PROGRAMMING AND DATA STRUCTURES

SORTING ALGORITHMS

HOURIA OUDGHIRI

FALL 2023

OUTLINE

- Sorting problem
- Types of sorting solutions
- Sorting algorithms
- Comparison of sorting algorithms

STUDENT LEARNING OUTCOMES

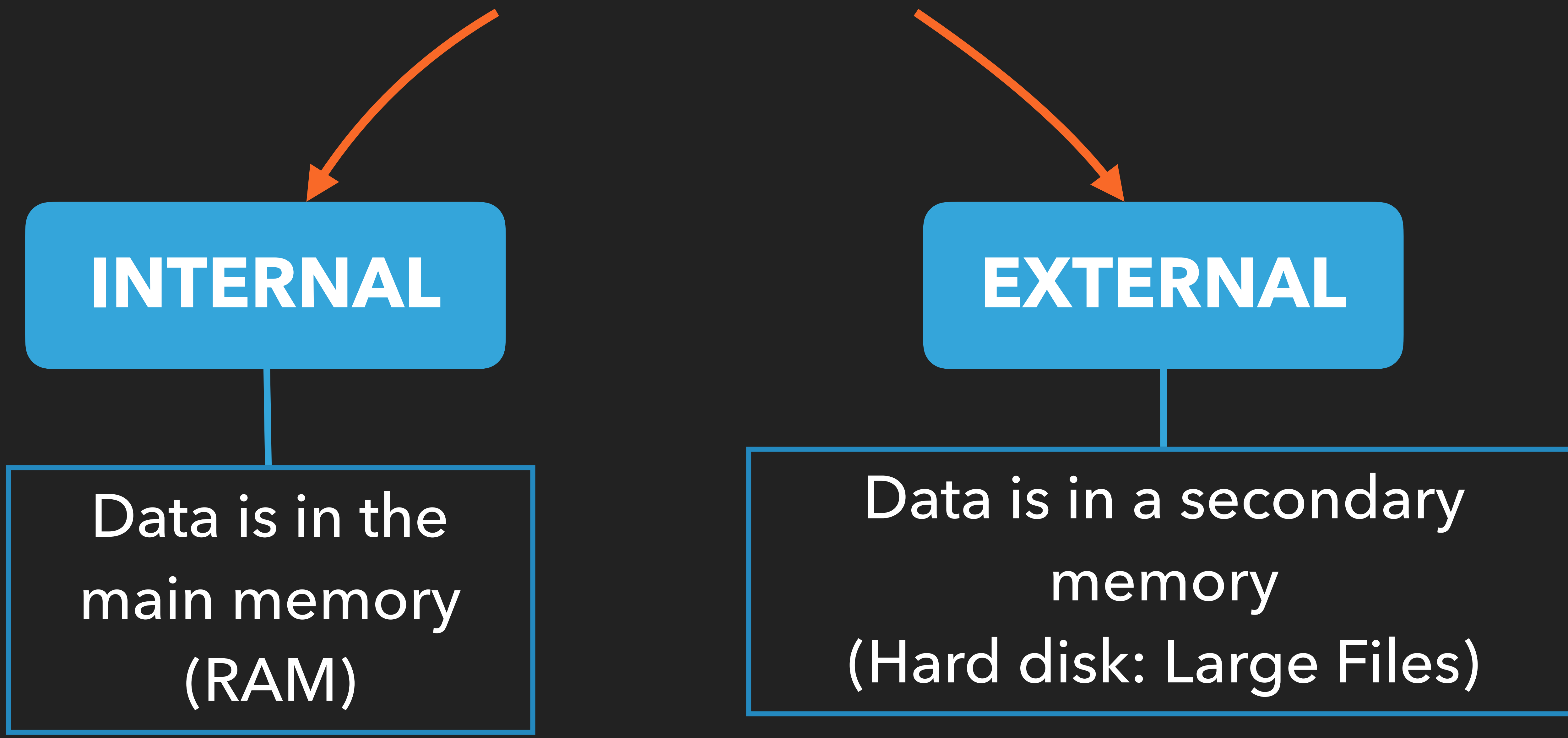
At the end of this chapter, you should be able to:

- List the types and categories of sorting
- Implement the different sorting algorithms
- Evaluate the complexity of the sorting algorithms
- Compare the sorting algorithms

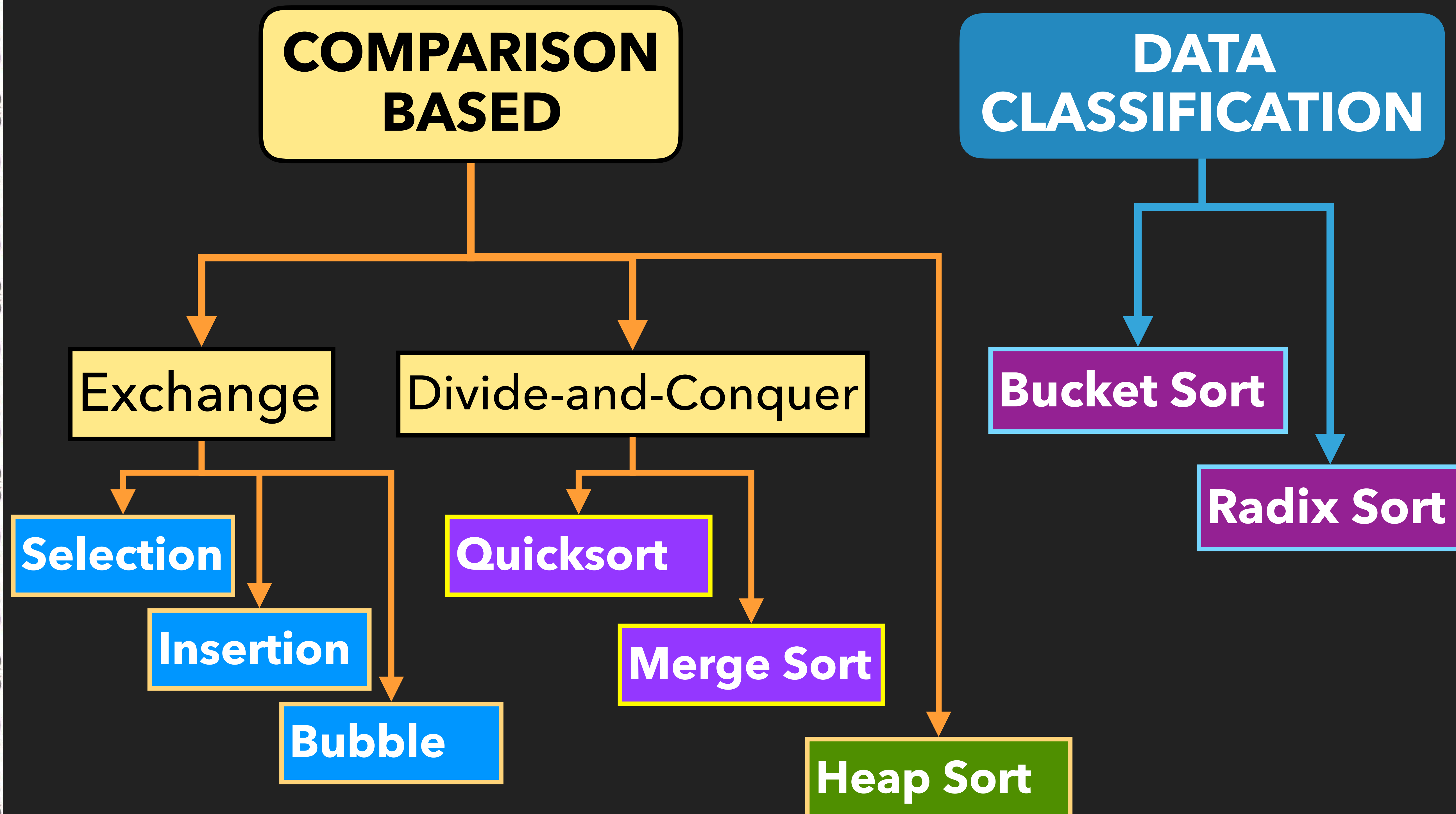
Sorting problem

- Given a list of data items, arrange the list in an ascending or descending order
- Commonly used in computer science to organize objects based on one specific criterion
- Allows using the efficient binary search algorithm
- Sorting is more complex than searching

Sorting types



Sorting categories



Selection Sort

67	33	21	84	49	50	75	min = 21
21	33	67	84	49	50	75	min = 33
21	33	67	84	49	50	75	min = 49
21	33	49	84	67	50	75	min = 50
21	33	49	50	67	84	75	min = 67
21	33	49	50	67	84	75	min = 75
21	33	49	50	67	75	84	

Selection Sort

```
Algorithm selectionSort
 for every element i (N size of the list)
  find the smallest element from i to N-1
  swap element i with the smallest element
End
```

Selection Sort

```
public static void selectionSort(int[] list) {
    int minIndex;
    for (int i=0; i<list.length-1; i++) {
        // Find the smallest element from i+1 to N
        int min = list[i];
        minIndex = i;
        for (int j=i+1; j<list.length; j++){
            if (list[j] < min){
                min = list[j];
                minIndex = j;
            }
        }
        // Swap the smallest element with element i
        swap(list, i, minIndex);
    }
}
```

Selection Sort

- Complexity Analysis

```
Iteration1(outer loop)
    (n)iterations(inner loop)
Iteration 2 (outer loop)
    (n-1) iterations (inner loop)
Iteration k (outer loop)
    (n-k+1) iterations (inner loop)
Iteration n-1 (outer loop)
    1 iteration (inner loop)
1 + 2 + … + (n) = n(n+1)/2
```

Time complexity: $O(n^2)$ – Quadratic growth
Space complexity: $O(1)$ – no additional space

Insertion Sort

67	33	21	84	49	50	75	Insert 33 at 0
33	67	21	84	49	50	75	Insert 21 at 0
21	33	67	84	49	50	75	Insert 84 at 3
21	33	67	84	49	50	75	Insert 49 at 2
21	33	49	67	84	50	75	Insert 50 at 3
21	33	49	50	67	84	75	Insert 75 at 5
21	33	49	50	67	75	84	

Insertion Sort

```
Algorithm insertionSort
   for every element i
     insert element i in the sorted list
     (0 to i-1)
   end for
End
```

Insertion Sort

```
public static void insertionSort(int[] list) {
    for (int i=1; i<list.length; i++) {
        //Insert element i in the sorted sub-list
        int currentVal = list[i];
        int j = i;
        while (j > 0 && currentVal < (list[j - 1])){
            // Shift element (j-1) into element (j)
            list[j] = list[j - 1];
            j--;
        }
        // Insert currentVal at index j
        list[j] = currentVal;
    }
}
```

Insertion Sort

- Complexity Analysis

```
Iteration1(outer loop)
    (1)iteration(inner loop)
Iteration 2 (outer loop)
    (2) iterations (inner loop)
Iteration k (outer loop)
    (k) iterations (inner loop)
Iteration n-1 (outer loop)
    n-1 iterations (inner loop)
1 + 2 + … + (n-1) = n(n-1)/2
```

Time complexity: $O(n^2)$ - Quadratic growth
Space complexity: $O(1)$

Bubble Sort

- At each iteration, exchange out of order pairs of elements until all elements are sorted
- Pushing the largest element to the end of the list

Bubble Sort

Pass 1

67	33	21	84	49	50	75	Out of order - swap
33	67	21	84	49	50	75	
33	67	21	84	49	50	75	Out of order - swap
33	21	67	84	49	50	75	
33	21	67	84	49	50	75	In order - No swap
33	21	67	84	49	50	75	Out of order - swap
33	21	67	49	84	50	75	
33	21	67	49	84	50	75	Out of order - swap
33	21	67	49	50	84	75	
33	21	67	49	50	84	75	Out of order - swap
33	21	67	49	50	75	84	

Bubble Sort

Pass 2

33 21 67 49 50 75 84	Out to order - swap
21 33 67 49 50 75 84	
21 33 67 49 50 75 84	In order - No swap
21 33 67 49 50 75 84	Out of order - swap
21 33 49 67 50 75 84	
21 33 49 67 50 75 84	Out of order - swap
21 33 49 50 67 75 84	
21 33 49 50 67 75 84	In order - No swap

Bubble Sort

Pass 3

21 33 49 50 67 75 84 In order - No swap

21 33 49 50 67 75 84 In order - No swap

21 33 49 50 67 75 84 In order - No swap

21 33 49 50 67 75 84 In order - No swap

No swaps: the list is sorted

Bubble Sort

```
Algorithm BubbleSort
  sorted = false
  last = N-1 (N size of the array)
  while (sorted is false)
    sorted = true
    for i=0 to last-1
       if(list[i] > list[i+1])
          swap(list[i], list[i+1])
          sorted = false
       end if
    end for
    last = last - 1;;
  end while
End
```

Bubble Sort

```
public static void bubbleSort(int[] list) {
  boolean sorted = false;
  for (int k=1; k < list.length && !sorted; k++) {
      sorted = true;
      for (int i=0; i<list.length-k; i++) {
          if (list[i] > list[i+1]) {
              swap(list, i, i+1);
              sorted = false;
          }
      }
  }
}
```

Bubble Sort

- Complexity Analysis

```
Iteration1(outer loop)
    (n-1)iteration(inner loop) to push the max
Iteration 2 (outer loop)
    (n-2) iterations (inner loop)
Iteration k (outer loop)
    (n-k) iterations (inner loop)
Iteration n-1 (outer loop)
    1 iterations (inner loop)
1 + 2 + … + (n-1) = n(n-1)/2
```

Time complexity: $O(n^2)$ - Quadratic growth

Space complexity: $O(1)$

Comparison (group 1)

Algorithm	Complexity	Performance Analysis
Selection Sort	**$O(n^2)$**	Simple Redundant Processing **Worst:** Same performance all the time
Insertion Sort	**$O(n^2)$**	Lower overhead than selection and bubble sort **Worst:** Reversed list
Bubble Sort	**$O(n^2)$**	Complex Inefficient for large sets **Worst:** Reversed list

Merge Sort

- Recursive sorting algorithm
- Uses divide-and-conquer strategy
- Split the list in halves recursively until obtaining lists with one element
- Merge the lists back in order

Merge Sort

67 33 21 84 49 50 75

[67 33 21] [84 49 50 75] Split

1. Merge sort the first half

[67] [33 21] [84 49 50 75] Split

[67] [33] [21] [84 49 50 75] Split

Merge Sort

67	33	21	84 49 50 75	
67	21 33		84 49 50 75	Merge
21 33 67			84 49 50 75	Merge

Merge Sort

2. Merge sort the second half

21 33 67	84 49 50 75		Split
21 33 67	84 49	50 75	Split
21 33 67	84 | 49	50 | 75	Split

Merge Sort

2. Merge sort the second half

21 33 67	84	49	50	75	
21 33 67	49 84		50 75		Merge
21 33 67	49 50 75 84				Merge
21 33 67	49 50 75 84				

3. Merge the two sorted halves

`21 33 49 50 67 75 80` Merge

Merge Sort

```
Algorithm MergeSort (recursive)

   Split the array in two halves
   MergeSort the first half
   MergeSort the second half
   Merge the two sorted halves

End
```

Merge Sort

```
public static void mergeSort(int[] list) {
  if (list.length > 1) {
    int[] firstHalf = new int[list.length/2];
    int[] secondHalf = new int[list.length -
                                  list.length/2];
    System.arraycopy(list, 0,
                     firstHalf, 0, list.length/2);
    System.arraycopy(list,list.length/2,
                     secondHalf, 0,
                     list.length-list.length/2);
    mergeSort(firstHalf);
    mergeSort(secondHalf);
    merge(firstHalf, secondHalf, list);
  }
}
```

Merge Sort

```
public static void merge(int[] list1, int[] list2,
                         int[] list) {
  int list1Index = 0, list2Index = 0, listIndex = 0;
  while(list1Index < list1.length &&
        list2Index < list2.length) {
    if (list1[list1Index] < list2[list2Index])
      list[listIndex++] = list1[list1Index++];
    else
      list[listIndex++] = list2[list2Index++];
  }
  // copy the remaining elements from list1 to list if any
  while(list1Index < list1.length)
    list[listIndex++] = list1[list1Index++];
  // copy the remaining elements from list2 to list if any
  while(list2Index < list2.length)
    list[listIndex++] = list2[list2Index++];
}
```

Merge Sort

- Complexity Analysis (Time)

```
Splitting the array in halves
   K = (log n) iterations (n/2^k = 1)

Merging halves
   (n) iterations (worst case)


Time compelxity: O(n log n) Log Linear
```

Merge Sort

- Complexity Analysis (Space)

```
Splitting the array in halves
 n/2, n/4, n/8, …

Total number of additional space
    (n/2 + n/4 + n/8 + …) ≃ n
```

Space complexity: O(n)

Quick Sort

- Recursive in-place sorting algorithm
- Uses divide-and-conquer strategy
- Divide the list in two partially sorted parts using a pivot
  - Part 1: all elements less than the pivot
  - Part 2: all elements greater than the pivot
- Repeat `quickSort` recursively on each part

Quick Sort

67 33 21 84 49 50 75 pivot = 67

67 33 21 84 49 50 75

67 33 21 84 49 50 75

67 33 21 84 49 50 75

67 33 21 49 84 50 75

67 33 21 49 50 84 75

50 33 21 49 | 67 | 84 75

Part1 Part2

Find position of the pivot

Quick Sort

67 33 21 84 49 50 75 pivot = 67

50 33 21 49 | 67 | 84 75

List1 List2

49 21 33 | 50 | 67 | 75 | 84

List3 List4

pivot (List1)= 50

pivot (List2) = 84

Quick Sort

67 33 21 84 49 50 75

$\boxed{50\ 33\ 21\ 49}$ 67 $\boxed{84\ 75}$

List1 List2

$\boxed{49\ 33\ 21}$ 50 67 $\boxed{75}$ 84

List3 List4

$\boxed{21\ 33}$ 49 50 67 $\boxed{75}$ 84 pivot (List3)= 49

List5

Quick Sort

21 33 49 50 67 75 84 pivot(list5) = 21

List7

21 33 49 50 67 75 84

21 33 49 50 67 75 84 Sorted list

Quick Sort

```
Algorithm QuickSort (recursive)
    Select a pivot
   Partition the array in two parts
   Part1: Elements less than the pivot
   Part2: Elements greater than the pivot
   QuickSort(part1)
   QuickSort(part2)
End
```

Quick Sort

```
public static void quickSort(int[] list) {
   quickSort(list, 0, list.length-1);
}
// Recursive Helper Method
public static void quickSort(int[] list,
                             int first, int last) {
  if (last > first) {
    int pivotIndex = partition(list, first, last);
    quickSort(list, first, pivotIndex-1);
    quickSort(list, pivotIndex+1, last);
  }
}
```

Quick Sort

```
public static int partition(int list[],
                            int first, int last){
   int pivot;
   int index, pivotIndex;
   pivot = list[first];// the first element
   pivotIndex = first;
   for (index = first + 1; index <= last; index++){
      if (list[index] < pivot){
         pivotIndex++;
         swap(list, pivotIndex, index);
      }
   }
   swap(list, first, pivotIndex);
   return pivotIndex;
}
```

Quick Sort

- Complexity Analysis

```
Partitioning the array in ~ halves
   (log n) iterations (average)

Arranging elements around the pivot
   (n) iterations - worst case
```

Time complexity: average case
O(n log n) - Log Linear

Quick Sort

✦ Complexity Analysis

```
Partitioning the array not in halves
    (n) iterations - worst case

Arranging elements around the pivot
    (n) iterations - worst case
```

Time complexity: $O(n^2)$- worst case

Space complexity: $O(1)$- No additional space

Quick Sort

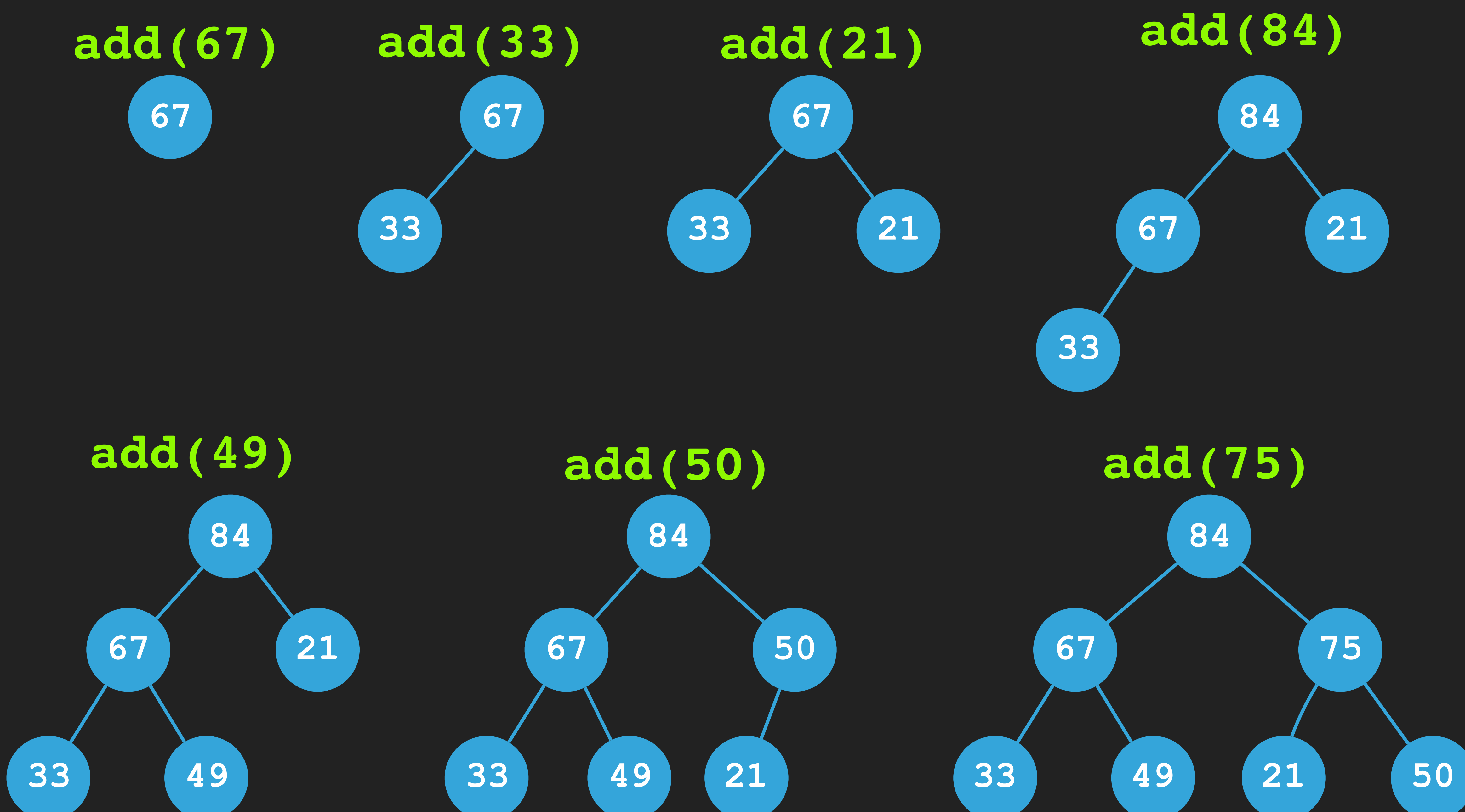
- How to select the pivot?
- Original data is random,
  pivot = first element
- Original data partially sorted, use "middle of the three" rule (median of first, middle, and last)

Heap Sort

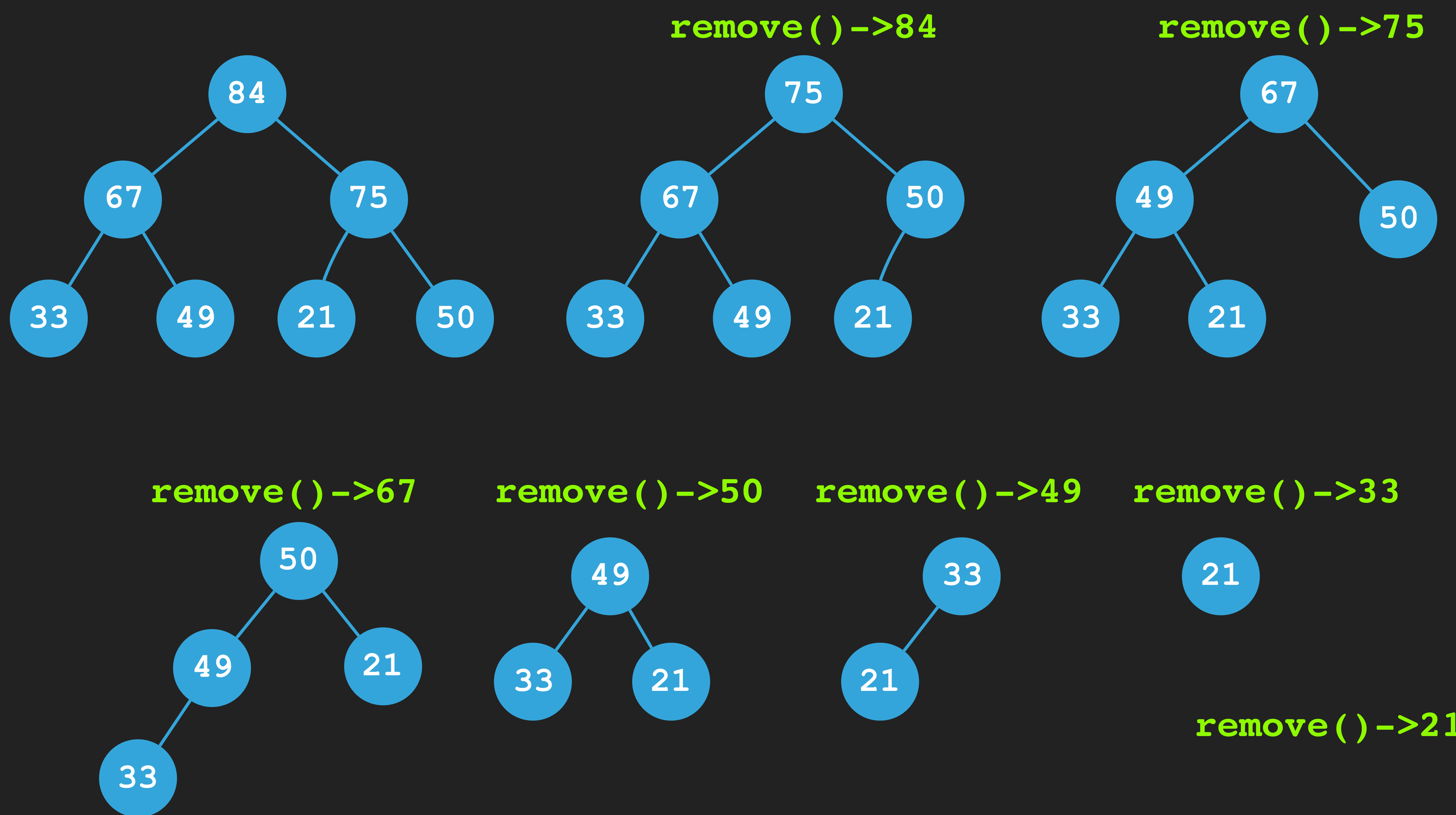
- Sorting using the heap data structure
  - Add the data to be sorted to the heap using **`add()`** method
  - Remove the elements from the heap using **`remove()`** method (elements are removed in ascending/descending order)

Heap Sort

67 33 21 84 49 50 75



Heap Sort



84 75 67 50 49 33 21

Heap Sort

```
Algorithm HeapSort
 create a heap h
 for each element i (0 to N-1) in list{
     h.add(element i)
 }
 for each index i (N-1 to 0) in list{
  list[i] = h.remove()
 }
End
```

Heap Sort

```
public static void heapSort(int[] list) {
    // Create a max heap
    Heap<Integer> heap = new Heap<>();
    // Add the elements of list to the heap
    for(int i=0; i<list.length; i++){
        heap.add(list[i]);
    }
    // Move the data from the heap back to list
    for (int i=list.length-1; i>=0; i--) {
        list[i] = heap.remove();
    }
}
```

Heap Sort

✦ Complexity Analysis (Time)

add() method ***O(log n)***
One path from a leaf to the root

remove() method ***O(log n)***
One path from the root to a leaf

add() and remove() are called **n** times ***O(n)***

Heap Sort: Worst case
O(n log n) - Log Linear

Heap Sort

- Complexity Analysis (Space)

```
 Heap with n nodes required to sort the
array list

Heap Sort space complexity: O(n)
```

Summary (group 2)

	Merge Sort	Quick Sort	Heap Sort
Type	Divide and Conquer Recursive	Divide and Conquer Recursive	Complete Binary Tree
Main task	Merging $O(n)$	Partitioning $O(n)$	Adding/removing nodes to/from heap
Time Complexity	*Worst case:* $O(n \log n)$	Average: $O(n \log n)$ Worst case: $O(n^2)$	Worst case: $O(n \log n)$
Space Complexity	Temporary arrays O(n)	In-place (No additional space)	Heap data structure O(n)

- Sorting algorithms are general - work for any type of data
- Sorting criterion defined in the method **`compareTo()`** or **`compare()`**
- Comparison based sorting cannot perform better than ***`O(n log n)`***
- How can we sort without comparing? Data Classification Algorithms

Bucket Sort

- For integers only
- Range of values to be sorted `[0, t]`
- Use `(t+1)` buckets
- An element equal to **`v`** is put in **`bucket v`**
- A bucket holds all the elements with the same value

Bucket Sort

67 33 21 84 49 50 75

Bucket 0	Bucket 21	Bucket 33	Bucket 49	Bucket 50	Bucket 67	Bucket 75	Bucket 84
NULL ...	21 ...	33 ...	49	50 ...	67 ...	75 ...	84

Bucket Sort

```
Algorithm bucketSort(list)
    create t+1 buckets
         (t is the maximum value in list)

    for each value in list (n)
      Assign value to bucket(value)

    for each bucket i (0 to t)
     Assign the elements of bucket i to list
```

Bucket Sort

```
public static void bucketSort(int[] list) {
    int t = max(list);
    ArrayList<ArrayList<Integer>> buckets;
    buckets = new ArrayList<>(t+1);
    // create t+1 buckets
    for(int i=0; i<t+1; i++)
      buckets.add(new ArrayList<>());

    //Distribute the data on the buckets
    for(int i=0; i<list.length; i++) {
      ArrayList<Integer> bucket = buckets.get(list[i]);
      bucket.add(list[i]);
    }
    // Move the data from the buckets back to the list
    int k = 0;
    for(int i=0; i<buckets.size(); i++) {
      ArrayList<Integer> bucket = buckets.get(i);
      for(int j=0; j<bucket.size(); j++)
        list[k++] = bucket.get(j);
    }
}
```

Bucket Sort

- Complexity Analysis

```
Create the buckets O(t)
Distribute data on the buckets O(n)
Move data from the buckets to list O(t)


Bucket Sort:
 Time Complexity: O(n+t) - Linear
 Space Complexity: O(n+t)

Not practical for large t
```

Radix Sort

- Bucket Sort with fixed number of buckets
- Radix-10: 10 buckets (decimal numbers)
- Divide data into subgroups based on their radix positions
- Bucket Sort is applied on each radix position

Radix Sort

67 33 21 84 49 50 75

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

Radix Sort

- Bucket sort using the ones position (10^0)

67 33 21 84 49 50 75

Bucket 0	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9
50	21		33	84	75		67		49

50 21 33 84 75 67 49

Radix Sort

- Bucket sort using the tens position (10^1)

50 21 33 84 75 67 49

Bucket 0	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9
		21	33	49	50	67	75	84	

21 33 49 50 67 75 84

Radix Sort

```
Algorithm radixSort(list)
   create 10 buckets
   max = number of digits of the largest value in list
   for each digit (0 to max-1){
    for each value in list
     Assign value to the bucket with number
                 (value % 10^(digit+1) / 10^digit)
    for each bucket i (0 to 9)
     Assign the elements of bucket i to list
    clear all buckets
   }
End
```

```
public static void radixSort(int[] list) {
        ArrayList<ArrayList<Integer>> buckets;
        buckets = new ArrayList<>();
        Integer maxValue = max(list);
        int digits = maxValue.toString().length();
        for(int d=0; d<digits; d++) {
            // create 10 buckets
            for(int j=0; j<10; j++) {
                buckets.add(new ArrayList<>());
            }
            //Distribute the data on the buckets
            for(int j=0; j<list.length; j++){
                // find the index of the bucket where list[j] should be placed
                int bucketIndex = (list[j] % (int)(Math.pow(10, d+1))) /
                                             (int)(Math.pow(10,d));
                ArrayList<Integer> bucket = buckets.get(bucketIndex);
                bucket.add(list[j]);
            }
            // Move the data from the buckets back to the list
            int k=0;
            for(int j=0; j<10; j++) {
                ArrayList<Integer> bucket = buckets.get(j);
                for(int l=0; l<bucket.size(); l++)
                    list[k++] = bucket.get(l);
            }
            // clear all the buckets for the next iteration
            buckets.clear();
        }
    }
```

Radix Sort

✦ Complexity Analysis

```
 Classifying the data into buckets O(n)

 Classifying for each position  O(d)

Radix Sort:
 Time Complexity: O(d.n)
 Space complexity: O(n)
 d: maximum number of radix positions
```

Testing the sorting algorithms

```
public class Test{
    public static void main(String[] args){
        int[] list = {67, 33, 21, 84, 49, 50, 75};
        print(list);
        Sort.selectionSort(list);
        System.out.print("Selection Sort: ");
        print(list);
        shuffle(list);
        print(list);
        Sort.insertionSort(list);
        System.out.print("Insertion Sort: ");
        print(list);
        shuffle(list);
        print(list);
        Sort.bubbleSort(list);
        System.out.print("Bubble Sort: ");
        print(list);
        shuffle(list);
        print(list);
        Sort.mergeSort(list);
        System.out.print("Merge Sort: ");
        print(list);
        shuffle(list);
        print(list);
        Sort.quickSort(list);
        System.out.print("Quick Sort: ");
        print(list);
        shuffle(list);
        print(list);
        Sort.heapSort(list);
        System.out.print("Heap Sort: ");
        print(list);
        shuffle(list);
        print(list);
        Sort.bucketSort(list);
        System.out.print("Bucket Sort: ");
        print(list);
        shuffle(list);
        print(list);
        Sort.radixSort(list);
        System.out.print("Radix Sort: ");
        print(list);
    }
}
```

Summary

	Quadratic	Log Linear			Linear		
Sorting Algorithm	**Selection Insertion Bubble Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**Bucket Sort (t buckets)**	**Radix Sort (d digits)**	**External Merge Sort**
Type	Exchange	Divide and Conquer		Binary Tree	Data Classification		Divide and Conquer
Time	$O(n^2)$	$O(n \log n)$	$O(n \log n)$ to $O(n^2)$	$O(n \log n)$	**O(n+t)**	**O(d.n)**	$O(n \log n)$
Space	No additional space	Require temporary array **O(n)**	No additional space	Heap **O(n)**	Require buckets		Require temporary files **O(n)**