

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 by size
- 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
 - **External** → can efficiently sort large # of keys
 - Most algorithms discussed in lecture are internal and based on arrays

Type of Sorting Algorithms

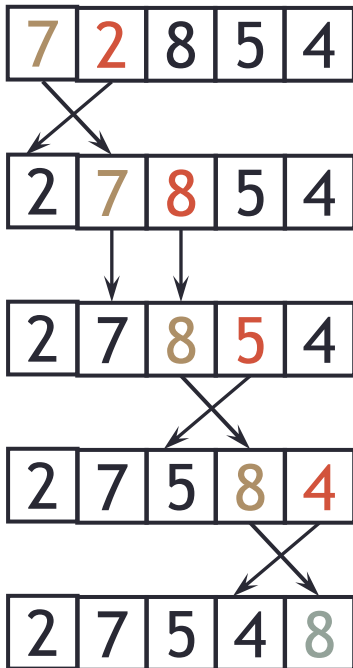
- **Comparison-based** and **Linear Algorithms**
 - Comparison-based Algorithms → Only uses pairwise key comparisons
 - Linear Algorithms → Uses additional properties of keys
- Comparison-based
 - Proven lower bound of $O(n \log(n))$
 - Examples
 - $O(n^2)$ → Bubblesort, Selection sort, Insertion sort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

Bubble Sort

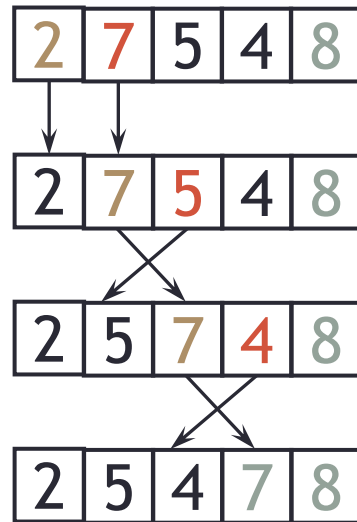
- Approach
 - Iteratively sweep through shrinking portions of list
 - Swap element **x** with its right neighbor if **x** is larger
- Performance
 - $O(n^2)$ average / worst case

Bubble Sort Example

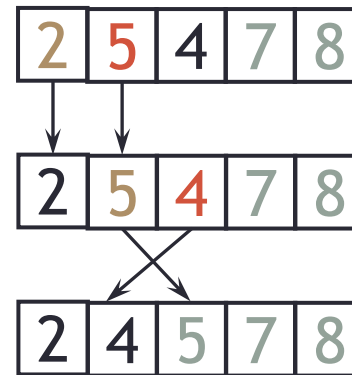
Sweep 1



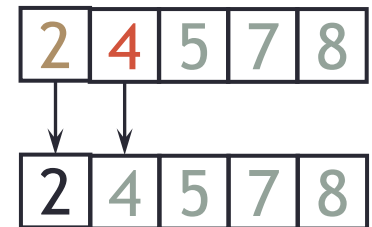
Sweep 2



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);  
}
```

**Swap with
right neighbor
if larger**



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

How can we improve it?

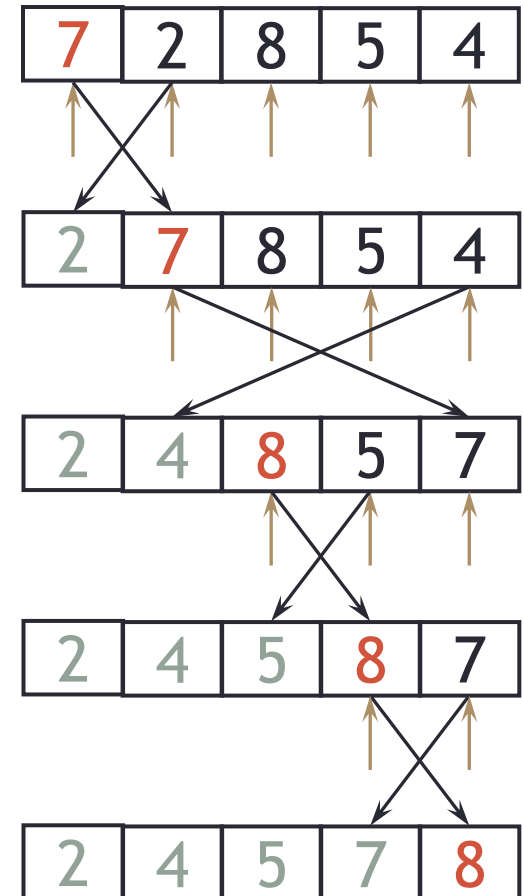
Selection Sort

- Approach

1. Iteratively sweep through shrinking portions of list
2. Select smallest element found in each sweep
3. 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 →

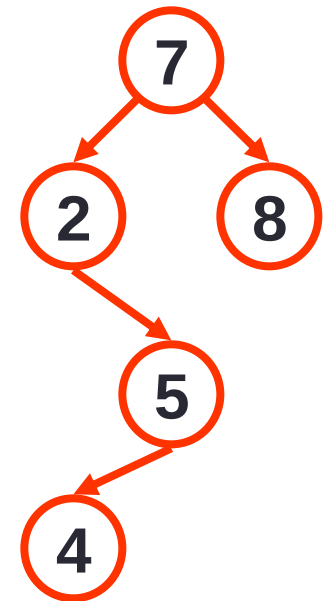
Find smallest element ←

Swap with smallest element found ←

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

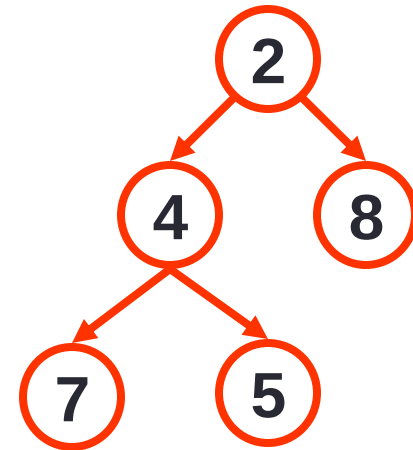


{ 7, 2, 8, 5, 4 }

Heap Sort

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

Heap



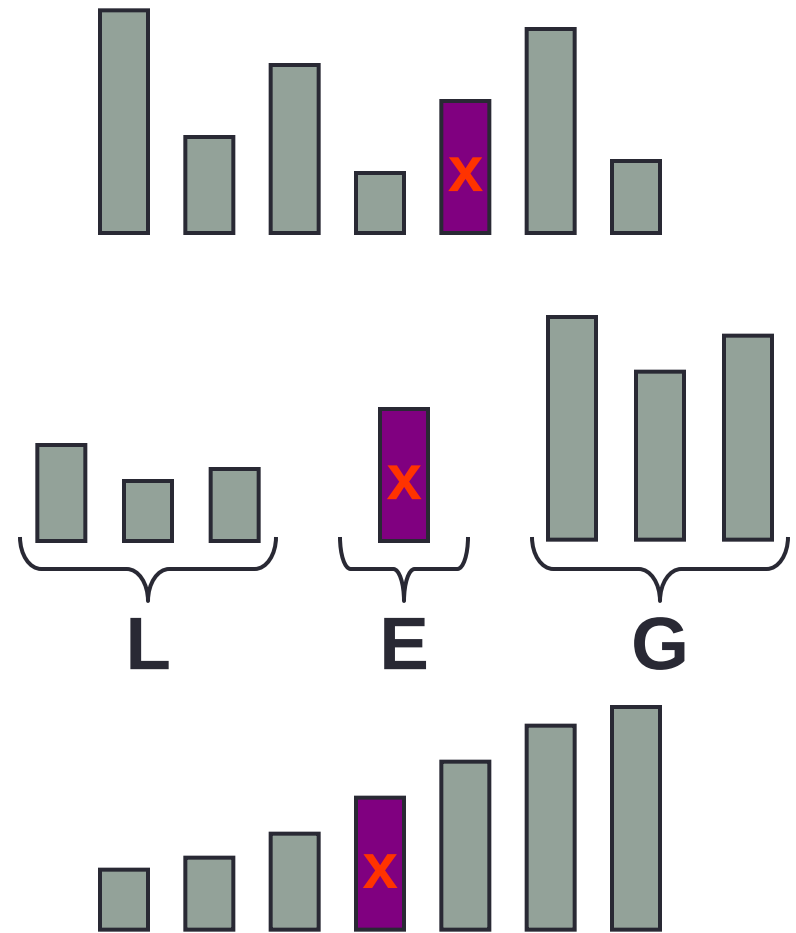
{ 7, 2, 8, 5, 4 }

Quick Sort

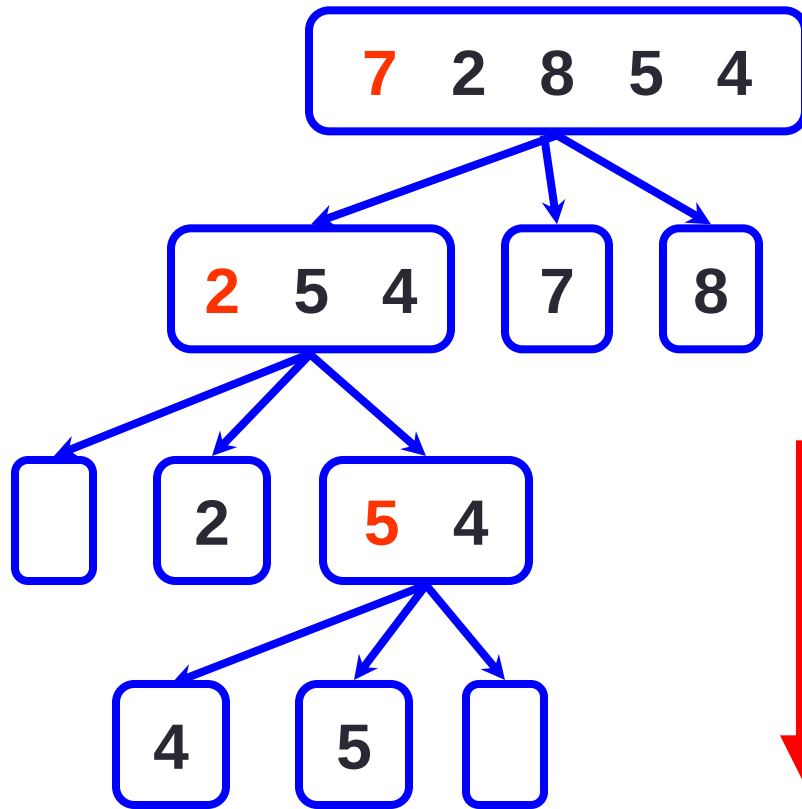
- Approach
 1. Select pivot value (near median of list)
 2. Partition elements (into 2 lists) using **pivot** value
 3. Recursively sort both resulting lists
 4. Concatenate resulting lists
 5. For efficiency pivot needs to partition list evenly
- Performance
 - $O(n \log(n))$ average case
 - $O(n^2)$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

Quick Sort Algorithm

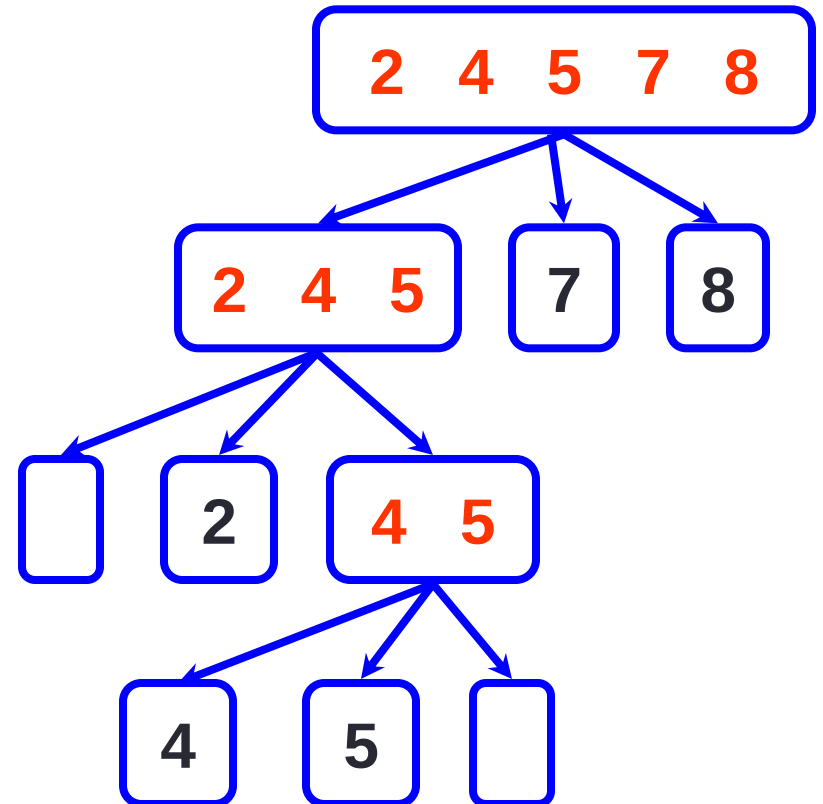
1. **If list below size K**
 - Sort w/ other algorithm (e.g. insertion sort)
2. Else pick pivot **x** and partition S into
 - L elements $< x$
 - E elements $= x$
 - G elements $> x$
3. Quicksort L & G
4. Concatenate L, E & G
 - If not sorting in place



Quick Sort Example



Partition & Sort



Result

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  
}
```

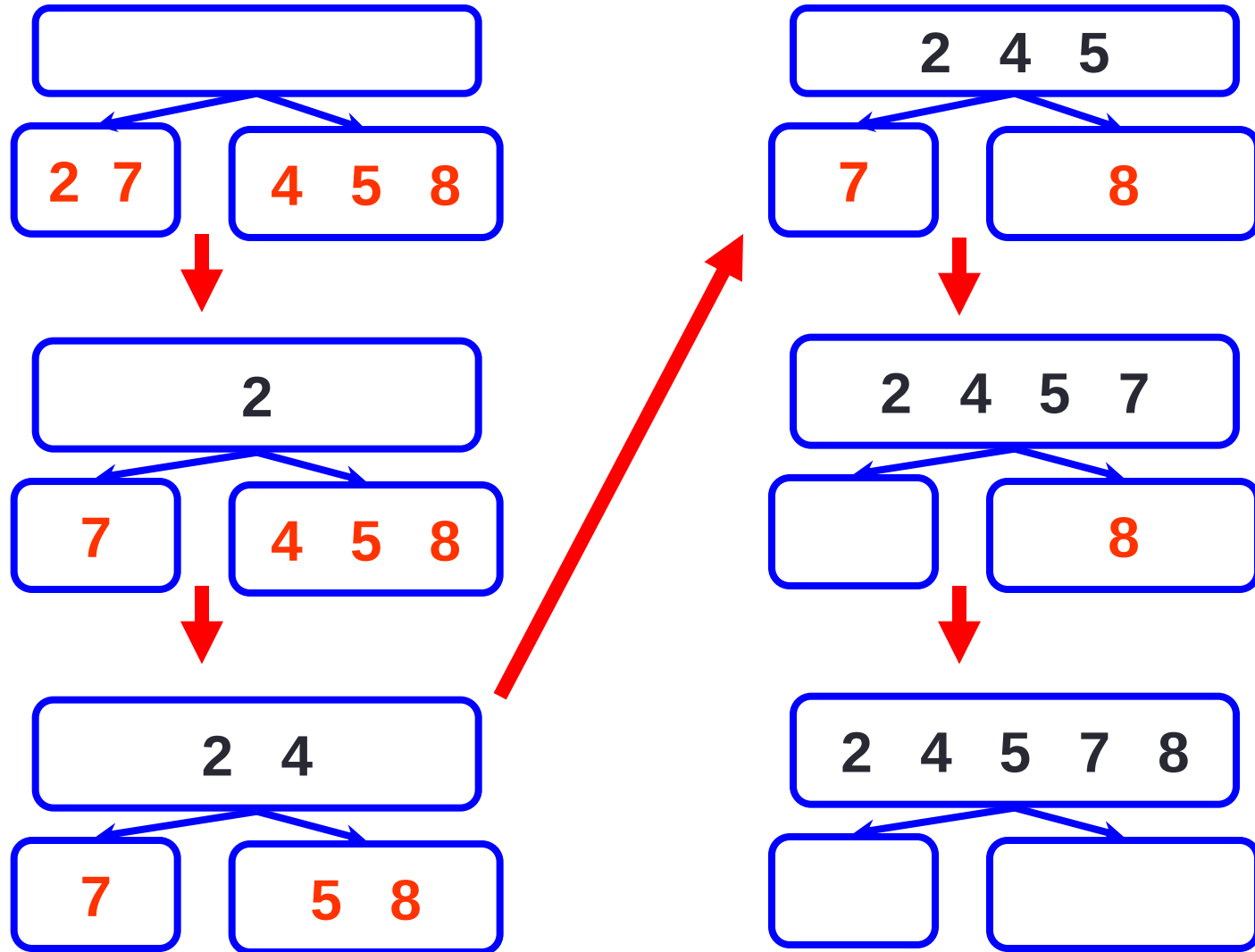
Quick Sort Code

```
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;  
}
```

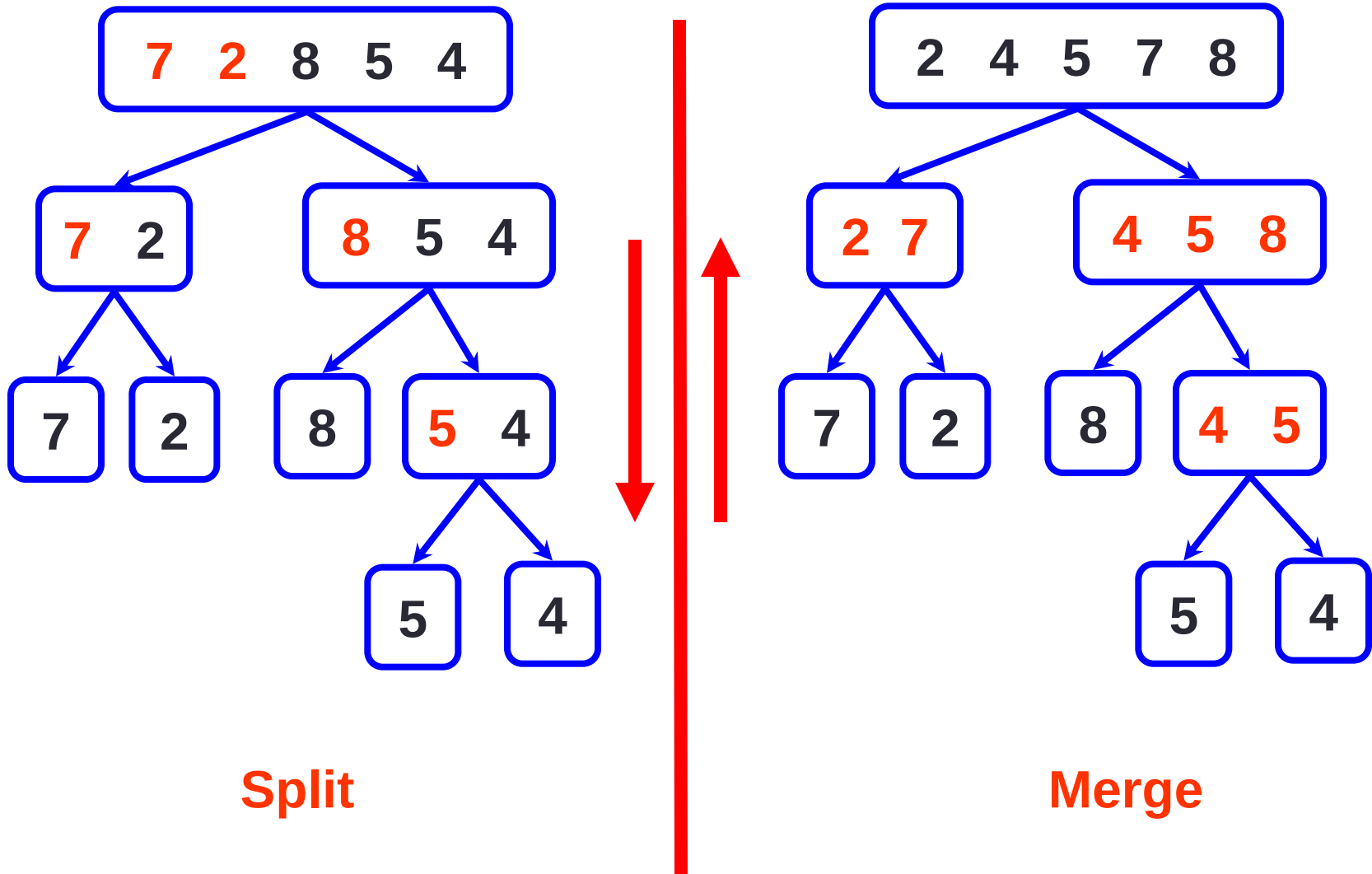
Merge Sort

- 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**

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];  
}
```

Sorting Properties

Name	Comparison Sort	Avg Case Complexity	Worst Case Complexity	In Place	Can be Stable
Bubble	✓	$O(n^2)$	$O(n^2)$	✓	✓
Selection	✓	$O(n^2)$	$O(n^2)$	✓	✓
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	✓	$O(n \log(n))$	$O(n \log(n))$		✓

Links

- President and Sorting
 - http://www.youtube.com/watch?v=k4RRi_ntQc8
- Sorting Algorithms Comparison
 - <http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html>
- Ineffective Sorts
 - <http://xkcd.com/1185/>
- Selection vs. Quicksort
 - <http://jtf.acm.org/demos/classroom/SortDemo.html>
- What different sorting algorithms sound like
 - <http://www.youtube.com/watch?v=t8g-iYGHpEA>
- Sorting Algorithms
 - <http://maven.smith.edu/~thiebaut/java/sort/demo.html>