

Sorting (chapter 7)

- Properties of Sorting Algorithms
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Radix Sort

Definition of Sorting

- A formal definition of sorting: For n records, each having a key K , the task of sorting is to find a permutation σ of these records such that

$$K_{\sigma(i)} \leq K_{\sigma(i+1)} \text{ for all } i < n$$

- If there are duplicate keys among the records, such permutations are not unique.
- A sorting algorithm is **stable** if it always generates permutations that preserve the original ordering of records of the same keys.
- **Internal sorting** vs. **external sorting**: Do we have huge amount of data to sort?

Insertion Sort

- Idea: In the k^{th} iteration, keep the first k items sorted.
 - Take the k^{th} item and insert it into the (already sorted) partial list of items $1 \sim k-1$ at the location that keeps the first k items sorted.
- Complexity: $O(n^2)$.
- Array implementation: Requires lots of item movements.
- Linked-list implementation: Needs time to determine the location to insert an item at.

Examples:

5	4	3	2	1
4	5	3	2	1
3	4	5	2	1
2	3	4	5	1
1	2	3	4	5

2	3	4	5	1
2	3	4	5	1
2	3	4	5	1
2	3	4	5	1
1	2	3	4	5

Quick Sort

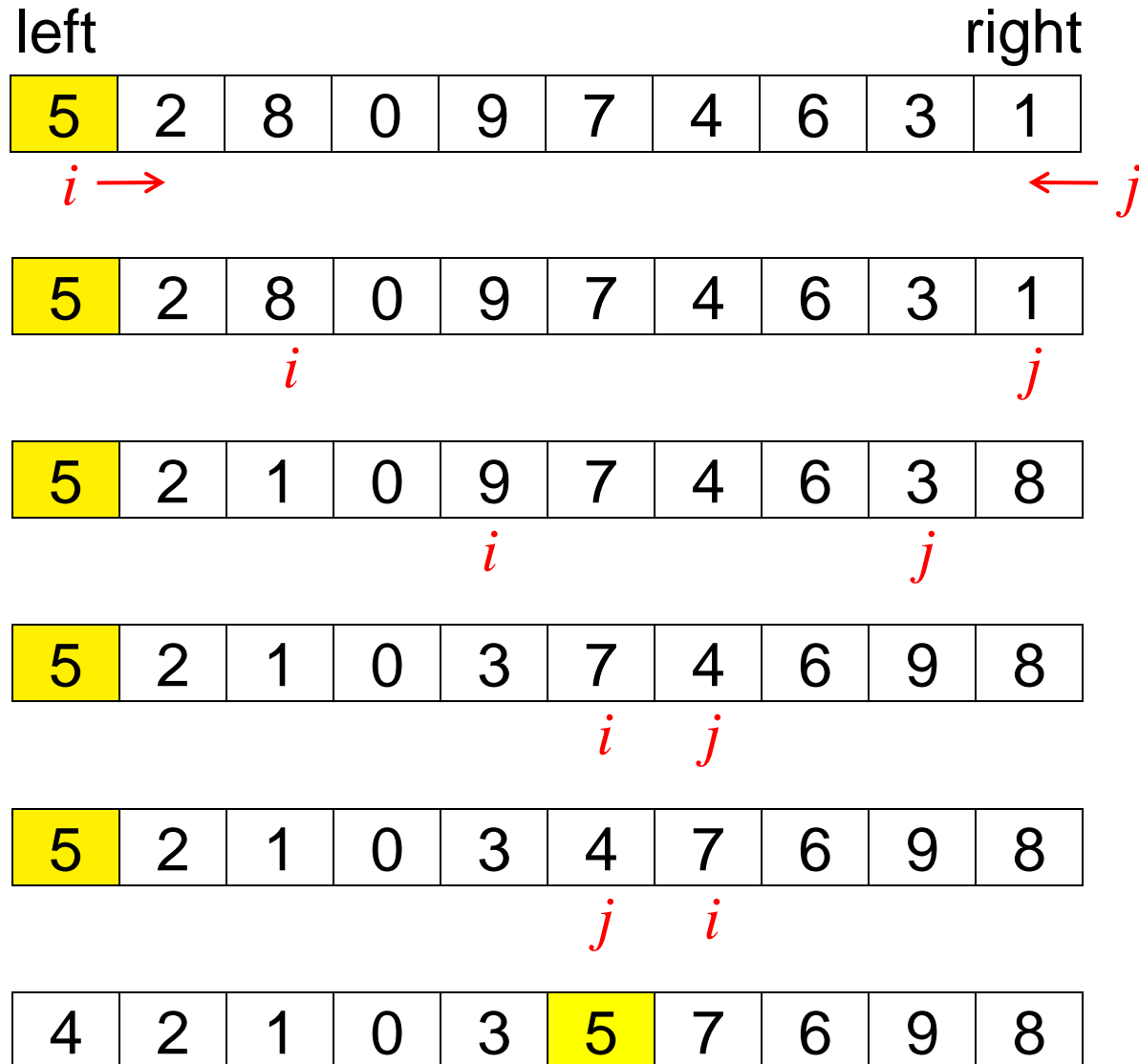
■ Idea:

- Take an item (usually the first item) as the **pivot**.
- (Partition step)
 - ◆ Find the first item from the left that is larger than the pivot and the first item from the right that is smaller than the pivot; swap the two items. Repeat until the whole list has been compared with the pivot.
 - ◆ Swap the pivot with an item so that all the items before pivot are no larger than the pivot, and all the items after the pivot are no smaller than the pivot.
- Recursively sort the partial lists before and after the pivot.
- Time complexity: $O(n^2)$ worst case, $O(n \log n)$ average case.
- Is not stable.

Quick Sort

```
template <class T>
void QuickSort(T *a, int left, int right)
{
    if (left < right) {
        int i=left, j=right+1;
        T pivot = a[left];
        // partition
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if (i < j) swap(a[i], a[j]);
        } while (i < j);
        swap (a[left], a[j]);
        // recursively sort the partial lists
        QuickSort(a, left, j - 1);
        QuickSort(a, j + 1, right);
    }
}
```

Quick Sort (Partition Example)



Quick Sort Example

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

Merge Sort

- Each step involves merging two lists that are already sorted.
 - Keep a pointer to the "current item" in each list, initialized to the first item.
 - In each iteration, copy the one with the smaller key among the two current items to the merged list. Shift its pointer.
 - When one of the list is done, copy the remaining items in the other list to the merged list.

Merging Two Lists (Example)

List 1

List 2

Merged List

1 5 26 77	11 15 59 61	
1 5 26 77	11 15 59 61	1
1 5 26 77	11 15 59 61	1 5
1 5 26 77	11 15 59 61	1 5 11
1 5 26 77	11 15 59 61	1 5 11 15
1 5 26 77	11 15 59 61	1 5 11 15 26
1 5 26 77	11 15 59 61	1 5 11 15 26 59
1 5 26 77	11 15 59 61	1 5 11 15 26 59 61
1 5 26 77	11 15 59 61	1 5 11 15 26 59 61 77

Recursive Merge Sort

- In each step, divides the list into two sub-lists of approximately equal sizes.
 - (Recursion) Sort the two sub-lists.
 - Merge the two sorted sub-lists.
- Time complexity: $O(n \log n)$.
- Is a stable sorting algorithm.

Recursive Merge Sort (Example)

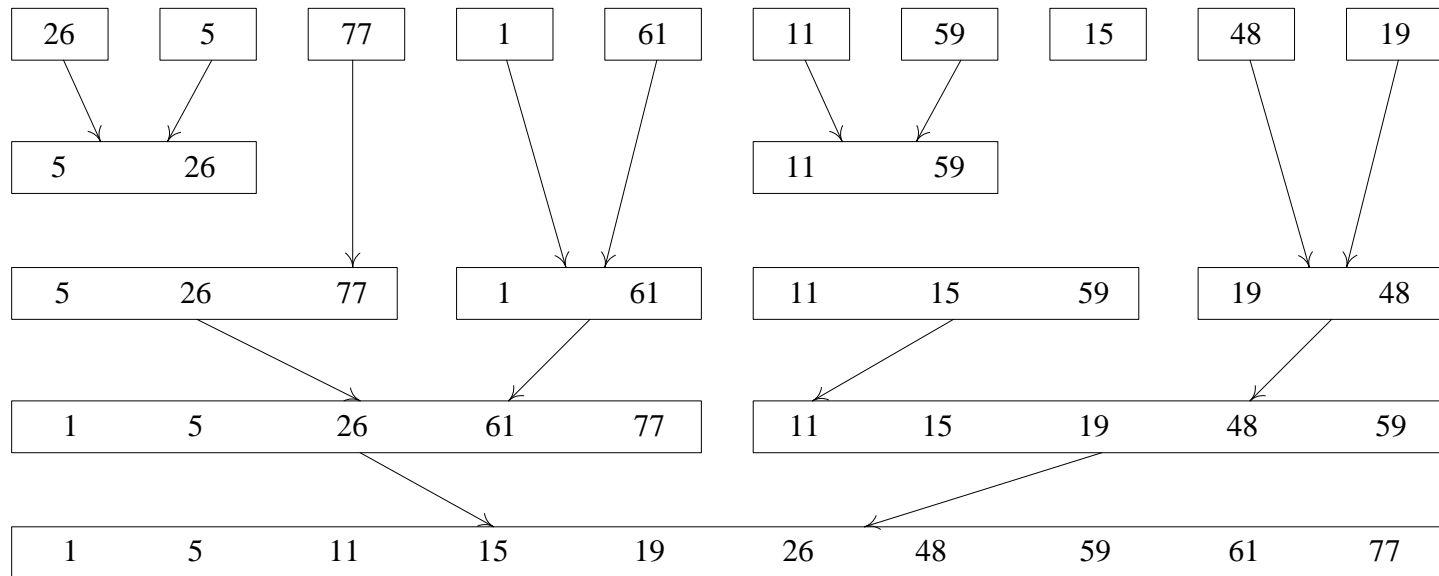
5	2	8	0	9	7	4	6	3	1
5	2	8	0	9	7	4	6	3	1
5	2	8	0	9	7	4	6	3	1
5	2	8	0	9	7	4	6	3	1
5	2	8	0	9	7	4	6	3	1
2	5	8	0	9	4	7	6	3	1
2	5	8	0	9	4	6	7	1	3
0	2	5	8	9	1	3	4	6	7
0	1	2	3	4	5	6	7	8	9

Issues of Merge Sort

- Main disadvantage: Extra space needed: $O(n)$.
 - Merge sort with $O(1)$ extra space is possible, but is more complicated and slower (time complexity $O(n (\log n)^2)$).
- Is not an **in-place** sorting algorithm.
 - Generally speaking, an in-place algorithm means that inputs and outputs share the same space, and only few (such as $O(1)$ or $O(\log n)$) extra spaces are needed.
 - Insertion sort and quick sort are generally considered in-place, although the definition is not strict.

Issues of Merge Sort

- We can do it recursively or iteratively, much like the binary search algorithm.
- The iterative version is a bottom-up approach:

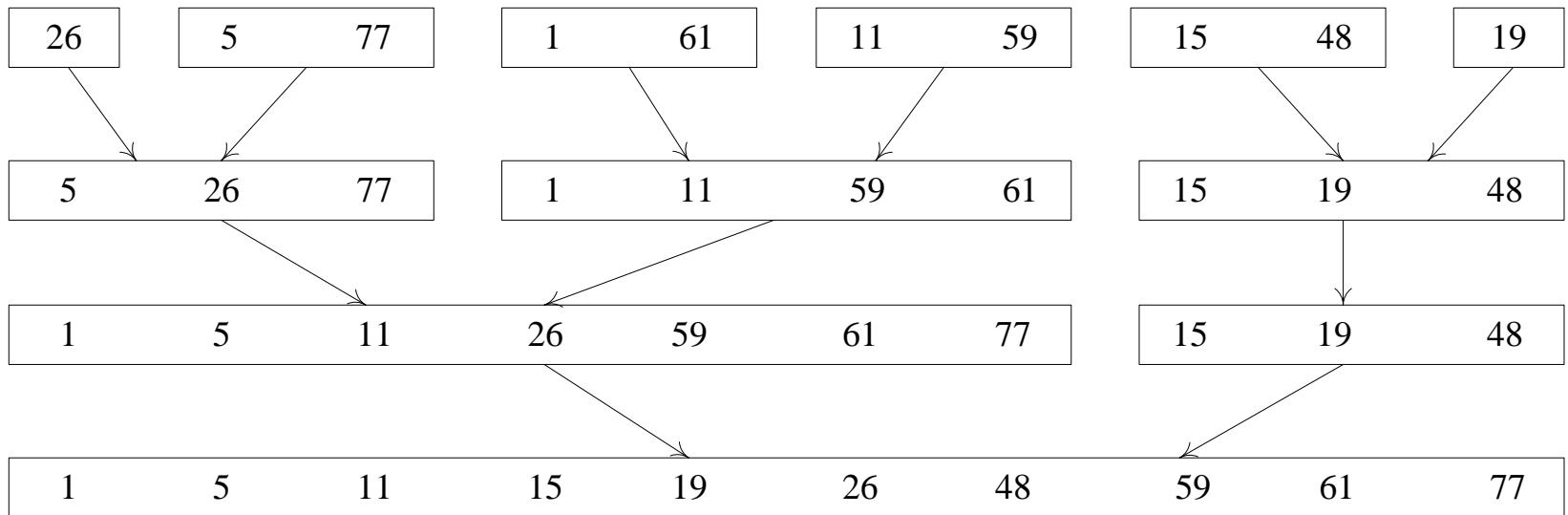


- Can work easily with linked lists. (Sorting linked lists is difficult for other sorting algorithms.)

Q: What is the space complexity when sorting linked lists?

Natural Merge Sort

- Goal: To take advantage of "sorted sublists" in the input.
- In the iterative version, start with the sorted sublists instead of the individual items.




- Best-case time complexity: $O(n)$.

Q: When does this happen?

Heap Sort

- Heap is a natural choice as the data structure for sorting.
 - A fast **selection sort**: Each selection takes $O(\log n)$ only.
- A simple implementation:
 - Initialize an empty heap.
 - Insert every item into the heap.
 - Remove the items one by one; they will be in a sorted order.
- Time complexity: $O(n \log n)$
- Extra space needed: $O(n)$ Q: Can we do better?

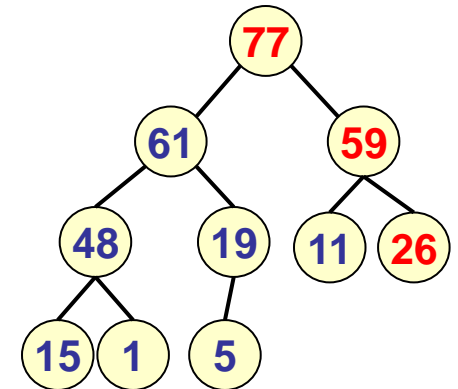
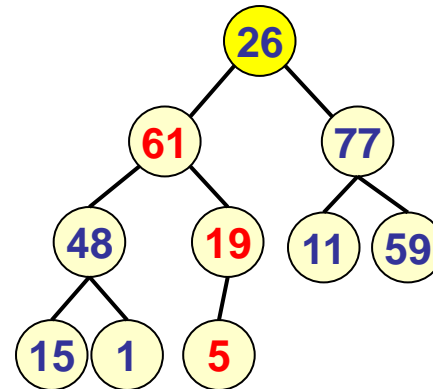
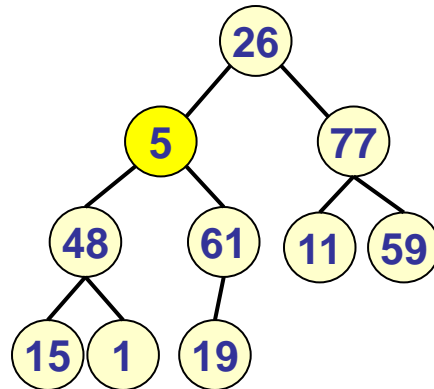
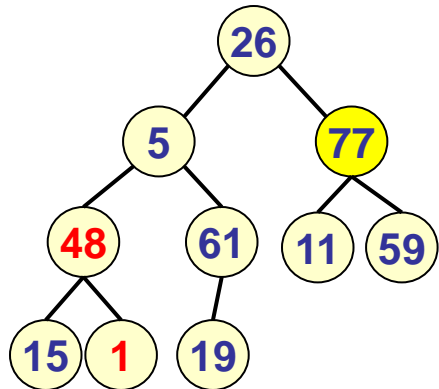
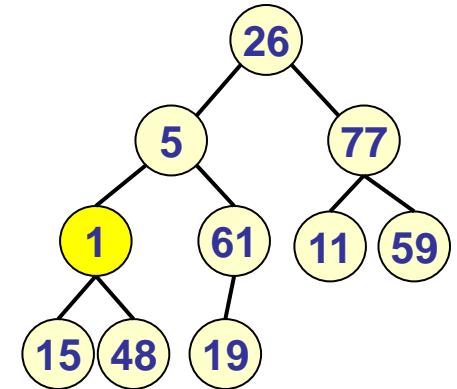
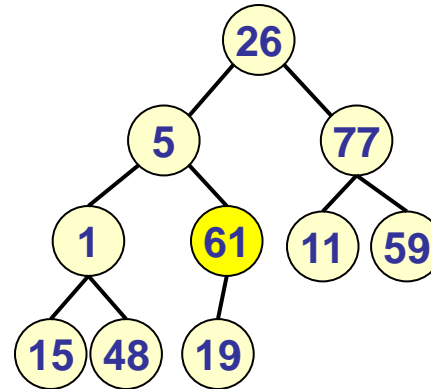
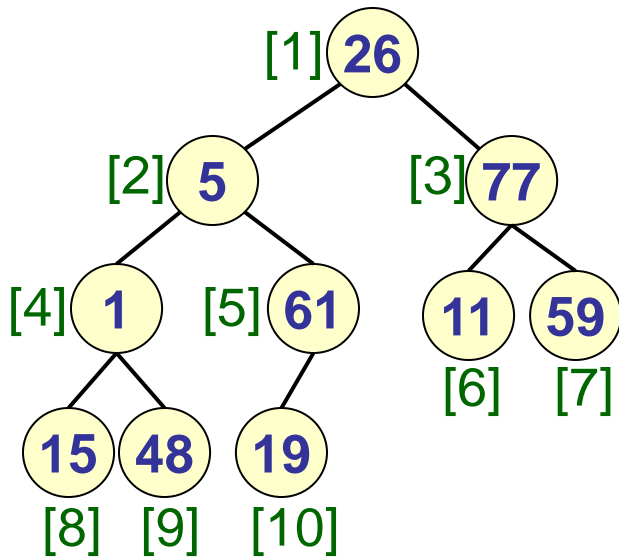
Heap Sort

- Idea: We can make heap sort in-place:
 - Remember that we use the array representation of binary trees for heaps.
 - We can treat our input array as a complete binary tree.
 - "**Heapify**" the binary tree: We move the items around so that the binary tree satisfies the heap property.
 - ◆ Make it a max heap  if we want to sort small-to-large, and vice versa.
 - In each iteration, do a "deletion" operation of the heap.

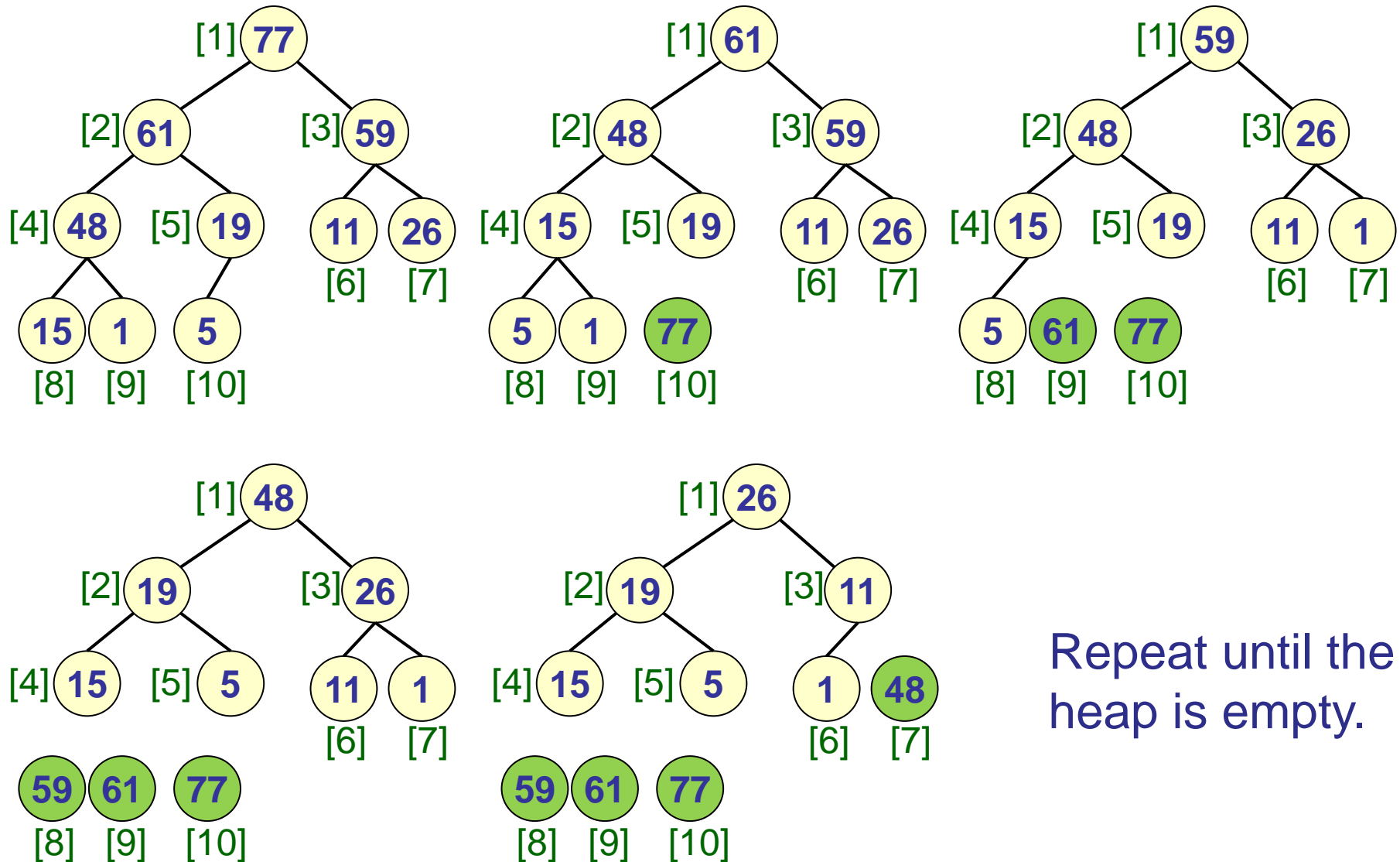
How to "Heapify" a Binary Tree?

- A recursive concept (for max heap):
 - For a non-leaf node **x**:
 - ◆ Assume that the two subtrees of **x** are already max heaps.
 - ◆ Swap **x** with its child with the larger key, if necessary. Repeat until **x** is at a place that satisfies the max heap property.
- With the array representation, this is easier done by scanning each item (as **x**) from the back to the front of the array. (Function *Adjust* in the textbook.)
- Time complexity: $O(n \log n)$.

How to "Heapify" a Binary Tree?



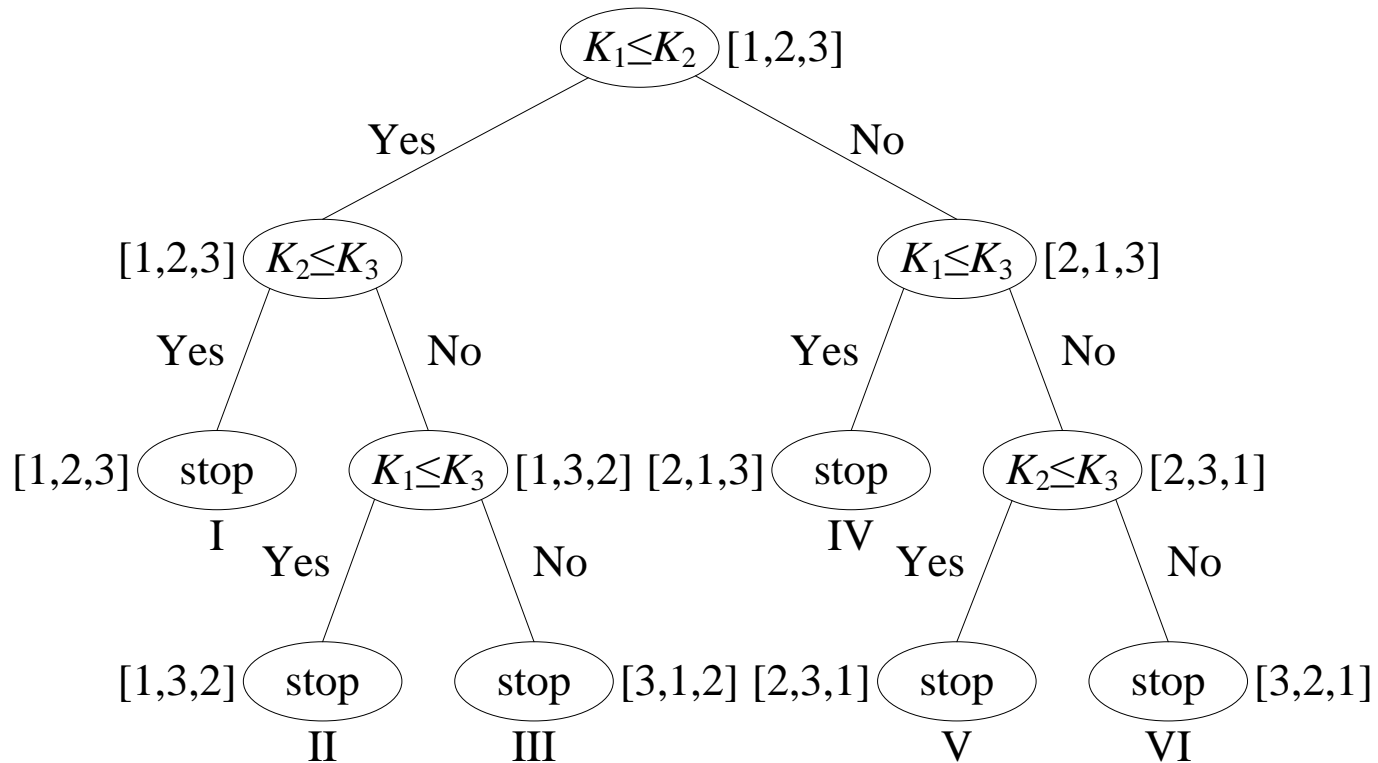
Now, the Heap Sort



Repeat until the heap is empty.

How Fast Can We Sort?

An illustration with decision trees, assuming that we can only compare and swap pairs of keys:



There are $n!$ possible leaves \rightarrow minimum depth is $O(\log_2(n!))$

Since $n! \geq (n/2)^{n/2}$, the minimum depth (number of comparisons) is $O(\log((n/2)^{n/2})) = O(n \log(n))$

How Fast Can We Sort?

A comparison of the time complexities of the four clustering methods so far:

Method	Worst	Average
Insertion	n^2	n^2
Heap	$n \log n$	$n \log n$
Merge	$n \log n$	$n \log n$
Quick	n^2	$n \log n$

However, the time complexity is not the whole story. The same time complexity does not imply the same performance in practice.

How Fast Can We Sort?


An experimental comparison of the actual time used by the four clustering methods so far:

n	Insertion	Heap	Merge	Quick
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.059	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

How Fast Can We Sort?

- For small n , insertion sort is fastest.
 - It can take advantage of partially sorted lists, using few item movements in such cases.
 - Faster than other $O(n^2)$ methods such as the bubble sort.
- For larger n , quick sort is the fastest.
 - If we need a stable sorting method, merge sort is usually the choice.
- Combining both in practical implementations:
 - In quick sort or merge sort, when a segment or a sublist has its length below a threshold (say, 100), use insertion sort on that segment or sublist instead of doing additional subdivision.

Comparison Sorts, or Else?

- The $O(n \log(n))$ bound applies to **comparison sorts**:
 - Keys of items are compared against each other.
 - Actions are taken based on the comparison: selection, swapping, etc.
- For many applications, we can do faster by not using comparisons.
 - Example: Assuming that you're sorting a deck of cards with the key being the suits:
 - How will you do this? And what's the time complexity, assuming that you have n cards in r suits?

More on Sorting Cards

- Now, how about sorting 100 (mixed) decks of cards according to the suits and then the face values?
 - Put the cards into 4 piles first, each having one suit.
 - Then for each pile, put its cards into 13 (smaller) piles, each having one face value.
 - Combine the piles in the correct order.

Q: Now, what's the time complexity?

Radix Sort

■ The idea of **radix sort**:

- Separate the sort key into several parts. For our example: suits and face values.
- The separate parts of a key have orders (precedences). For our example: Suit is more important than face value.
- The sorting occurs in several passes, each involving only one part of the key. For our example: First the suit, then the face value.
- Each pass involves distributing the items into a finite set of bins (piles in our example).
- Complexity for **n** items, **d** passes, and **r** bins per pass: **$O(d(n+r))$** .

MSD and LSD Radix Sort

When sorting on multiple keys (or multiple parts of a key, as in radix sort), we can use two different ordering of the keys:

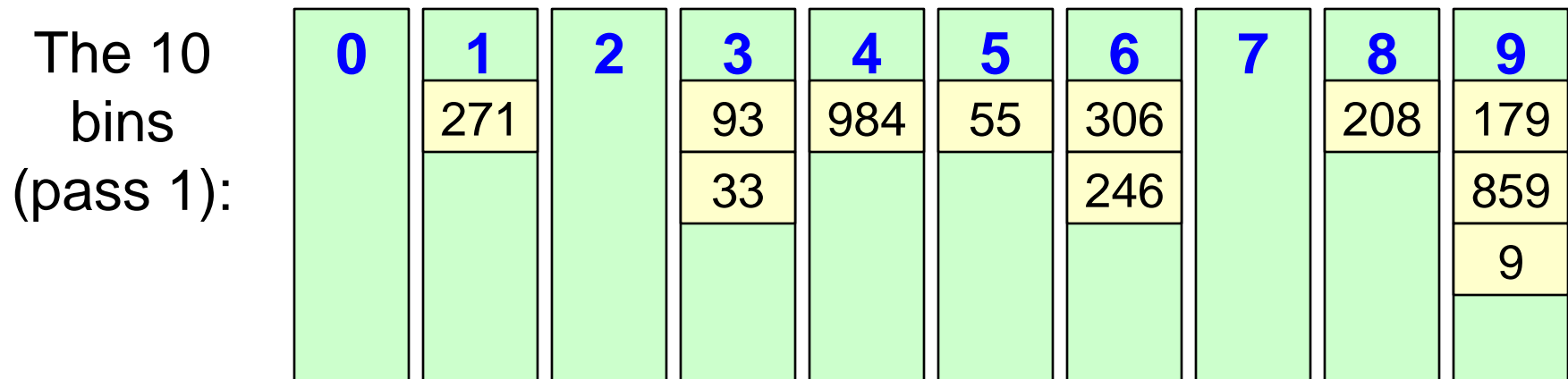
- **MSD** (most-significant-digit first): more intuitive
 - Separate items into bins of the most significant key first.
 - For the items in each bin, separate them into bins of the second most significant key.
 - Repeat this (recursively) for all other keys.
- **LSD** (least-significant-digit first): more simple
 - Separate items into bins of the least significant key first.
 - Concatenate the items into a single set, and then separate them into bins of the second least significant key.
 - Repeat this for all other keys.

Illustration of LSD Radix Sort

Consider the radix-10 (10 bins in each pass) sorting of integer keys, with each pass corresponding to a digit:

Original:

179	208	306	93	859	984	55	9	271	33	246
-----	-----	-----	----	-----	-----	----	---	-----	----	-----



Each bin has a queue (to keep the items in FIFO order).

Result (pass 1):

271	93	33	984	55	306	246	208	179	859	9
-----	----	----	-----	----	-----	-----	-----	-----	-----	---

Illustration of LSD Radix Sort

Now the second pass:

Result
(pass 1):

271	93	33	984	55	306	246	208	179	859	9
-----	----	----	-----	----	-----	-----	-----	-----	-----	---

The 10
bins
(pass 2):

0	1	2	3	4	5	6	7	8	9
306			33	246	55		271	984	93
208					859		179		
9									

Each bin has a queue (to keep the items in FIFO order).

Result
(pass 2):

306	208	9	33	246	55	859	271	179	984	93
-----	-----	---	----	-----	----	-----	-----	-----	-----	----

Illustration of LSD Radix Sort

Now the third pass:

Result
(pass 2):

306	208	9	33	246	55	859	271	179	984	93
-----	-----	---	----	-----	----	-----	-----	-----	-----	----

The 10
bins
(pass 3):

0	1	2	3	4	5	6	7	8	9
9	179	208	306					859	984
33		246							
55		271							
93									

Each bin has a queue (to keep the items in FIFO order).

Result
(pass 3):

9	33	55	93	179	208	246	271	306	859	984
---	----	----	----	-----	-----	-----	-----	-----	-----	-----

A Note on External Sorting

- Consider **Big Data** ...
- When we can not put everything being processed in the main memory, moving data in and out of the memory becomes the main bottleneck.
 - As a matter of fact, for today's PCs, when we can not put everything being processed in the cache, moving data between the cache and the main memory becomes the main bottleneck.
- Algorithms have to adapt to this.
 - It is better to have algorithms that work *locally*.
- For sorting: **Merge sort** is usually the method of choice here.