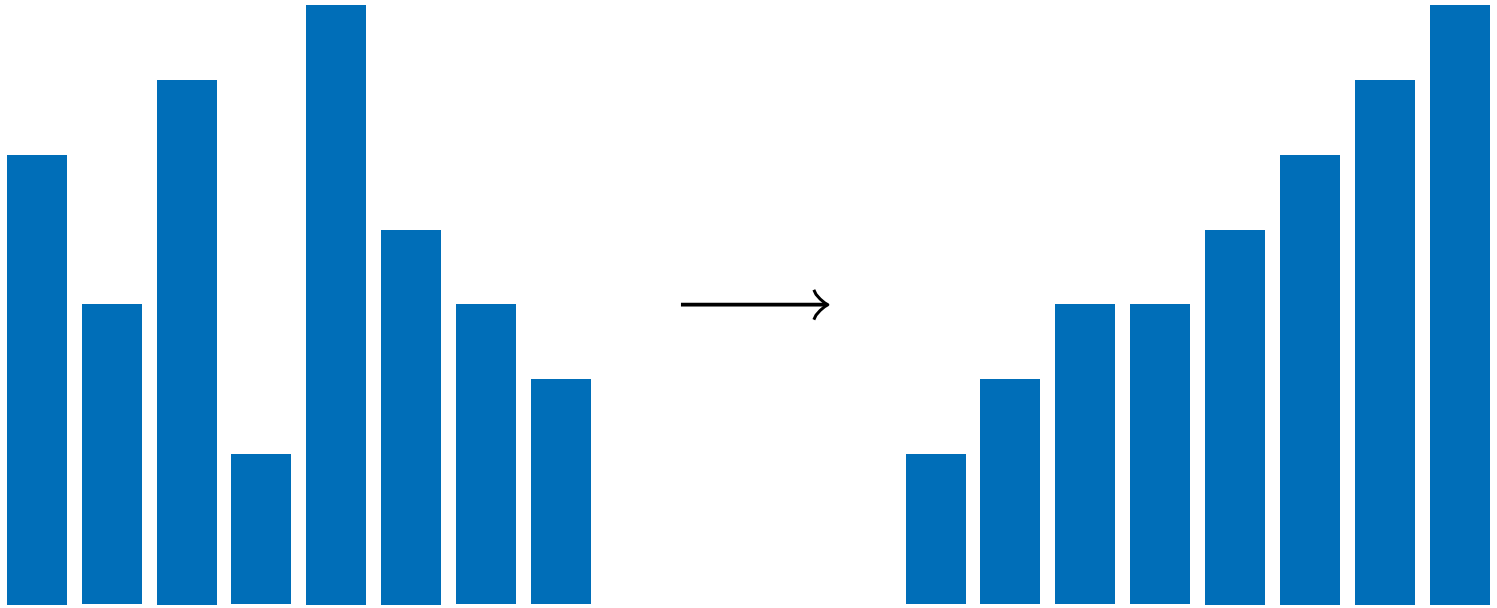
Simple Sorting Algorithms

Lecture 17



Why Sorting?

- Sorting data is an important step of many efficient algorithms

- Sorted data allows for more efficient queries (binary search)

https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/sorting

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://www.toptal.com/developers/sorting-algorithms

Sorting Problem

Input: List A of n elements

Output: Permutation A' of elements in A such that all elements of A' are in non-decreasing order.

For any two indexes i, j: if i < j, then A'[i] <= A'[j]

Recap: Simple Sorting Algorithms

- Insertion sort
- Selection sort
- Bubble sort

Insertion Sort

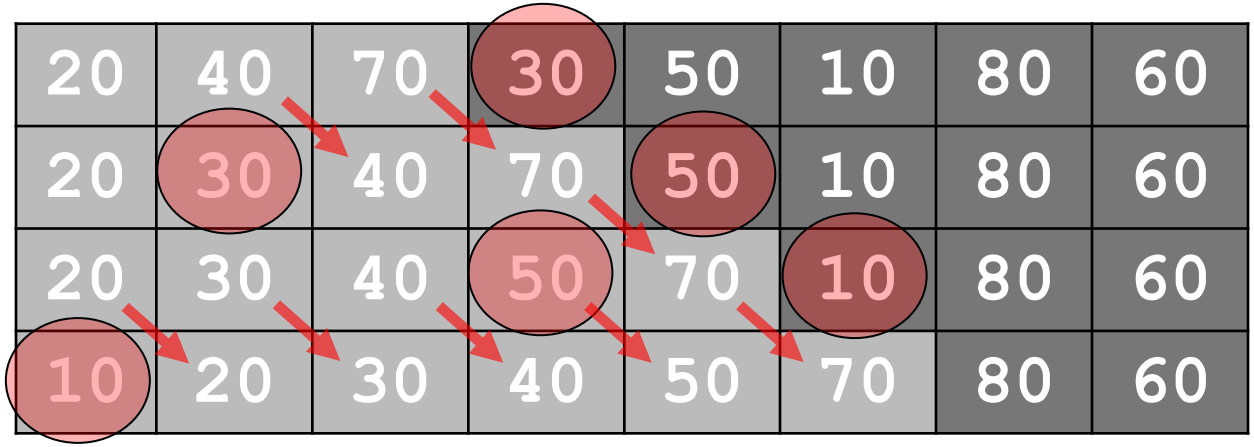
- "Remove" the items one at a time from the original array and "Insert" them into a new array, putting them into the correct relative sorted order as you insert
- We could accomplish this by using two arrays as implied above, but that would double our memory requirements
- We'd rather be able to sort in place:

  Use only a constant amount of extra memory

Insertion Sort: in-place

- We are going to keep 2 logical parts in the array:

- In each iteration, we will take the next item out of the UNSORTED section and put it into its correct relative location in the SORTED section



Insertion sort: code

```
public static void insertionSort(int[] a, int first, int last) {
 int unsortedIndex;  // each element of a becomes the unsorted in turn

 for (unsortedIndex = first + 1; unsortedIndex <= last; unsortedIndex++) {
        // Assertion: a[first] <= a[first + 1] <= ... <= a[unsorted - 1]

        int unsorted = a[unsortedIndex];

        insertInOrder(unsorted, a, first, unsortedIndex - 1);
 } // end for
} // end insertionSort
```

- "Insert" each item in array into its correct spot

```
private static void insertInOrder(int element, int[] a,
                                  int begin, int end) {
 int index;
 // searching for the correct placve for element in the sorted part
 //shifting values to the right
 for (index = end; (index >= begin) && (element < a[index]); index--){
        a[index + 1] = a[index];
 } // end for

 a[index + 1] = element;  // insert
} // end insertInOrder
```

- Find correct spot for current element
- Note that this goes from back to front
- Shift to the right always goes from back to front

Insertion sort: implementation details

- Initial method has only the input array as a param
- This calls an overloaded version with start and end index values as params – allows us to sort only part of the array if we want
- Each iteration in this method brings one more item from the unsorted portion of the array into the sorted portion
- It does this by calling another method to actually move the value into its correct spot
- Values are shifted from left to right, leaving a "hole" in the spot where the item should be

Running time of Insertion Sort

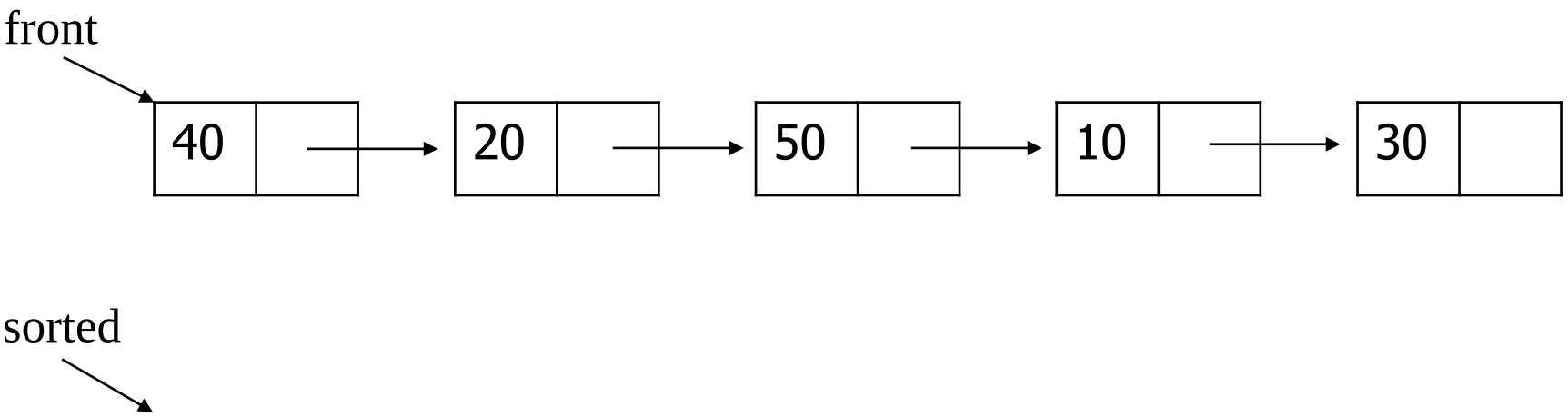
- Key instruction: comparisons of array items
- What is the WORST possible input?

  REVERSE SORTED data – think why?
- Consider each iteration of the insertionSort loop
  - when unsorted = 1, 1 comparison in insertInOrder method
  - when unsorted = 2, 2 comparisons in insertInOrder method
  - …
  - when unsorted = N-1, N-1 comps in insertInOrder method
- Overall we get $1 + 2 + \ldots + N-1 = (N-1)(N)/2 = O(N^2)$
- This is the worst case

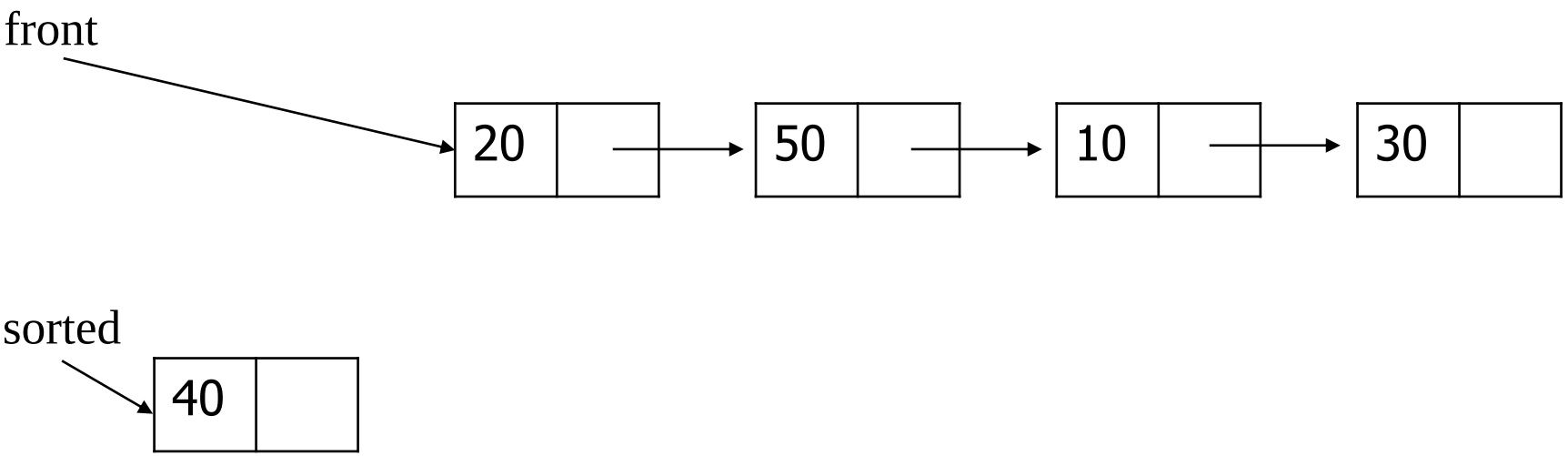
Can input be a Linked List?

- Yes – in fact it is probably more natural with a linked list
- At each iteration simply remove the front node from the input list, and "insert it in order" into a second, new list
- This is still “in-place”: we are not creating ANY new nodes – just moving the ones we have around

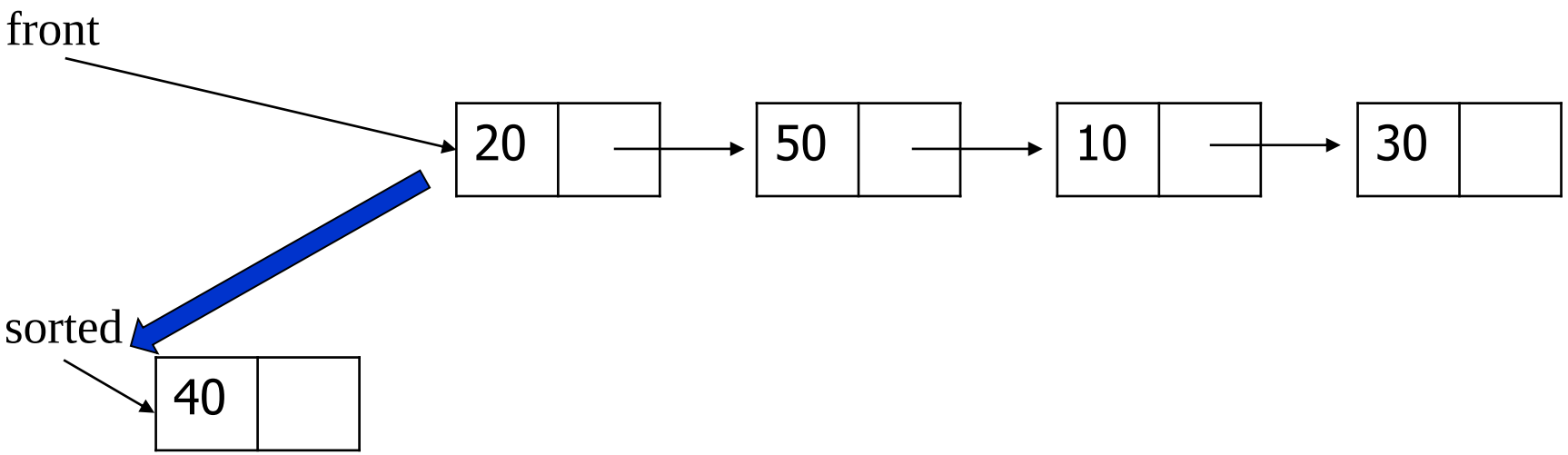
Insertion Sort of a Linked List



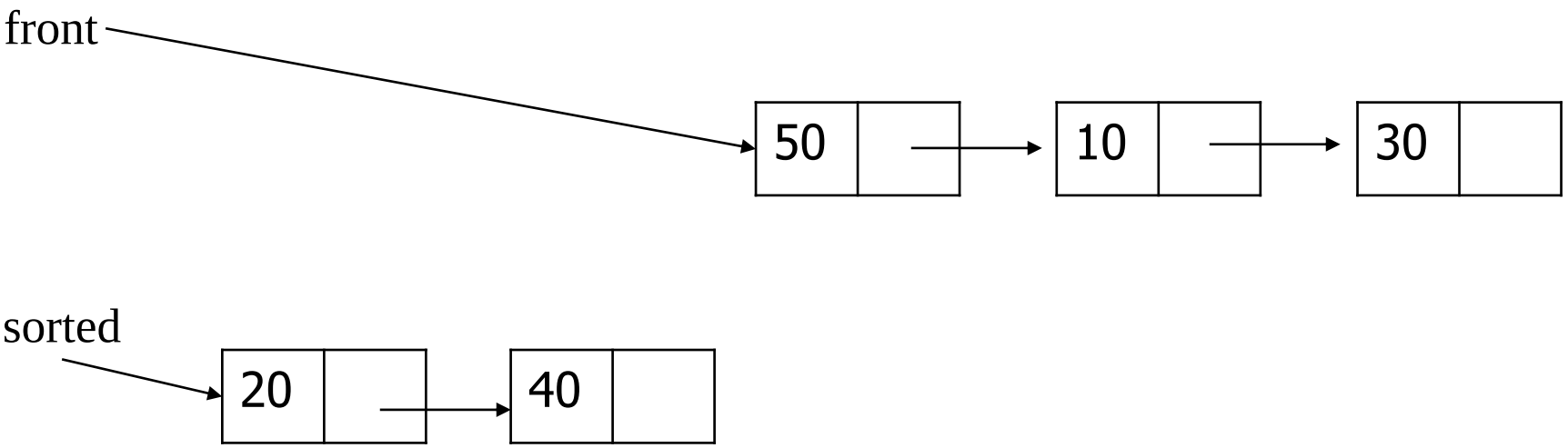
Insertion Sort of a Linked List



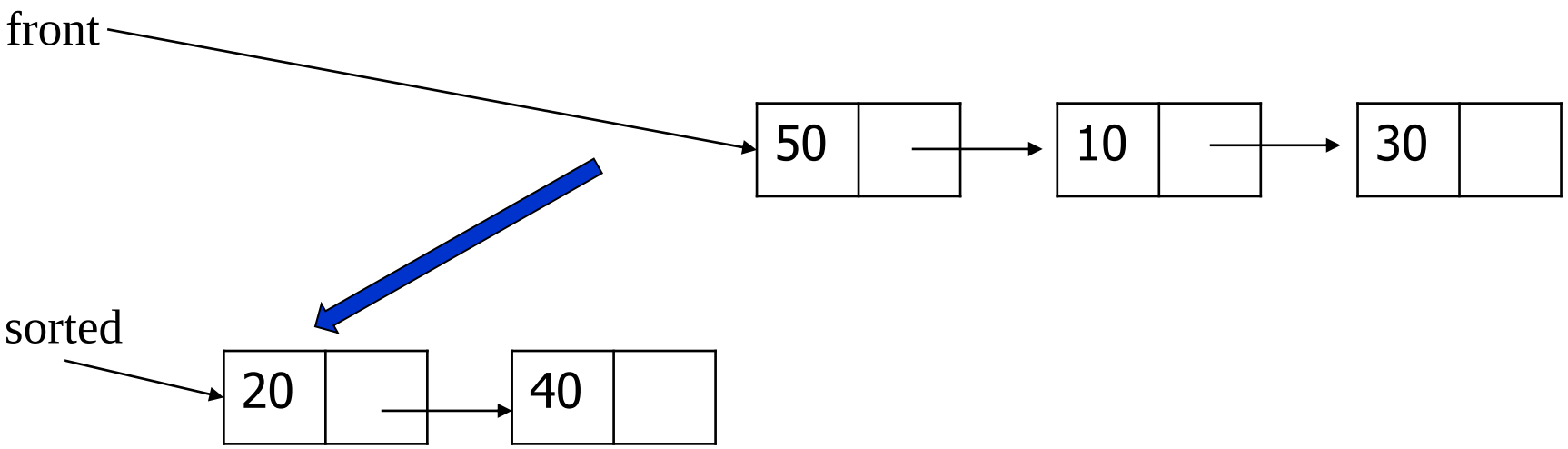
Insertion Sort of a Linked List



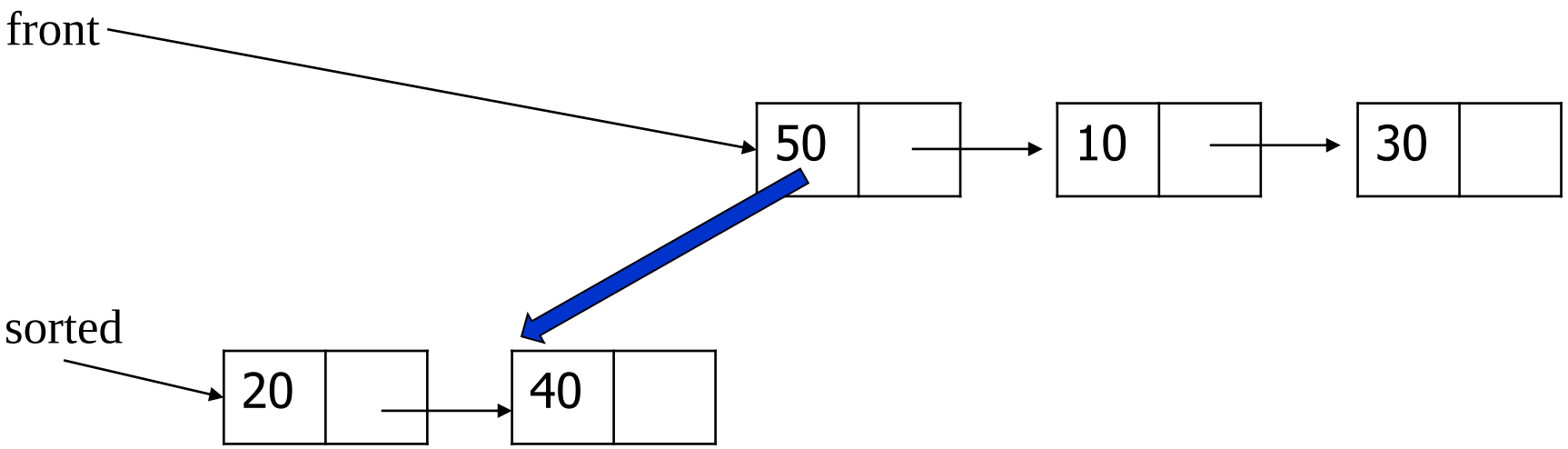
Insertion Sort of a Linked List



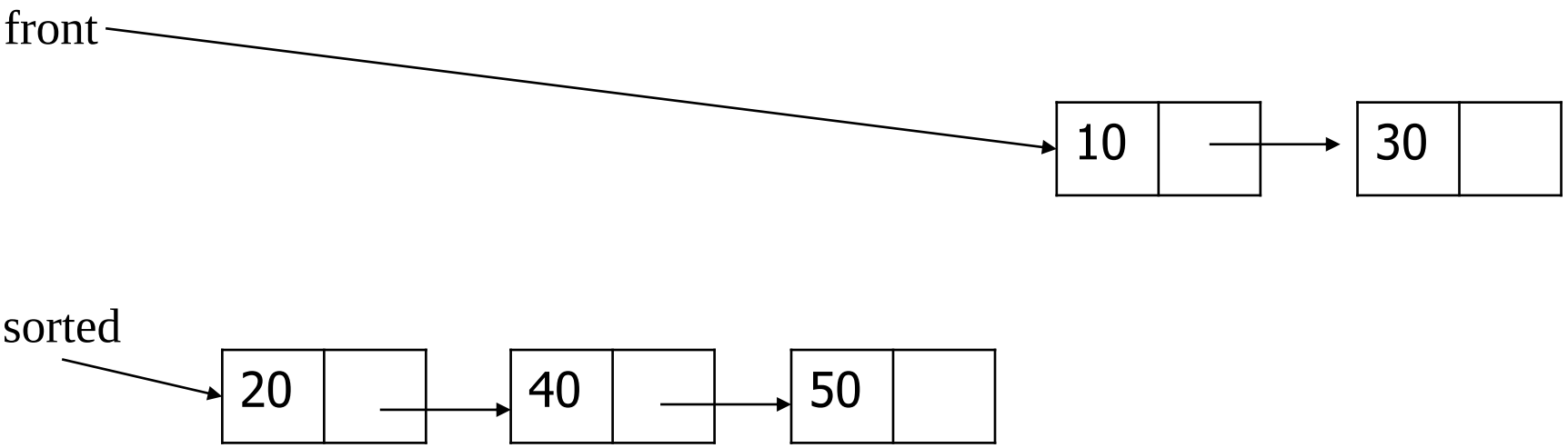
Insertion Sort of a Linked List



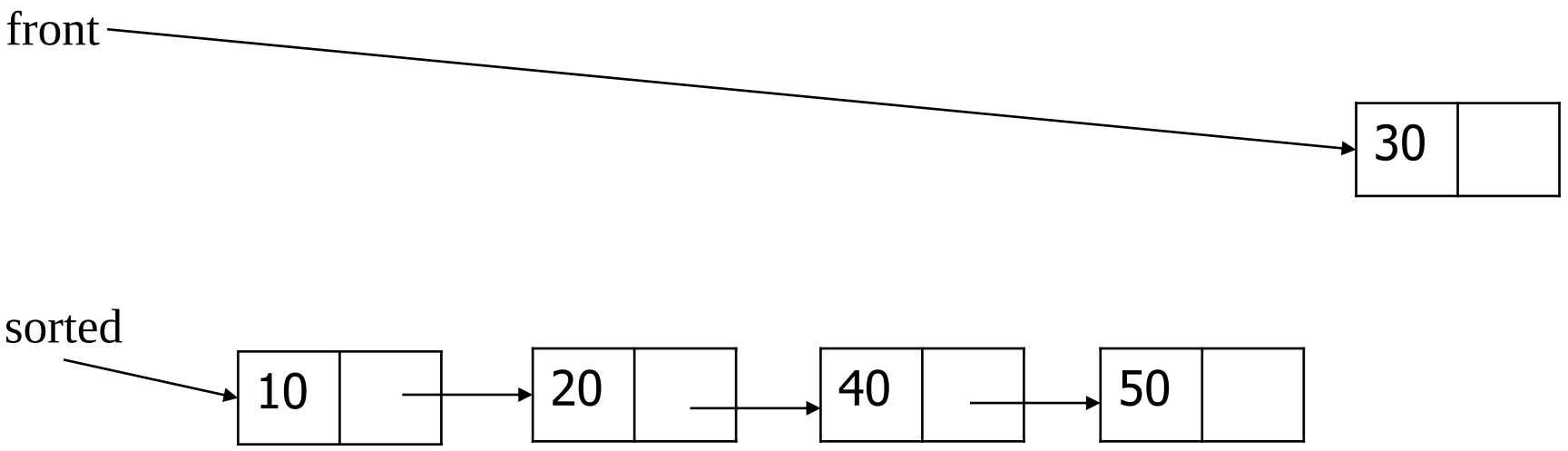
Insertion Sort of a Linked List



Insertion Sort of a Linked List



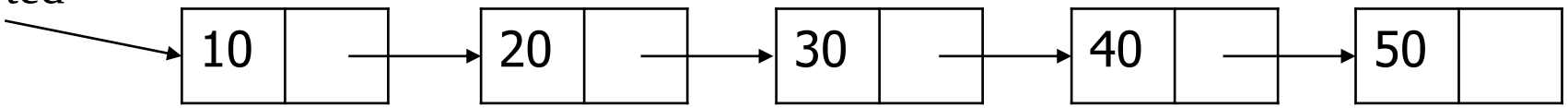
Insertion Sort of a Linked List



Insertion Sort of a Linked List

front

sorted



- Run-time?
- What is the worst case input now?

Insertion Sort of a Linked List

- Each node is removed from the original list and is "inserted" into the second list in the proper location
- We insert from front to back of the sorted list, comparing the data as we go
- In this case the worst case is when the correct position of every new node is at the END of the list
  - This would occur if the data was initially sorted (!)
- The run-time is the same as for the array implementation: $O(N^2)$

Other sorts: Selection Sort

- At iteration *i* of the outer loop, find the *i*-th smallest item and swap it into location *i*
  - *i* = 0 : find 0th smallest and swap into location 0
  - *i* = 1 : find 1th smallest and swap into location 1
  - …
  - *i* = N-2 : find (N-2)-th smallest and swap into loc N-2
- Also implementation is simple using 2 nested for loops (or method calls, as shown in text)
- We looked at the Selection Sort code when we learned about generic sorting: LINK

Running time: $O(N^2)$

Other sorts: Bubble Sort

For k from N-1 to 2 :

    For j from 0 to k-1

        Item j is compared to item j+1

        If item j is greater than item j+1, they are out of order

            In this case we swap them

    By the end of iteration k the largest item is “Bubbled up” into position k.

Early stopping: when no more swaps

Running time: $O(N^2)$

Bubble Sort example

One iteration:

0	1	2	3	4	5	6
50	**30**	40	70	10	80	20
30	**50**	**40**	70	10	80	20
30	40	**50**	**70**	10	80	20
30	40	50	**70**	**10**	80	20
30	40	50	10	**70**	**80**	20
30	40	50	10	70	**80**	**20**
30	40	50	10	70	20	**80**

Total: N iterations

Running time: $O(N^2)$

Recursive Implementations

- Textbook also discusses recursive implementations of *InsertionSort* and *SelectionSort*
- As with Sequential Search, this is more to demonstrate an idea rather than something that we would actually do
- These recursive versions are not divide and conquer, there is no efficiency or implementation motivation to doing them recursively
- Read over these explanations and convince yourselves that the recursive versions do the same thing as the iterative versions and have the same $O(N^2)$ running time

Comparing runtime in practice

- Note diff between *InsertionSort* and *SelectionSort*:
- Both have two nested "for" loops

- In *InsertionSort*, "inner" for loop will stop when the item is at the correct insertion point
  - Running time is very different for different inputs
  - Best-case O(N): for an almost sorted input

- In *SelectionSort* we don't have an early stopping condition:
  - Thus it doesn't matter how the data is initially organized
  - There is no "worst case" or "best case“: all cases iterate the same number of times and do the same number of comparisons: $O(N^2)$

Idea for improving Insertion Sort

- What makes its performance poor?
- Consider what happens within each iteration:
  - **Either nothing** (if items are already in order)
  - Or the **value moves of 1 location** (it only moves several positions)
  - If the element is greatly out of order, it will take a lot of adjacent comparisons to find its place in the sorted part
- If we can swap the current item farther than by one position, perhaps we can improve the performance

Shell sort

- Rather than comparing adjacent items, we compare items that are farther away from each other
- Specifically, we compare and "sort" items that are *K* locations apart for some *K*
  - i.e. we *InsertionSort* subarrays of our original array that are K locations apart
- We gradually reduce K from a large value to a small one, ending with *K* = 1
  - Note that when *K* = 1 the algorithm is straight *InsertionSort*

Shell sort: example

40	20	70	60	50	10	80	30
40	10	70	30	50	20	80	60

K = 4

40	10	70	30	50	20	80	60
40	10	50	20	70	30	80	60

K = 2

40	10	50	20	70	30	80	60
10	20	30	40	50	60	70	80

K = 1

The idea is that by the time K = 1, most of the data will be almost sorted and will not have to move far

Shell sort: code 1/2

```
public static void shellSort(int[] a) {
        int n = a.length;
        int first = 0;
        int last = n-1;

        int gap = n / 2;// initial gap is n/2
        if (gap % 2 == 0)
                gap ++;

        // Continue until the gap is zero
        while (gap > 0) {
        // for each position inside the gap
        //do insertion sort on the corresponding subarray
                for (int begin = first; begin < first + gap; begin++){
                        insertionSortSubarray(a, begin, last, gap);
                }

                gap = gap / 2;// reduce gap
                if (gap > 0 && gap % 2 == 0)
                        gap ++;
        } // end while
} // end shellSort
```

Shell sort: code 2/2

```
private static void insertionSortSubarray (int [] a,
                                           int start, int end, int gap) {
        int unsortedIndex, index;
        // go through all elements in the subarray
        //starting from the second
        for (unsortedIndex = start + gap; unsortedIndex <= end;
                                  unsortedIndex=unsortedIndex+gap){
                int unsorted = a[unsortedIndex];
                index = unsortedIndex - gap;
                while ((index >= start) && unsorted < a[index]){
                        a[index + gap] = a[index];
                        index = index - gap;
                } // end while

                a[index + gap] = unsorted;
        } // end for
}
```

ShellSort: performance

- It seems like this algorithm will actually be worse than InsertionSort :
  - Its last "iteration" is a full InsertionSort
  - Previous iterations do InsertionSorts of subarrays with the total of N elements
- Yet when timed it actually outperforms InsertionSort
- Recall that the original *InsertionSort* actually **has a (O(N)) performance in the best case** – ShellSort moves the data toward this best case

ShellSort: big O

- We move the data less time and when we reach the last step the data is mostly sorted and does not require many steps to find the final position of an item
- Precise analysis is tricky, and depends on the values for K (initial value and how it is updated)
- If you always use K to be odd, it can be shown that *ShellSort* has time complexity **$O(N^{3/2})$** [Compare to N^2 for large N]

- See textbook for more details and Wikipedia

- Simple sorting algorithms:
  - InsertionSort – $O(N^2)$
  - SelectionSort – $O(N^2)$
  - BubbleSort – $O(N^2)$
  - ShellSort – $O(N^{1.5})$

- For a small number of items, their simplicity makes them ok to use
- But for a large number of items, this is not a good run-time

Can we do better?

Yes, using divide-and-conquer