CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Sorting

Department of Computer Science
University of Maryland, College Park

Sorting

- Goal
  - Arrange elements in predetermined order
    - Based on key for each element
  - Derived from ability to compare two keys
- Properties
  - **Stable** → relative order of equal keys unchanged
    - Stable: 3, 1, 4, 3, 3, 2 → 1, 2, 3, 3, 3, 4
    - Unstable: 3, 1, 4, 3, 3, 2 → 1, 2, 3, 3, 3, 4
  - **In-place** → uses only constant additional space
  - **Internal** → all data being sorted fits into main memory
  - **External** → required when the data being sorted do not fit into main memory
  - **Adaptive** → Algorithm takes into account whether the data is sorted or partially sorted
    - Runs faster the more sorted the data is initially
- Most algorithms discussed in lecture are internal and based on arrays

Type of Sorting Algorithms

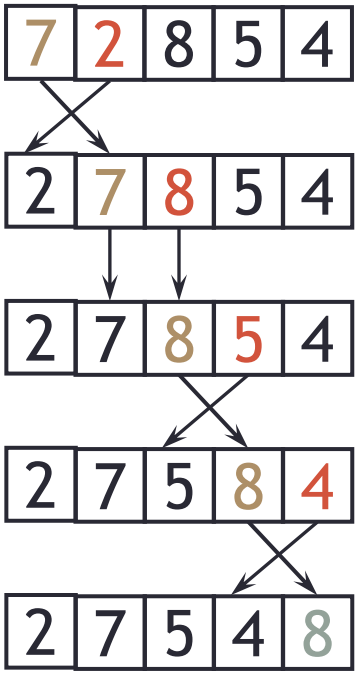
- Comparison-based and Linear Algorithms
  - Comparison-based Algorithms →Only uses pairwise key comparisons
  - Linear Algorithms →Uses additional properties of keys
- **Comparison-based**
  - Proven lower bound of Omega(n log(n))
  - Examples
    - $O(n^2)$ →Bubblesort, Selection sort, Insertion sort
    - O(nlog(n)) → Treesort, Heapsort, Quicksort, Mergesort
- **Linear Algorithms**
  - Counting sort
  - Bucket (bin) sort
  - Radix sort

Bubble Sort

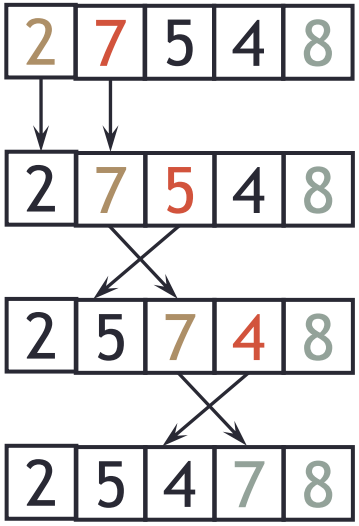
- Approach
  - Iteratively sweep through shrinking portions of list
  - Swap element x with its right neighbor if x is larger
  - After each pass largest element placed at the bottom (lighter elements bubble up)
- Performance
  - $O(n^2)$ average / worst case

Bubble Sort Example

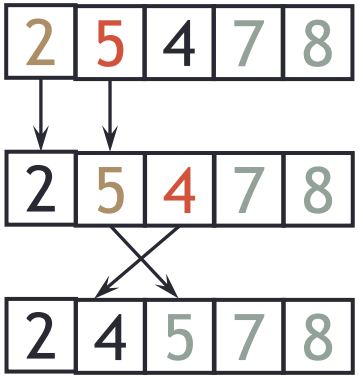
Sweep 1



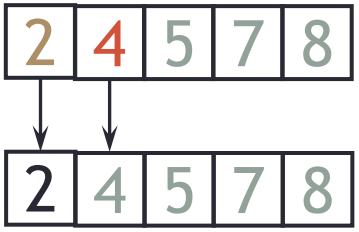
Sweep 2



Sweep 3



Sweep 4



Bubble Sort Code

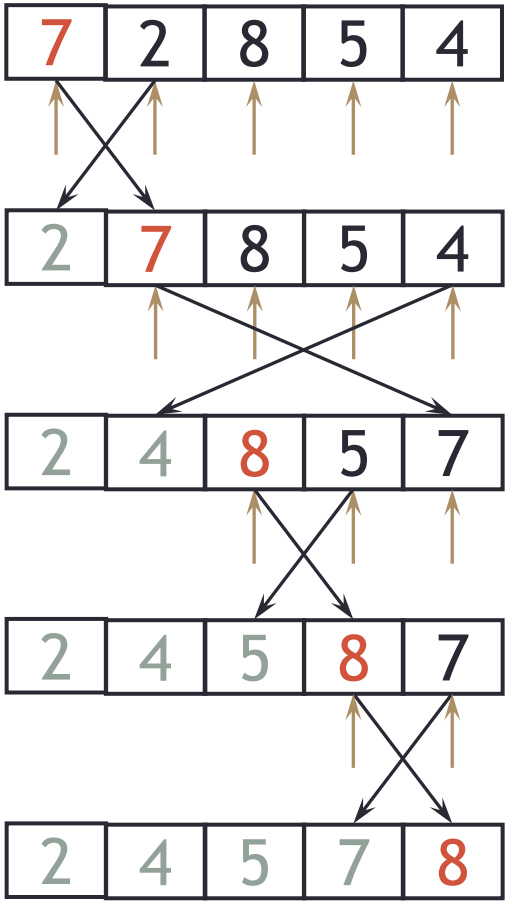
```
void bubbleSort(int[] a) {
    int outer, inner;

    for (outer = a.length - 1; outer > 0; outer--)
       for (inner = 0; inner < outer; inner++)
          if (a[inner] > a[inner + 1])
            swap(a, inner, inner+1);
}

void swap(int a[ ], int x, int y) {
    int temp = a[x];

    a[x] = a[y];
    a[y] = temp;
}
```

Swap with right neighbor if larger

How can we improve it?

Selection Sort

- Approach
  - Iteratively sweep through shrinking portions of list
  - Select smallest element found in each sweep
  - Swap smallest element with front of current list
- Performance
  - $O(n^2)$ average / worst case



Selection Sort Code

```
void selectionSort(int[] a) {
    int outer, inner, min;

    for (outer = 0; outer < a.length - 1; outer++) {
       min = outer;
       for (inner = outer + 1; inner < a.length; inner++) {
          if (a[inner] < a[min]) {
             min = inner;
          }
       }
       swap(a, outer, min);
    }
}
```

Sweep through array

Find smallest element

Swap with smallest element found

Insertion Sort

- Similar to method used when sorting cards
- Although is $O(n^2)$ it is the preferred algorithm when data is nearly sorted or when the size of the data to sort is small (due to low overhead). For these reasons is often used in algorithms like quicksort when the size of the data is small
- Has a simple implementation
- More efficient in practice than most simple quadratic algorithms
- **It is adaptive**

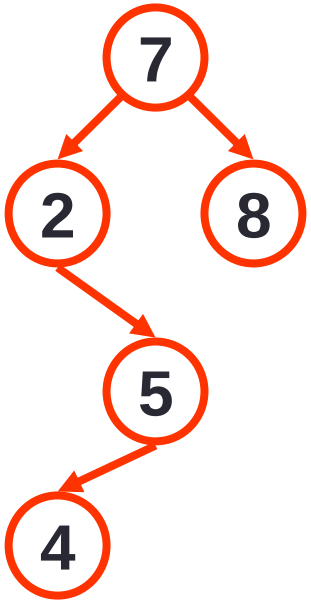
Insertion Sort Code

```
void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int j, temp = a[i];

        for (j = i - 1; j >= 0 && temp < a[j]; j--) {
            a[j + 1] = a[j];
        }
        a[j + 1] = temp;
    }
}
```

Tree Sort

- Approach
  - Insert elements in binary search tree
  - List elements using inorder traversal
- Performance
  - Binary search tree
    - O(n log(n)) average case
    - O(n^2) worst case
  - Balanced binary search tree
    - O(n log(n)) average / worst case

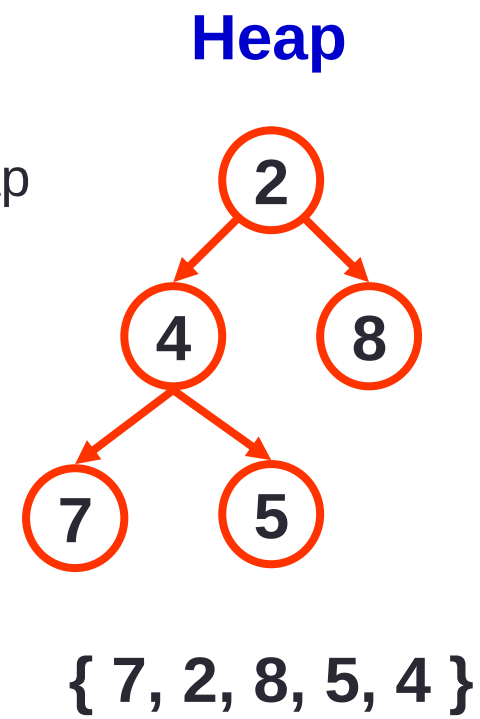
Binary search tree



{ 7, 2, 8, 5, 4 }

Heap Sort

- Approach
  - Insert elements in heap
  - Remove smallest element in heap, repeat
  - List elements in order of removal from heap
- Performance
  - O(n log(n)) average / worst case



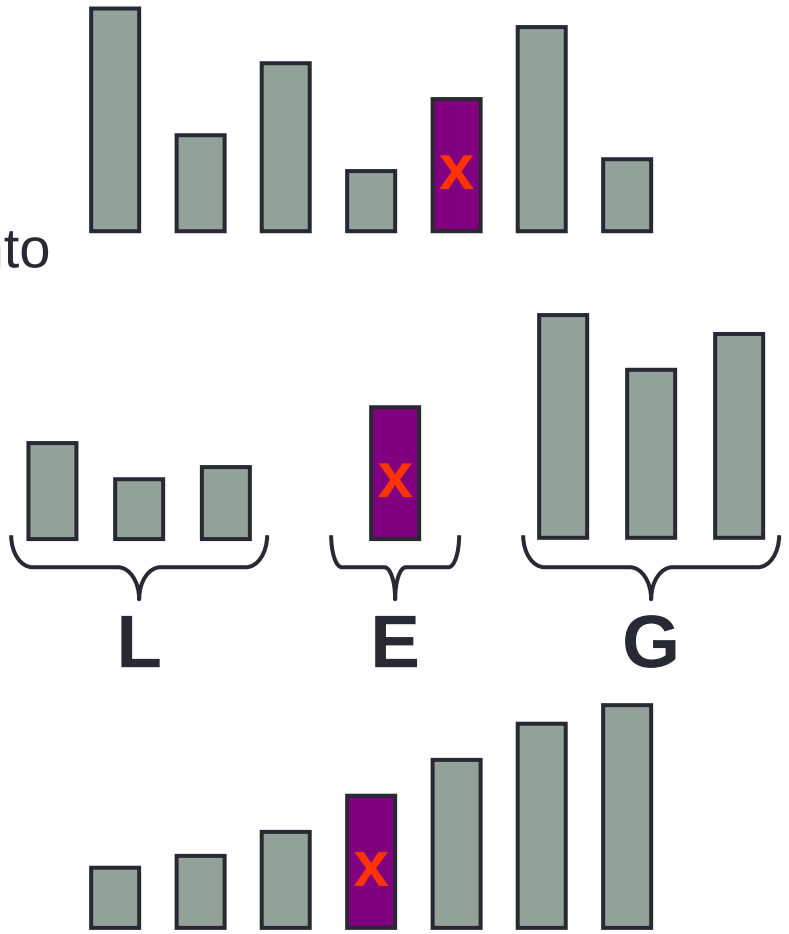
{ 7, 2, 8, 5, 4 }

Quick Sort

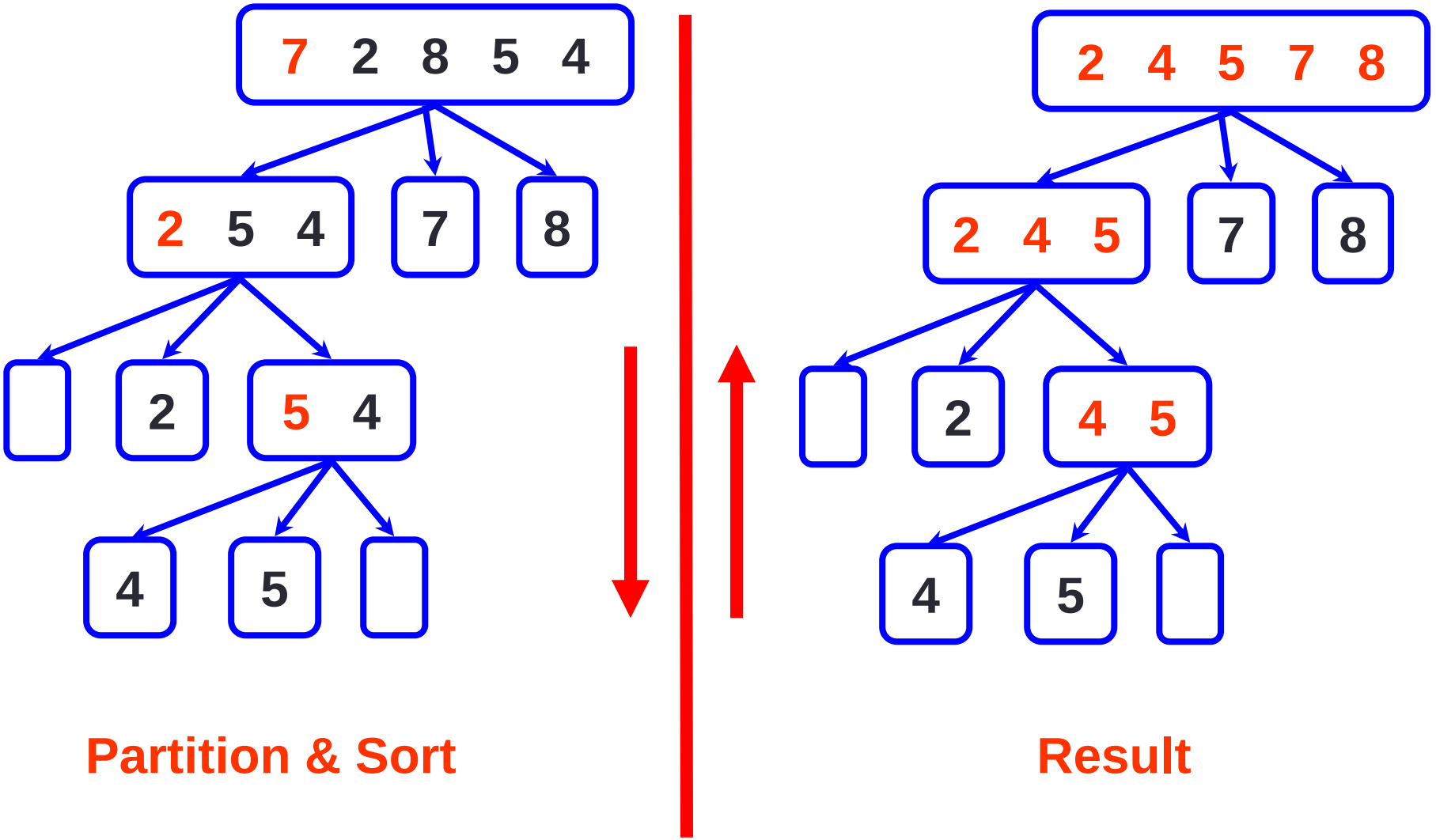
- Approach
  - Select pivot value (near median of list)
  - Partition elements (into 2 lists) using pivot value
  - Recursively sort both resulting lists
  - Concatenate resulting lists
  - **For efficiency pivot needs to partition list evenly**
- Performance
  - O(n log(n)) average case
  - $O(n^2)$ worst case
- Used by Arrays.sort
- Runs faster than mergesort in most cases

Quick Sort Algorithm

1. If list below size K
   - Sort w/ other algorithm
     - e.g., insertion sort
2. Else pick pivot x and partition S into
   - L elements < x
   - E elements = x
   - G elements > x
3. Quicksort L & G
4. Concatenate L, E & G
   - If not sorting in place



Quick Sort Example



Quick Sort Code

```
void quickSort(int[] a, int x, int y) {
    int pivotIndex;

    if ((y - x) > 0) {
        pivotIndex = partitionList(a, x, y);
        quickSort(a, x, pivotIndex - 1);
        quickSort(a, pivotIndex + 1, y);
    }
}

int partitionList(int[] a, int first, int last) {
    … // partitions list and returns index of pivot
}
```

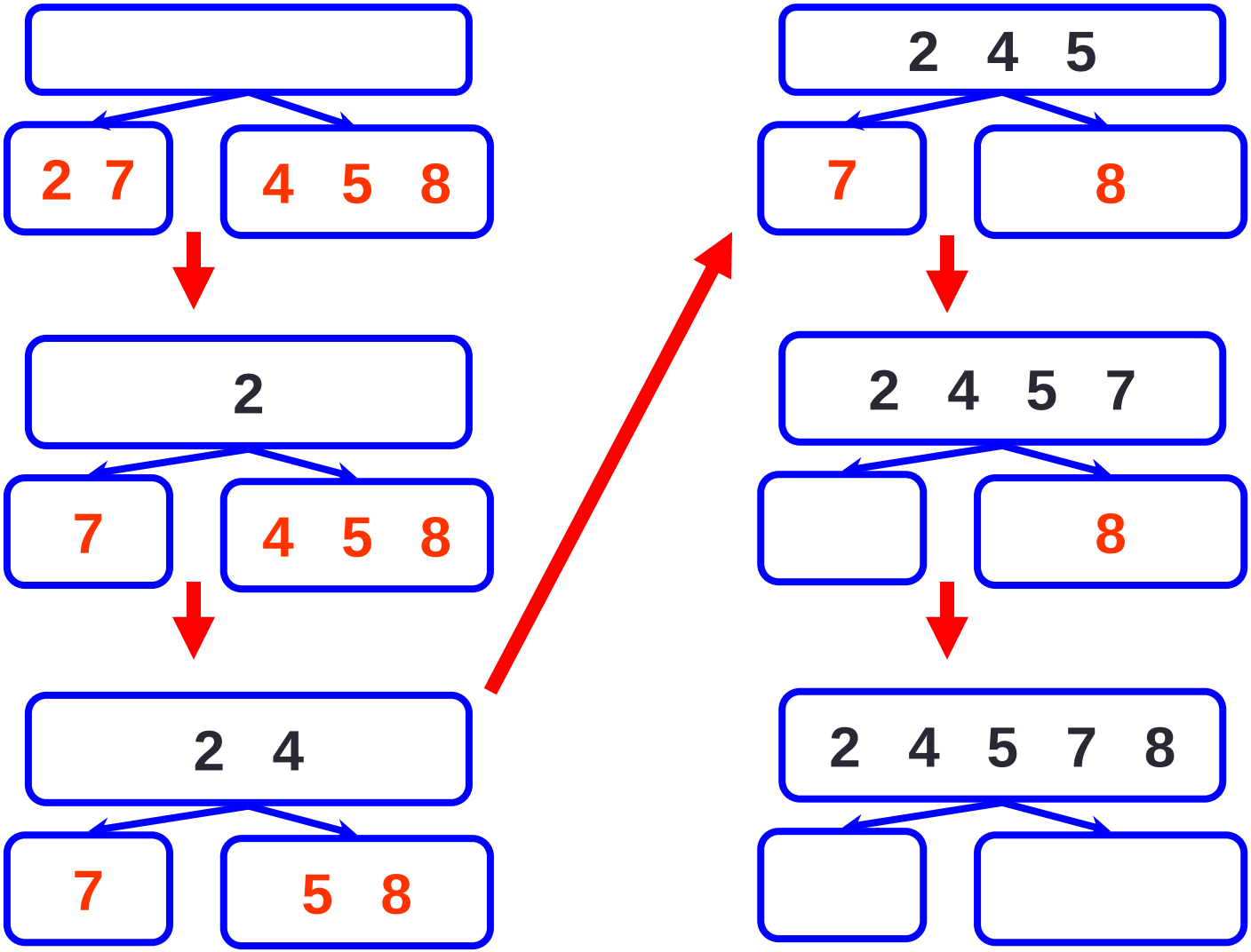
Quick Sort Code

```
int partitionList(int a[], int first, int last) {
   int i, pivot, border;

   pivot = a[first];
   border = first;
   for (i = first + 1; i <= last; i++) {
      if (a[i] <= pivot) {
         border++;
         swap(a, border, i);
      }
    }
    swap(a, first, border);

    return border;
}
```

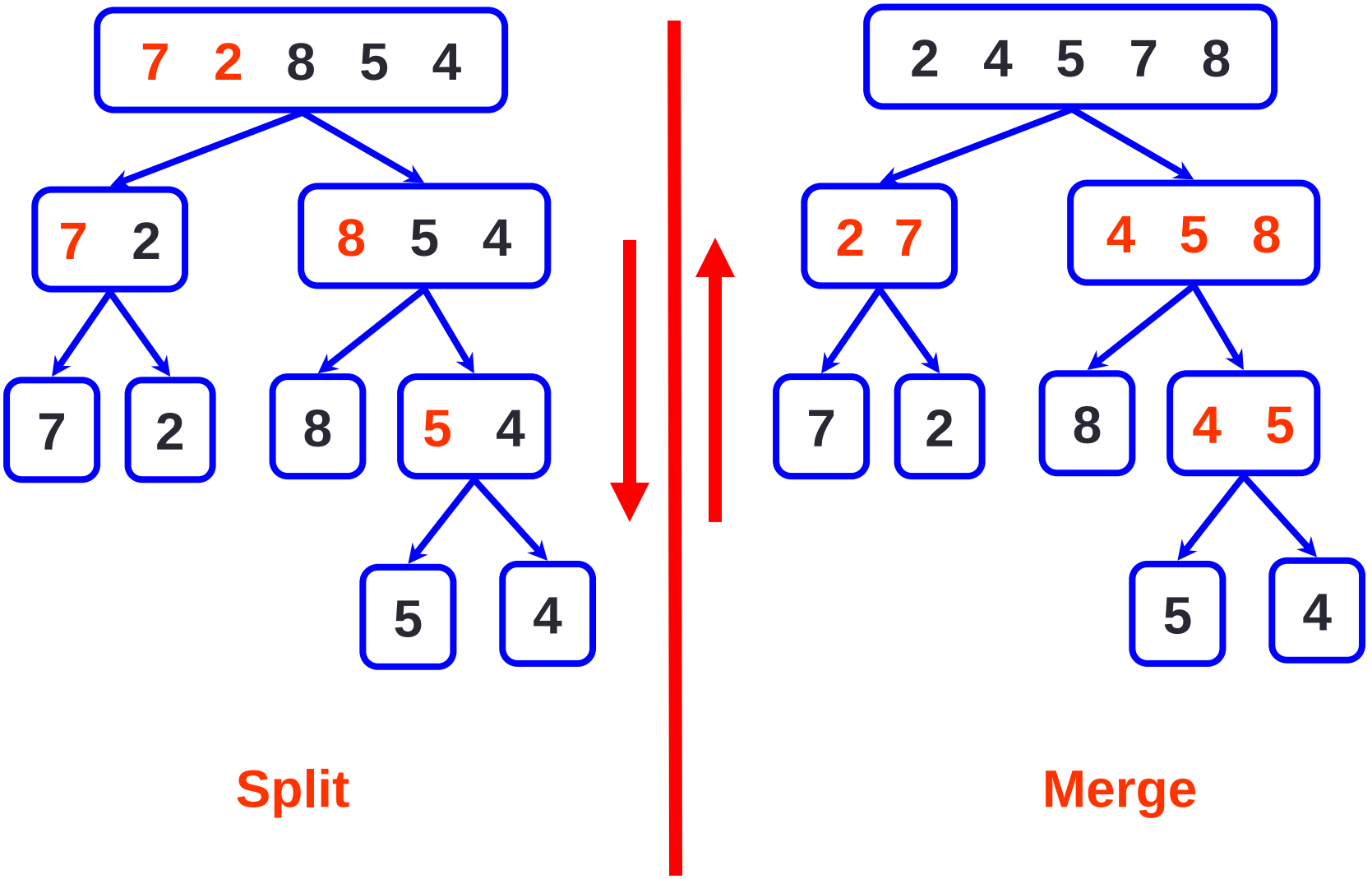
Merge Sort

- Approach
  1. Partition list of elements into 2 lists
  2. Recursively sort both lists
  3. Given 2 sorted lists, merge into 1 sorted list
     - Examine head of both lists
     - Move smaller to end of new list
- Performance
  - O(n log(n)) average / worst case
- Used by Collections.sort
- External and internal sorting algorithm
- Often preferred for sorting a linked list
- Divide and conquer algorithm
- Uses additional memory to perform merge step (**not in-place**)
- Even though merge sort and quick sort are O(nlog(n)) algorithms, quick sort is usually faster in practice and does not require additional memory

Merge Example

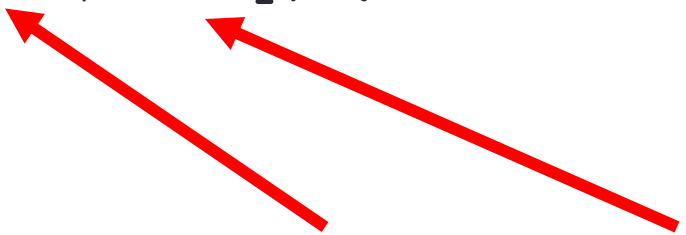


Merge Sort Example



Merge Sort Code

```
void mergeSort(int[] a, int x, int y) {
    int mid = (x + y) / 2;

    if (x != y) {
        mergeSort(a, x, mid);
        mergeSort(a, mid + 1, y);
        merge(a, x, y, mid);
    }
}

void merge(int[] a, int x, int y, int mid) {
    … // merges 2 adjacent sorted lists in array
}
```



Lower end of array region to be sorted

Upper end of array region to be sorted

Merge Sort Code

```
void merge(int[] a, int x, int y, int mid) {
   int j, size = y - x + 1, left = x, right = mid + 1;
   int[] tmp = new int[a.length];

   for (j = 0; j < size; j++)
      if (left > mid)
         tmp[j] = a[right++];
      else
         if (right > y || a[left] < a[right])
            tmp[j] = a[left++];
         else
            tmp[j] = a[right++];

   for (j = 0; j < size; j++)
      a[x + j] = tmp[j];
}
```

Radix Sort

- **Does not compare array entries**
- It is **not** suitable as a general-purpose sorting algorithm
- Approach
  - Decomposes a key C into its components (C_1, C_2, … , C_n) and uses each component to organize to which buckets keys are assigned
    - For an integer we use the digits that make up the integer
    - For a word we use the letters that make up the word
- **Running time O(n), but it cannot sort all types of data**

Radix Sort (Example)

- B(i) - stands for Bucket for digit i
- Original - 122, 397, 220, 017, 512
- Important: Each bucket must retain the order in which it receives values
- **Step 1**: Distribute values into buckets according to rightmost digit
  - 12**2**, 39**7**, 22**0**, 01**7**, 51**2**
  - B(0){22**0**}, B(1){}, B(2){12**2**, 51**2**}, B(3){}, B(4){}, B(5){}, B(6){}, B(7){39**7**, 01**7**}, B(8){}, B(9){}
  - 220, 122, 512, 397, 017
- **Step 2**: Distribute values into buckets according to the next (left) digit
  - 2**2**0, 1**2**2, 5**1**2, 3**9**7, 0**1**7
  - B(0){}, B(1){5**1**2, 0**1**7}, B(2){2**2**0, 1**2**2}, B(3){}, B(4){}, B(5){}, B(6){}, B(7){}, B(8){}, B(9){3**9**7}
  - 512, 017, 220, 122, 397
- **Step 3**: Distribute values into buckets according to the leftmost digit
  - **5**12, **0**17, **2**20, **1**22, **3**97
  - B(0){**0**17}, B(1){**1**22}, B(2){**2**20}, B(3){**3**97}, B(4){}, B(5){**5**12}, B(6){}, B7{}, B8{}, B9{}
  - **Sorted** → 017, 122, 220, 397, 512

Radix Sort Algorithm for Integer Array

- Reference: Data Structures & Abstractions with Java, 5th Edition, Carrano, Henry, ISBN 9780134831695
- **radixSort** method
  - Sorts array[first..last] of positive integers in ascending order where maxDigits is the number of digits in the longest integer. Bucket can be a queue.

```
radixSort(array, first, last, maxDigits) {
   for (i = 0 to maxDigits - 1) {     // d iterations, processing of digits from right to left
      Clear bucket[0], bucket[1], … , bucket[9]
      for (index = first to last) { // n iterations, processing all elements of array
         digit = digit i of a[index]
         Place a[index] at end of bucket[digit]
      }
      // Completes a step
      Place contents of bucket[0], bucket[1], … , bucket[9] into array a
   }
}
```

- For **d** digits for each integer and array with **n** integers: O(**d** x **n**). As long as **d** is fixed and much smaller than **n** radix sort is O(**n**)
- **Radix sort disadvantages**: number of buckets depends on the possible components of a key (e.g., integers 10 buckets, words 26 buckets)

Sorting Properties

Name	Compari-son Sort	Avg Case Complexity	Worst Case Complexity	In Place	Can be Stable
Bubble	√	$O(n^2)$	$O(n^2)$	√	√
Selection	√	$O(n^2)$	$O(n^2)$	√	√
Insertion	√	$O(n^2)$	$O(n^2)$	√	√
Tree	√	O(n log(n))	$O(n^2)$		
Heap	√	O(n log(n))	O(n log(n))		
Quick	√	O(n log(n))	$O(n^2)$	√	
Merge	√	O(n log(n))	O(n log(n))		√
Radix		O(n)	O(n)		√

Links

- Sorting Algorithms Animations
  - https://www.toptal.com/developers/sorting-algorithms/
  - https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- President and Sorting
  - http://www.youtube.com/watch?v=k4RRi_ntQc8
- Ineffective Sorts
  - http://xkcd.com/1185/
- What different sorting algorithms sound like
  - http://www.youtube.com/watch?v=t8g-iYGHpEA
- Big-O Complexity for Array Sorting Algorithms
  - http://bigocheatsheet.com/