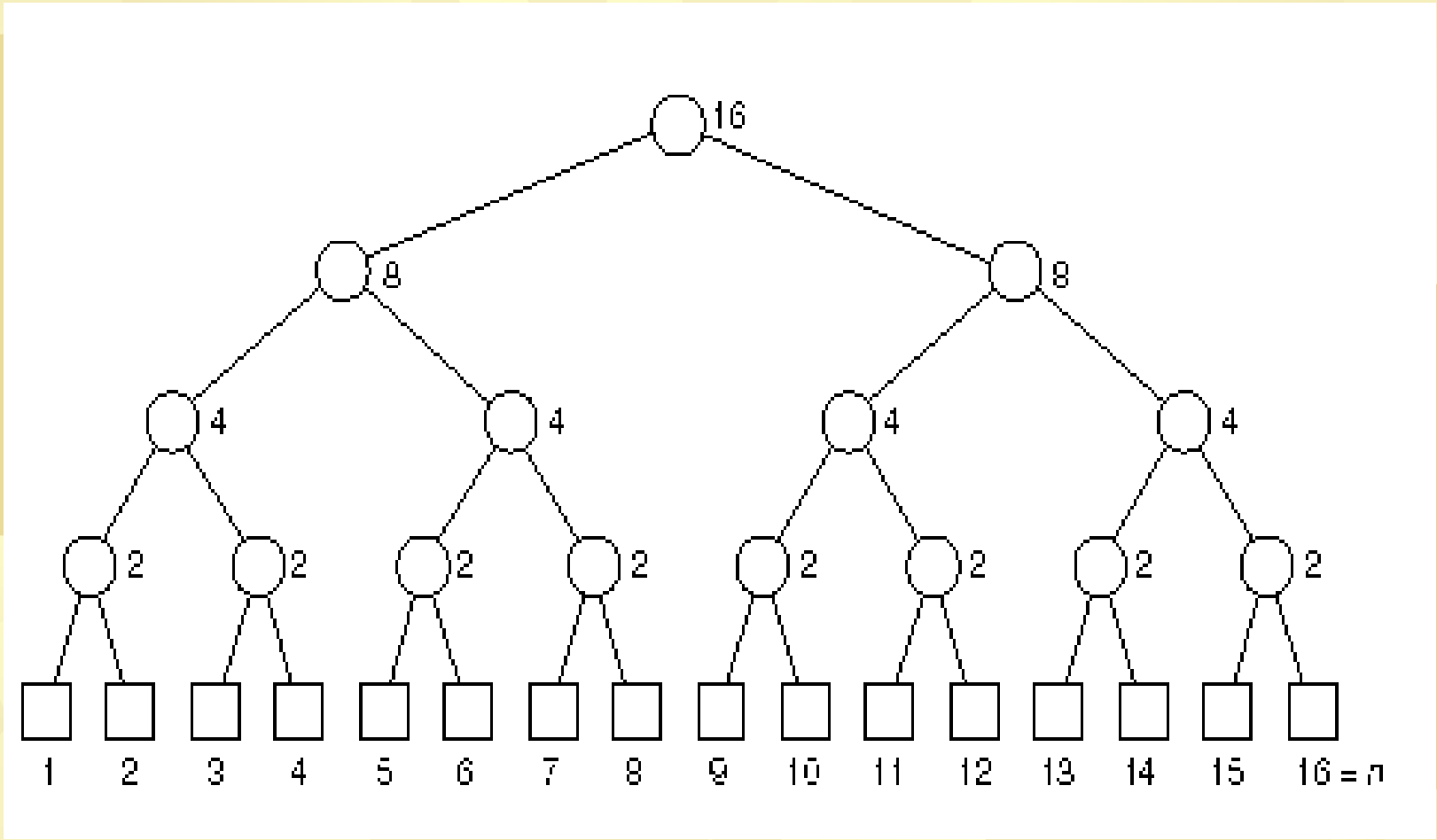


Divide Linked List in Half

- After merging



Analysis of Mergesort



Analysis of Mergesort

Lower bound	$\lg n! \approx n \lg n - 1.44n + O(\log n)$
Mergesort	$n \lg n - 1.1583n + 1$
Insertion sort	$1/4n^2 + O(n)$

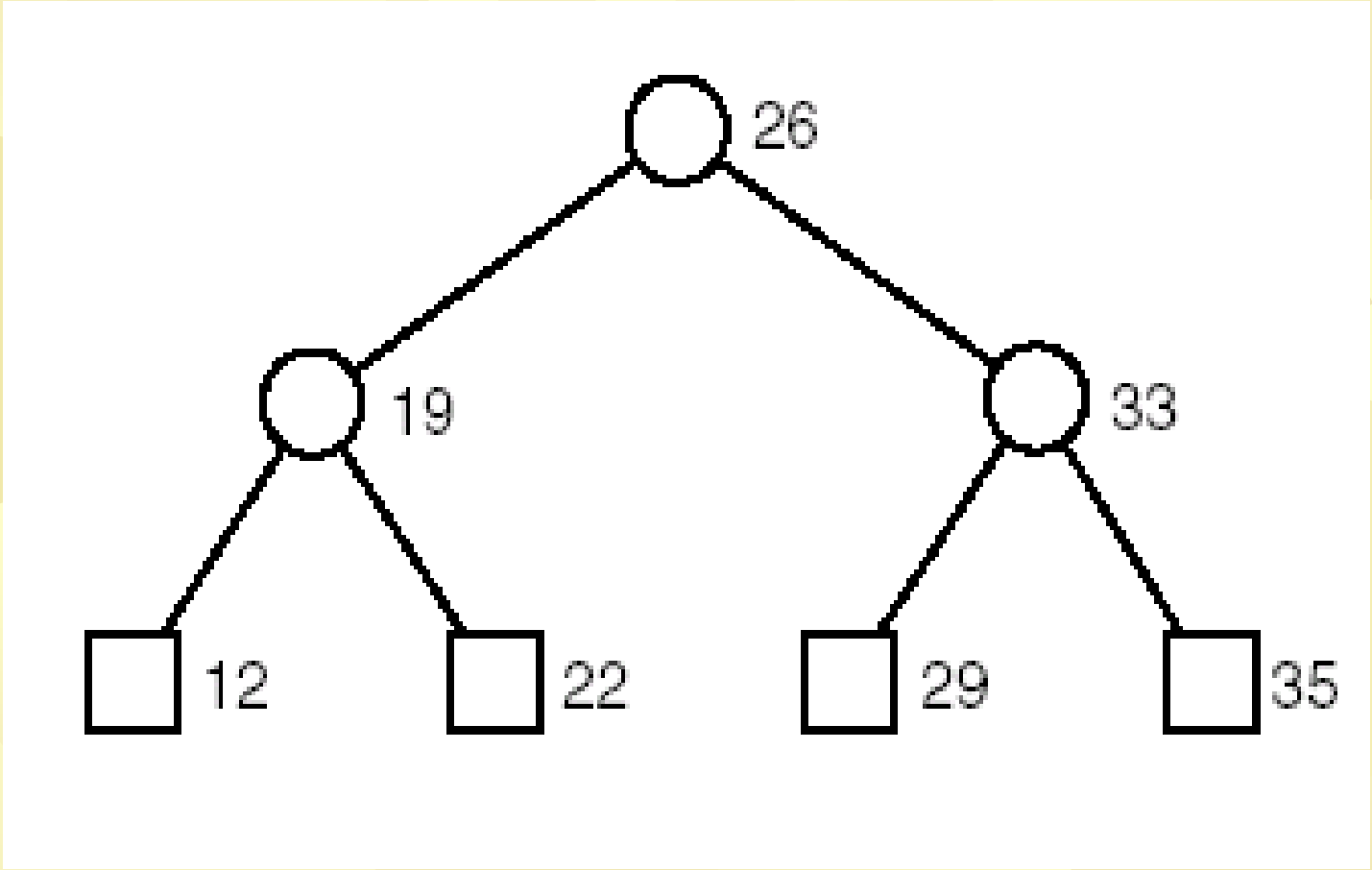
Quick sort of 7 Numbers

- Sort (26, 19, 33, 35, 29, 12, 22)
  - Partition into (19, 12, 22) and (33, 35, 29); pivot = 26

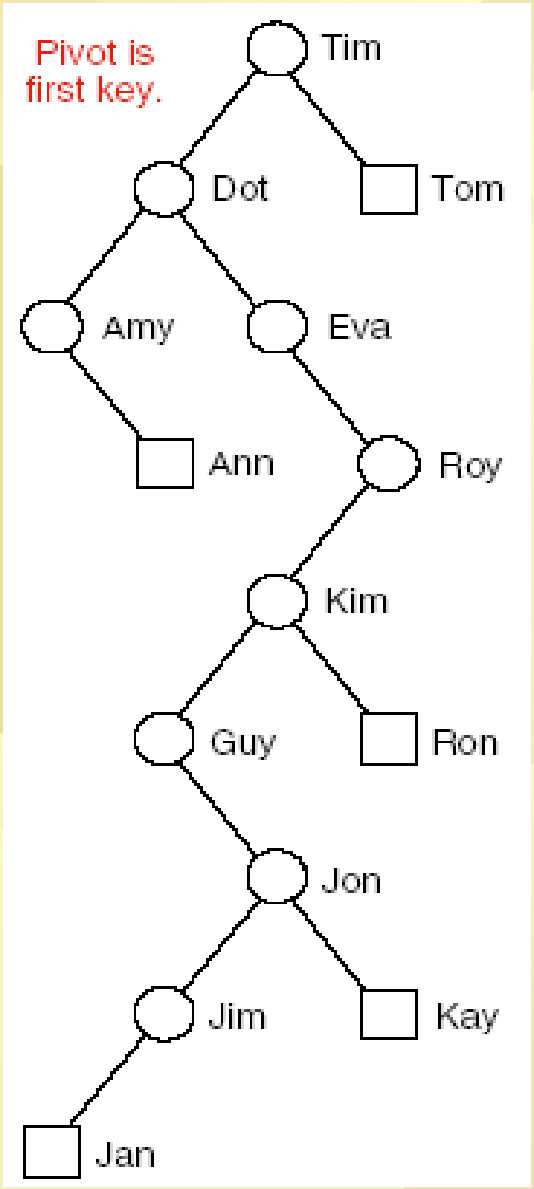
Sort (19, 12, 22)	Sort (33, 35, 29)
Partition into (12) and (22);	Partition into (29) and (35);
pivot = 19	pivot = 33
Sort (12)	Sort (29)
Sort (22)	Sort (35)
Combine into (12, 19, 22)	Combine into (29, 33, 35)

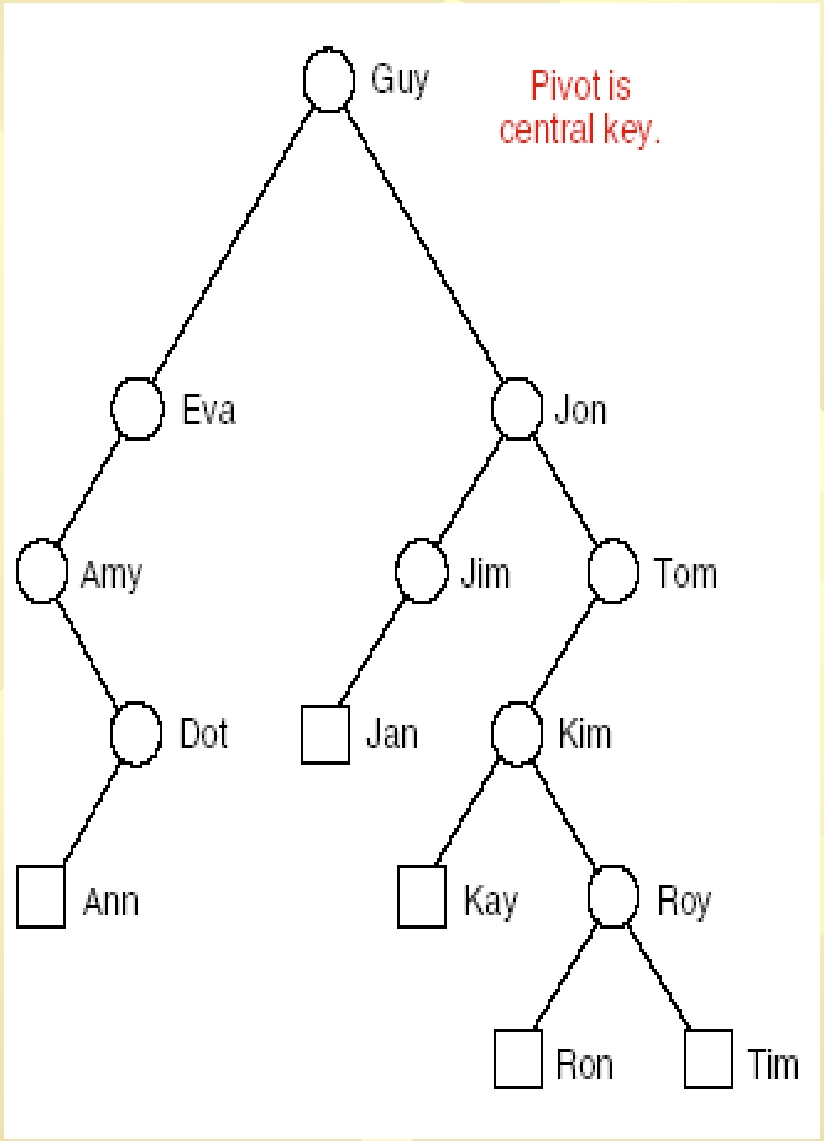
  - Combine into (12, 19, 22, 26, 29, 33 35)

Quick sort of 7 Numbers



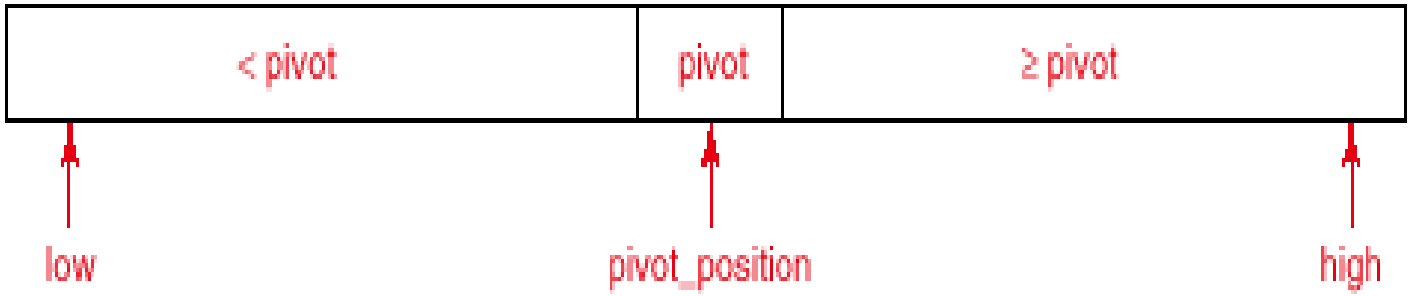
Quick sort



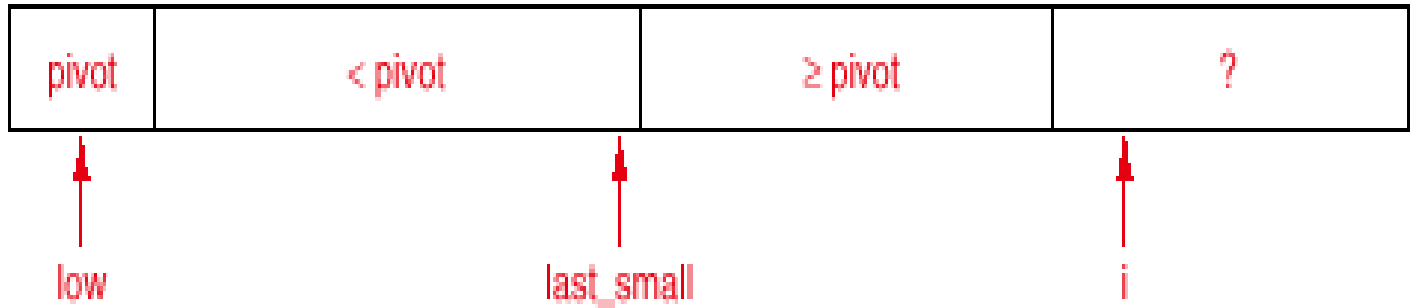


Quicksort for Contiguous Lists

Goal (postcondition)

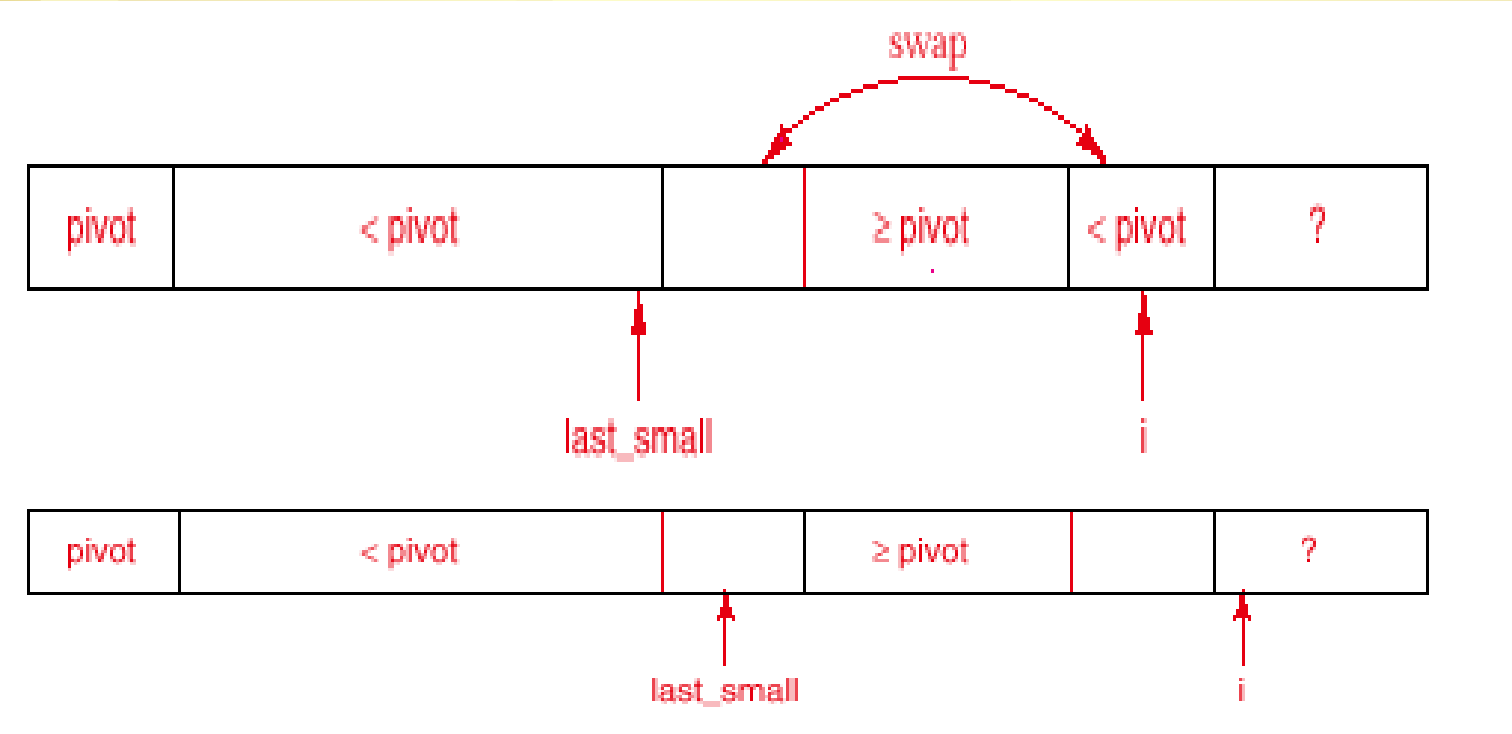


Loop invariant



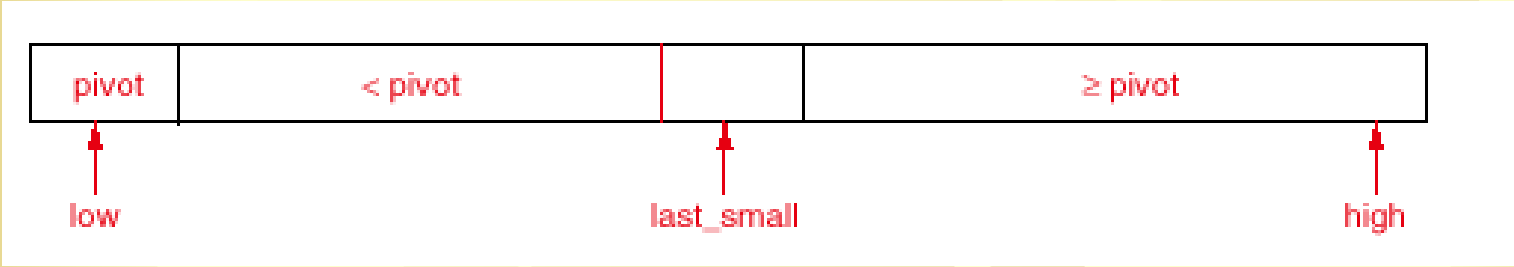
Quicksort for Contiguous Lists

Restore the invariant



Quicksort for Contiguous Lists

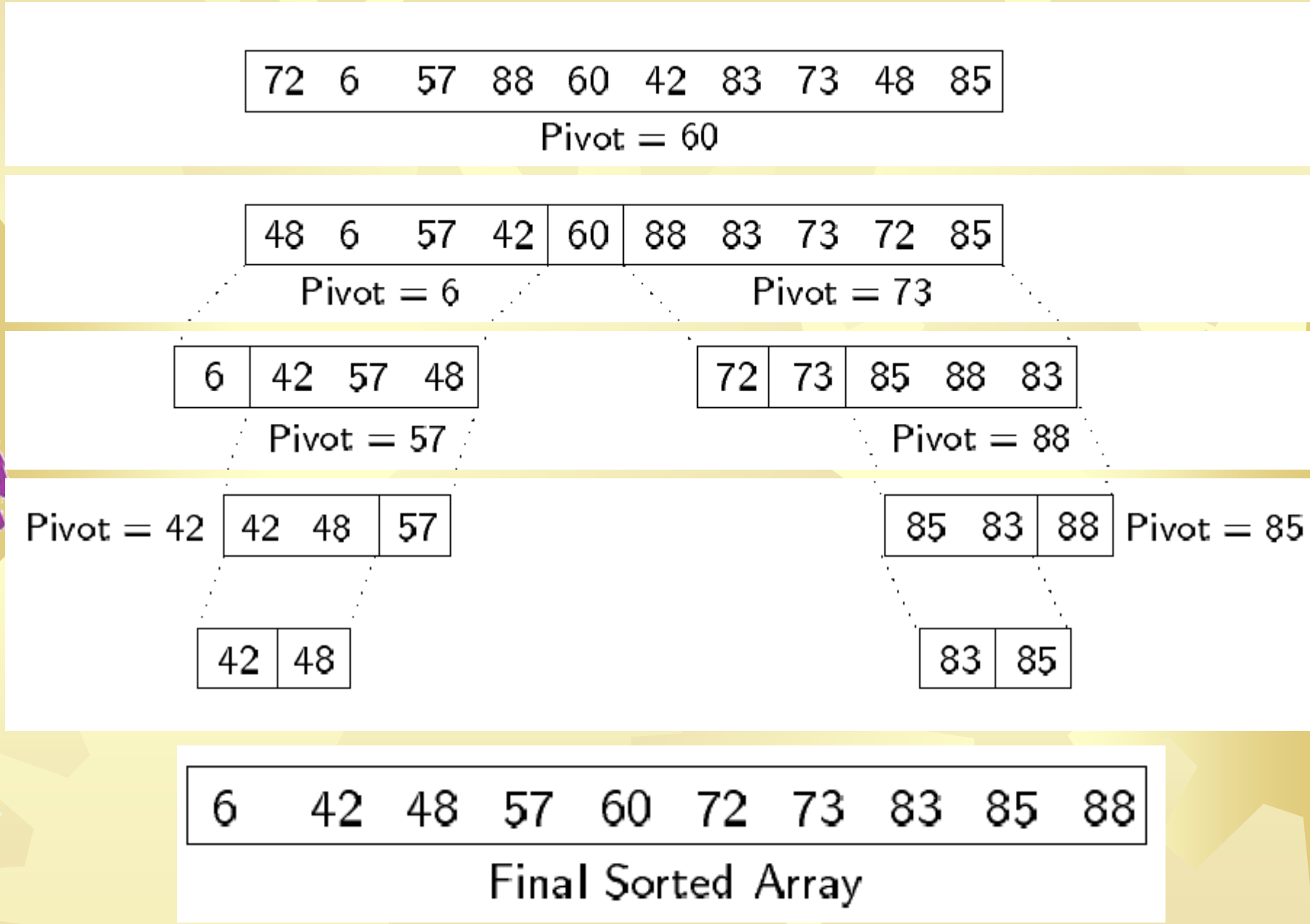
Final position



Quicksort Example

Initial	72	6	57	88	85	42	83	73	48	60
	l									r
Pass 1	72	6	57	88	85	42	83	73	48	60
	l								r	
Swap 1	48	6	57	88	85	42	83	73	72	60
	l								r	
Pass 2	48	6	57	88	85	42	83	73	72	60
				l		r				
Swap 2	48	6	57	42	85	88	83	73	72	60
				l		r				

Swap 3	48	6	57	85 r	42 l	88	83	73	72	60
Pass 3	48	6	57	42 r	85 l	88	83	73	72	60
Reverse Swap	48	6	57	42 r	85 l	88	83	73	72	60


72 6 57 88 60 42 83 73 48 85
Pivot = 60
48 6 57 42 60 88 83 73 72 85
Pivot = 6
Pivot = 73
6 42 57 48
72 73 85 88 83
Pivot = 57
Pivot = 88
Pivot = 42
42 48 57
85 83 88
Pivot = 85
42 48
83 85
6 42 48 57 60 72 73 83 85 88
Final Sorted Array

Analysis of Quicksort

- Worst-case analysis:
  - If the pivot is chosen poorly, one of the partitioned sublists may be empty and the other reduced by only one entry. In this case, quicksort is slower than either insertion sort or selection sort.

Analysis of Quicksort

- Choice of pivot:
  - First or last entry: Worst case appears for a list already sorted or in reverse order.
  - Central entry: Poor cases appear only for unusual orders.
  - Random entry: Poor cases are very unlikely to occur.

Analysis of Quicksort

- Average-case analysis

In its average case, quicksort performs $C(n) = 2n \ln n + O(n) \approx 1.39n \lg n + O(n)$ comparisons of keys in sorting a list of n entries in random order.

Mathematical Analysis

- Take a list of n distinct keys in random order.
- Let C(n) be the average number of comparisons of keys required to sort the list.
- If the pivot is the p^{th} smallest key, let C(n, p) be the average number of key comparisons done.
- The partition function does n - 1 key comparisons, and the recursive calls do C(p – 1) and C(n – p) comparisons.
- For n≥ 2, we have C(n, p)= n - 1+ C(p – 1)+ C(n –p).

Mathematical Analysis

- Averaging from p = 1 to p = n gives,

  for n ≥ 2,

$$C(n) = n - 1 + \frac{2}{n}(C(0) + C(1) + \cdots + C(n-1))$$

An equation of this form is called a *recurrence relation* because it expresses the answer to a problem in terms of earlier, smaller cases of the same problem.

Mathematical Analysis

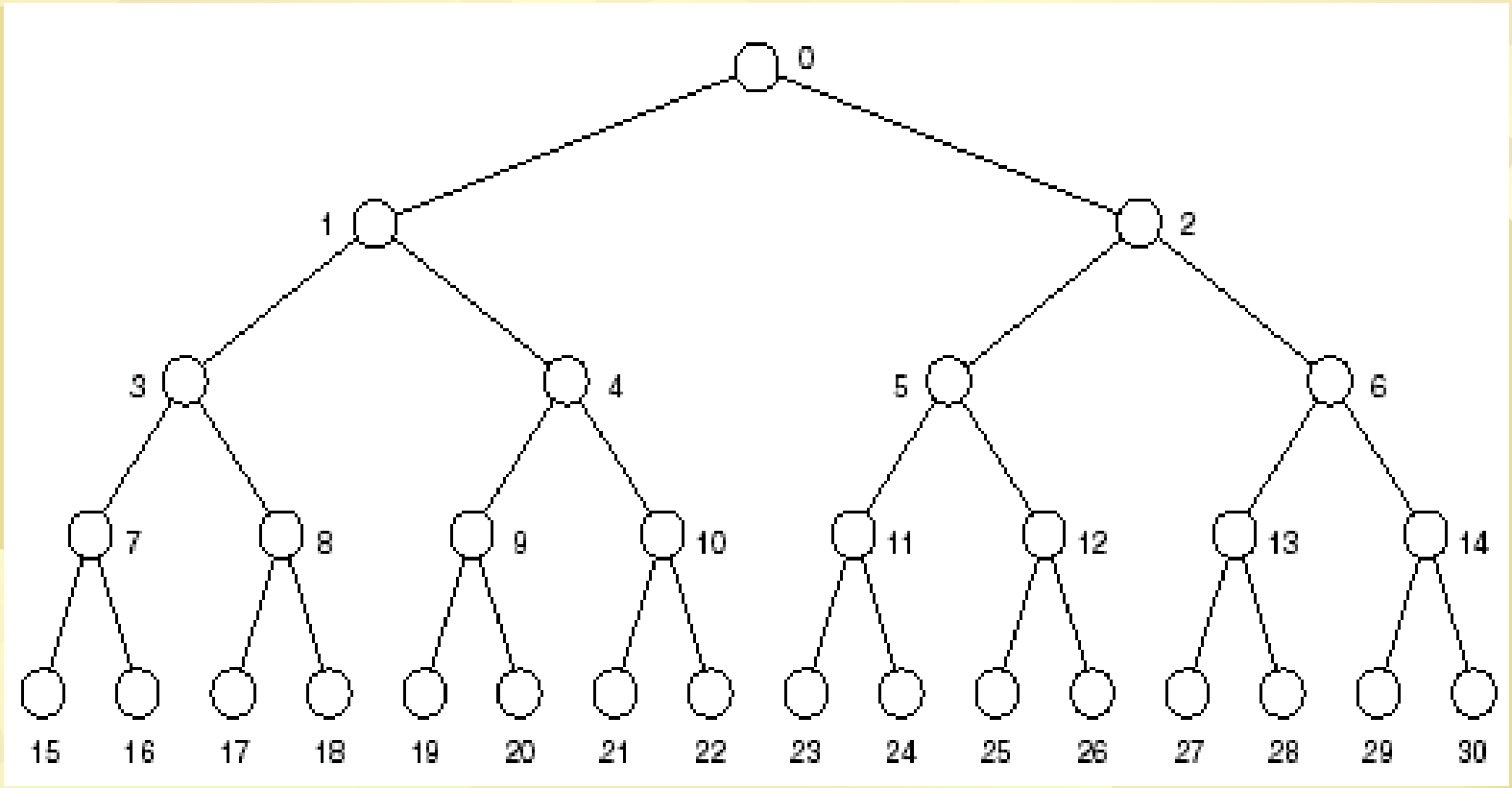
- Write down the same recurrence relation for case n-1; multiply the first by n and the second by n - 1; and subtract:

$$\frac{C(n)-2}{n+1} = \frac{C(n-)-2}{n} + \frac{2}{n+1}$$

- Use the equation over and over to reduce to the case C(1)= 0:

$$\frac{C(n)-2}{n+1} = -1 + 2\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{4} + \frac{1}{3}\right)$$

Trees, Lists, and Heaps

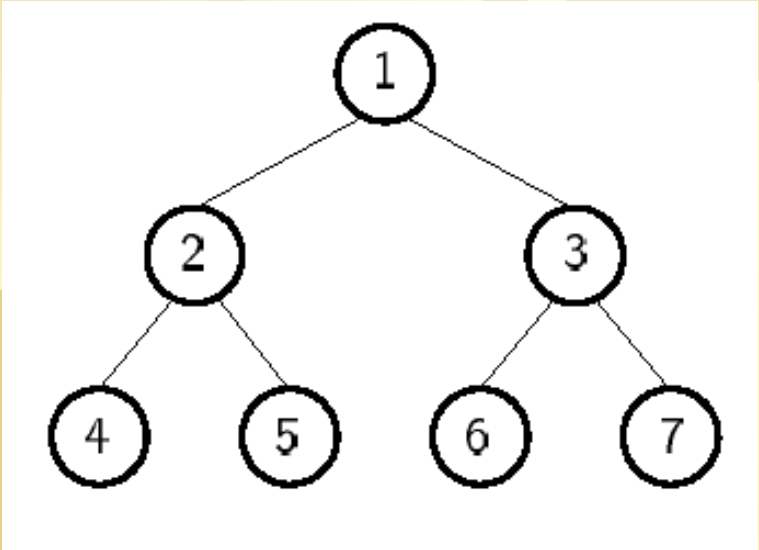


Trees, Lists, and Heaps

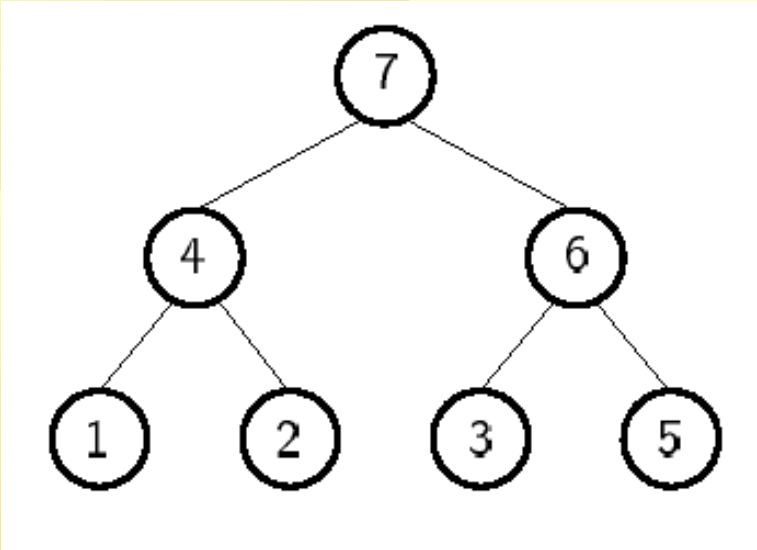
- If the root of the tree is in position 0 of the list, then the left and right children of the node in position k are in positions 2k + 1 and 2k + 2 of the list, respectively. If these positions are beyond the end of the list, then these children do not exist.
- DEFINITION: A *heap* is a list in which each entry contains a key, and, for all positions k in the list, the key at position k is at least as large as the keys in positions 2k + 1 and 2k + 2, provided these positions exist in the list.

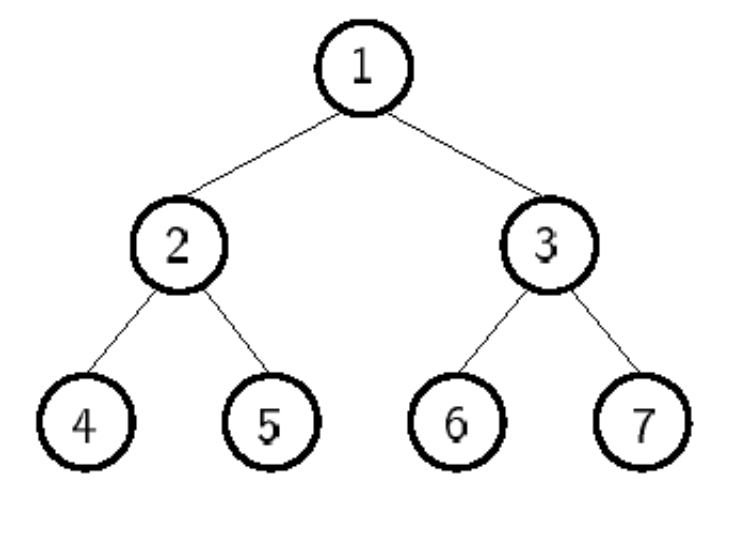
Heaps

- Heap: Complete binary tree with the Heap Property:
  - Min-heap: all values less than child values.
  - Max-heap: all values greater than child values.
- The values in a heap are partially ordered.
- Heap representation: normally the array based complete binary tree representation.

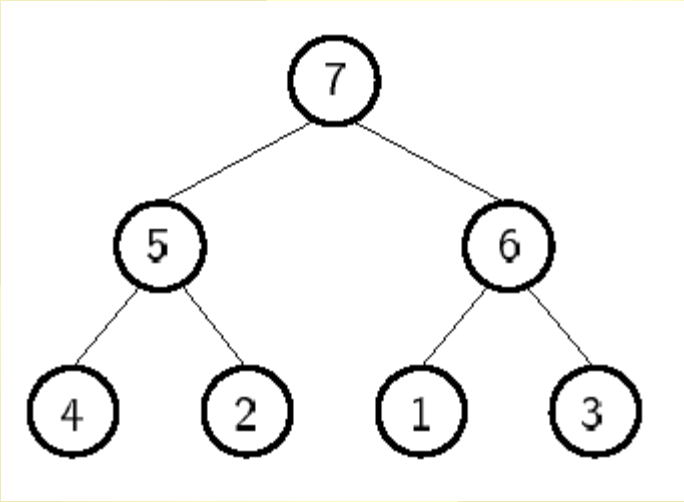


requires exchanges (4-2), (4-1), (2-1),(5-2), (5-4), (6-3), (6-5), (7-5), (7-6).





requires exchanges (5-2), (7-3), (7-1),(6-1)



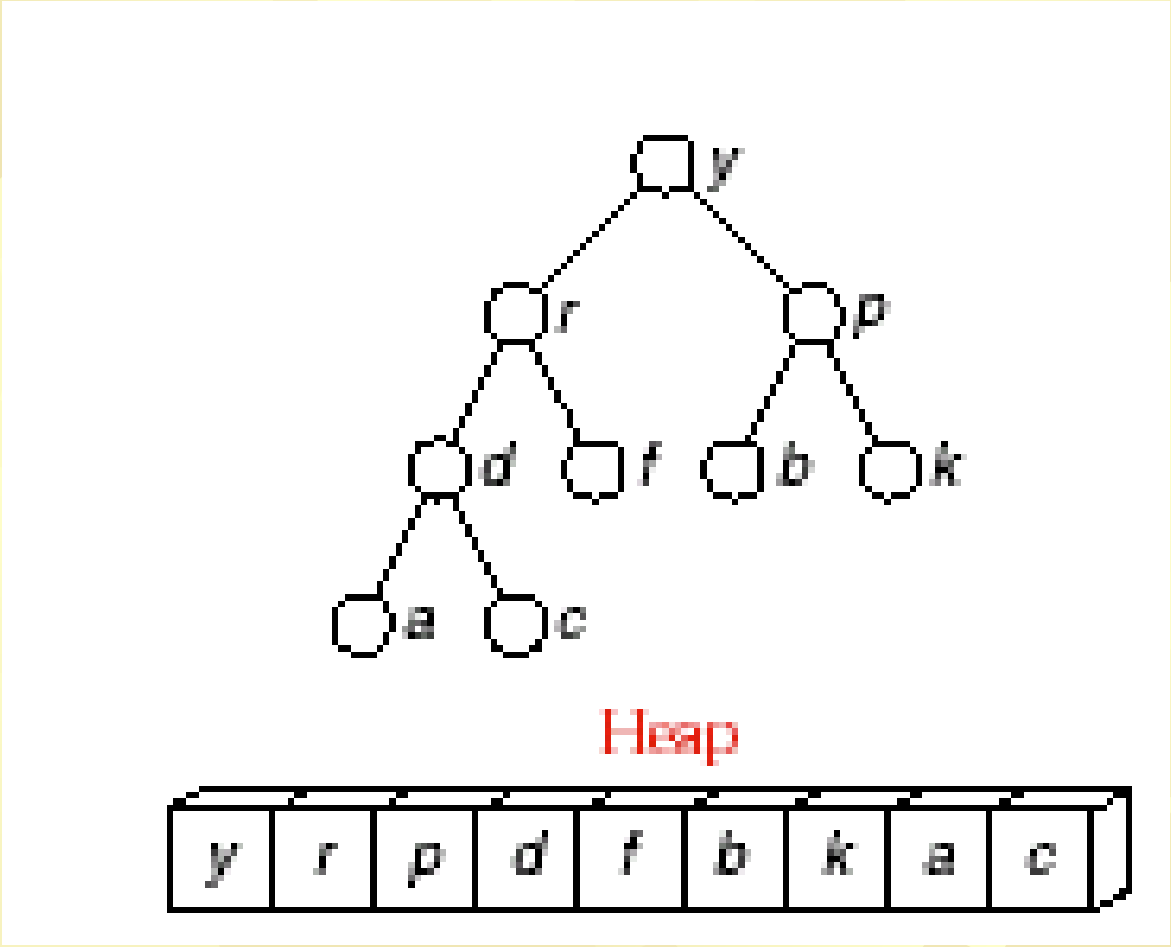
Construction

- For fast heap construction
  - Work from high end of array to low end.
  - Call siftdown for each item.
  - Don't need to call siftdown on leaf nodes.

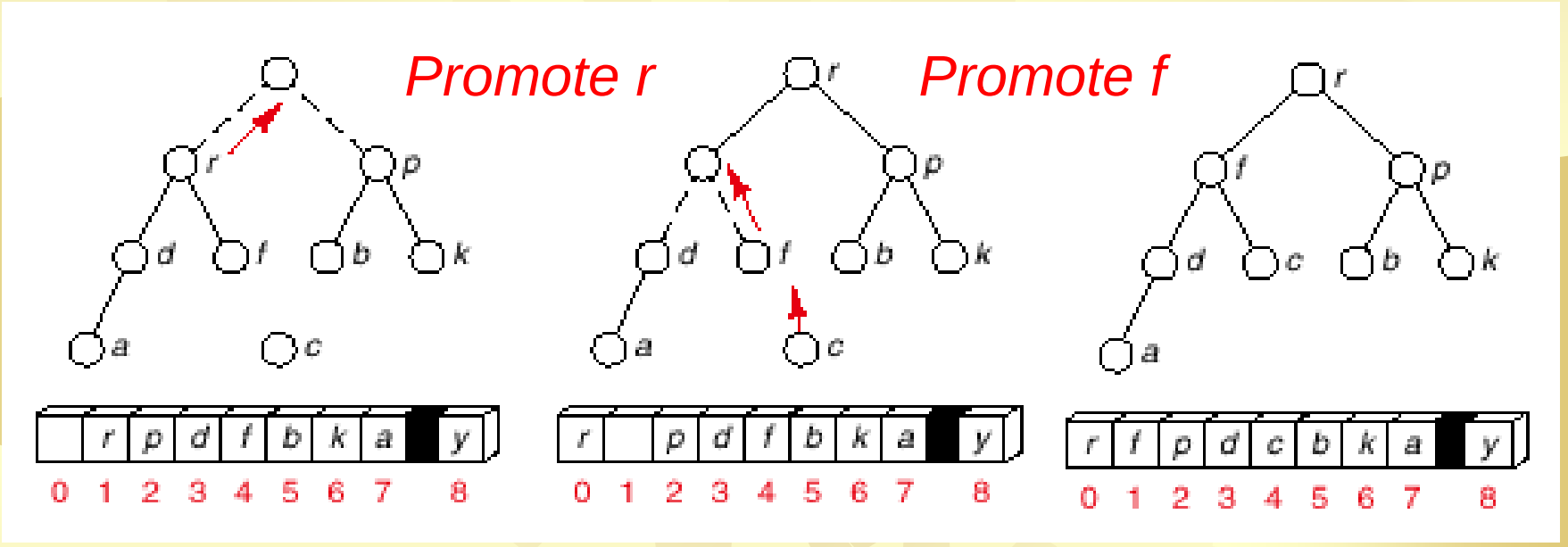
Trees, Lists, and Heaps

- REMARK Many C++ manuals and textbooks refer to the area used for dynamic memory as the “heap”; this use of the word “heap” has nothing to do with the present definition.
- *Heapsort* is suitable only for contiguous lists.

Trees, Lists, and Heaps

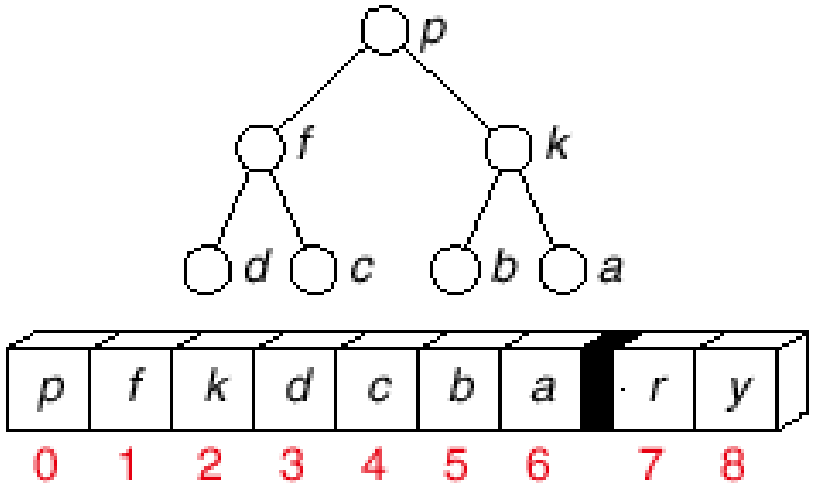


Trees, Lists, and Heaps

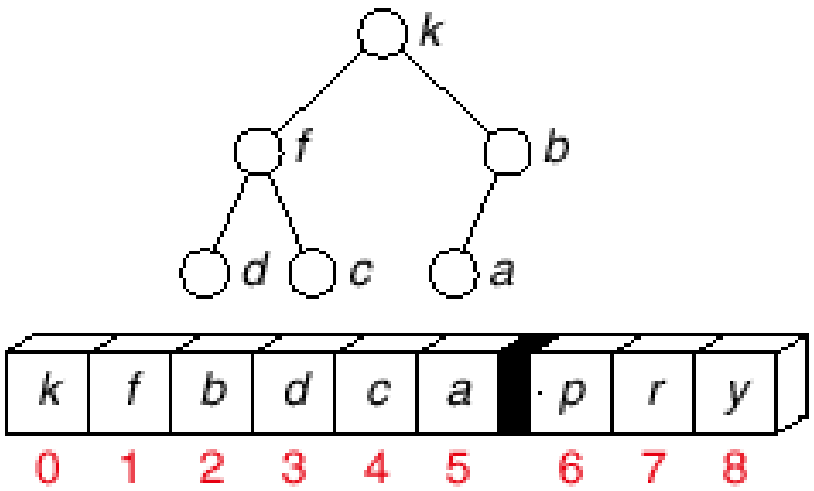


Trees, Lists, and Heaps

Remove r,
Promote p, k
Reinsert a:

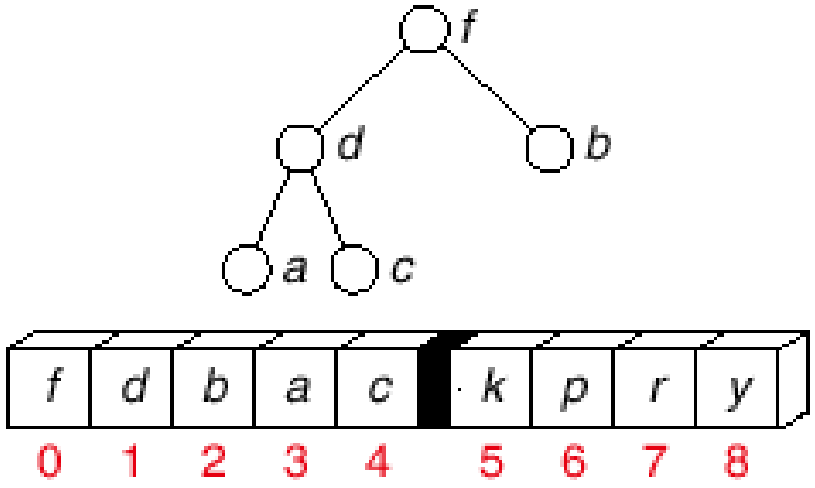


Remove p
Promote k, b
Reinsert a:

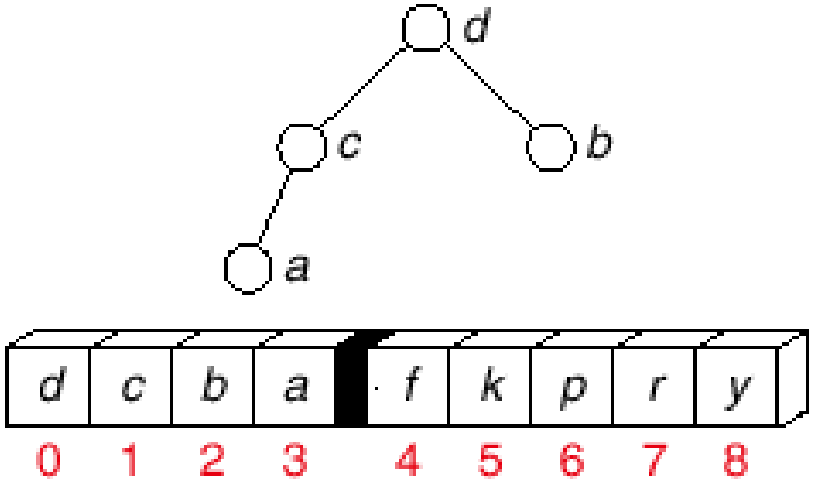


Trees, Lists, and Heaps

Remove k,
Promote f, d
Reinsert a:

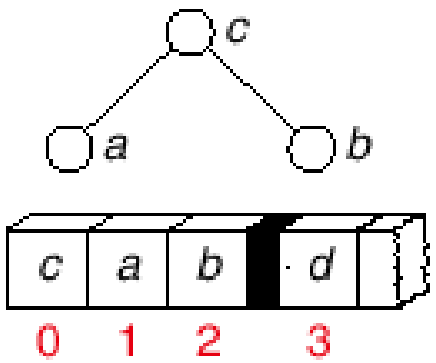


Remove f,
Promote d
Reinsert c:

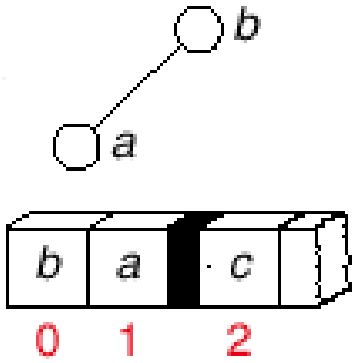


Trees, Lists, and Heaps

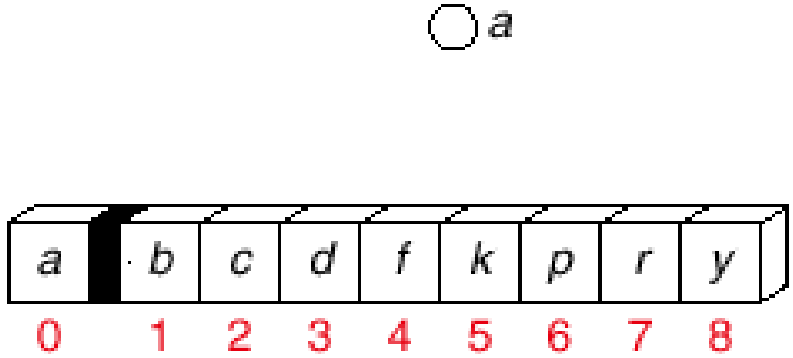
Remove d, Promote c Reinsert d:



Remove c, Promote b



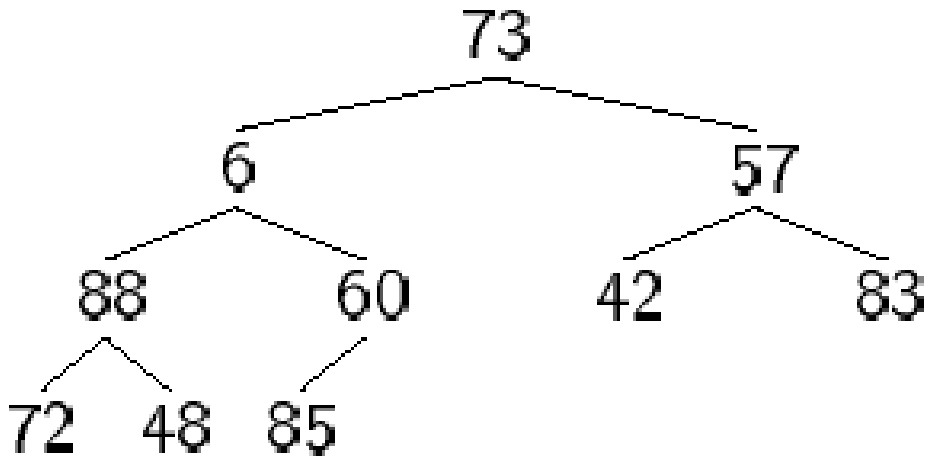
Remove b, Promote a



Heapsort Example

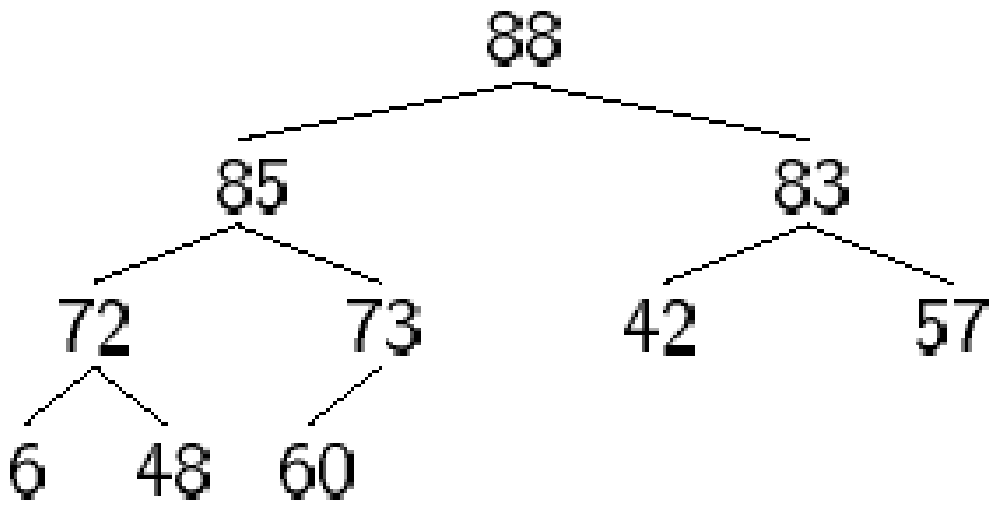
Original Numbers

73	6	57	88	60	42	83	72	48	85



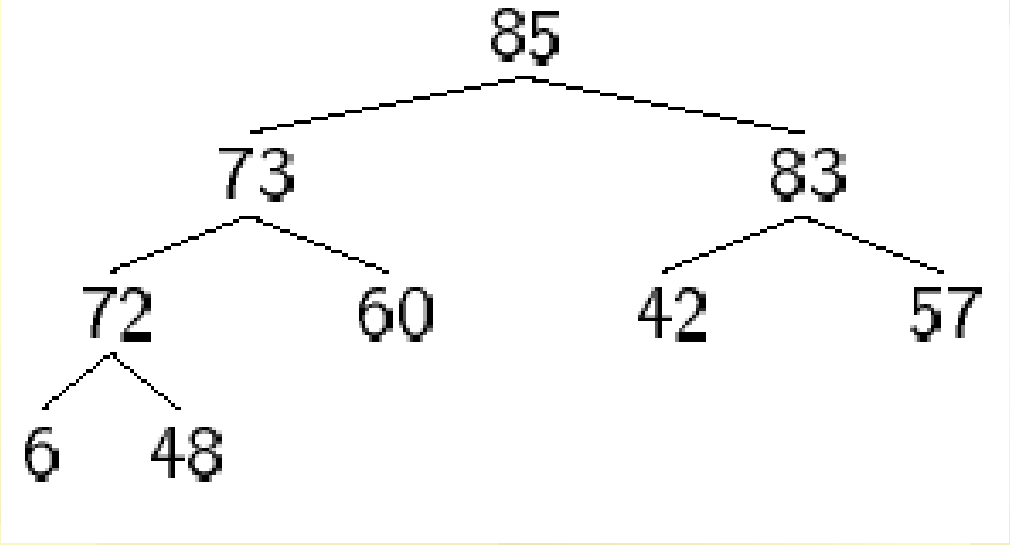
Build Heap

88	85	83	72	73	42	57	6	48	60



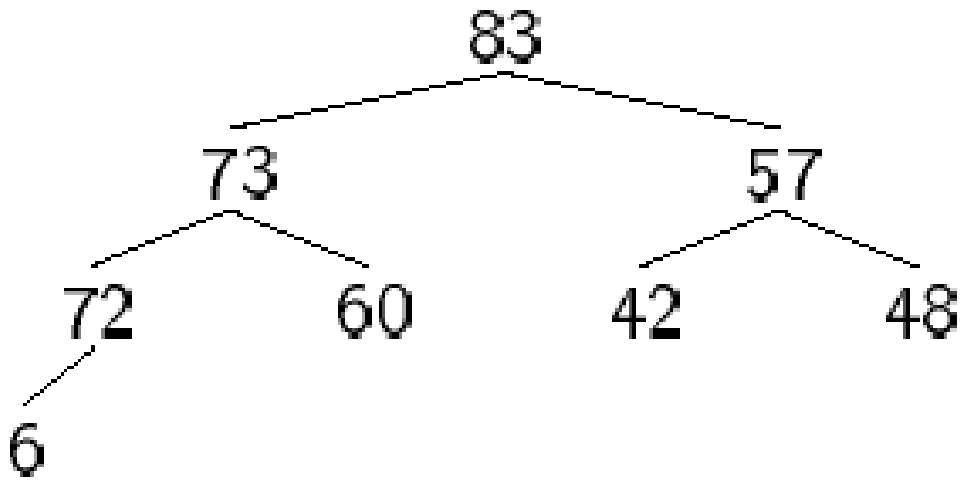
Remove 88

85	73	83	72	60	42	57	6	48	88



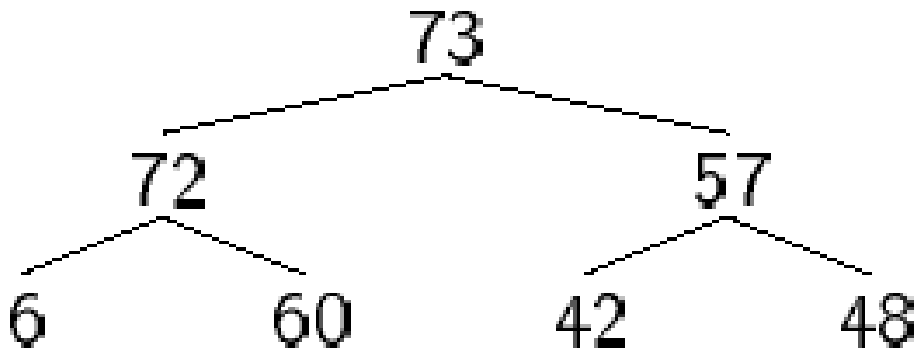
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Analysis of Heapsort

- In its worst case for sorting a list of length n, heapsort performs

  $2n \lg n + O(n)$

  comparisons of keys,

- it performs

  $n \lg n + O(n)$

  assignments of entries.

Priority Queues

- DEFINITION: A *priority queue* consists of entries, such that each entry contains a key called the *priority* of the entry. A priority queue has only two operations other than the usual creation, clearing, size, full, and empty operations:
  - Insert an entry.
  - Remove the entry having the largest (or smallest) key.
- If entries have equal keys, then any entry with the largest key may be removed first.

Priority Queues

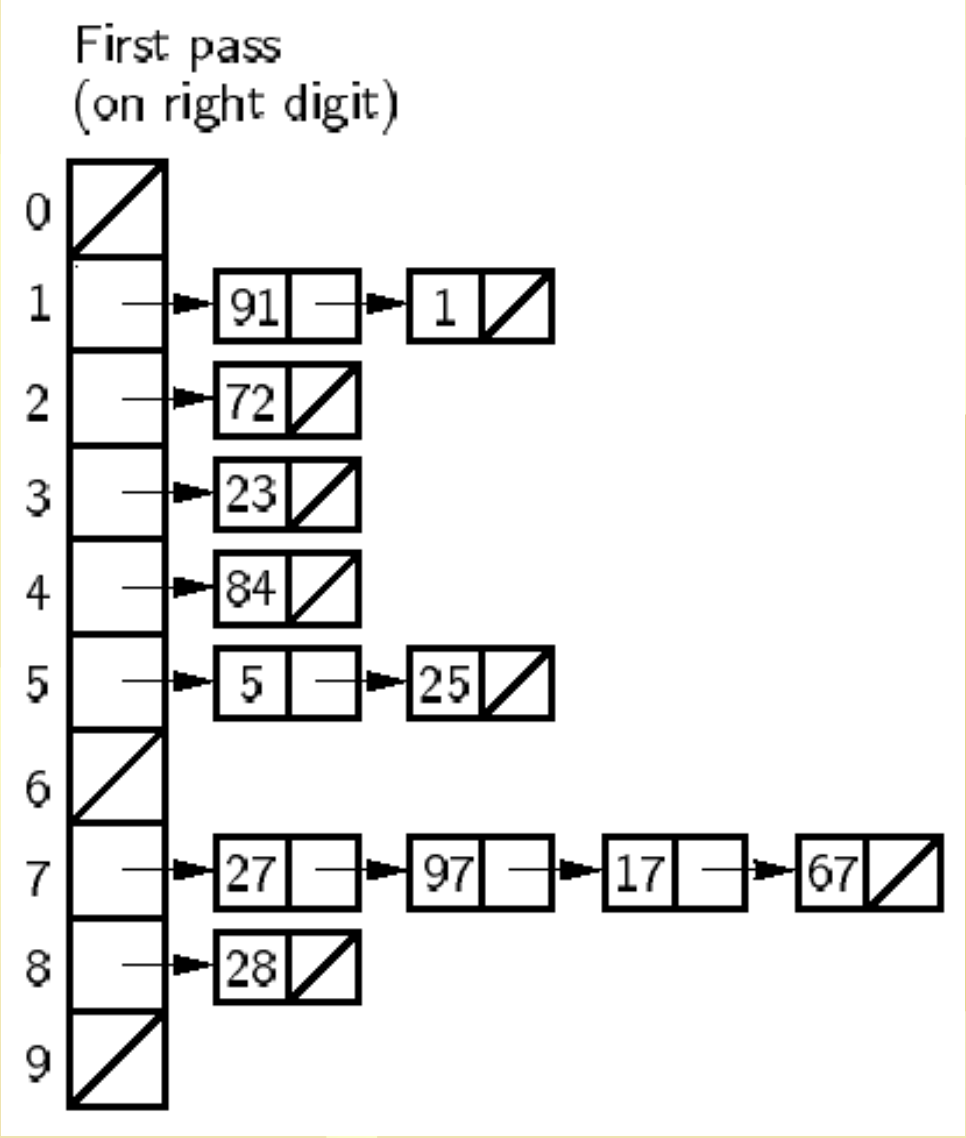
- A priority queue stores objects, and on request releases the object with greatest value.
- Example: Scheduling jobs in a multi-tasking operating system.
- The priority of a job may change, requiring some reordering of the jobs.
- Implementation: use a heap to store the priority queue.
- To support priority reordering, delete and re-insert. Need to know index for the object.

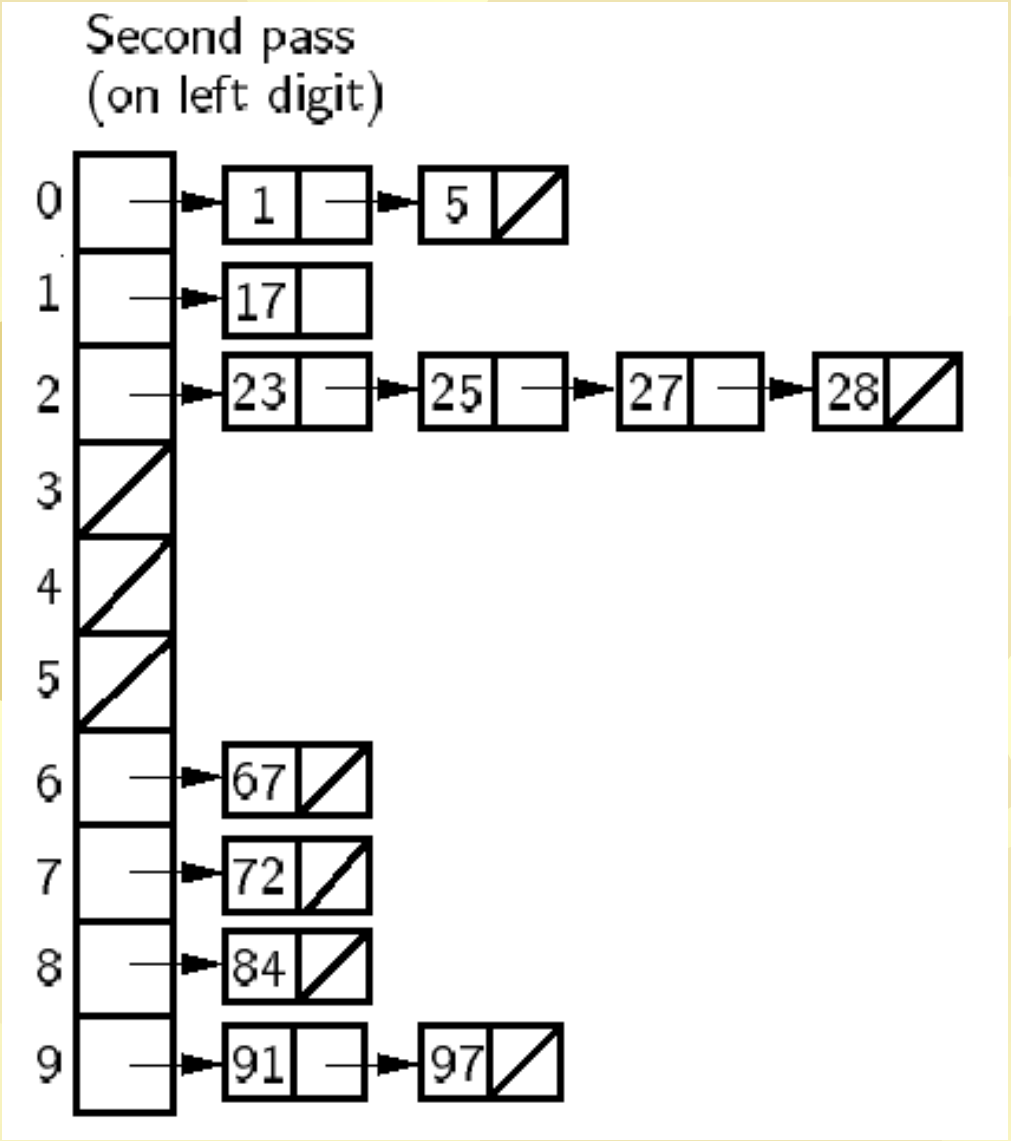
Radix Sort

- Initial List:

  27 91 1 97
  17 23 84 28
  72 5 67 25

- **Result of first pass:**
  91 1 72 23
  84 5 25 27
  97 17 67 28





- **Result of second pass: 1 17 5 23 25 27 28 67 72 84 91 97**

Radix Sort Example

- Initial Input: Array A

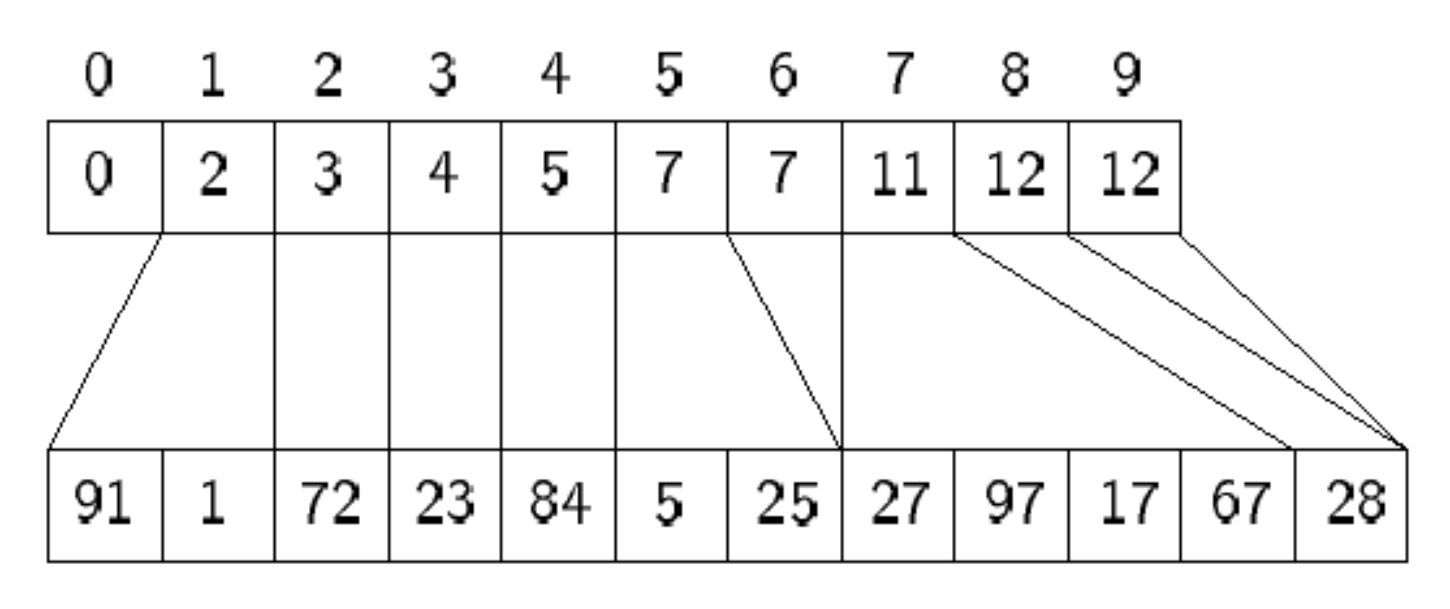
27	91	1	97	17	23	84	28	72	5	67	25

- First pass values for Count. rtok = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

- Count array: Index positions for Array B.

End of Pass 1: Array A.

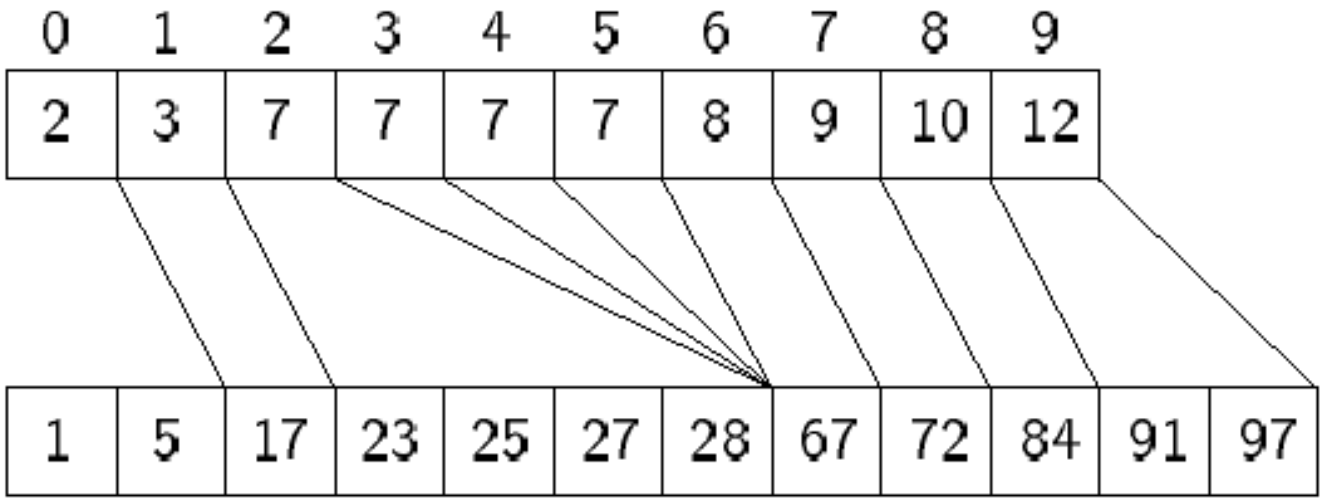


- Second pass values for Count. rtok = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

- Count array: Index positions for Array B.

End of Pass 2: Array A.



Comparison of Sorting Methods

- Use of space
- Use of computer time
- Programming effort
- Statistical analysis
- Empirical testing

Pointers and Pitfalls

- Many computer systems have a general-purpose sorting utility.If you can access this utility and it proves adequate for your application, then use it rather than writing a sorting program from scratch.

Pointers and Pitfalls

- In choosing a sorting method, take into account the ways in which the keys will usually be arranged before sorting, the size of the application, the amount of time available for programming,the need to save computer time and space, the way in which the data structures are implemented, the cost of moving data, and the cost of comparing keys.

Pointers and Pitfalls

- Divide-and-conquer is one of the most widely applicable and most powerful methods for designing algorithms. When faced with a programming problem, see if its solution can be obtained by first solving the problem for two (or more) problems of the same general form but of a smaller size. If so, you may be able to formulate an algorithm that uses the divide-and conquer method and program it using recursion.

Pointers and Pitfalls

- Mergesort, quicksort, and heapsort are powerful sorting methods,more difficult to program than the simpler methods, but much more efficient when applied to large lists. Consider the application carefully to determine whether the extra effort needed to implement one of these sophisticated algorithms will be justified.

Pointers and Pitfalls

- Priority queues are important for many applications. Heaps provide an excellent implementation of priority queues.
- Heapsort is like an insurance policy: It is usually slower than quicksort, but it guarantees that sorting will be completed in O(n log n) comparisons of keys, as quicksort cannot always do.