

# Chapter 10 - Sorting

One of the most **important concepts** and **common applications** in computing.

23	78	45	8	32	56
----	----	----	---	----	----



8	23	32	45	56	78
---	----	----	----	----	----

# Sorting

- **Internal sort**: all data are held in **primary memory** during the sorting process.
- **External sort**: primary memory for data currently being sorted and secondary storage for data that do not fit in primary memory.

Số liệu

thư viện

# Sorting

**Sort stability:** data with equal keys maintain their relative input order in the output.

78	8	45	8	32	56
----	---	----	---	----	----

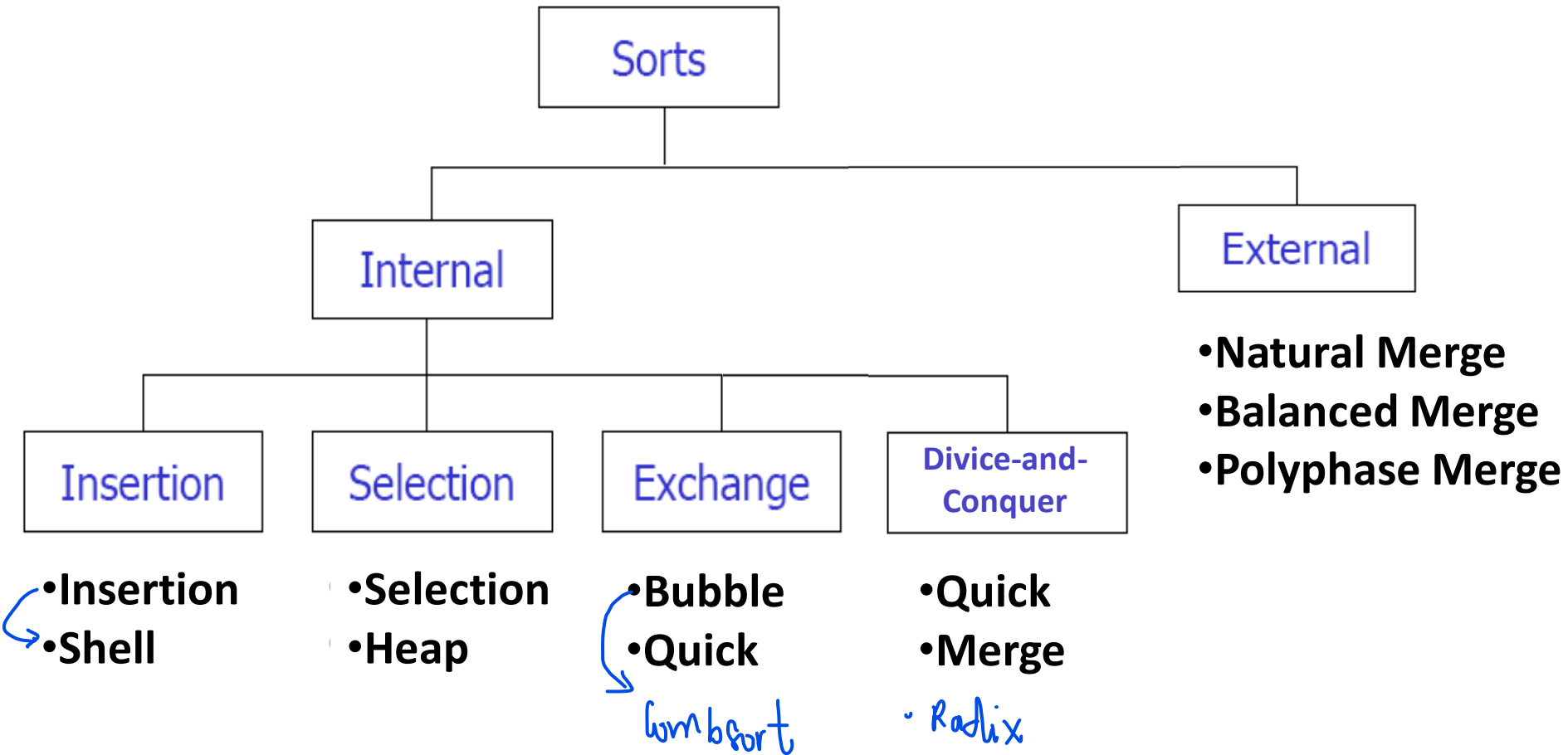


8	8	32	45	56	78
---	---	----	----	----	----

# Sorting

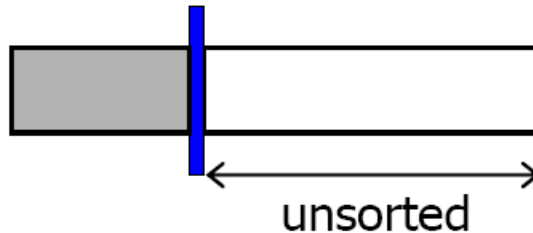
- **Sort efficiency**: a measure of the relative efficiency of a sort = number of **comparisons** + number of **moves**

# Sorting

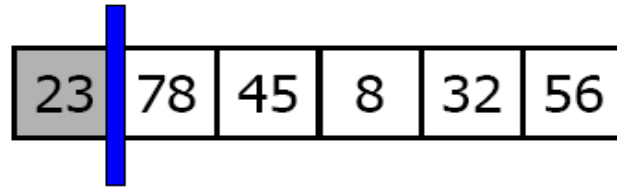


# Straight Insertion Sort

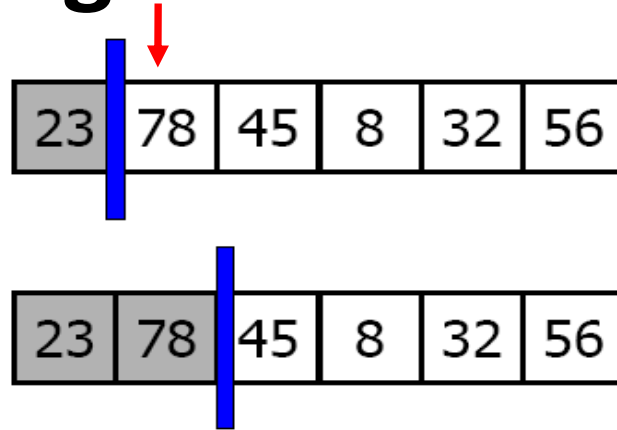
- The list is divided into two parts: **sorted** and **unsorted**.
- In each pass, the first element of the unsorted sublist is **inserted** into the sorted sublist.



# Straight Insertion Sort

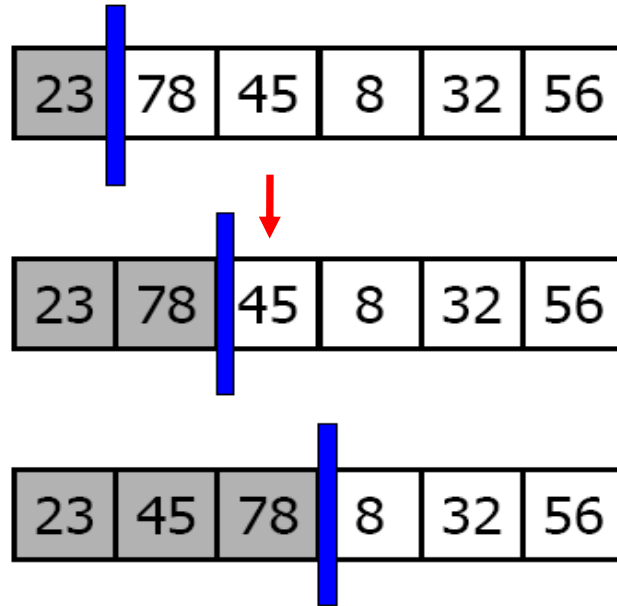


# Straight Insertion Sort

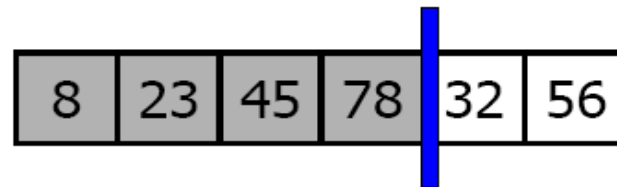
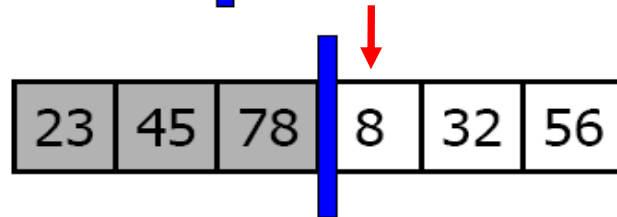
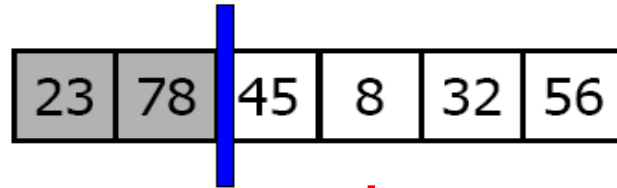
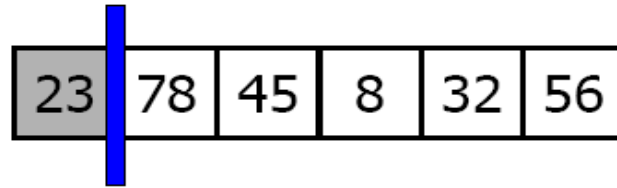




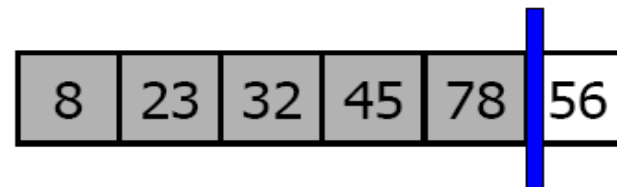
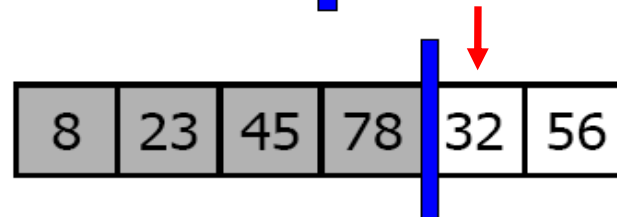
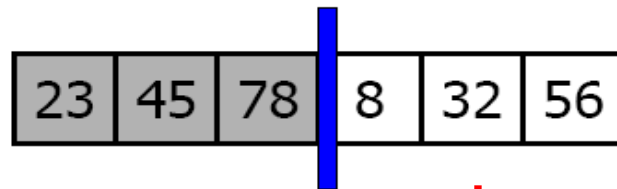
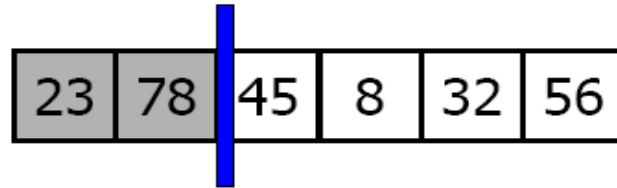
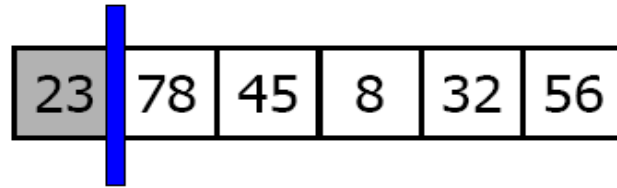
# Straight Insertion Sort



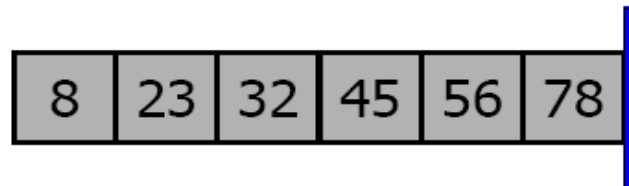
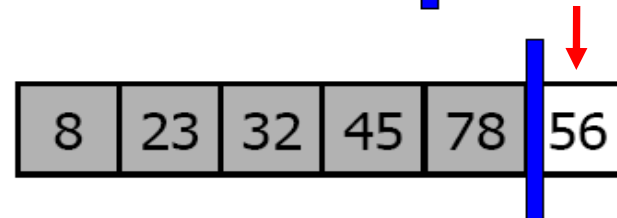
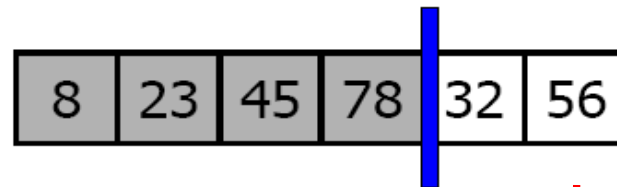
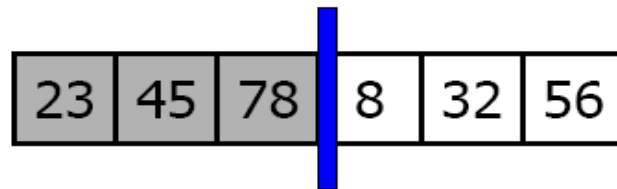
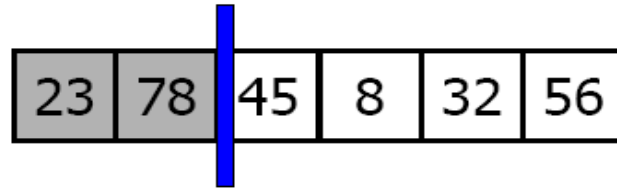
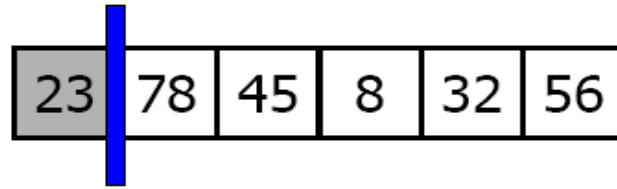
# Straight Insertion Sort



# Straight Insertion Sort



# Straight Insertion Sort



# Straight Insertion Sort

Algorithm **InsertionSort** ()

Sorts the contiguous list using straight insertion sort

**Post** sorted list.

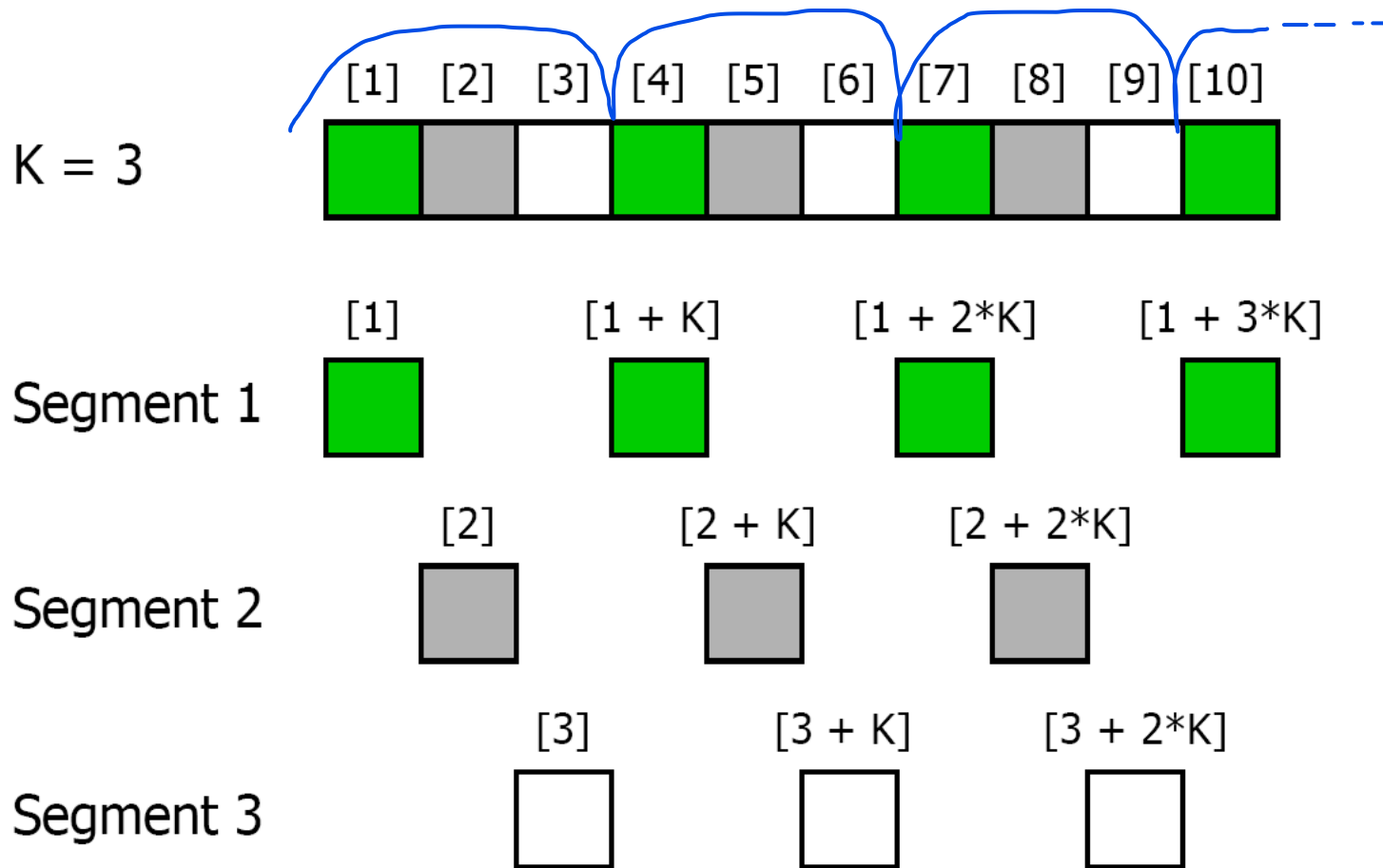
1. **if** (**count** > 1)
  1. **current** = 1
  2. **loop** (**current** < **count** )
    1. **temp** = **data**<sub>current</sub>
    2. **walker** = **current**-1
    3. **loop** (**walker** >= 0) AND (**temp**.key < **data**<sub>walker</sub>.key)
      1. **data**<sub>walker+1</sub> = **data**<sub>walker</sub>
      2. **walker** = **walker** -1
    4. **data**<sub>walker+1</sub> = **temp**
    5. **current** = **current** + 1

End InsertionSort

# Shell Sort

- Named after its creator Donald L. Shell (1959).
- Given a list of  $N$  elements, the list is divided into  $K$  segments ( $K$  is called the increment).
- Each segment contains  $N/K$  or more elements.
- Segments are dispersed throughout the list.
- Also is called diminishing-increment sort

# Shell Sort



# Shell Sort

23	78	45	8	32	56
----	----	----	---	----	----

- For the value of **K** in each iteration, sort the **K** segments.
- After each iteration, **K** is reduced until it is **1** in the final iteration.



# Example of Shell Sort

Unsorted

Sublists incr. 5

5-Sorted

Recombined

Tim  
Dot  
Eva  
Roy  
Tom  
Kim  
Guy  
Amy  
Jon  
Ann  
Jim  
Kay  
Ron  
Jan

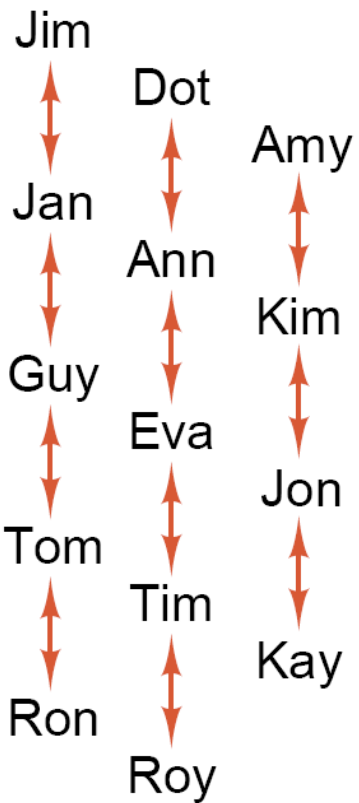
Tim  
Dot  
Eva  
Roy  
Tom  
Kim  
Guy  
Amy  
Jon  
Ann  
Jim  
Kay  
Ron  
Jan

Jim  
Dot  
Amy  
Jan  
Ann  
Kim  
Guy  
Eva  
Jon  
Tom  
Tim  
Kay  
Ron  
Roy

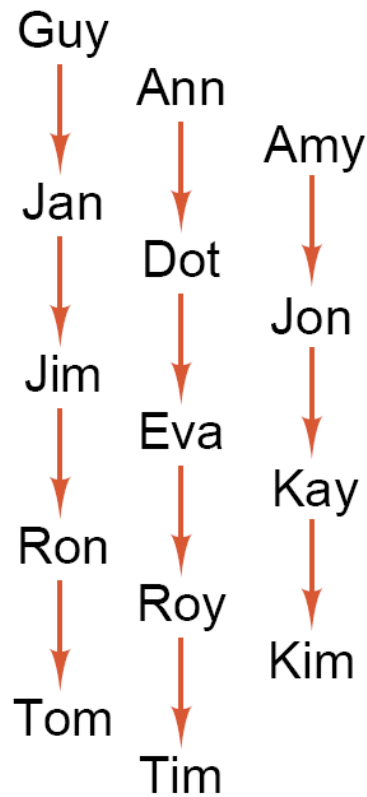
Jim  
Dot  
Amy  
Jan  
Ann  
Kim  
Guy  
Eva  
Jon  
Tom  
Tim  
Kay  
Ron  
Roy

# Example of Shell Sort

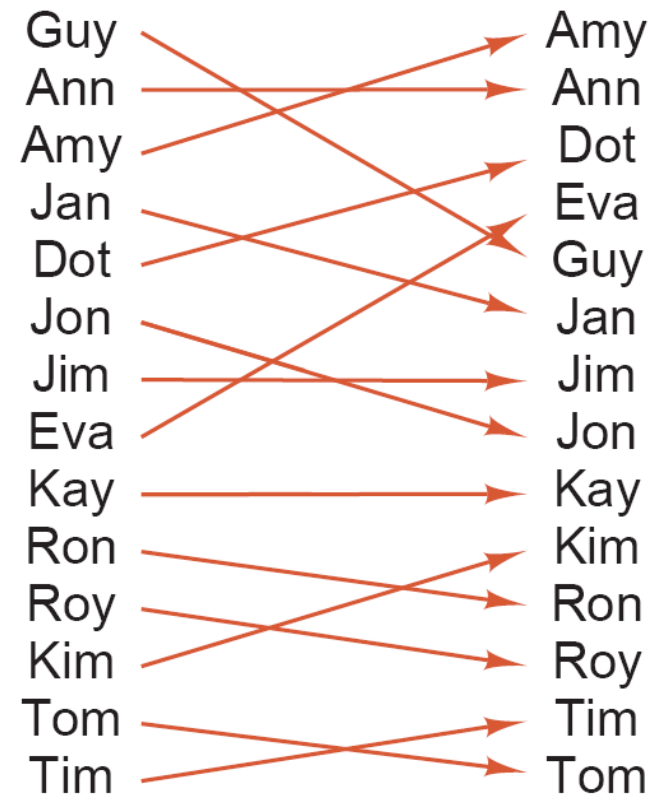
Sublists incr. 3



3-Sorted



List incr. 1



Sorted

# Choosing incremental values

- From more of the comparisons, it is better when we can receive more new information.
- Incremental values should not be multiples of each other, otherwise, the same keys compared on one pass would be compared again at the next.
- The final incremental value must be 1.

# Choosing incremental values

Incremental values may be:

1, 4, 13, 40, 121, ...

$$\left[ \begin{array}{l} k_t = 1 \\ k_{i-1} = 3 * k_i + 1 \\ t = \lceil \log_3(n) \rceil - 1 \end{array} \right.$$

or :

1, 3, 7, 15, 31,...

$$\left[ \begin{array}{l} k_t = 1 \\ k_{i-1} = 2 * k_i + 1 \\ t = \lceil \log_2(n) \rceil - 1 \end{array} \right.$$

# Shell Sort

Algorithm **ShellSort** ()

Sorts the contiguous list using Shell sort

**Post** sorted list.

1.  $k = \text{first\_incremental\_value}$  *interval =  $N/2$*
2. **loop** ( $k \geq 1$ )
  1.  $\text{segment} = 1$  *step*
  2. **loop** ( $\text{segment} \leq k$ )
    1. **SortSegment**(segment)
    2.  $\text{segment} = \text{segment} + 1$
  3.  $k = \text{next\_incremental\_value}$

End ShellSort

# Shell Sort

Algorithm **SortSegment**(val **segment** <int>, val **k** <int>)

Sorts the segment beginning at **segment** using insertion sort, step between elements in the segment is **k**.

**Post** sorted segment.

1. **current** = **segment** + **k** *→ interval*
2. **loop** (**current** < **count**) *current ≡ i | count ≡ N*
  1. **temp** = **data**[**current**]
  2. **walker** = **current** - **k** *walker ≡ j*
  3. **loop** (**walker** ≥ 0) AND (**temp**.key < **data**[**walker**].key)
    1. **data**[**walker** + **k**] = **data**[**walker**]
    2. **walker** = **walker** - **k**
  4. **data**[**walker** + **k**] = **temp**
  5. **current** = **current** + **k**

End SortSegment

# Insertion Sort Efficiency

- Straight insertion sort:

$$f(n) = n(n + 1)/2 = O(n^2)$$

- Shell sort:

$$O(n^{1.25}) \quad \text{Empirical study}$$

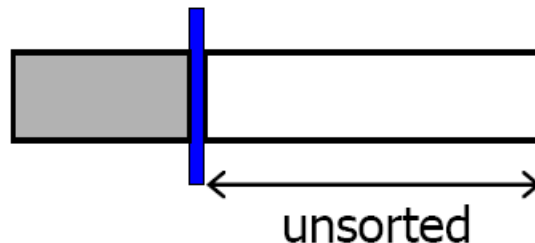
# Selection Sort

- In each pass, the smallest/largest item is selected and placed in a sorted list.

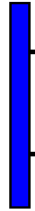


# Straight Selection Sort

- The list is divided into two parts: **sorted** and **unsorted**.
- In each pass, in the unsorted sublist, the smallest element is **selected** and **exchanged** with the first element.

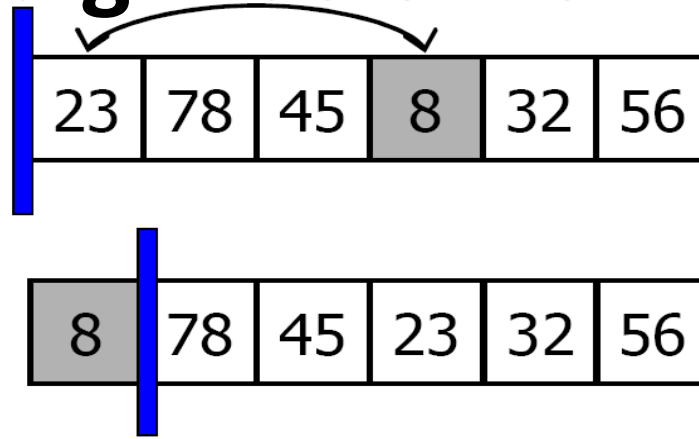


# Straight Selection Sort

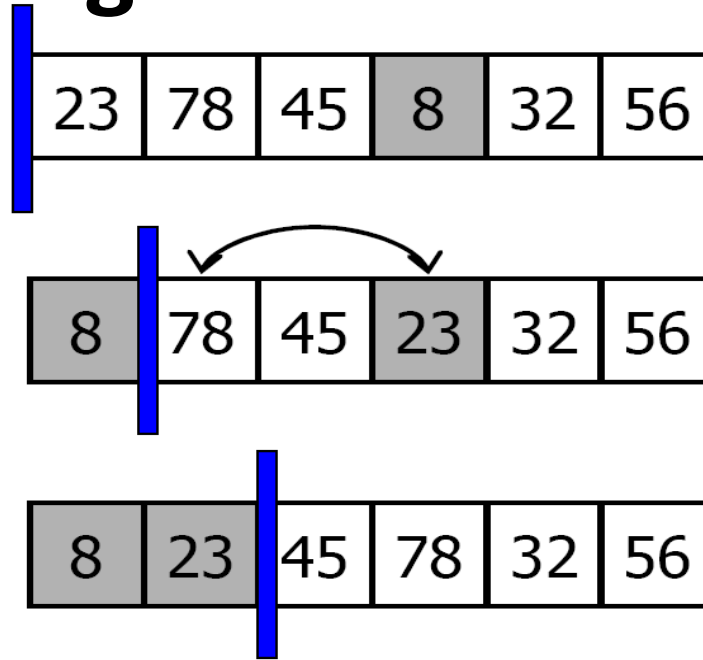


23	78	45	8	32	56
----	----	----	---	----	----

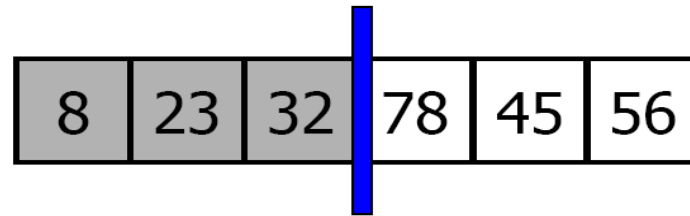
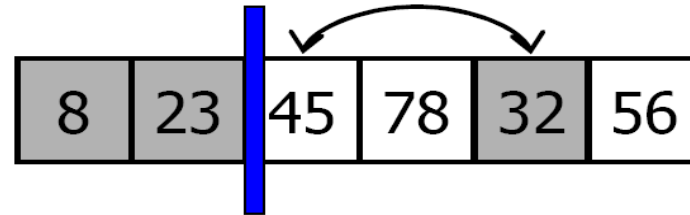
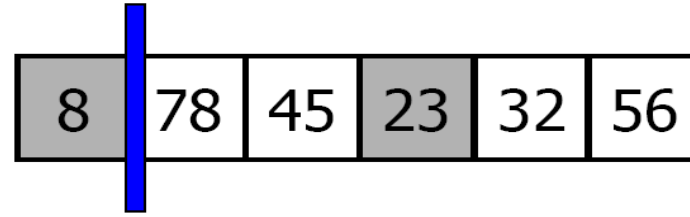
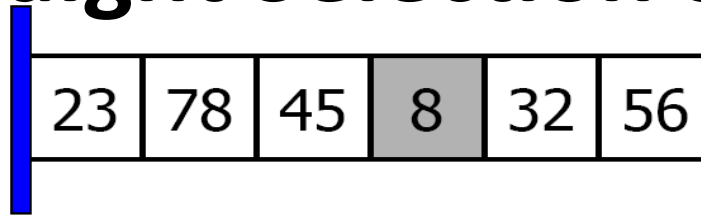
# Straight Selection Sort



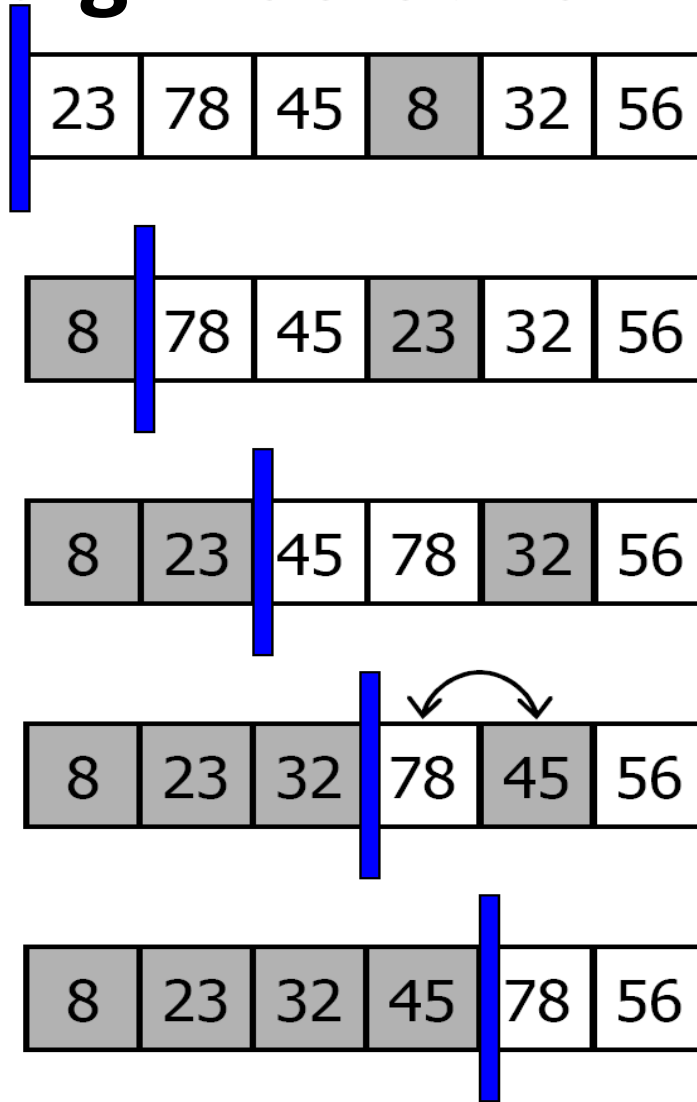
# Straight Selection Sort



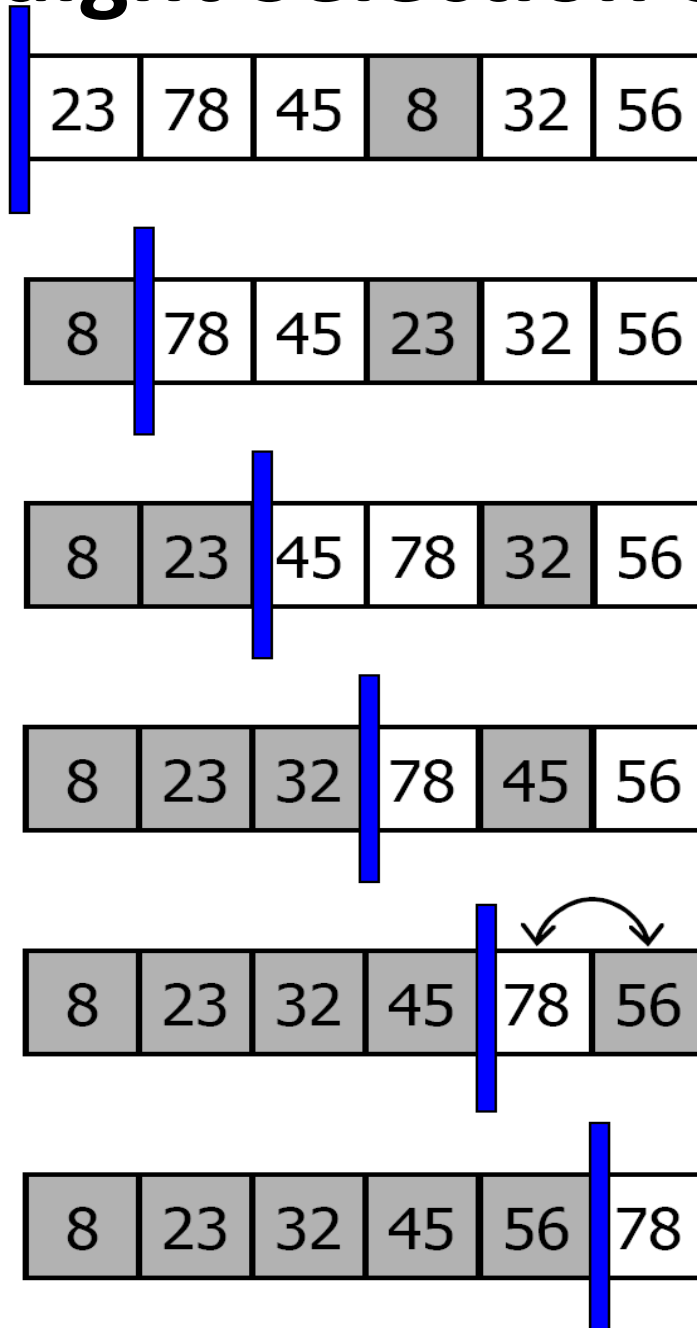
# Straight Selection Sort



# Straight Selection Sort



# Straight Selection Sort



# Selection Sort

Algorithm **SelectionSort** ()

Sorts the contiguous list using straight selection sort

Post sorted list.

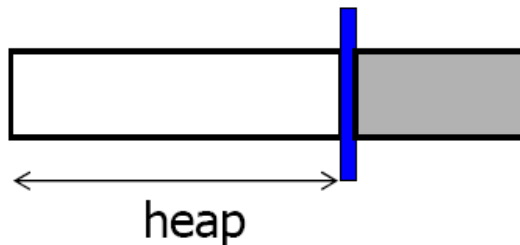
1. current = 0
2. **loop** (current < count - 1)
  1. smallest = current
  2. walker = current + 1
  3. **loop** (walker < count)
    1. if (data [walker].key < data [smallest].key)
      1. smallest = walker
    2. walker = walker+1
  4. swap(current, smallest)
  5. current = current + 1

End SelectionSort

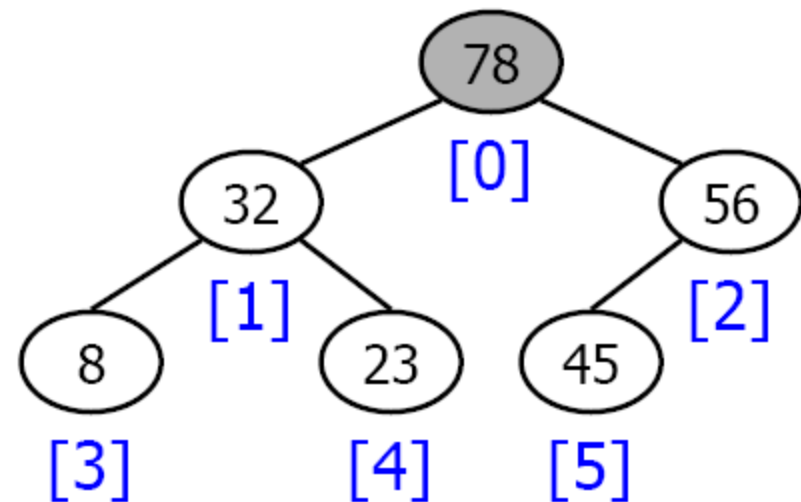
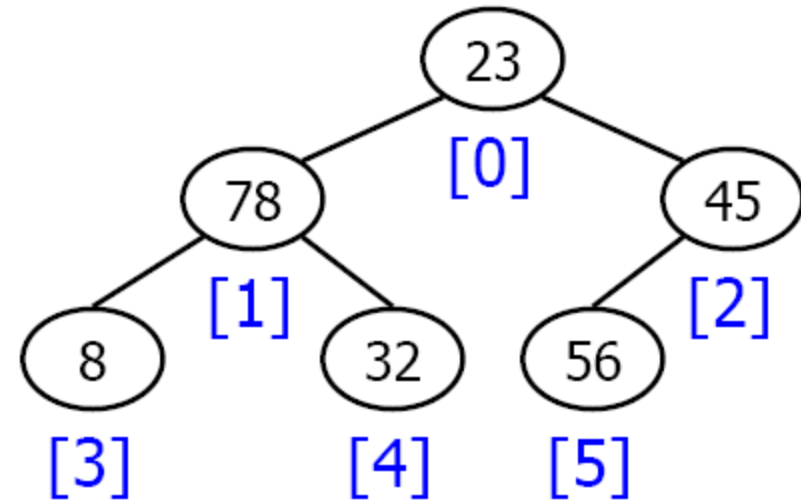
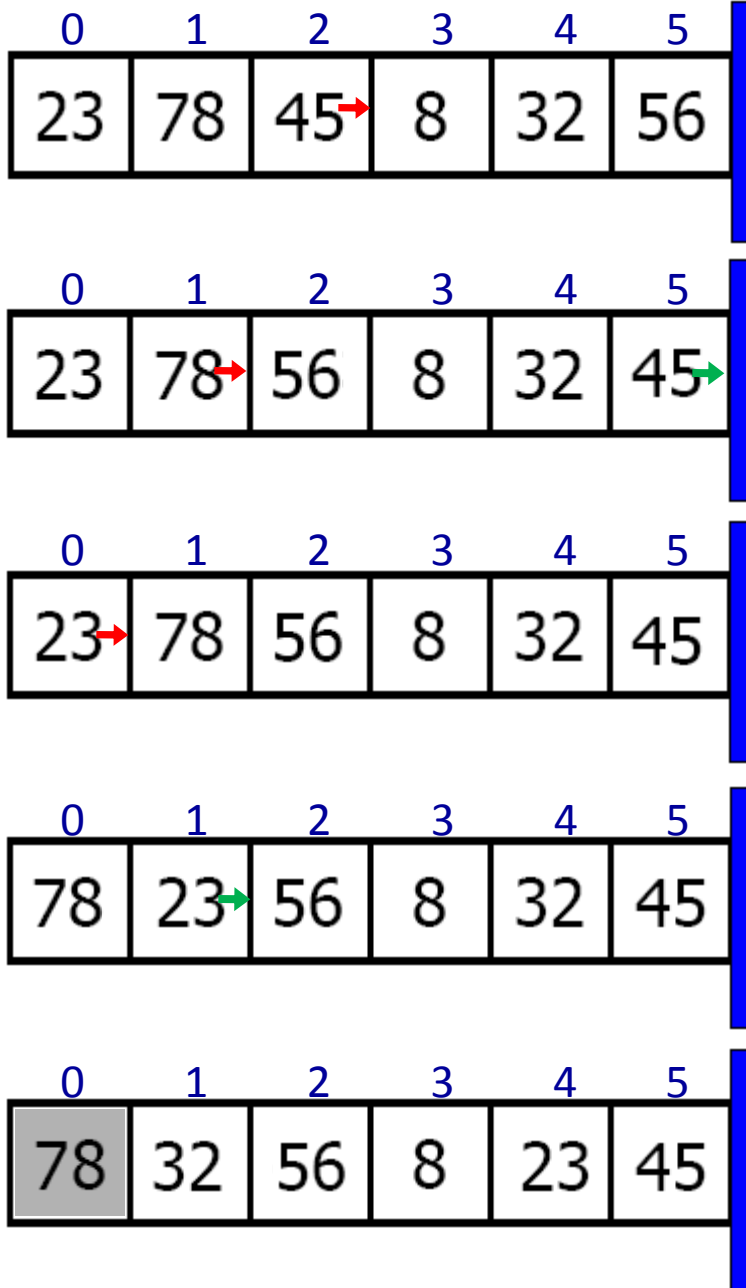


# Heap Sort

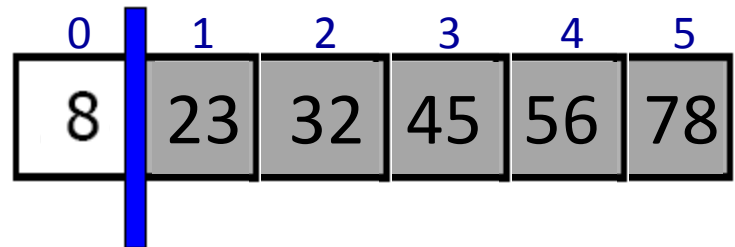
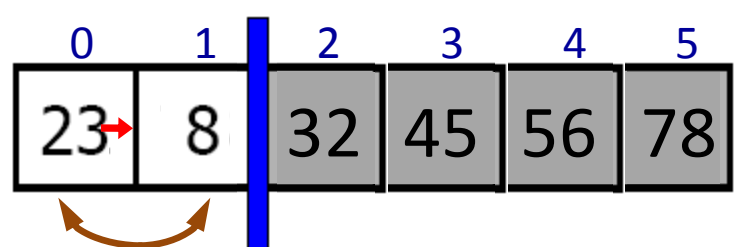
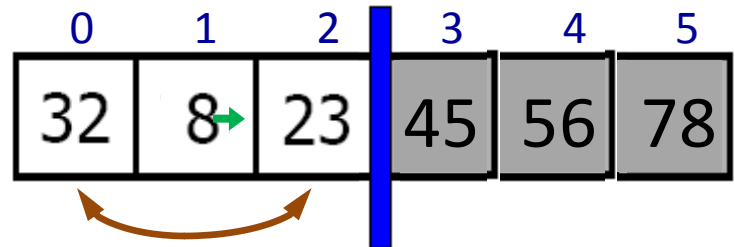
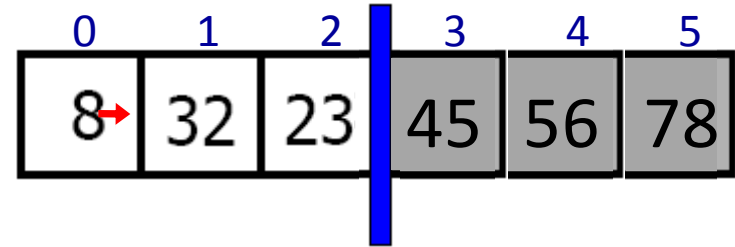
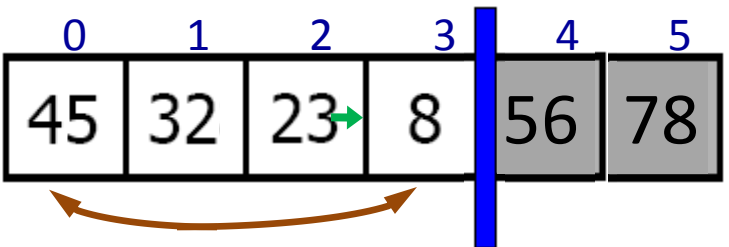
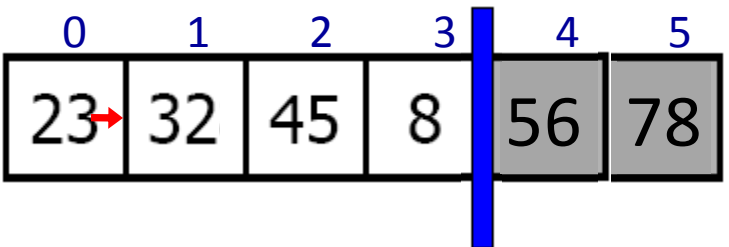
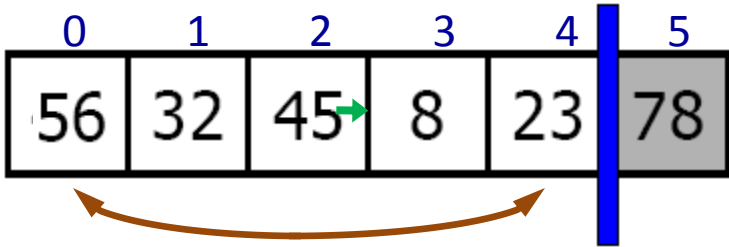
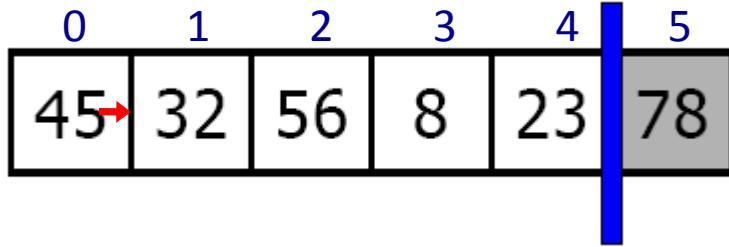
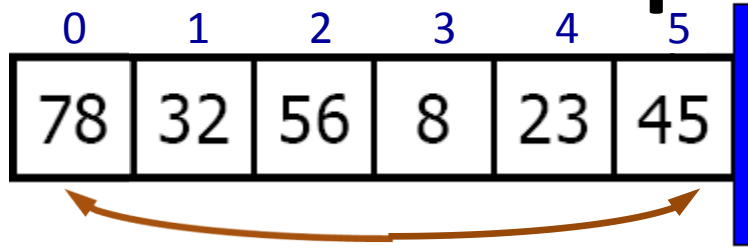
- The unsorted sublist is organized into a **heap**.
- In each pass, in the unsorted sublist, the largest element is **selected** and **exchanged** with the last element.  
Then the heap is **reheaped**.



# Build Heap (first stage)



# Heap Sort (second stage)



# Heap Sort

Algorithm **HeapSort** ()

Sorts the contiguous list using heap sort.

**Post** sorted list.

**Uses** Recursive function **ReheapDown**.

1. position = **count** / 2 - 1 *// Build Heap*

2. **loop** (position >= 0)

1. **ReheapDown**(position, **count**-1)

2. position = position - 1

3. last = **count** - 1 *// second stage of heapsort*

4. **loop** (last > 0)

1. swap(0, last)

2. last = last - 1

3. **ReheapDown**(0, last - 1)

End HeapSort

# Selection Sort Efficiency

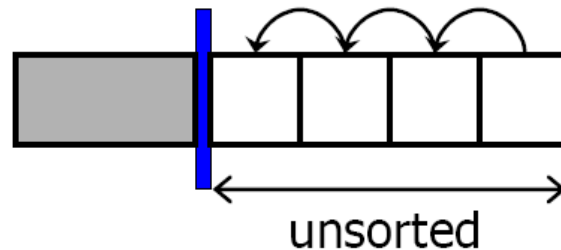
- Straight selection sort:  $O(n^2)$
- Heap sort:  $O(n \log_2 n)$

# Exchange Sort

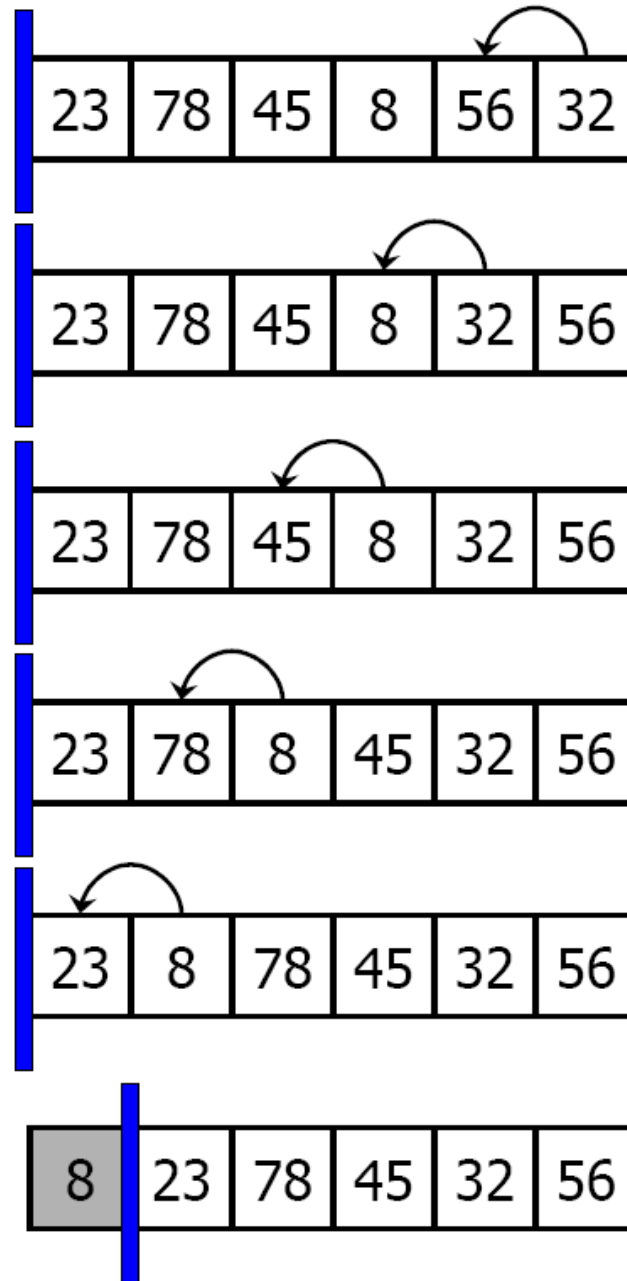
- In each pass, elements that are out of order are **exchanged**, until the entire list is sorted.
- **Exchange** is extensively used.

# Bubble Sort

- The list is divided into two parts: **sorted** and **unsorted**.
- In each pass, the smallest element is **bubbled** from the unsorted sublist and moved to the sorted sublist.

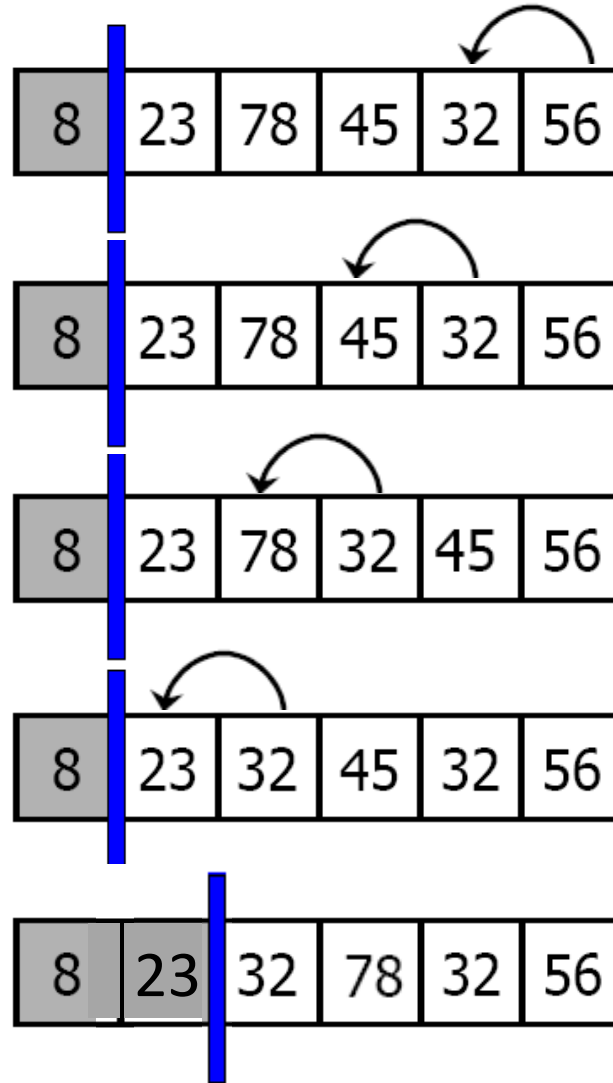


# Bubble Sort





# Bubble Sort



# Bubble Sort

Algorithm **BubbleSort** ()

Sorts the contiguous list using straight bubble sort

Post sorted list.

1. current = 0
2. flag = FALSE
3. **loop** (current < count) AND (flag = FALSE)
  1. walker = count - 1
  2. flag = TRUE
  3. **loop** (walker > current)
    1. if (data [walker].key < data [walker-1].key)
      1. flag = FALSE
      2. swap(walker, walker - 1)
    2. walker = walker - 1
  4. current = current + 1

End BubbleSort

# Exchange Sort efficiency

- Bubble sort:

$$f(n) = n(n + 1)/2 = O(n^2)$$

# Divide-and-conquer sorting

Algorithm **DivideAndConquer()**

1. **if** (the list has length greater than 1)
  1. partition the list into lowlist, highlist
  2. lowlist. **DivideAndConquer()**
  3. highlist. **DivideAndConquer()**
  4. **combine**(lowlist, highlist)

End DivideAndConquer

# Divide-and-conquer sorting

	Partition	Combine
<b>Merge Sort</b>	easily	hard
<b>Quick Sort</b>	hard	easily

# Quick Sort

Algorithm **QuickSort()**

Sorts the contiguous list using quick sort.

**Post** Sorted list.

**Uses** function recursiveQuickSort.

1. recursiveQuickSort(0, count -1)

End QuickSort

# Quick Sort

Algorithm **recursiveQuickSort**(val **low** <int>, val **high** <int>)

Sorts the **contiguous list** using quick sort.

**Pre** low and high are valid positions in contiguous list.

**Post** Sorted list.

**Uses** functions recursiveQuickSort, Partition.

**1.** if (low < high) *// Otherwise, no sorting is needed.*

1. pivot\_position = **Partition**(low, high)

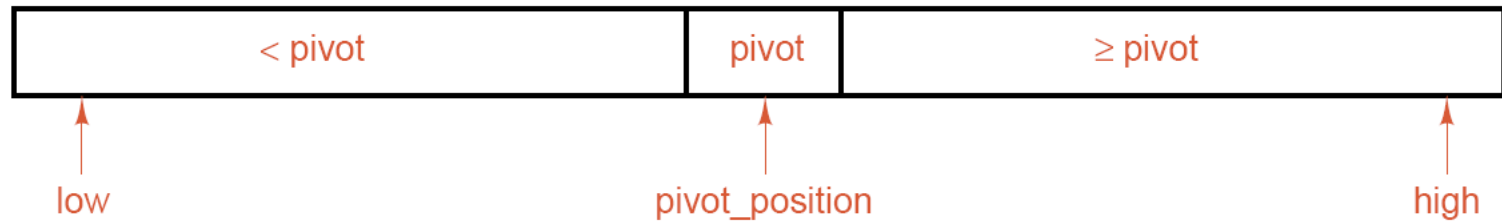
2. **recursiveQuickSort**(low, pivot\_position - 1)

3. **recursiveQuickSort**(pivot\_position + 1, high)

End recursiveQuickSort

# Partition Algorithm

- Given a pivot value, the partition rearranges the entries in the list as below:

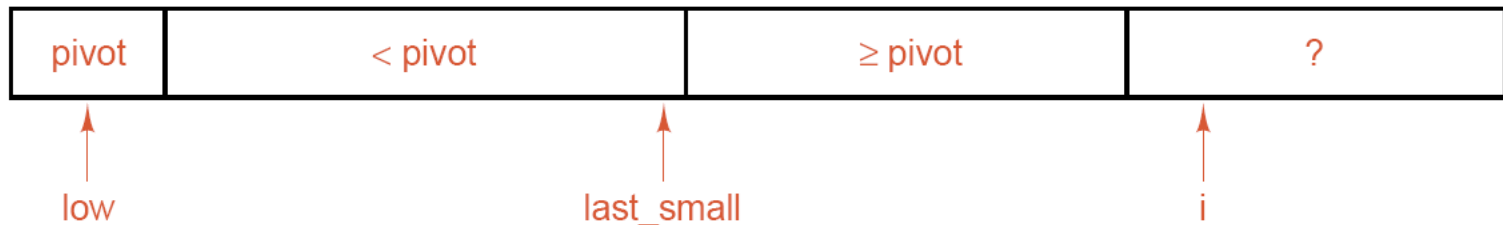




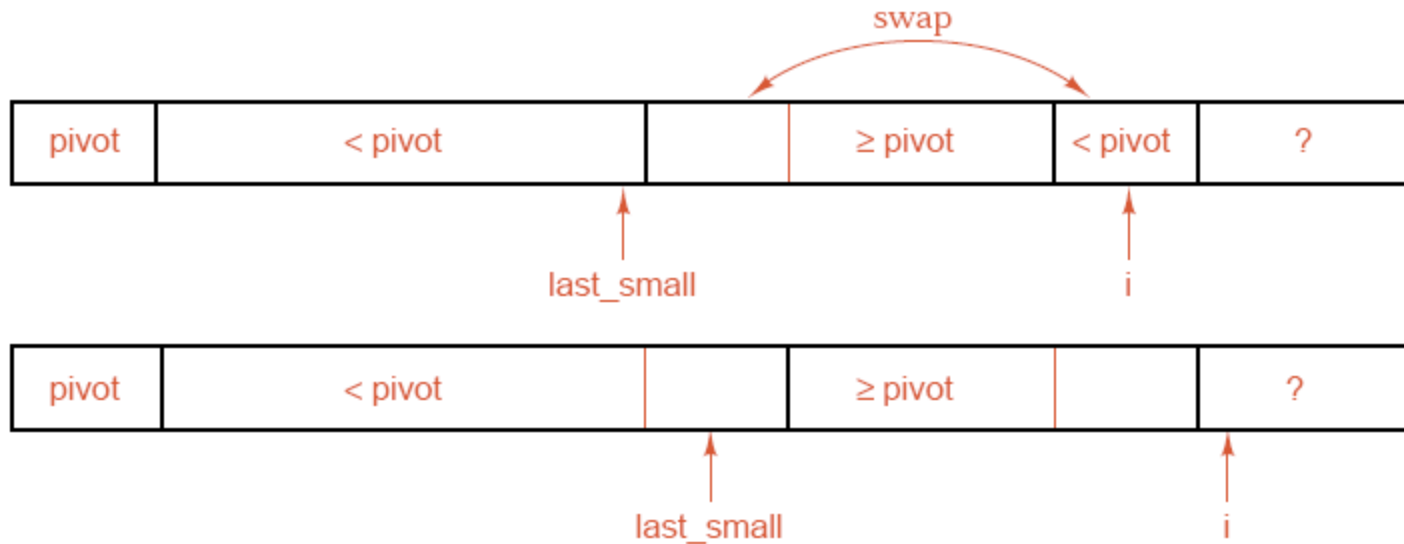
# Partition Algorithm

Algorithm:

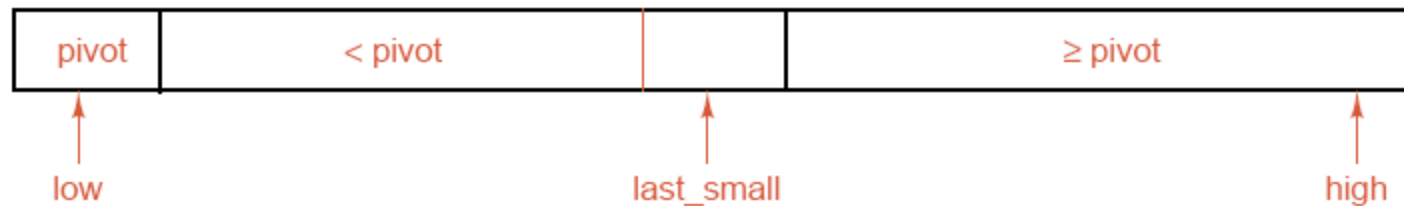
- Temporarily leave the pivot value at the first position.
- use a for loop running on a variable  $i$ ,  $last\_small$  is the position all entries at or before it have keys less than pivot.
- if the entry at  $i \geq pivot$ ,  $i$  can be increased.
- Otherwise,  $last\_small$  is increased and two entries at position  $last\_small$  and  $i$  are swapped:



# Partition Algorithm



- When the loop terminates:



- At last, swap the pivot from position low to position last\_small.

# Partition in Quick Sort

<integer> **Partition**(val **low** <integer>, val **high** <integer>)

Partitions the entries between indices **low** and **high** to two sublists.

**Pre**      **low** and **high** are valid positions in contiguous list, with **low**≤**high**.

**Post**     The center entry in the range between indices **low** and **high** of the list has been chosen as a pivot.

All entries of the list between indices **low** and **high**, inclusive, have been rearranged so that those with keys less than the pivot come before the pivot , and the remaining entries come after the pivot. The final position of the pivot is returned.

**Uses**     Function swap(val i <integer>, val j <integer>) interchanges entries in positions i and j.

<integer> **Partition**(val **low** <integer>, val **high** <integer>)

*// i is used to scan through the list.*

*// last\_small is the position of the last key less than pivot*

1. swap (**low**, (**low**+**high**)/2) *// First entry is now pivot.*

2. pivot = entry<sub>low</sub>

3. last\_small = **low**

4. i = **low** + 1

5. **loop** (i <= **high**)

*//entry<sub>j</sub>.key < pivot, when low < j <= last\_small*

*// entry<sub>j</sub>.key >= pivot, when last\_small < j < i*

1. **if** (data<sub>i</sub> < pivot)

1. last\_small = last\_small + 1

2. swap(last\_small, i) *// Move large entry to right and small to left.*

6. swap(**low**, last\_small) *// Put the pivot into its proper position.*

7. return last\_small

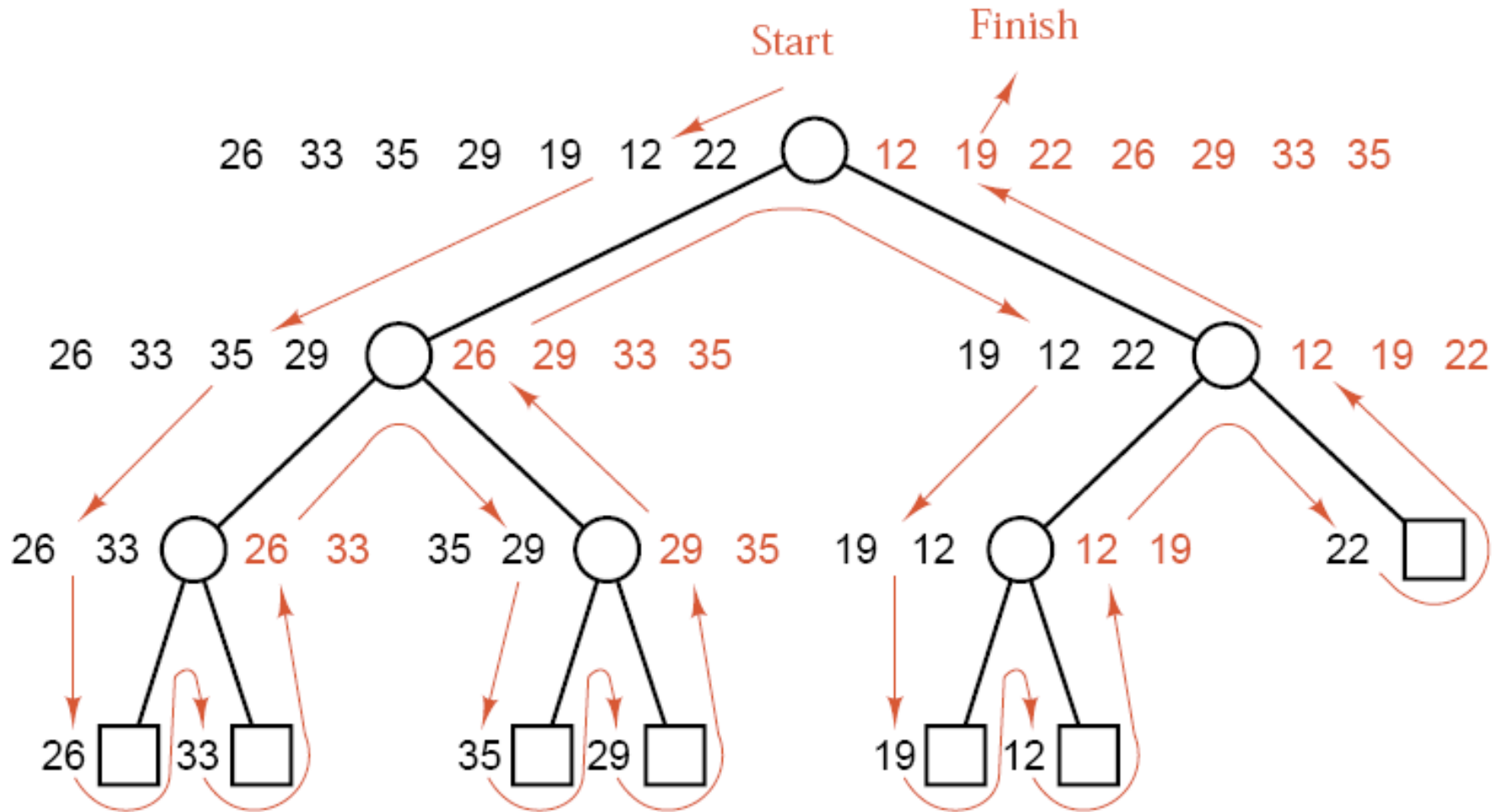
End Partition

# Quick Sort Efficiency

- Quick sort:

$O(n \log_2 n)$

# Merge Sort



# Merge Sort

Algorithm **MergeSort()** *// for linked list*

Sorts the linked list using merge sort

**Post** sorted list.

**Uses** recursiveMergeSort.

1. recursiveMergeSort(head)

End MergeSort

# Merge Sort

Algorithm **recursiveMergeSort**(ref **sublist** <pointer>)

Sorts the **linked list** using recursive merge sort.

**Post** The nodes referenced by **sublist** have been rearranged so that their keys are sorted into nondecreasing order.

The pointer parameter **sublist** is reset to point at the node containing the smallest key.

**Uses** functions recursiveMergeSort, Divide, Merge.

1. **if** (**sublist** is not NULL) AND (**sublist**->link is not NULL)
  1. **Divide**(**sublist**, second\_list)
  2. **recursiveMergeSort**(**sublist**)
  3. **recursiveMergeSort**(secondlist)
  4. **Merge**(**sublist**, secondlist)

End recursiveMergeSort



# Merge Sort

Algorithm **Divide**(val **sublist** <pointer>, ref **secondlist** <pointer>)

Divides the list into two halves.

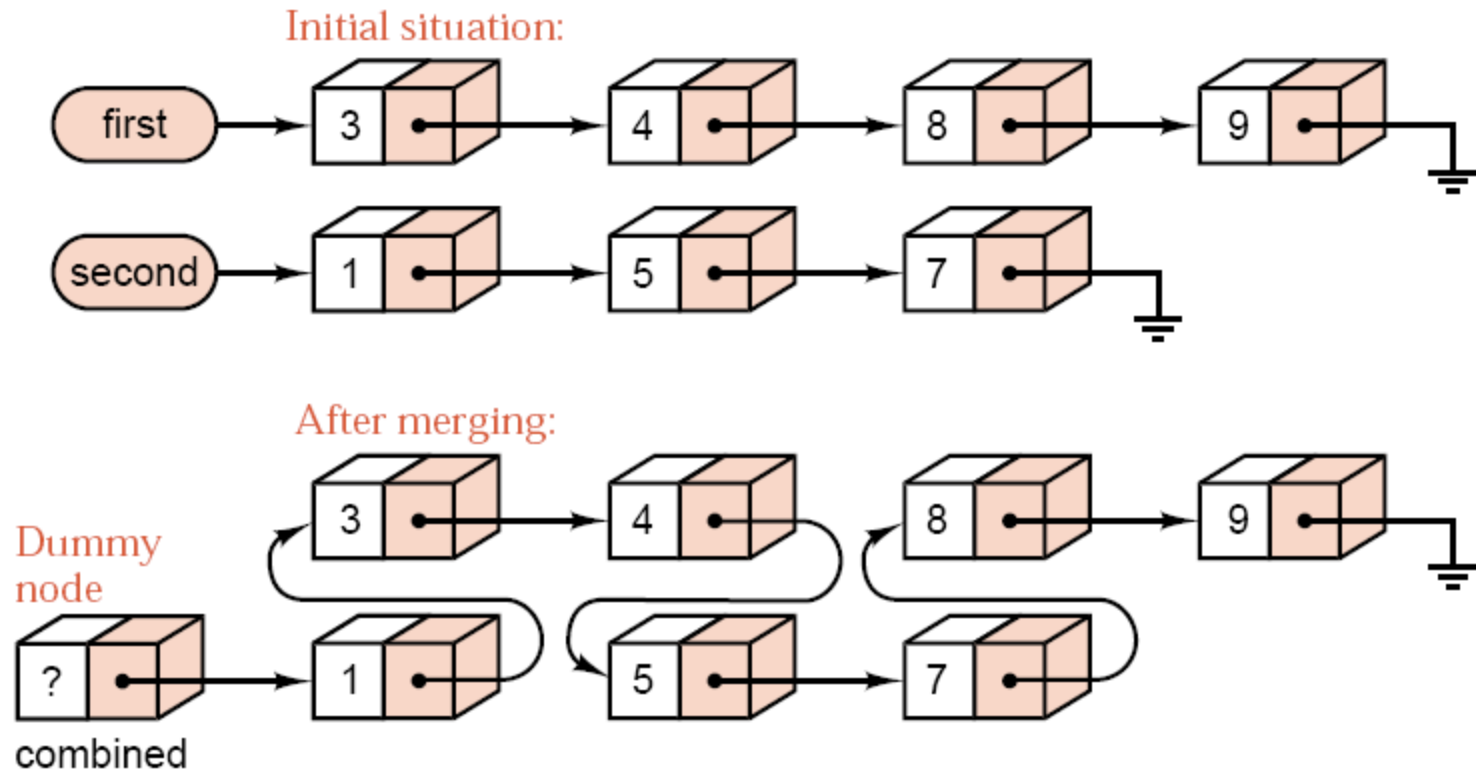
**Pre** **sublist** is not NULL.

**Post** The list of nodes referenced by **sublist** has been reduced to its first half, and **secondlist** points to the second half of the sublist. If the sublist has an odd number of entries, then its first half will be one entry larger than its second.

1. midpoint = sublist
2. position = **sublist**->link    *// Traverse the entire list*
3. **loop** (position is not NULL)    *// Move position twice for midpoint's one move.*
  1. position = position->link
  2. if (position is not NULL)
    1. midpoint = midpoint->link
    2. position = position->link
4. **secondlist** = midpoint->link
5. midpoint->link = NULL

End Divide

# Merge two sublists



# Merge two sublists

Algorithm **Merge** (ref **first** <pointer>, ref **second** <pointer>)

Merges two sorted lists to a sorted list.

**Pre** **first** and **second** point to ordered lists of nodes.

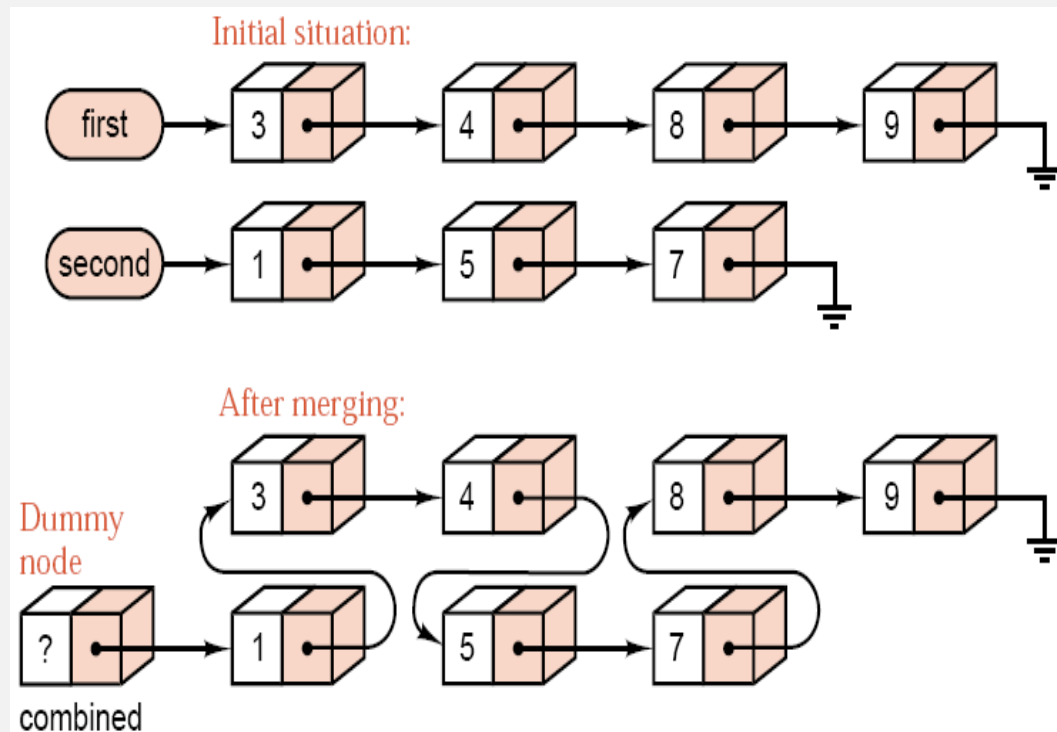
**Post** **first** points to an ordered list containing all nodes that were referenced by **first** and **second**. **Second** became NULL.

Algorithm **Merge** (ref **first** <pointer>, ref **second** <pointer>)

*// lastSorted is a pointer points to the last node of sorted list.*

*// combined is a dummy first node, points to merged list.*

1. lastSorted = address of combined
2. **loop** (**first** is not NULL) AND (**second** is not NULL) *// Attach node with smaller key*
  1. **if** (**first**->data.key <= **second**->data.key)
    1. lastSorted->link = **first**
    2. lastSorted = **first**
    3. **first** = **first**->link *// Advance to the next unmerged node*
  2. **else**
    1. lastSorted->link = **second**
    2. lastSorted = **second**
    3. **second** = **second**->link
3. **if** (**first** is NULL)
  1. lastSorted->link = **second**
  2. **second** = NULL
4. **else**
  1. lastSorted->link = **first**
5. **first** = combined.link



End Merge