

# Sorting

Paul Fodor

CSE260, Computer Science B: Honors

Sony Brook University

<http://www.cs.stonybrook.edu/~cse260>

# Objectives

- To study and analyze time complexity of various sorting algorithms
  - To design, implement, and analyze *insertion sort*
  - To design, implement, and analyze *bubble sort*
  - To design, implement, and analyze *merge sort*
  - To design, implement, and analyze *quick sort*
  - To design and implement a *binary heap*
  - To design, implement, and analyze *heap sort*
  - To design, implement, and analyze *bucket sort* and *radix sort*
  - To design, implement, and analyze *external sort* for files that have a large amount of data

# What data to sort?

- The data to be sorted might be integers, doubles, characters, or objects that are comparable.
- For simplicity, this section assumes:
  - data to be sorted are *integers*
  - data are sorted in *ascending* order
  - data are stored in an *array*
    - The programs can be easily modified to sort other types of data, to sort in descending order, or to sort data in an **ArrayList** or a **LinkedList**.
- The Java API contains several overloaded sort methods for sorting primitive type values and objects in the **java.util.Arrays** and **java.util.Collections** class.

# Insertion Sort

```
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted
```

The *insertion sort* algorithm sorts a list of values by repeatedly inserting an unsorted element into a sorted sublist until the whole list is sorted.

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.



Step 2: The sorted sublist is {2, 9}. Insert 5 into the sublist.



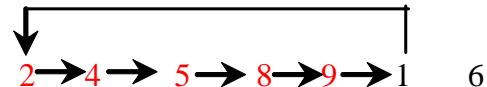
Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.



Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.



Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.



Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.



Step 7: The entire list is now sorted.



# How to Insert 4 in sorted order?

list    [0] [1] [2] [3] [4] [5] [6]  
      [ 2   5   9   4 ]

Step 1: Save 4 to a temporary variable `currentElement`

list    [0] [1] [2] [3] [4] [5] [6]  
      [ 2   5       9 ]

Step 2: Move `list[2]` to `list[3]`

list    [0] [1] [2] [3] [4] [5] [6]  
      [ 2       5   9 ]

Step 3: Move `list[1]` to `list[2]`

list    [0] [1] [2] [3] [4] [5] [6]  
      [ 2   4   5   9 ]

Step 4: Assign `currentElement` to `list[1]`

# InsertionSort - From Idea to Solution

```
for (int i = 1; i < list.length; i++) {  
    insert list[i] into a sorted sublist list[0..i-1] so that  
    list[0..i] is sorted  
}
```

Expand



```
int currentElement = list[i];  
int k;  
for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {  
    list[k + 1] = list[k];  
}  
// Insert the current element into list[k + 1]  
list[k + 1] = currentElement;
```

# InsertionSort - From Idea to Solution

```
public static void insertionSort(int[] list) {  
    for (int i = 1; i < list.length; i++) {  
        int currentElement = list[i];  
        int k;  
        for (k = i - 1; k >= 0 && list[k] > currentElement; k--)  
            list[k + 1] = list[k];  
        // Insert the current element into list[k + 1]  
        list[k + 1] = currentElement;  
    }  
}  
  
int[] list = {1, 9, 4, 6, 5, -4};  
InsertionSort.insertionSort(list);
```

To insert the first element, we need 1 comparison and at most 1 swap

To insert the last element, we need  $n-1$  comparisons and at most  $n-1$  swaps

$$\begin{aligned} T(n) &= (2 + c) + (2 \times 2 + c) + \cdots + (2 \times (n - 1) + c) \\ &= 2(1 + 2 + \cdots + n - 1) + c(n - 1) \\ &= 2 \frac{(n - 1)n}{2} + cn - c = n^2 - n + cn - c \\ &= O(n^2) \end{aligned}$$

```

public class InsertionSort {
    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        long startTime = System.currentTimeMillis();
        insertionSort(a);
        long endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + "ms");
    }
    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
    public static void insertionSort(int[] list) {
        for (int i = 1; i < list.length; i++) {
            int currentElement = list[i];
            int k;
            for (k = i - 1; k >= 0 && list[k] > currentElement; k--)
                list[k + 1] = list[k];
            // Insert the current element into list[k + 1]
            list[k + 1] = currentElement;
        }
    }
}

```

3641ms



# Bubble Sort

- Repeatedly steps through the list to be sorted, compares each **pair of adjacent items** and **swaps them if they are in the wrong order**
  - After the first pass, the last element becomes the largest in the array
  - After the second pass, the second-to-last element becomes the second largest in the array
    - It is called "bubble" sort because the large values gradually "*bubble*" to the top (end of the array)
    - It is also called "*sinking sort*" because the smaller values gradually "*sink*" their way to the bottom (beginning of the array)

```
for (int k = 1; k < list.length; k++) {  
    // Perform the kth pass  
    for (int i = 0; i < list.length - k; i++) {  
        if (list[i] > list[i + 1])  
            // swap list[i] with list[i + 1];  
            ...  
    }  
}
```

# Bubble Sort Optimization

- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted
  - If no swap takes place in a pass, there is no need to perform the next pass, because all the elements are already sorted!
  - We can use this property to improve the previous algorithm:

```
boolean needNextPass = true;
for (int k = 1; k < list.length && needNextPass; k++) {
    // Array may be sorted and next pass not needed
    needNextPass = false;
    // Perform the kth pass
    for (int i = 0; i < list.length - k; i++) {
        if (list[i] > list[i + 1]) {
            // swap list[i] with list[i + 1];
            needNextPass = true; // Next pass still needed
        }
    }
}
```

# Bubble Sort Example

- Example:

2 9 5 4 8 1	2 5 4 8 1 9	2 4 5 1 8 9	2 4 1 5 8 9	1 2 4 5 8 9
2 5 9 4 8 1	2 4 5 8 1 9	2 4 5 1 8 9	2 1 4 5 8 9	
2 5 4 9 8 1	2 4 5 8 1 9	2 4 1 5 8 9		
2 5 4 8 9 1	2 4 5 1 8 9			
2 5 4 8 1 9				
(a) 1st pass	(b) 2nd pass	(c) 3rd pass	(d) 4th pass	(e) 5th pass

- In the **best case**, the bubble sort algorithm needs just the first pass to find that the array is already sorted—no next pass is needed. Since the number of comparisons is  $n - 1$  in the first pass, the best-case time for a bubble sort is  $O(n)$ .

- Worse case:**  $(n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n^2}{2} - \frac{n}{2}$   
time:  $O(n^2)$

```

public class BubbleSort {
    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        long startTime = System.currentTimeMillis();
        bubbleSort(a);
        long endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + "ms");
    }
    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
    public static void bubbleSort(int[] list) {
        boolean needNextPass = true;
        for (int k = 1; k < list.length && needNextPass; k++) {
            // Array may be sorted and next pass not needed
            needNextPass = false;
            // Perform the kth pass
            for (int i = 0; i < list.length - k; i++) {
                if (list[i] > list[i + 1]) {
                    int temp = list[i];
                    list[i] = list[i + 1];
                    list[i + 1] = temp;
                    needNextPass = true; // Next pass still needed
                }
            }
        }
    }
}

```

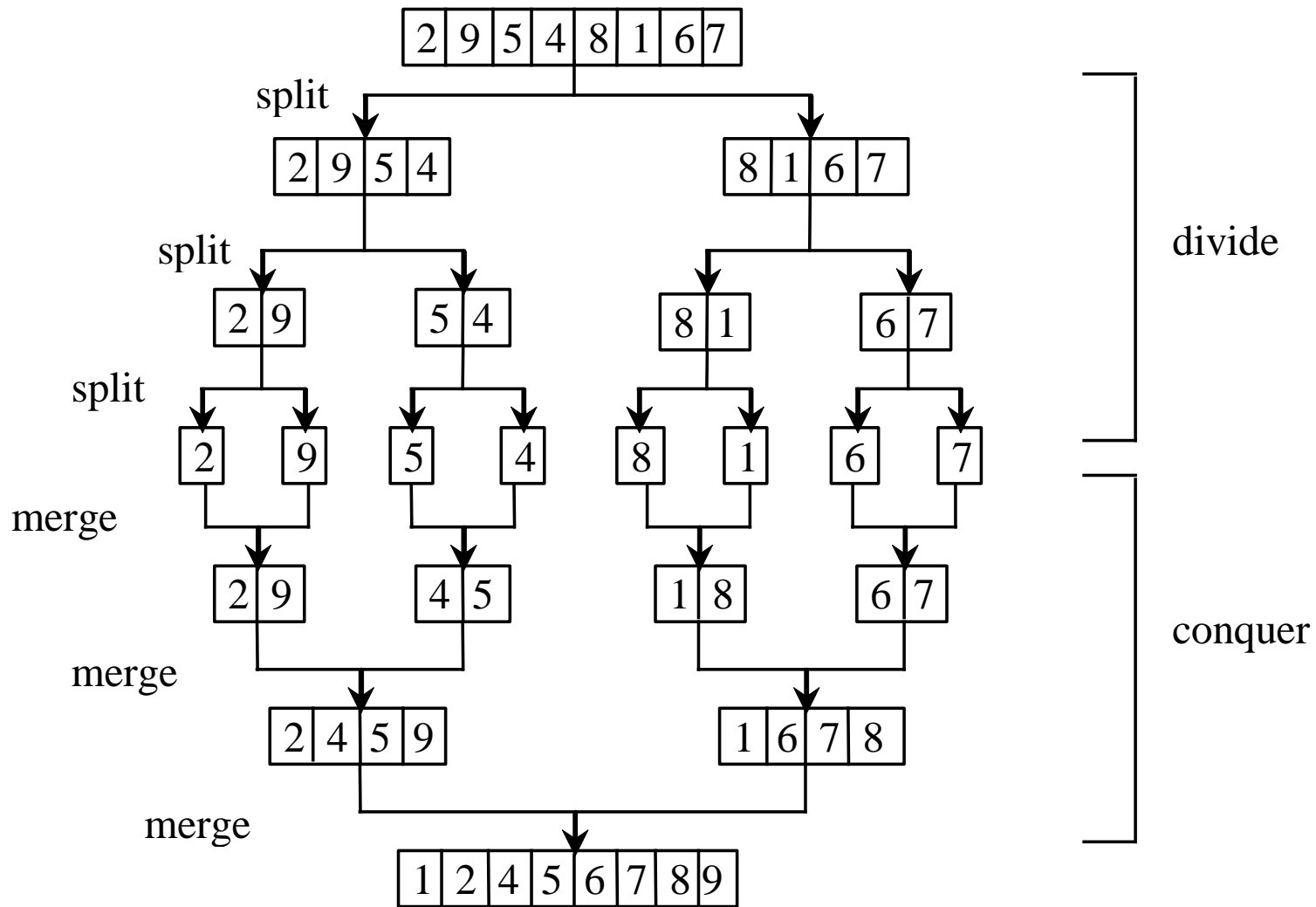
14650ms

# Merge Sort

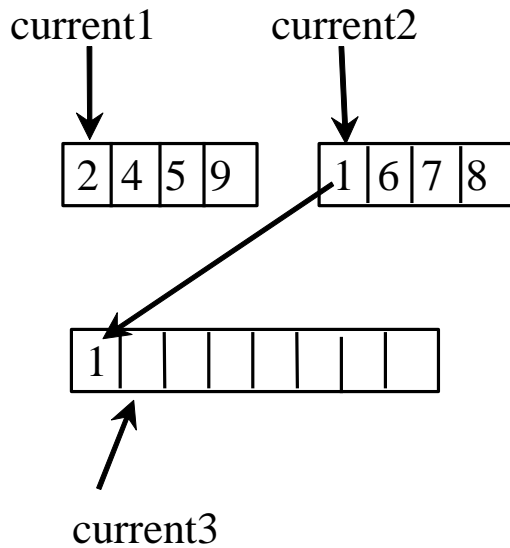
- *Merge sort* is a **divide and conquer algorithm** that was invented by John von Neumann in 1945
  - Divide the unsorted list into  **$n$**  sublists, each containing 1 element (a list of 1 element is considered sorted)
  - Repeatedly **merge** sorted sublists to produce new sorted sublists until there is only 1 sublist remaining
    - This will be the sorted list.



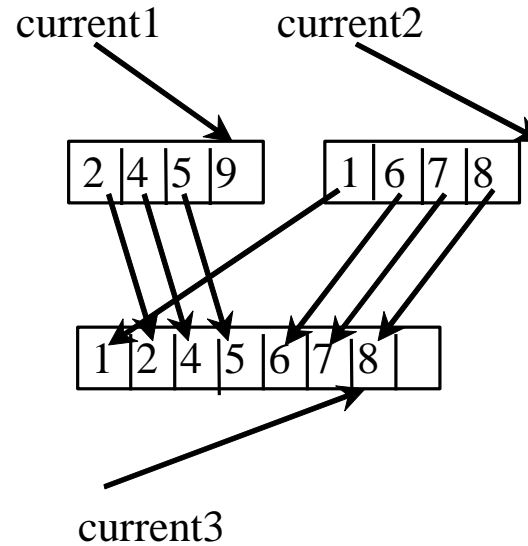
# Merge Sort



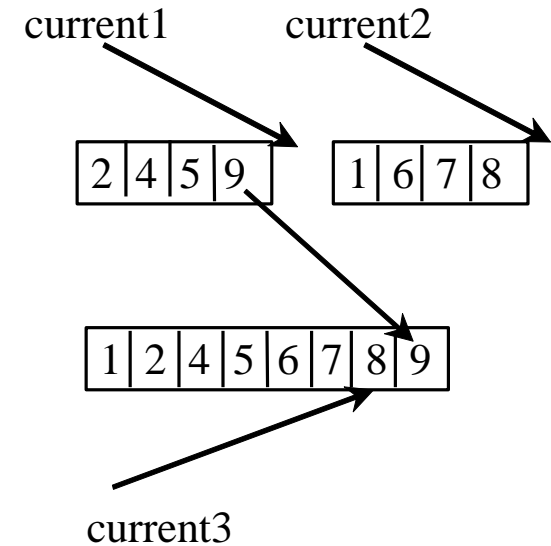
# How to Merge Two Sorted Lists



(a) After moving 1 to temp



(b) After moving all the elements in list2 to temp



(c) After moving 9 to temp

# Merge Sort Recursive pseudocode

```
mergeSort(list) :
```

```
    firstHalf = mergeSort(firstHalf) ;
```

```
    secondHalf = mergeSort(secondHalf) ;
```

```
    list = merge(firstHalf, secondHalf) ;
```



```

public static void mergeSort(int[] list) {
    if (list.length > 1) {
        // Merge sort the first half
        int[] firstHalf = new int[list.length / 2];
        System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
        mergeSort(firstHalf);

        // Merge sort the second half
        int secondHalfLength = list.length - list.length / 2;
        int[] secondHalf = new int[secondHalfLength];
        System.arraycopy(list, list.length / 2, secondHalf, 0,
            secondHalfLength);
        mergeSort(secondHalf);

        // Merge firstHalf with secondHalf into list
        merge(firstHalf, secondHalf, list);
    }
}

```

```
/** Merge two sorted lists */
public static void merge(int[] list1, int[] list2, int[] temp){
    int current1 = 0; // Current index in list1
    int current2 = 0; // Current index in list2
    int current3 = 0; // Current index in temp

    while (current1 < list1.length && current2 < list2.length) {
        if (list1[current1] < list2[current2])
            temp[current3++] = list1[current1++];
        else
            temp[current3++] = list2[current2++];
    }

    while (current1 < list1.length)
        temp[current3++] = list1[current1++];

    while (current2 < list2.length)
        temp[current3++] = list2[current2++];
}
```

# Example:

```
public static void main(String[] args) {  
    int[] list = {14,12,2,3,2,-2,1,3,6,5};  
    mergeSort(list);  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + ", ");  
}
```

-2, 1, 2, 2, 3, 3, 5, 6, 12, 14

```

public class MergeSort {
    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        long startTime = System.currentTimeMillis();
        mergeSort(a);
        long endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + "ms");
    }
    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
    public static void mergeSort(int[] list) {
        if (list.length > 1) {
            // Merge sort the first half
            int[] firstHalf = new int[list.length / 2];
            System.arraycopy(list, 0, firstHalf, 0, list.length / 2);
            mergeSort(firstHalf);

            // Merge sort the second half
            int secondHalfLength = list.length - list.length / 2;
            int[] secondHalf = new int[secondHalfLength];
            System.arraycopy(list, list.length / 2, secondHalf, 0, secondHalfLength);
            mergeSort(secondHalf);

            // Merge firstHalf with secondHalf into list
            merge(firstHalf, secondHalf, list);
        }
    }
}

```

```
public static void merge(int[] list1, int[] list2, int[] temp) {  
    int current1 = 0; // Current index in list1  
    int current2 = 0; // Current index in list2  
    int current3 = 0; // Current index in temp  
  
    while (current1 < list1.length && current2 < list2.length) {  
        if (list1[current1] < list2[current2])  
            temp[current3++] = list1[current1++];  
        else  
            temp[current3++] = list2[current2++];  
    }  
  
    while (current1 < list1.length)  
        temp[current3++] = list1[current1++];  
  
    while (current2 < list2.length)  
        temp[current3++] = list2[current2++];  
}
```

16ms

# Merge Sort Time

- Let  $T(n)$  denote the time required for sorting an array of  $n$  elements using merge sort.
- Without loss of generality, assume  $n$  is a power of 2.
- The merge sort algorithm splits the array into two subarrays, sorts the subarrays using the same algorithm recursively, and then merges the subarrays

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \textit{mergetime}$$

- The first  $T(n/2)$  is the time for sorting the first half of the array and the second  $T(n/2)$  is the time for sorting the second half

# Merge Sort Time

- To merge two sorted subarrays, it takes at most  $n-1$  comparisons to compare the elements from the two subarrays and  $n$  moves to move elements to the temporary array

$$\begin{aligned}T(n) &= 2T\left(\frac{n}{2}\right) + 2n - 1 = 2\left(2T\left(\frac{n}{4}\right) + 2\frac{n}{2} - 1\right) + 2n - 1 \\&= 2^2 T\left(\frac{n}{2^2}\right) + 2n - 2 + 2n - 1 \\&= 2^k T\left(\frac{n}{2^k}\right) + 2n - 2^{k-1} + \dots + 2n - 2 + 2n - 1 \\&= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + 2n - 2^{\log n - 1} + \dots + 2n - 2 + 2n - 1 \\&= n + 2n \log n - 2^{\log n} + 1 \\&= 2n \log n + 1 \\&= O(n \log n)\end{aligned}$$

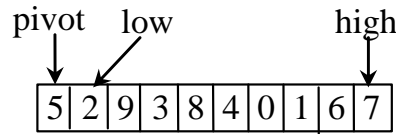
# Quick Sort

- *Quick sort*, developed by C. A. R. Hoare in 1962, works as follows:
  - The algorithm selects an element, called the *pivot*, in the array
  - **Divide** the array into two parts such that all the elements in the first part are **less** than or equal to the pivot and all the elements in the second part are **greater** than the pivot
  - Recursively apply the quick sort algorithm to the first part and then the second part

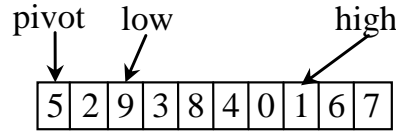




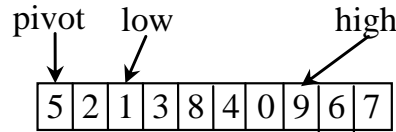
# Partition with forward and backward search



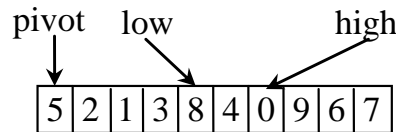
(a) Initialize pivot, low, and high



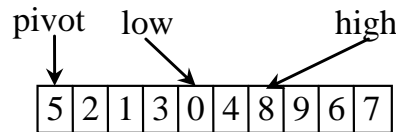
(b) Search forward and backward



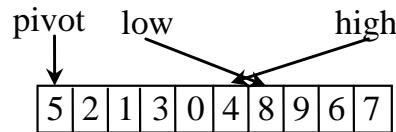
(c) 9 is swapped with 1



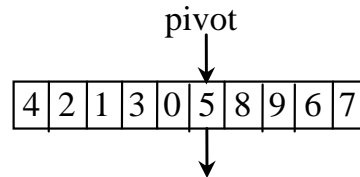
(d) Continue search



(e) 8 is swapped with 0



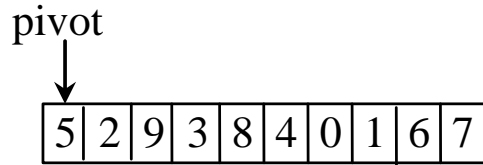
(f) when  $\text{high} < \text{low}$ , search is over



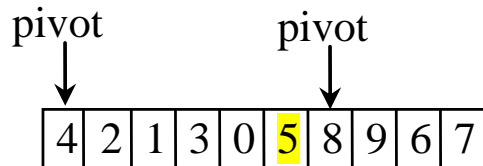
(g) pivot is in the right place

The index of the pivot is returned

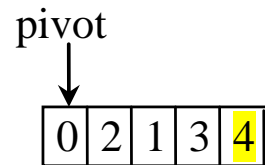
# Quick Sort



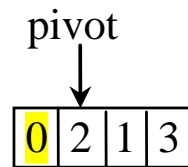
(a) The original array



(b) The original array is partitioned



(c) The partial array (4 2 1 3 0) is partitioned



(d) The partial array (0 2 1 3) is partitioned



(e) The partial array (2 1 3) is partitioned

```

public static void quickSort(int[] list) {
    quickSort(list, 0, list.length - 1);
}

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);
    }
}

public static int partition(int[] list, int first, int last) {
    int pivot = list[first]; // Choose the first element as pivot
    int low = first + 1; // Index for forward search
    int high = last; // Index for backward search
    while (high > low) {
        // Search forward from left
        while (low <= high && list[low] <= pivot)
            low++;
        // Search backward from right
        while (low <= high && list[high] > pivot)
            high--;
    }
}

```

```

// Swap two elements in the list
if (high > low) {
    int temp = list[high];
    list[high] = list[low];
    list[low] = temp;
}
}

// Account for duplicated elements:
while (high > first && list[high] >= pivot)
    high--;

// Swap pivot with list[high]
if (pivot > list[high]) {
    list[first] = list[high];
    list[high] = pivot;
    return high;
} else {
    return first;
}
}

```

```

public class QuickSort {
    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);

        long startTime = System.currentTimeMillis();
        quickSort(a);
        long endTime = System.currentTimeMillis();

        System.out.println((endTime - startTime) + "ms");
    }
    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
    public static void quickSort(int[] list) {
        quickSort(list, 0, list.length - 1);
    }
    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);
        }
    }
    public static int partition(int[] list, int first, int last) {
        int pivot = list[first]; // Choose the first element as pivot
        int low = first + 1; // Index for forward search
        int high = last; // Index for backward search
        while (high > low) {
            // Search forward from left
            while (low <= high && list[low] <= pivot)
                low++;

```

```

// Search backward from right
while (low <= high && list[high] > pivot)
    high--;
// Swap two elements in the list
if (high > low) {
    int temp = list[high];
    list[high] = list[low];
    list[low] = temp;
}
}

// Account for duplicated elements:
while (high > first && list[high] >= pivot)
    high--;

// Swap pivot with list[high]
if (pivot > list[high]) {
    list[first] = list[high];
    list[high] = pivot;
    return high;
} else {
    return first;
}
}
}

```

16ms

# Quick Sort Time

- To **partition** an array of  **$n$**  elements, it takes  **$n-1$**  comparisons and  **$n$**  moves in the worst case
- So, the time required for partition is  **$O(n)$**

# Worst-Case Time

- In the worst case, each time the pivot divides the array into one big subarray with the other empty
- The size of the big subarray is one less than the one before divided
- The algorithm requires:

$$(n-1) + (n-2) + \dots + 2 + 1 = O(n^2)$$



# Best-Case Time

- In the best case, each time the pivot divides the array into two parts of about the same size
- Let  $T(n)$  denote the time required for sorting an array of  $n$  elements using quick sort. So,

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n = O(n \log n)$$

# Average-Case Time

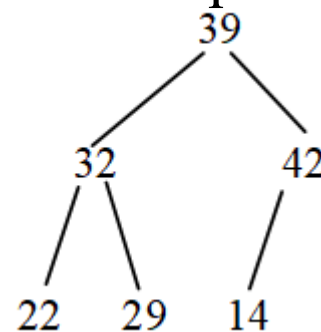
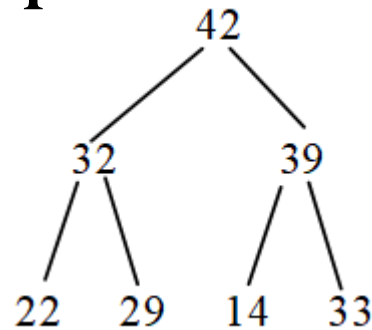
- On the average, each time the pivot will not divide the array into two parts of the same size nor one empty part
- Statistically, the sizes of the two parts are very close  $\Rightarrow$  the average time is also  $O(n \log n)$
- Both merge sort and quick sort employ the divide-and-conquer approach
  - For merge sort, the bulk of the work is to merge two sublists
  - Merge sort is more efficient than quick sort in the worst case, but the two are equally efficient in the average case
  - Merge sort requires a temporary array for sorting two subarrays
  - **Quick sort does not need additional array space. Thus, quick sort is more space efficient than merge sort.**

# Binary tree

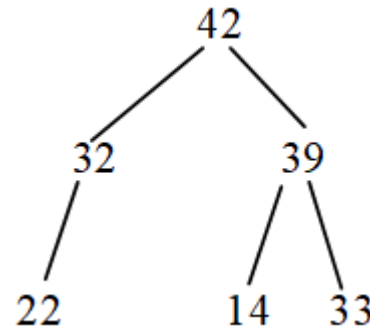
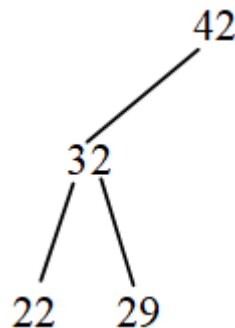
- A *binary tree* is a hierarchical structure: it either is empty or it consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*
  - The *length* of a path is the number of the edges in the path
  - The *depth* of a node is the length of the path from the root to that node

# Complete Binary Tree

- A binary tree is *complete* if every level of the tree is full except that the **last level may not be full** and **all the leaves on the last level are placed left-most**. Examples of complete binary trees:

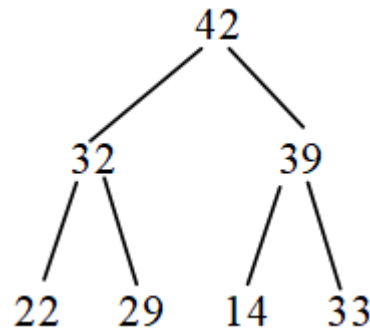


- Not complete:

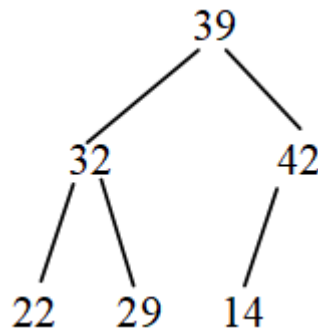


# Binary Heap

- A *binary heap* is a binary tree with the following properties:
  - It is a complete binary tree, and
  - Each node is greater than or equal to any of its children
    - Example:



- Example: not a heap, because the root (39) is less than its right child (42).

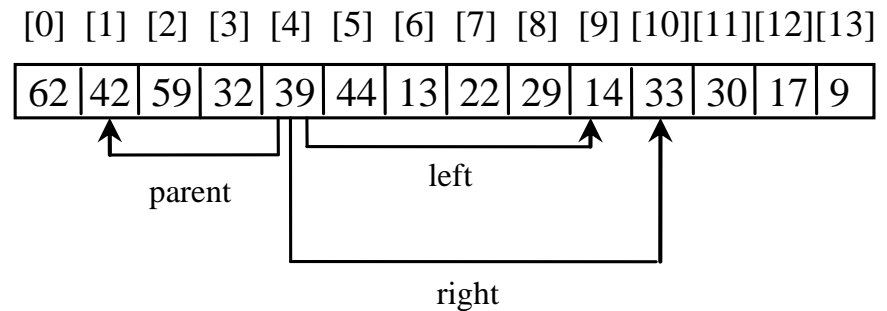
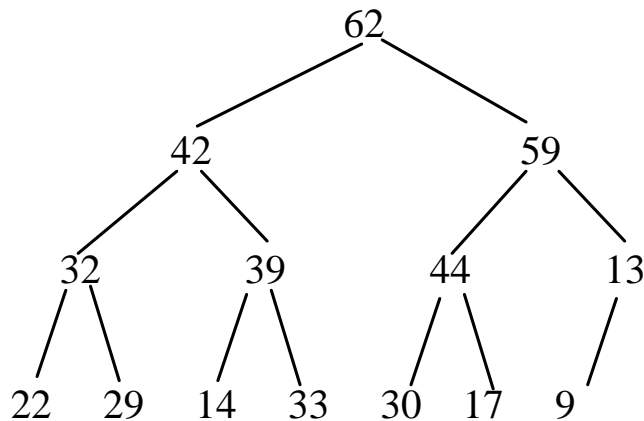


# Heap Sort

- *Heap sort* uses a binary heap: it first adds all the elements to a heap and then removes the largest elements successively to obtain a sorted list
- Heap is a useful data structure for designing efficient sorting algorithms and priority queues.

# Storing a Heap

- A heap can be stored in an **ArrayList** or an array if the heap size is known in advance
- For a node at position  **$i$** , its left child is at position  **$2i+1$**  and its right child is at position  **$2i+2$** , and its parent is at index  **$(i-1)/2$** 
  - For example: the root is at position **0**, and its two children are at positions **1** and **2**
    - The node for element **39** is at position **4**, so its left child (element **14**) is at **9** ( **$2*4+1$** ), its right child (element **33**) is at **10** ( **$2*4+2$** ), and its parent (element **42**) is at **1** ( **$(4-1)/2$** ).



# Adding Elements to the Heap

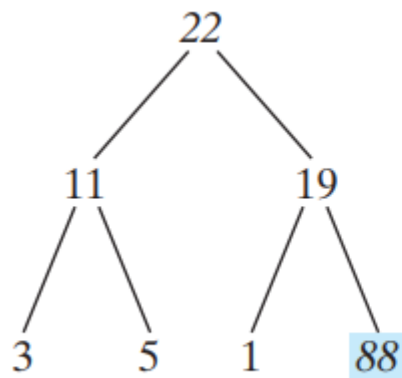
- To add a new node to the heap, first add it to the end of the heap and then rebuild the tree with this algorithm:

```
Let the last node be the current node;  
while (the current node is greater than its parent) {  
    Swap the current node with its parent;  
    Now the current node is one level up;  
}
```

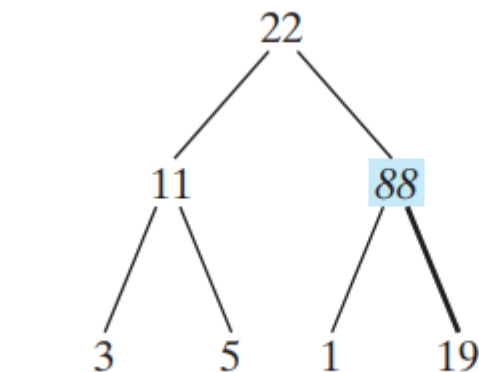


# Adding Elements to the Heap

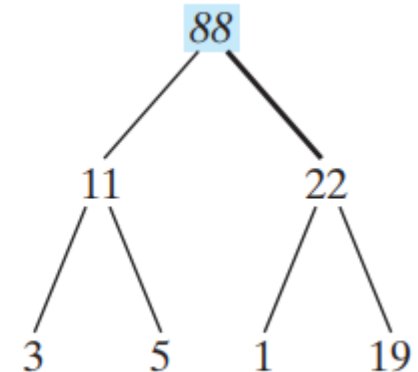
- Adding 88 into a heap:
  - Place the new node 88 at the end of the tree
  - Swap 88 with 19
  - Swap 88 with 22



(a) Add 88 to a heap



(b) After swapping 88 with 19



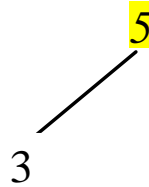
(c) After swapping 88 with 22

# Adding Elements to the Heap

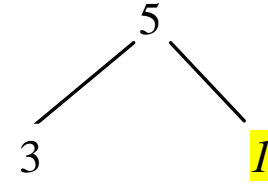
- Suppose a heap is initially empty. after adding numbers 3, 5, 1, 19, 11, and 22 in this order

3

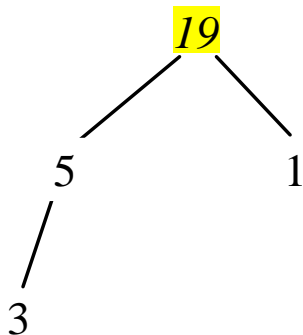
(a) After adding 3



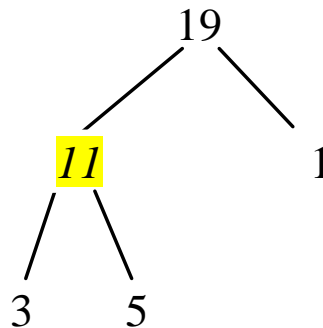
(b) After adding 5



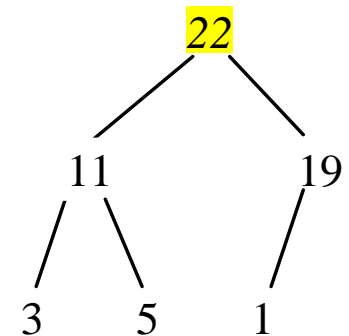
(c) After adding 1



(d) After adding 19



(e) After adding 11



(f) After adding 22

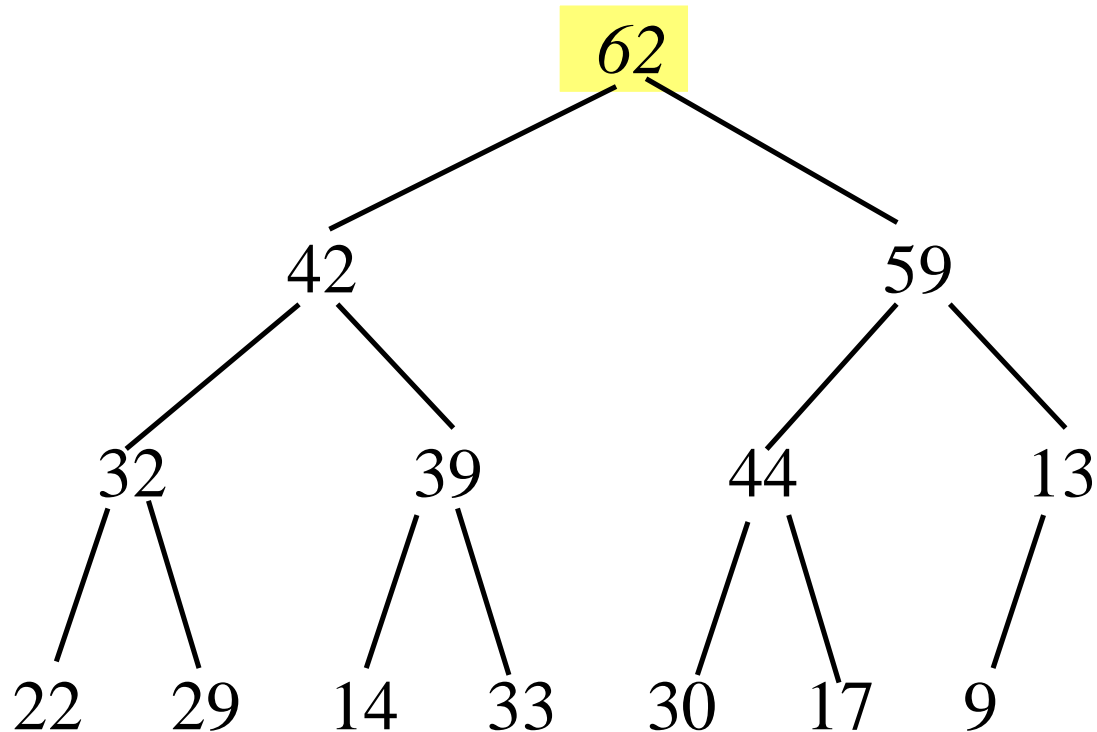
# Removing the Root and Rebuild the Heap

- Often you need to remove the maximum element, which is the root in a heap
  - After the root is removed, the tree must be rebuilt to maintain the heap property using this algorithm:

```
Move the last node to replace the root;  
Let the root be the current node;  
while (the current node has children and the  
current node is smaller than one of its children) {  
    Swap the current node with the larger of its  
    children;  
    Now the current node is one level down;  
}
```

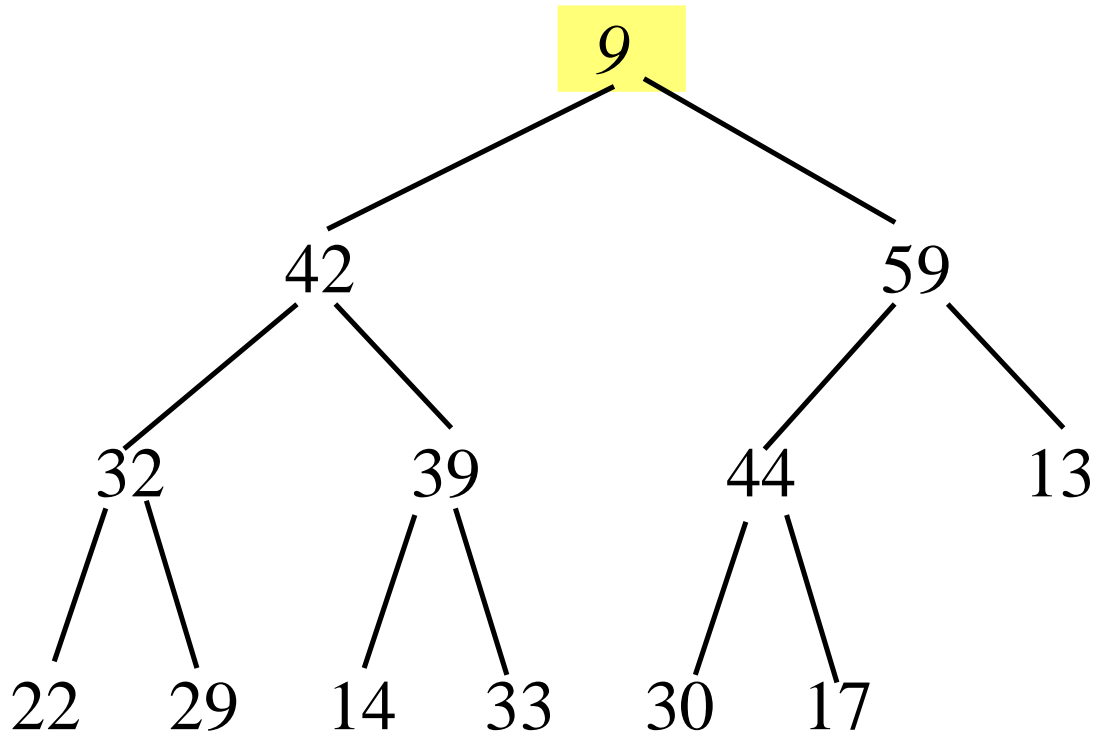
# Removing the Root and Rebuild the Heap

- Removing root 62 from the heap

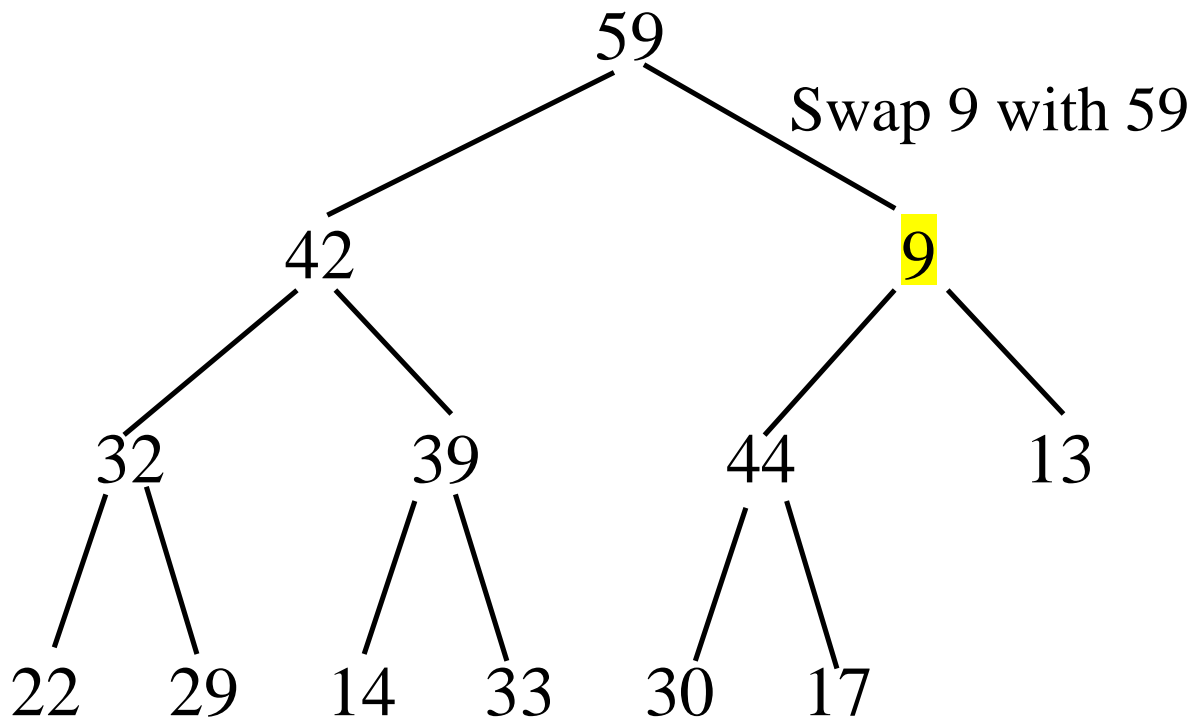


# Removing the Root and Rebuild the Heap

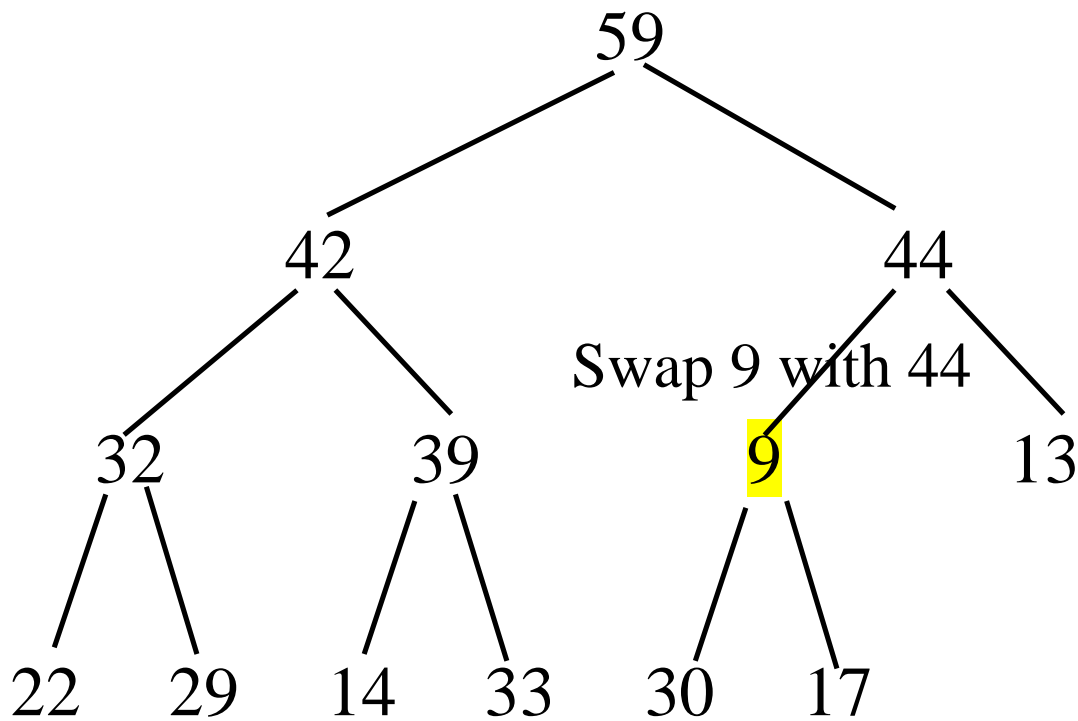
Move 9 to root



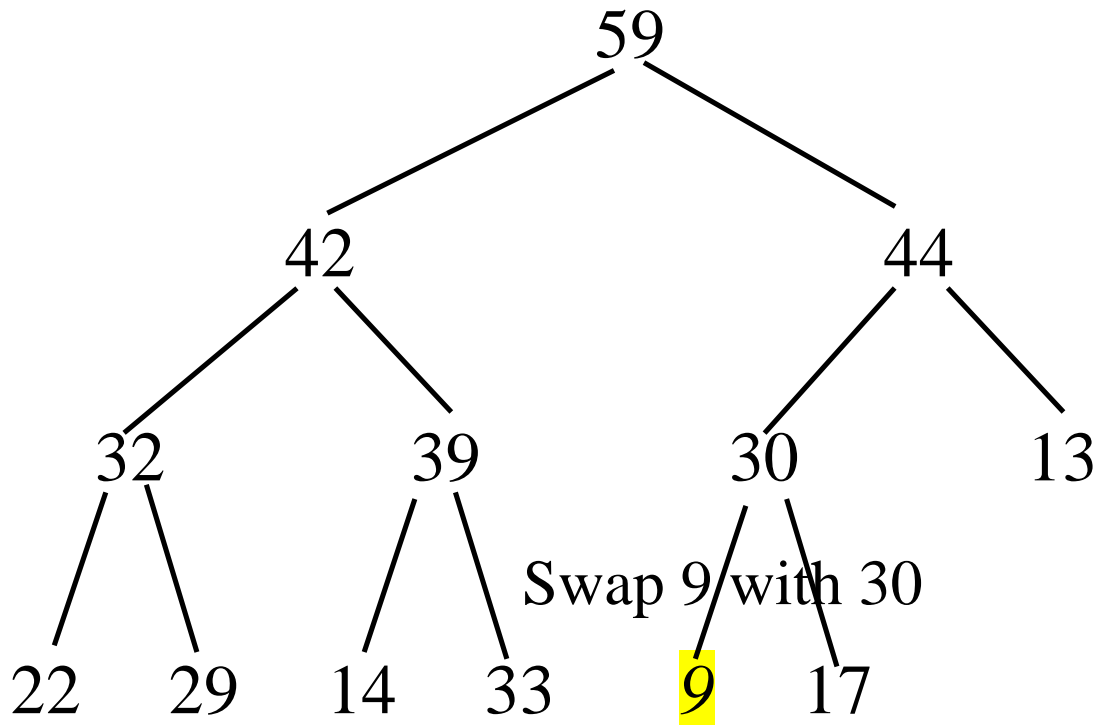
# Removing the Root and Rebuild the Heap



# Removing the Root and Rebuild the Heap



# Removing the Root and Rebuild the Heap





# The Heap Class

**Heap<E extends Comparable<E>>**

-list: java.util.ArrayList<E>

+Heap()  
+Heap(objects: E[])  
+add(newObject: E): void  
+remove(): E  
+getSize(): int

Creates a default empty heap.

Creates a heap with the specified objects.

Adds a new object to the heap.

Removes the root from the heap and returns it.

Returns the size of the heap.

```

public class Heap<E extends Comparable> {
    private java.util.ArrayList<E> list = new java.util.ArrayList<E>();
    /** Create a default heap */
    public Heap() {
    }
    /** Create a heap from an array of objects */
    public Heap(E[] objects) {
        for (int i = 0; i < objects.length; i++)
            add(objects[i]);
    }
    /** Add a new object into the heap */
    public void add(E newObject) {
        list.add(newObject); // Append to the heap
        int currentIndex = list.size() - 1; // The index of the last node
        while (currentIndex > 0) {
            int parentIndex = (currentIndex - 1) / 2;
            // Swap if the current object is greater than its parent
            if (list.get(currentIndex).compareTo(
                list.get(parentIndex)) > 0) {
                E temp = list.get(currentIndex);
                list.set(currentIndex, list.get(parentIndex));
                list.set(parentIndex, temp);
            } else break; // the tree is a heap now
            currentIndex = parentIndex;
        }
    }
}

```

```

/** Remove the root from the heap */
public E remove() {
    if (list.size() == 0) return null;

    E removedObject = list.get(0);
    list.set(0, list.get(list.size() - 1));
    list.remove(list.size() - 1);

    int currentIndex = 0;
    while (currentIndex < list.size()) {
        int leftChildIndex = 2 * currentIndex + 1;
        int rightChildIndex = 2 * currentIndex + 2;

        // Find the maximum between two children
        if (leftChildIndex >= list.size()) break; // The tree is a heap
        int maxIndex = leftChildIndex;
        if (rightChildIndex < list.size()) {
            if (list.get(maxIndex).compareTo(
                list.get(rightChildIndex)) < 0) {
                maxIndex = rightChildIndex;
            }
        }
    }
}

```

```

// Swap if the current node is less than the maximum
if (list.get(currentIndex).compareTo(
    list.get(maxIndex)) < 0) {
    E temp = list.get(maxIndex);
    list.set(maxIndex, list.get(currentIndex));
    list.set(currentIndex, temp);
    currentIndex = maxIndex;
}
else
    break; // The tree is a heap
}

return removedObject;
}

/** Get the number of nodes in the tree */
public int getSize() {
    return list.size();
}
}

```

# Heap Sort

```
public class HeapSort {
    public static <E extends Comparable> void heapSort(E[] list) {
        // Create a Heap of integers
        Heap<E> heap = new Heap<E>();

        // Add elements to the heap
        for (int i = 0; i < list.length; i++)
            heap.add(list[i]);

        // Remove elements from the heap
        for (int i = list.length - 1; i >= 0; i--)
            list[i] = heap.remove();
    }

    /** A test method */
    public static void main(String[] args) {
        Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
        heapSort(list);
        for (int i = 0; i < list.length; i++)
            System.out.print(list[i] + " ");
    }
}
```

```

public class Sorting {
    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        Integer[] b = new Integer[a.length];
        for(int i=0; i<b.length; i++)
            b[i] = a[i];

        long startTime = System.currentTimeMillis();
        HeapSort.heapSort(b);
        long endTime = System.currentTimeMillis();

        System.out.println((endTime - startTime) + "ms");
    }

    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
}

```

76ms

# Heap Sort Time

- Let  $h$  denote the height for a heap of  $n$  elements. Since a heap is a complete binary tree, the first level has 1 node, the second level has 2 nodes, the  $k$ th level has  $2^{(k-1)}$  nodes, the  $(h-1)$ th level has  $2^{(h-2)}$  nodes, and the  $h$ th level has at least one node and at most  $2^{(h-1)}$  nodes. Therefore, the number of nodes  $n$  is:

$$1 + 2 + \dots + 2^{h-2} < n \leq 1 + 2 + \dots + 2^{h-2} + 2^{h-1}$$

$$2^{h-1} - 1 < n \leq 2^h - 1$$

$$2^{h-1} < n + 1 \leq 2^h$$

$$\log 2^{h-1} < \log(n + 1) \leq \log 2^h$$

$$h - 1 < \log(n + 1) \leq h$$

- Thus,  $\log(n + 1) \leq h < \log(n + 1) + 1$
- Hence, the height of the heap is  **$O(\log n)$**

# Heap Add and Remove Time

- Since the add method traces a path from a leaf to a root, it takes at most  $h = \log n$  steps to add a new element to the heap.
- Thus, the total time for constructing an initial heap is  $O(n \log n)$  for an array of  $n$  elements.
- Since the remove method traces a path from a root to a leaf, it takes at most  $h = \log n$  steps to rebuild a heap after removing the root from the heap.



# Heap Sort Time

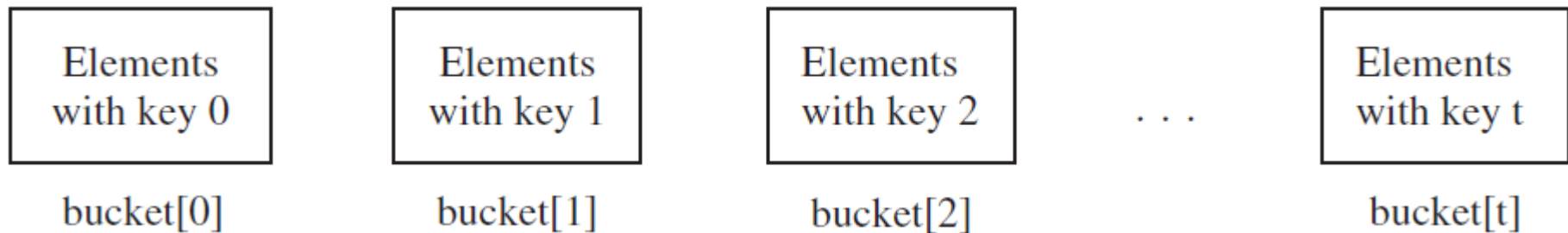
- Heap Sort Time: since the remove method is invoked  $n$  times, the total time for producing a sorted array from a heap is  $O(n \log n)$
- Merge sort requires a temporary array for merging two subarrays; a heap sort does not need additional array space.
- Therefore, a heap sort is more space efficient than a merge sort.

# Bucket Sort and Radix Sort

- All sort algorithms discussed so far are general sorting algorithms that work for **any types of keys** (e.g., integers, strings, and any comparable objects)
  - These algorithms sort the elements by comparing their keys
  - The lower bound for general sorting algorithms is  $O(n \log n)$
  - So, no sorting algorithms based on comparisons can perform better than  $O(n \log n)$
- However, if the keys are small integers, you can use bucket sort without having to compare the keys

# Bucket Sort

- The *bucket sort* algorithm works as follows:
  - Assume the keys are in the range from **0** to **t**
  - We need **t+1** buckets labeled **0**, **1**, ..., and **t**
  - If an element's key is **i**, the element is put into the bucket **i**
  - Each bucket holds the elements with the same key value



- You can use an **ArrayList** to implement a bucket element

# Bucket Sort

```
public static <E> void bucketSort(E[] list) {
    java.util.ArrayList<E>[] bucket = new java.util.ArrayList[t+1];
    // Distribute the elements from list to buckets
    for (int i = 0; i < list.length; i++) {
        // Assume element has the getKey() method
        int key = getKey(list[i]); // list[i].getKey()
        if (bucket[key] == null)
            bucket[key] = new java.util.ArrayList<E>();
        bucket[key].add(list[i]);
    }
    // Now move the elements from the buckets back to list
    int k = 0; // k is an index for list
    for (int i = 0; i < bucket.length; i++) {
        if (bucket[i] != null) {
            for (int j = 0; j < bucket[i].size(); j++)
                list[k++] = (E) (bucket[i].get(j));
        }
    }
}
```

# Bucket Sort

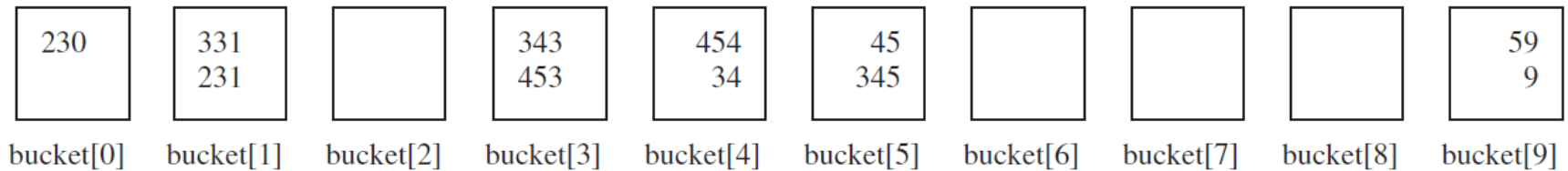
- Takes  $O(n + t)$  time to sort the list and uses  $O(n + t)$  space, where  $n$  is the list size
- If  $t$  is too large, using the bucket sort is not desirable
  - Instead, you can use a radix sort
- The radix sort is based on the bucket sort, but a radix sort uses only ten buckets
- Bucket sort is *stable*, meaning that if two elements in the original list have the same key value, their order is not changed in the sorted list.
  - That is, if element **e1** and element **e2** have the same key and **e1** precedes **e2** in the original list, **e1** still precedes **e2** in the sorted list

# Radix Sort

- Assume that the keys are positive integers
- The idea for the radix sort is to divide the keys into subgroups based on their radix positions
- It applies a bucket sort repeatedly for the key values on radix positions, starting from the least-significant position

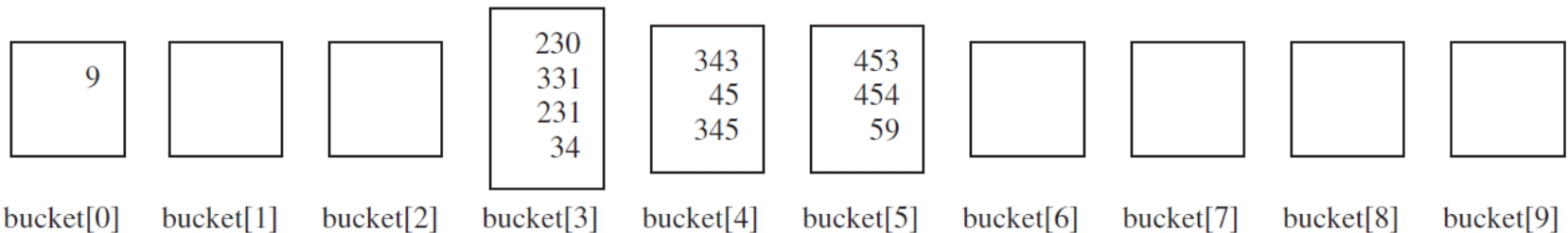
# Radix Sort

- Sort 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9

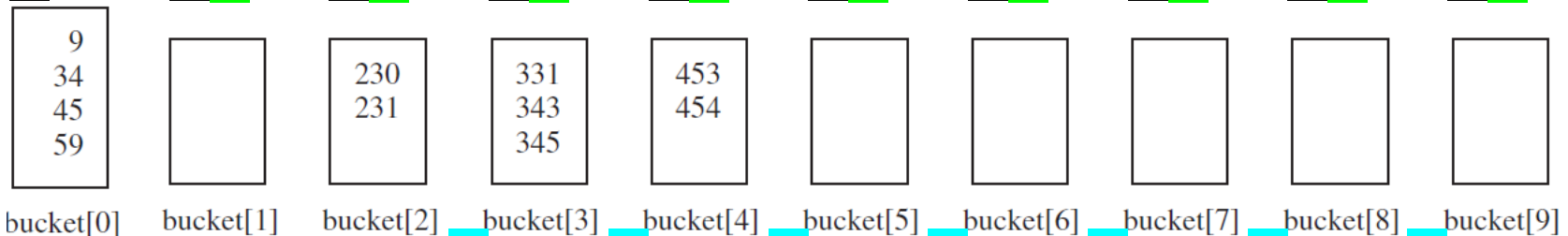


- Remove elements from the buckets:

230, 331, 231, 343, 453, 454, 34, 45, 345, 59, 09



009, 230, 331, 231, 034, 343, 045, 345, 453, 454, 59



9, 34, 45, 59, 230, 231, 331, 343, 345, 453, 454

# Radix Sort

- Radix sort takes  $O(d*n)$  time to sort  $n$  elements with integer keys, where  $d$  is the maximum number of the radix positions among all keys



```

public class RadixSort {
    public static void main(String[] args) {
        int size = 100000;
        int[] a = new int[size];
        randomInitiate(a);
        long startTime = System.currentTimeMillis();
        // radix sort
        for(int i=0; i<6; i++) {
            radix = i;
            bucketSort(a);
        }
        long endTime = System.currentTimeMillis();
        System.out.println((endTime - startTime) + "ms");
    }
    private static void randomInitiate(int[] a) {
        for (int i = 0; i < a.length; i++)
            a[i] = (int) (Math.random() * a.length);
    }
    static int radix = 1;
    public static int getKey(int n) {
        for(int i=0; i<radix; i++)
            n = n / 10;
        return n % 10;
    }
    static int t = 10;
}

```

```

public static void bucketSort(int[] list) {
    java.util.ArrayList<Integer>[] bucket = new java.util.ArrayList[t+1];
    // Distribute the elements from list to buckets
    for (int i = 0; i < list.length; i++) {
        // Assume element has the getKey() method
        int key = getKey(list[i]);
        if (bucket[key] == null)
            bucket[key] = new java.util.ArrayList<Integer>();
        bucket[key].add(list[i]);
    }
    // Now move the elements from the buckets back to list
    int k = 0; // k is an index for list
    for (int i = 0; i < bucket.length; i++) {
        if (bucket[i] != null) {
            for (int j = 0; j < bucket[i].size(); j++)
                list[k++] = (int) (bucket[i].get(j));
        }
    }
}

```

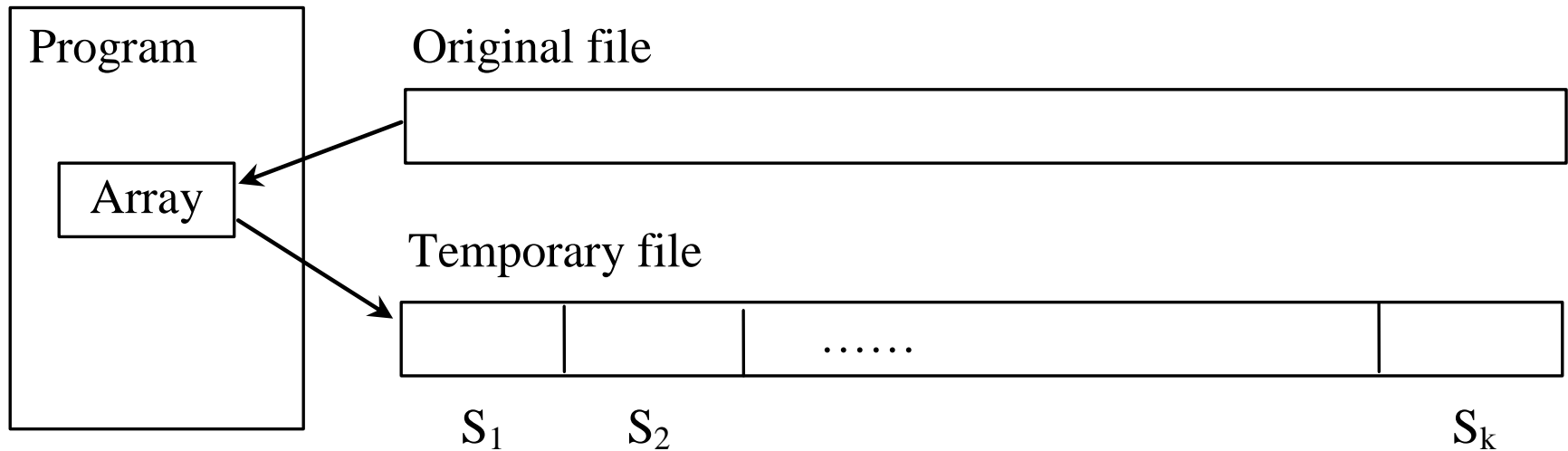
433ms

# External Sort

- All the sort algorithms discussed in the preceding sections assume that all data to be sorted is available at one time in internal memory such as an array
- To sort data stored in an external file, you may first bring data to the memory, then sort it internally.
- However, if the file is too large, all data in the file cannot be brought to memory at one time

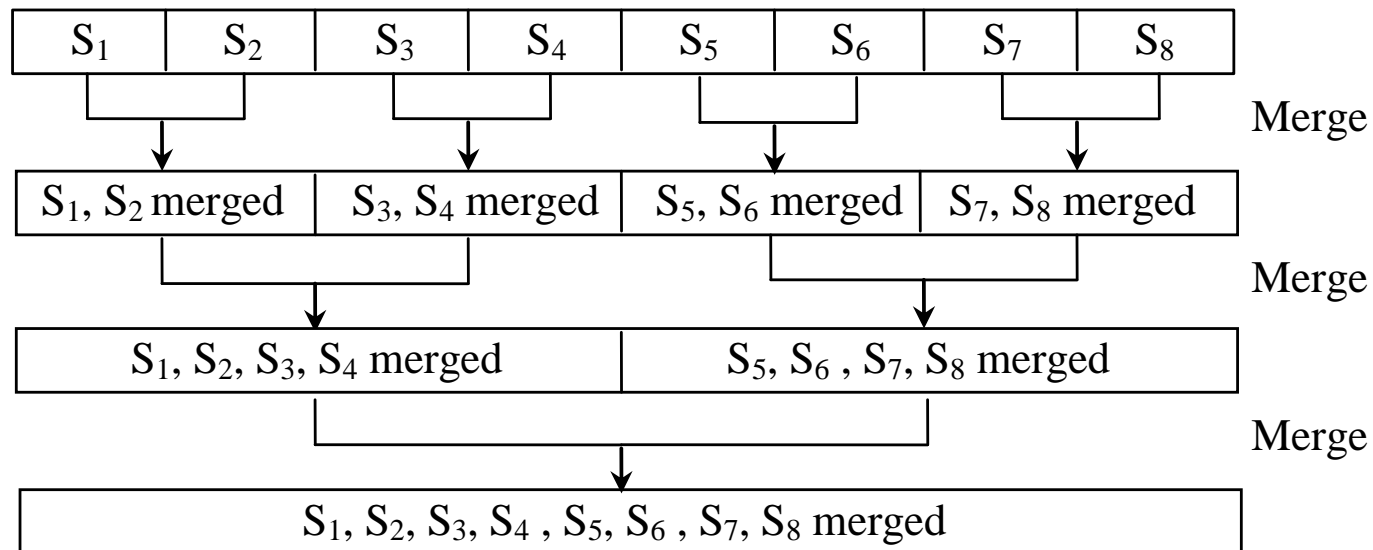
# Phase I

- Repeatedly bring data from the file to an array, sort the array using an internal sorting algorithm, and output the data from the array to a temporary file



# Phase II

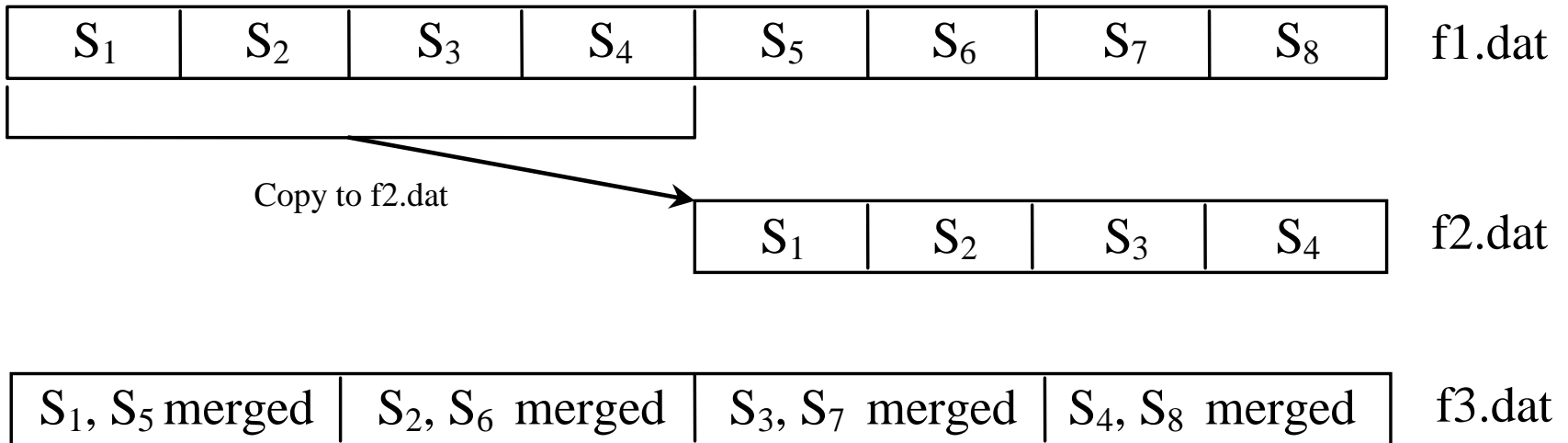
- **Merge** pairs of sorted segments (e.g., S1 with S2, S3 with S4, ..., and so on) into a larger sorted segment and save the new segment into a new temporary file
- Continue the same process until one sorted segment results



# Implementing Phase II

- Each merge step merges two sorted segments to form a new segment
- The new segment doubles the number elements - so the number of segments is reduced by half after each merge step
- A segment is too large to be brought to an array in memory
  - To implement a merge step, copy half number of segments from file f1.dat to a temporary file f2.dat.
  - Then merge the first remaining segment in f1.dat with the first segment in f2.dat into a temporary file named f3.dat.

# Implementing Phase II



# External Sort Complexity

- In the external sort, the dominating cost is that of I/O.
- Assume **n** is the number of elements to be sorted in the file
  - In Phase I, **n** number of elements are read from the original file and output to a temporary file, therefore, the I/O for Phase I is  $O(\mathbf{n})$ .
  - In Phase II, before the first merge step, the number of sorted segments is  $\mathbf{n/c}$ , where **c** is **MAX\_ARRAY\_SIZE**
    - Each merge step reduces the number of segments by half: after the first merge step, the number of segments is  $\mathbf{n/c * 1/2 = n / (2 * c)}$



# External Sort Complexity

- After the second merge step, the number of segments is  $n/2c * 1/2 = n/(2^2c)$
- After the third merge step the number of segments is  $n/2^3c$
- After  $\log(n/c)$  merge steps, the number of segments is reduced to 1 -> Therefore, the total number of merge steps is  $\log(n/c)$
- In each merge step, half the number of segments are read from file f1 and then written into a temporary file f2 - the remaining segments in f1 are merged with the segments in f2
  - The number of I/Os in each merge step is  $O(n)$
- Since the total number of merge steps is  $\log(n/c)$ , the total number of I/Os is:  $O(n) * \log(n/c) = O(n \log n)$

```

import java.io.*;

public class CreateLargeFile {
    public static void main(String[] args) throws Exception {
        DataOutputStream output = new DataOutputStream(
            new BufferedOutputStream(new FileOutputStream("largedata.dat")));
        for (int i = 0; i < 800004; i++)
            output.writeInt((int) (Math.random() * 1000000));
        output.close();

        // Display first 100 numbers
        DataInputStream input =
            new DataInputStream(new FileInputStream("largedata.dat"));
        for (int i = 0; i < 100; i++)
            System.out.print(input.readInt() + " ");
        input.close();
    }
}

```

```

import java.io.*;

public class SortLargeFile {
    public static final int MAX_ARRAY_SIZE = 43;
    public static final int BUFFER_SIZE = 100000;

    public static void main(String[] args) throws Exception {
        // Sort largedata.dat to sortedfile.dat
        sort("largedata.dat", "sortedfile.dat");

        // Display the first 100 numbers in the sorted file
        displayFile("sortedfile.dat");
    }

    /** Sort data in source file and into target file */
    public static void sort(String sourcefile, String targetfile)
        throws Exception {
        // Implement Phase 1: Create initial segments
        int numberOfSegments =
            initializeSegments(MAX_ARRAY_SIZE, sourcefile, "f1.dat");

        // Implement Phase 2: Merge segments recursively
        merge(numberOfSegments, MAX_ARRAY_SIZE,
            "f1.dat", "f2.dat", "f3.dat", targetfile);
    }
}

```

```

/** Sort original file into sorted segments */
private static int initializeSegments
    (int segmentSize, String originalFile, String f1)
    throws Exception {
    int[] list = new int[segmentSize];
    DataInputStream input = new DataInputStream(
        new BufferedInputStream(new FileInputStream(originalFile)));
    DataOutputStream output = new DataOutputStream(
        new BufferedOutputStream(new FileOutputStream(f1)));
    int numberOfSegments = 0;
    while (input.available() > 0) {
        numberOfSegments++;
        int i = 0;
        for ( ; input.available() > 0 && i < segmentSize; i++) {
            list[i] = input.readInt();
        }
        // Sort an array list[0..i-1]
        java.util.Arrays.sort(list, 0, i);
        // Write the array to f1.dat
        for (int j = 0; j < i; j++) {
            output.writeInt(list[j]);
        }
    }
    input.close();
    output.close();
    return numberOfSegments;
}

```

```

private static void merge(int numberOfSegments, int segmentSize,
    String f1, String f2, String f3, String targetfile)
    throws Exception {
    if (numberOfSegments > 1) {
        mergeOneStep(numberOfSegments, segmentSize, f1, f2, f3);
        merge((numberOfSegments + 1) / 2, segmentSize * 2,
            f3, f1, f2, targetfile);
    }
    else { // Rename f1 as the final sorted file
        File sortedFile = new File(targetfile);
        if (sortedFile.exists()) sortedFile.delete();
        new File(f1).renameTo(sortedFile);
    }
}

private static void mergeOneStep(int numberOfSegments,
    int segmentSize, String f1, String f2, String f3)
    throws Exception {
    DataInputStream f1Input = new DataInputStream(
        new BufferedInputStream(new FileInputStream(f1), BUFFER_SIZE));
    DataOutputStream f2Output = new DataOutputStream(
        new BufferedOutputStream(new FileOutputStream(f2), BUFFER_SIZE));

    // Copy half number of segments from f1.dat to f2.dat
    copyHalfToF2(numberOfSegments, segmentSize, f1Input, f2Output);
    f2Output.close();
}

```

```

// Merge remaining segments in f1 with segments in f2 into f3
DataInputStream f2Input = new DataInputStream(
    new BufferedInputStream(new FileInputStream(f2), BUFFER_SIZE));
DataOutputStream f3Output = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f3), BUFFER_SIZE));

mergeSegments(numberOfSegments / 2,
    segmentSize, f1Input, f2Input, f3Output);

f1Input.close();
f2Input.close();
f3Output.close();
}

/** Copy first half number of segments from f1.dat to f2.dat */
private static void copyHalfToF2(int numberOfSegments,
    int segmentSize, DataInputStream f1, DataOutputStream f2)
    throws Exception {
    for (int i = 0; i < (numberOfSegments / 2) * segmentSize; i++) {
        f2.writeInt(f1.readInt());
    }
}

```

```

/** Merge all segments */
private static void mergeSegments(int numberOfSegments,
    int segmentSize, DataInputStream f1, DataInputStream f2,
    DataOutputStream f3) throws Exception {
    for (int i = 0; i < numberOfSegments; i++) {
        mergeTwoSegments(segmentSize, f1, f2, f3);
    }

    // If f1 has one extra segment, copy it to f3
    while (f1.available() > 0) {
        f3.writeInt(f1.readInt());
    }
}

```

```

/** Merges two segments */
private static void mergeTwoSegments(int segmentSize,
    DataInputStream f1, DataInputStream f2,
    DataOutputStream f3) throws Exception {
    int intFromF1 = f1.readInt();
    int intFromF2 = f2.readInt();
    int f1Count = 1;
    int f2Count = 1;

```

```

while (true) {
    if (intFromF1 < intFromF2) {
        f3.writeInt(intFromF1);
        if (f1.available() == 0 || f1Count++ >= segmentSize) {
            f3.writeInt(intFromF2);
            break;
        } else {
            intFromF1 = f1.readInt();
        }
    } else {
        f3.writeInt(intFromF2);
        if (f2.available() == 0 || f2Count++ >= segmentSize) {
            f3.writeInt(intFromF1);
            break;
        } else {
            intFromF2 = f2.readInt();
        }
    }
}
while (f1.available() > 0 && f1Count++ < segmentSize) {
    f3.writeInt(f1.readInt());
}
while (f2.available() > 0 && f2Count++ < segmentSize) {
    f3.writeInt(f2.readInt());
}

```



```
/** Display the first 100 numbers in the specified file */
public static void displayFile(String filename) {
    try {
        DataInputStream input =
            new DataInputStream(new FileInputStream(filename));
        for (int i = 0; i < 100; i++)
            System.out.print(input.readInt() + " ");
        input.close();
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}
```