Chapter 23 Sorting

Objectives

- To study and analyze time complexity of various sorting algorithms (§§23.2–23.7).
- To design, implement, and analyze insertion sort (§23.2).
- To design, implement, and analyze bubble sort (§23.3).
- To design, implement, and analyze merge sort (§23.4).
- To design, implement, and analyze quick sort (§23.5).
- To design and implement a binary heap (§23.6).
- To design, implement, and analyze heap sort (§23.7).
- To design, implement, and analyze bucket sort and radix sort (§23.8).
- To design, implement, and analyze external sort for files that have a large amount of data (§23.9).

why study sorting?

Sorting is a classic subject in computer science. There are three reasons for studying sorting algorithms.

- First, sorting algorithms illustrate many creative approaches to problem solving and these approaches can be applied to solve other problems.
- Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays.
- Third, sorting algorithms are excellent examples to demonstrate algorithm performance.

what data to sort?

The data to be sorted might be integers, doubles, characters, or objects. §7.8, "Sorting Arrays," presented selection sort and insertion sort for numeric values. The selection sort algorithm was extended to sort an array of objects in §11.5.7, "Example: Sorting an Array of Objects." 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. For simplicity, this section assumes:

- data to be sorted are integers,
- data are sorted in ascending order, and
- 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.

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.

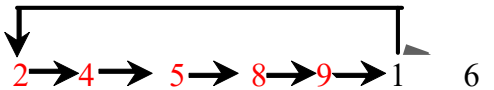
Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step2: 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.



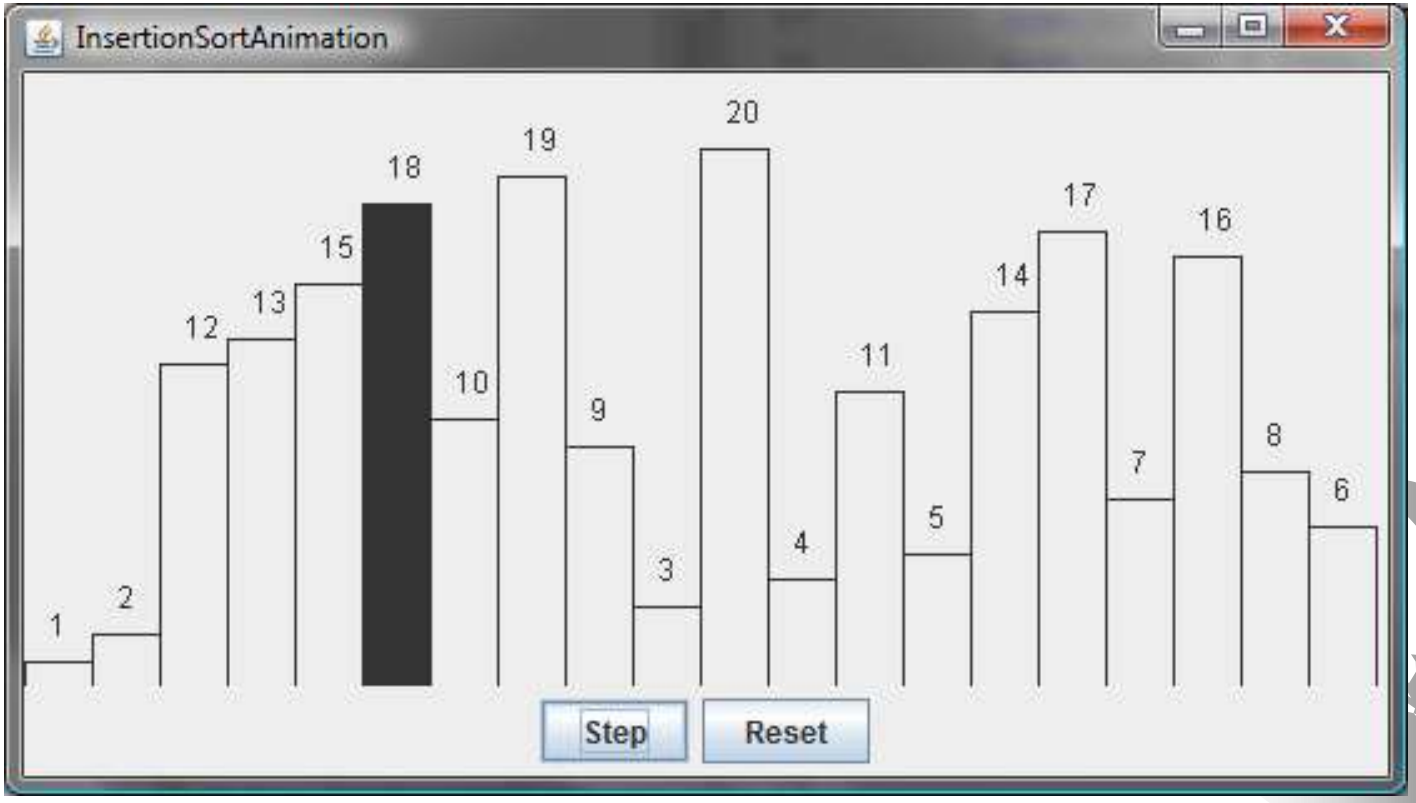
Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.

Step 7: The entire list is now sorted.

animation

Insertion Sort Animation

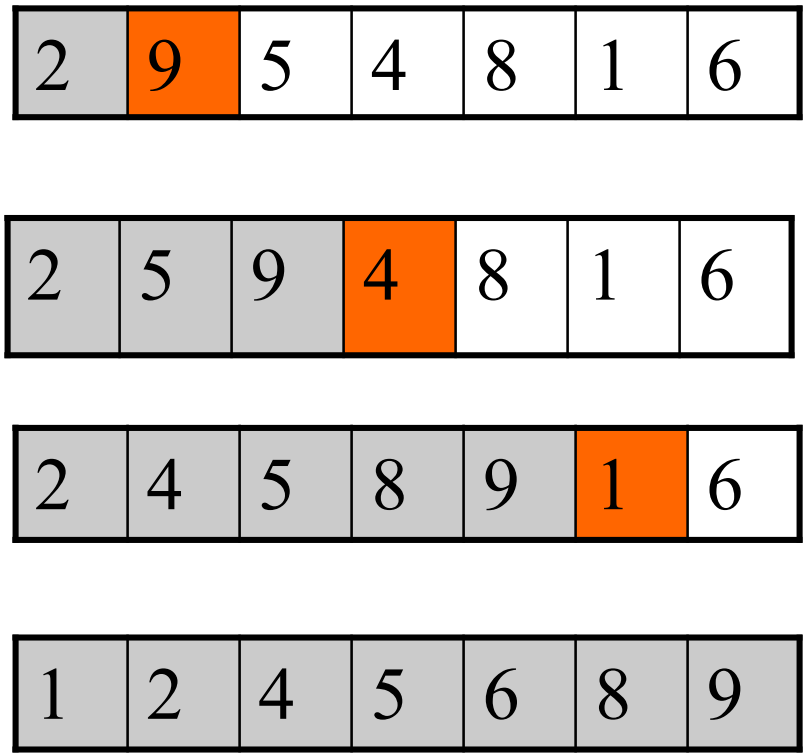
http://www.cs.armstrong.edu/liang/animation/web/InsertionSort.html



animation

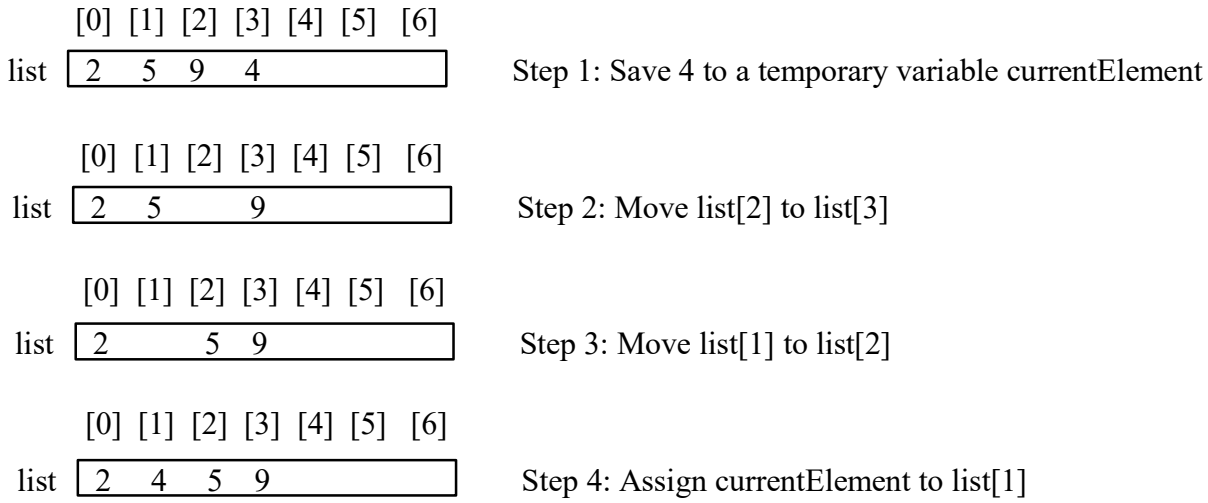
Insertion Sort

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



How to Insert?

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.



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
}
```

```
list[0]

list[0] list[1]

list[0] list[1] list[2]

list[0] list[1] list[2] list[3]

list[0] list[1] list[2] list[3] ...
```

From Idea to Solution

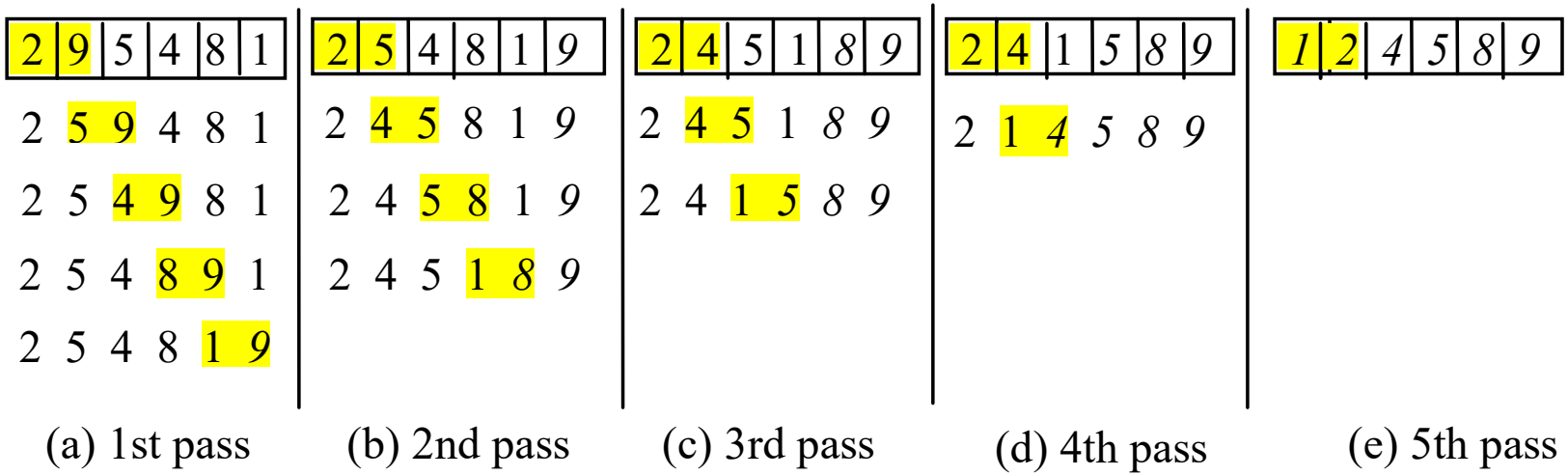
```
for (int i = 1; i < list.length; i++) {
  insert list[i] into a sorted sublist list[0..i-1] so that
  list[0..i] is sorted
}
```

Expand

```
double 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;
```

InsertSort Run

Bubble Sort



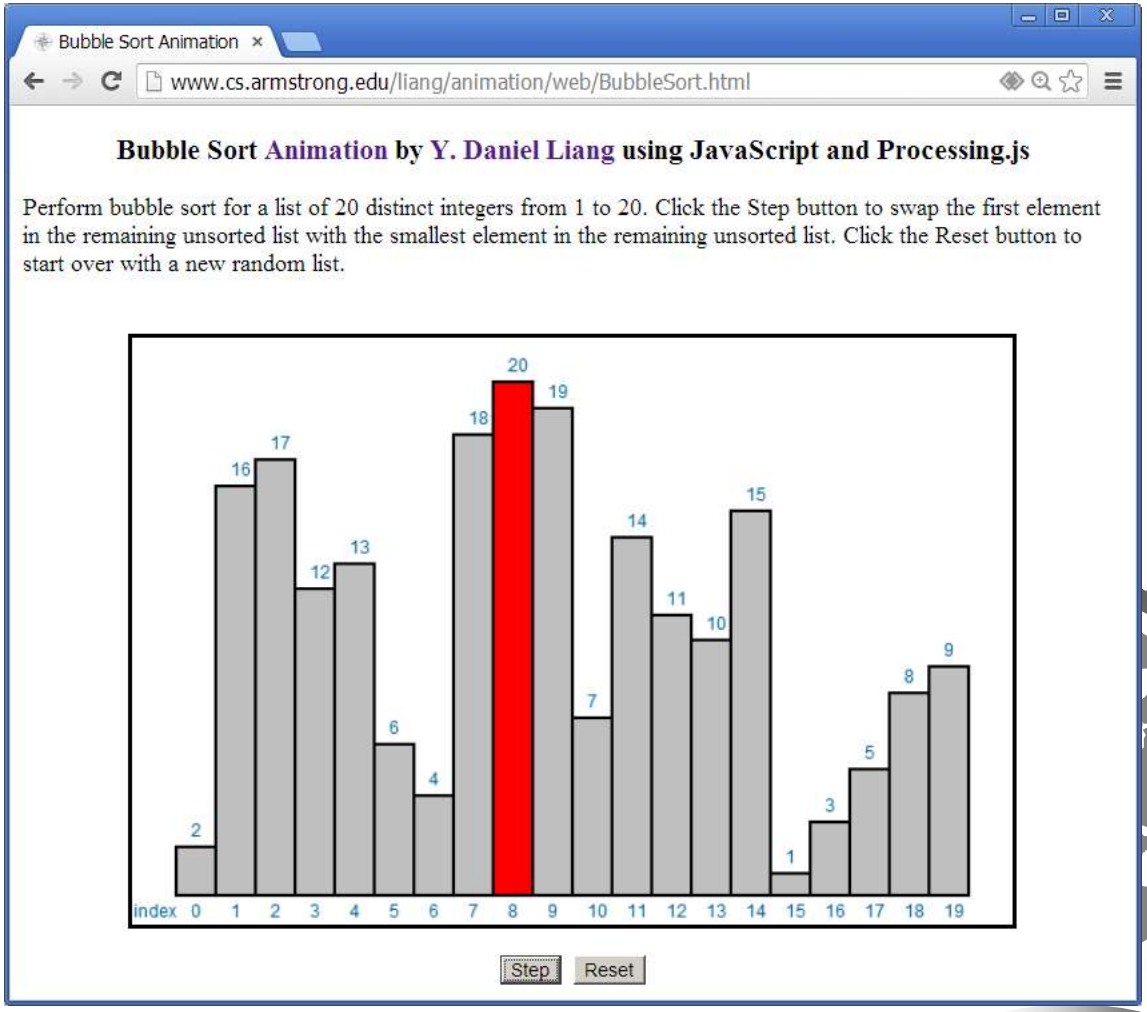
Bubble sort time: $O(n^2)$

$$(n-1)+(n-2)+...+2+1=\frac{n^2}{2}-\frac{n}{2}$$

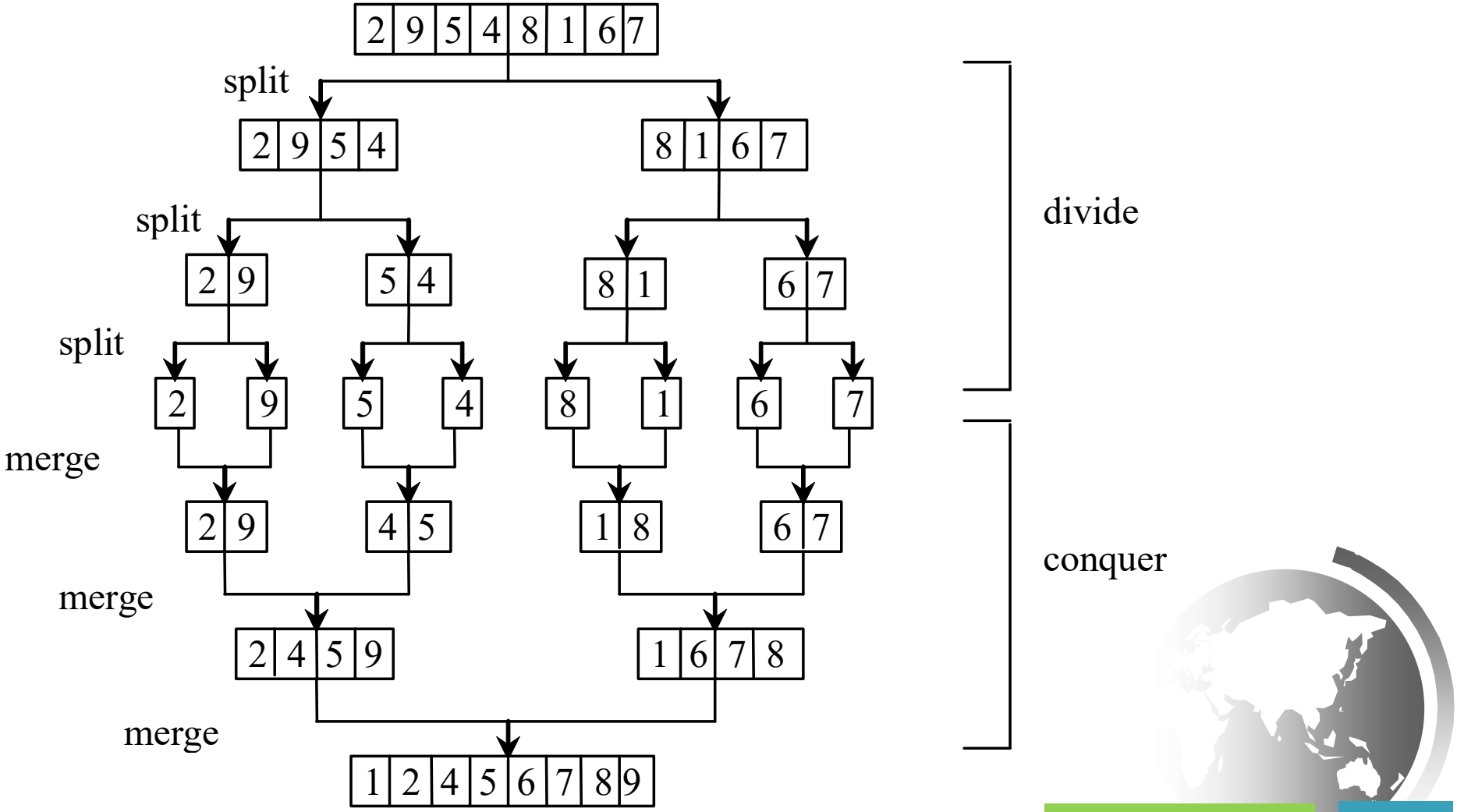
BubbleSort Run

Bubble Sort Animation

http://www.cs.armstrong.edu/liang/animation/web/BubbleSort.html



Merge Sort

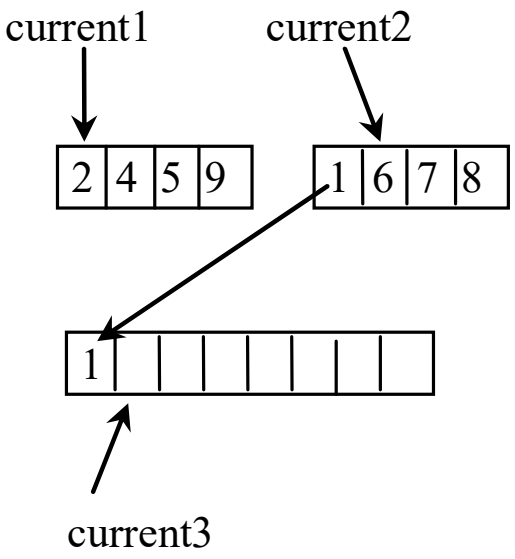

2 9 5 4 8 1 6 7
split
2 9 5 4
8 1 6 7
split
2 9
5 4
8 1
6 7
split
2
9
5
4
8
1
6
7
merge
2 9
4 5
1 8
6 7
merge
2 4 5 9
1 6 7 8
merge
1 2 4 5 6 7 8 9
divide
conquer

MergeSort
Run

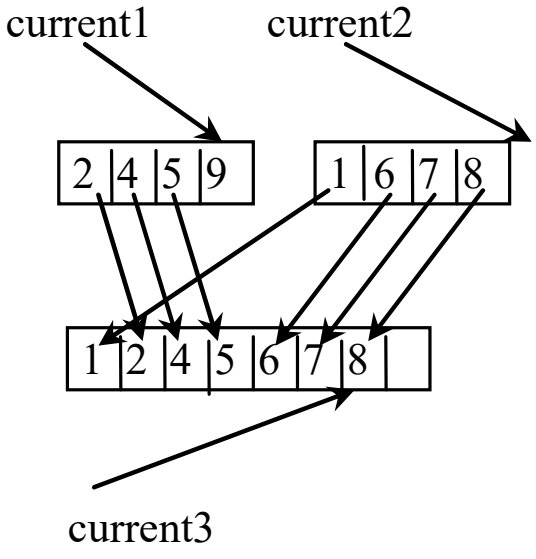
Merge Sort

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

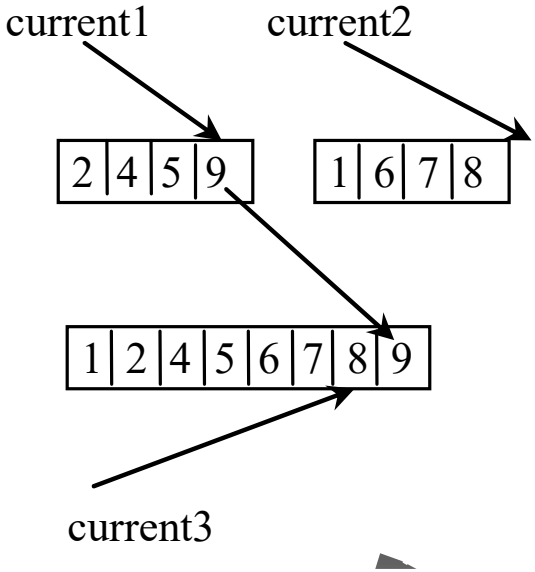
Merge Two Sorted Lists



(a) After moving 1 to temp



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



(c) After moving 9 to temp

Animation for Merging Two Sorted Lists

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. So,

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

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

Merge Sort Time

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. To merge two 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. So, the total time is *2n-1*. Therefore,

$$T(n) = 2T(\frac{n}{2}) + 2n - 1 = 2(2T(\frac{n}{4}) + 2\frac{n}{2} - 1) + 2n - 1 = 2^2 T(\frac{n}{2^2}) + 2n - 2 + 2n - 1$$

$$= 2^k T(\frac{n}{2^k}) + 2n - 2^{k-1} + ... + 2n - 2 + 2n - 1$$

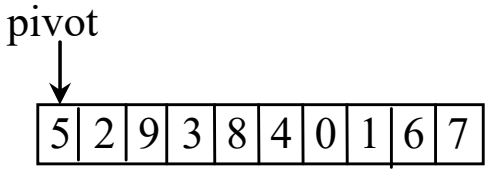
$$= 2^{\log n} T(\frac{n}{2^{\log n}}) + 2n - 2^{\log n - 1} + ... + 2n - 2 + 2n - 1$$

$$= n + 2n\log n - 2^{\log n} + 1 = 2n\log n + 1 = O(n\log n)$$

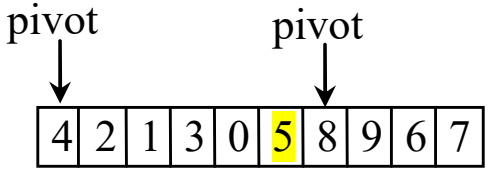
Quick Sort

Quick sort, developed by C. A. R. Hoare (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.

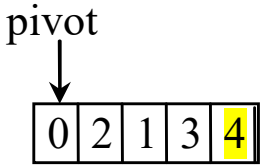
Quick Sort



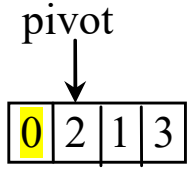
(a) The original array



(b)The original array is partitioned



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

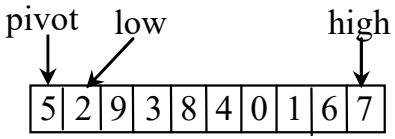


(d) The partial array (0 2 1 3) is partitioned

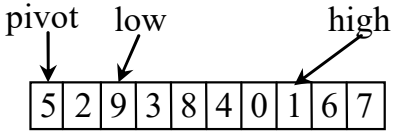
(e) The partial array (2 1 3) is partitioned

Partition

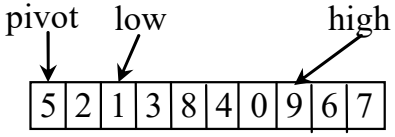
Animation for partition



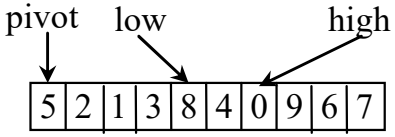
(a) Initialize pivot, low, and high



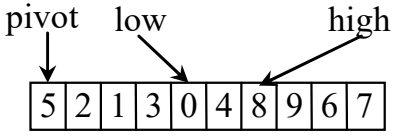
(b) Search forward and backward



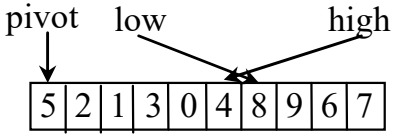
(c) 9 is swapped with 1



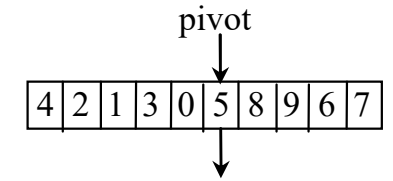
(d) Continue search



(e) 8 is swapped with 0



(f) when high < low, search is over



(g) pivot is in the right place

QuickSort

Run

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 $O(n^2)$ time:

$$(n-1)+(n-2)+...+2+1=O(n^2)$$

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 elements using quick sort. So,

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

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. So the average time is *O(nlogn)*. The exact average-case analysis is beyond the scope of this book.

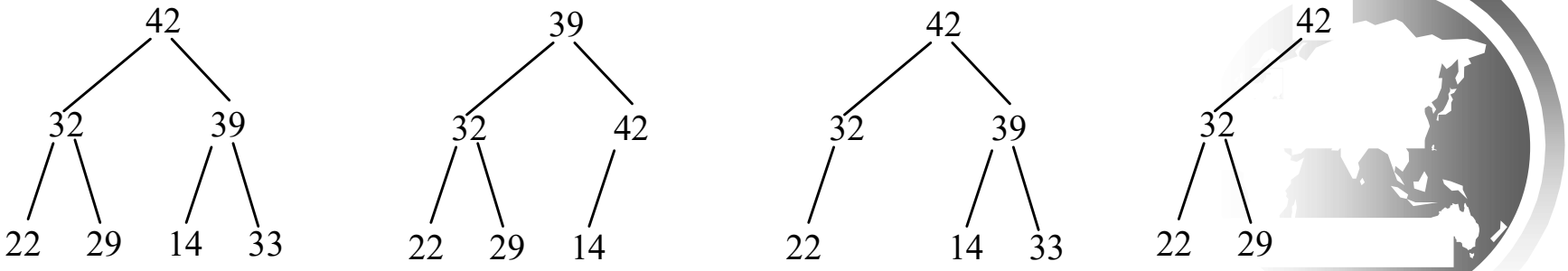
Heap

Heap is a useful data structure for designing efficient sorting algorithms and priority queues. A *heap* is a binary tree with the following properties:

- It is a complete binary tree.
- Each node is greater than or equal to any of its children.

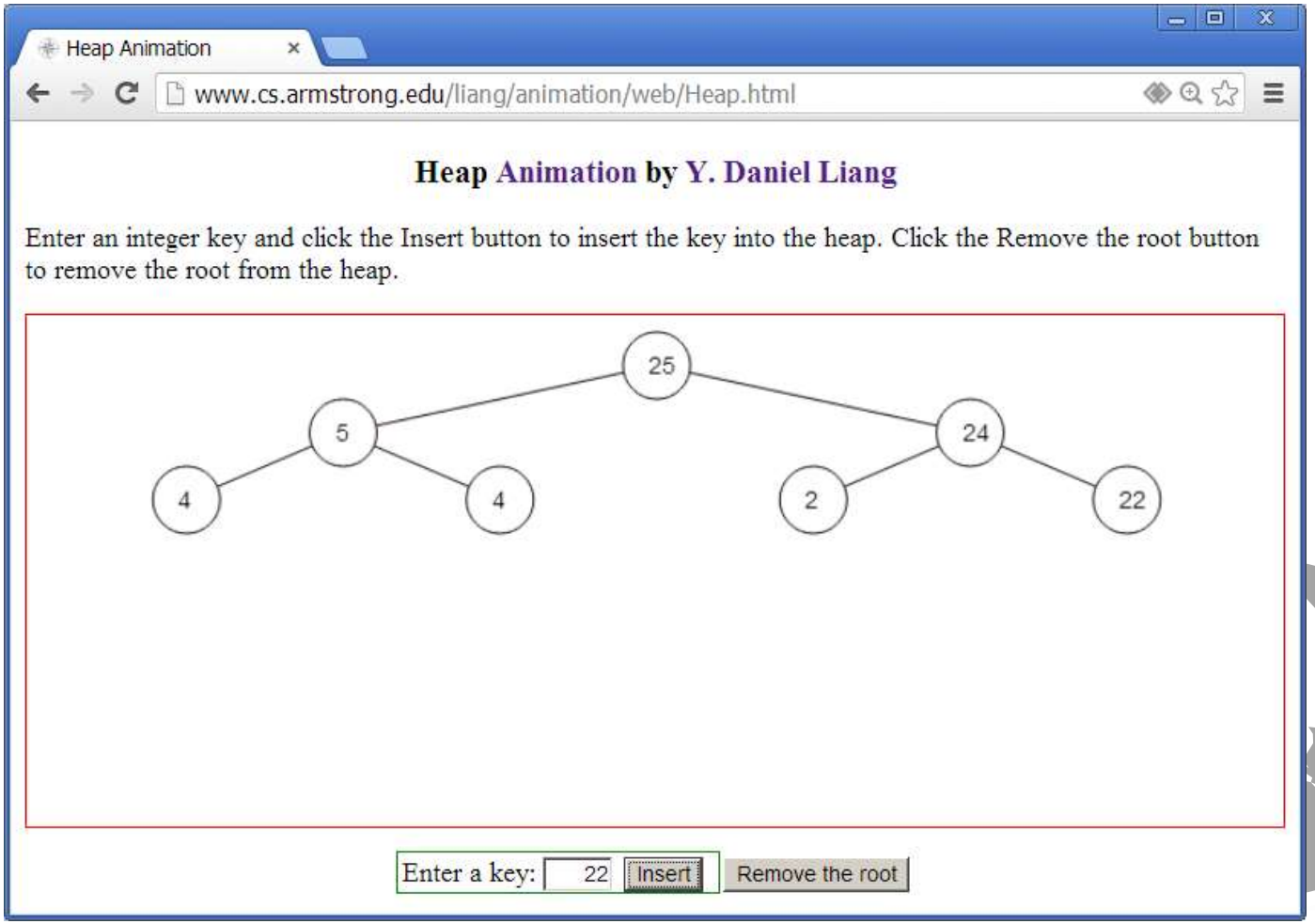
Complete Binary Tree

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. For example, in the following figure, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap, because the root (39) is less than its right child (42).



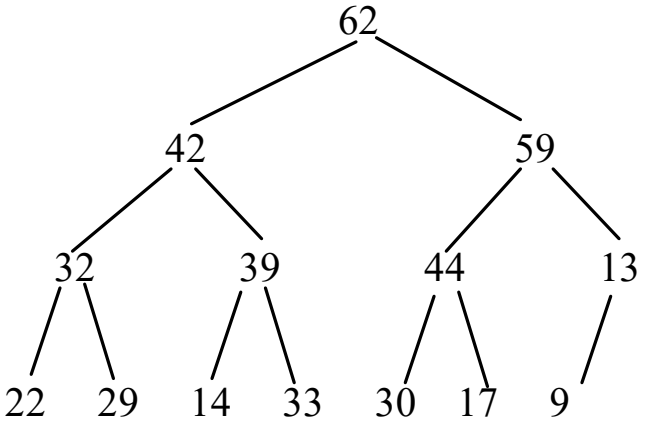
See How a Heap Works

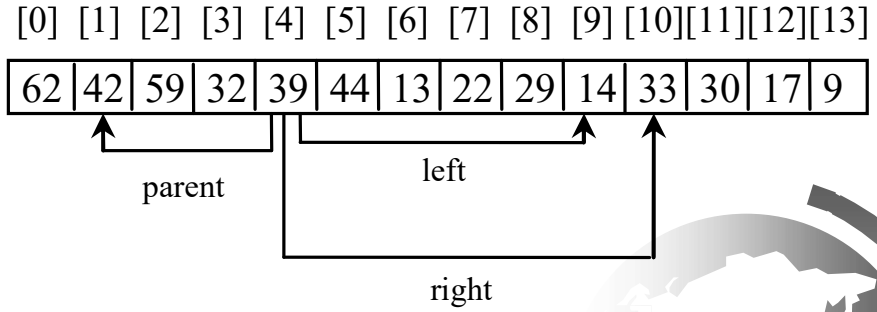
http://www.cs.armstrong.edu/liang/animation/web/Heap.html



Representing a Heap

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 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*).

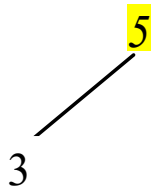




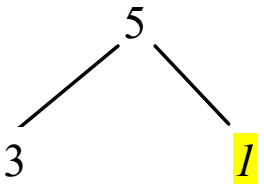
Adding Elements to the Heap

Adding 3, 5, 1, 19, 11, and 22 to a heap, initially empty

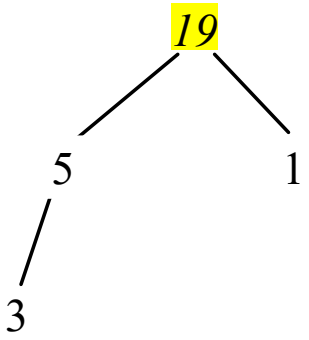
(a) After adding 3



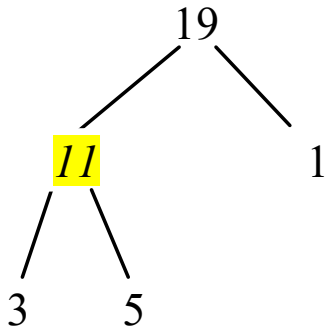
(b) After adding 5



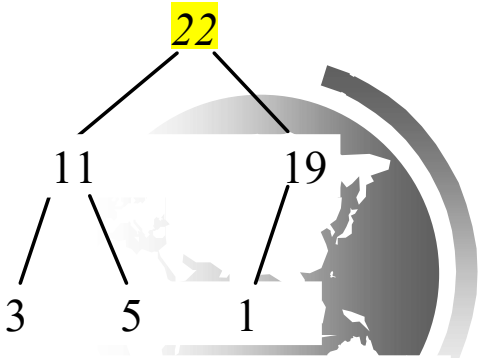
(c) After adding 1



(d) After adding 19



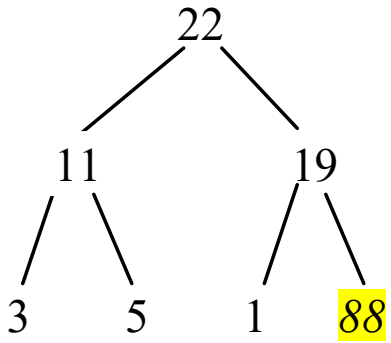
(e) After adding 11



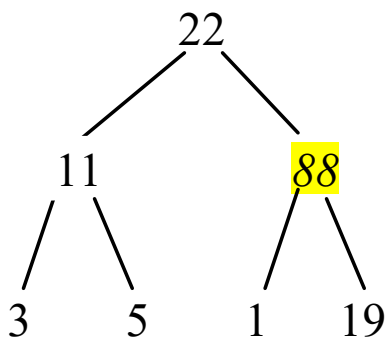
(f) After adding 22

Rebuild the heap after adding a new node

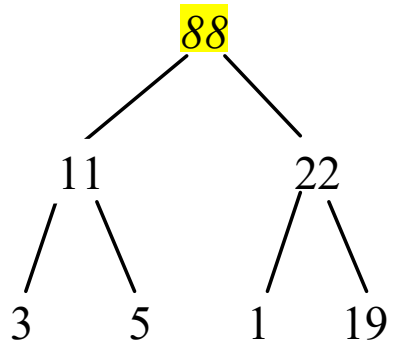
Adding 88 to the heap



(a) Add 88 to a heap



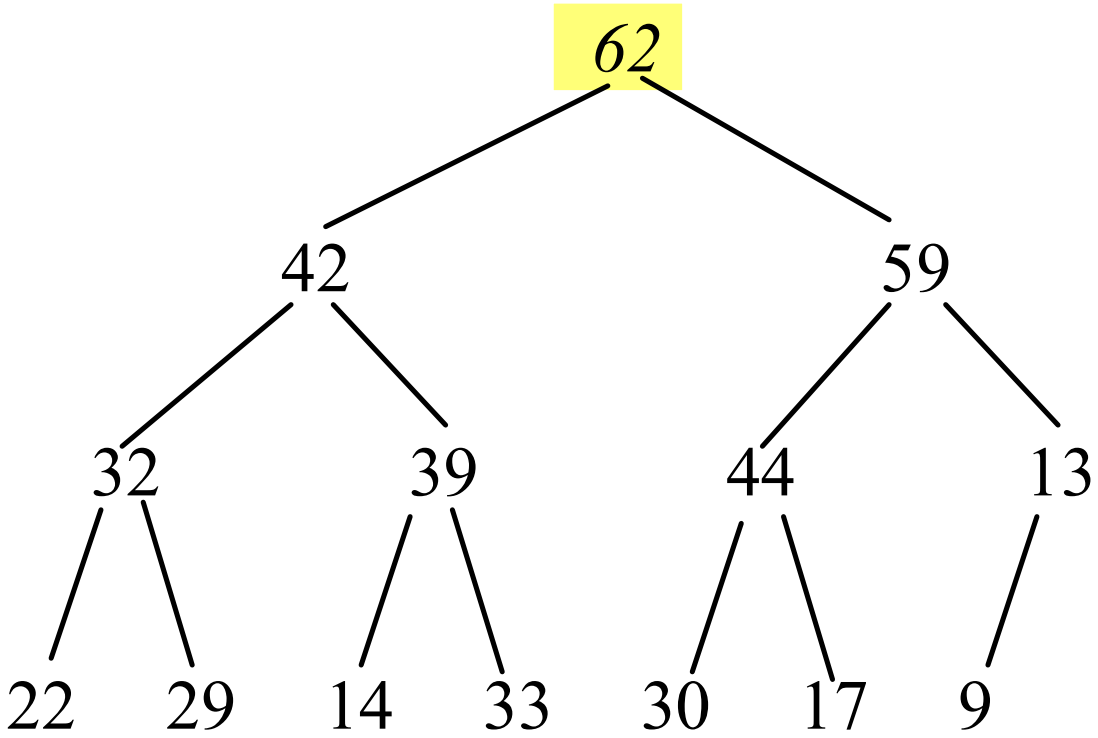
(b) After swapping 88 with 19



(b) After swapping 88 with 22

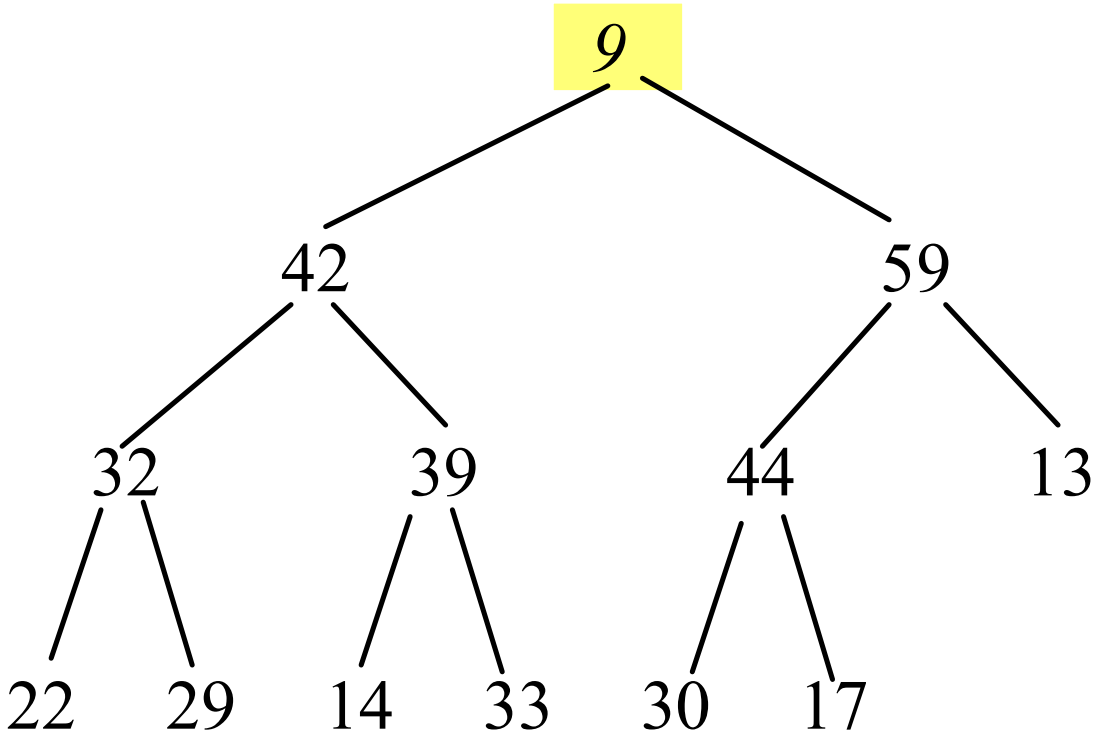
Removing the Root and Rebuild the Tree

Removing root 62 from the heap



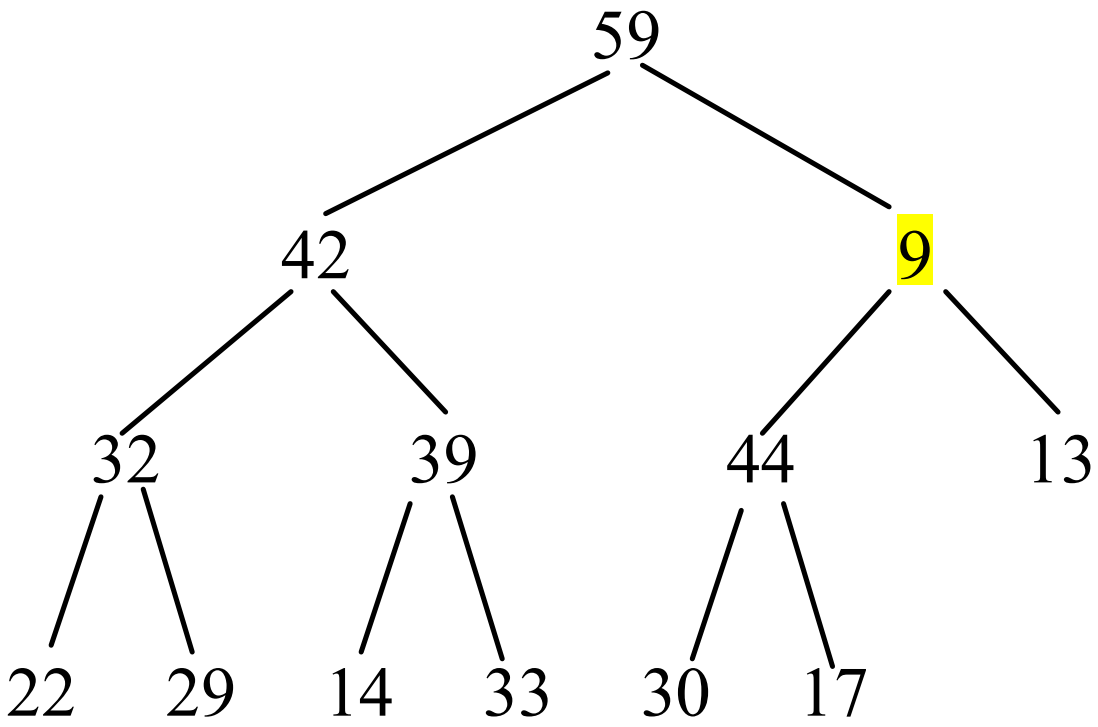
Removing the Root and Rebuild the Tree

Move 9 to root



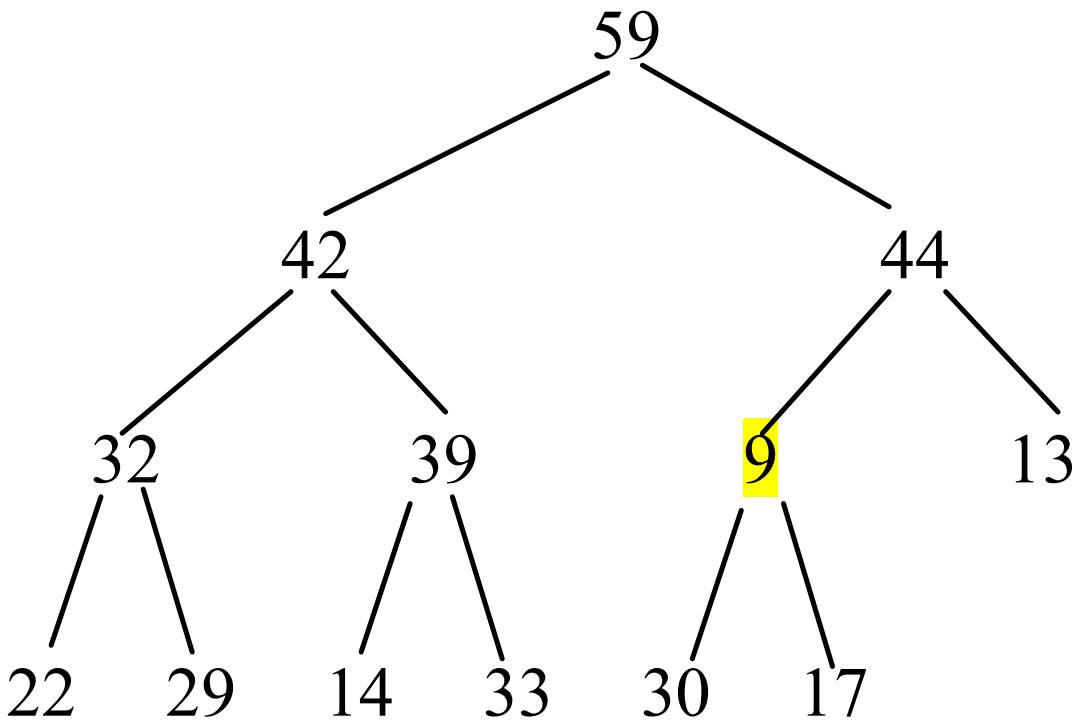
Removing the Root and Rebuild the Tree

Swap 9 with 59



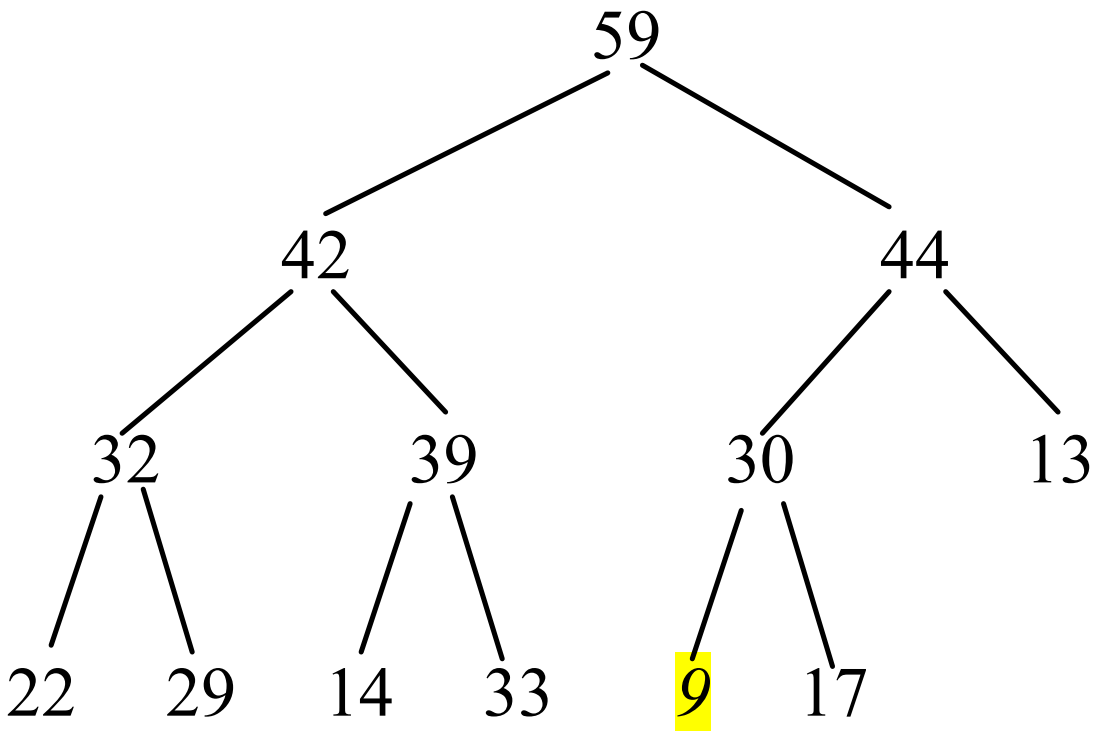
Removing the Root and Rebuild the Tree

Swap 9 with 44

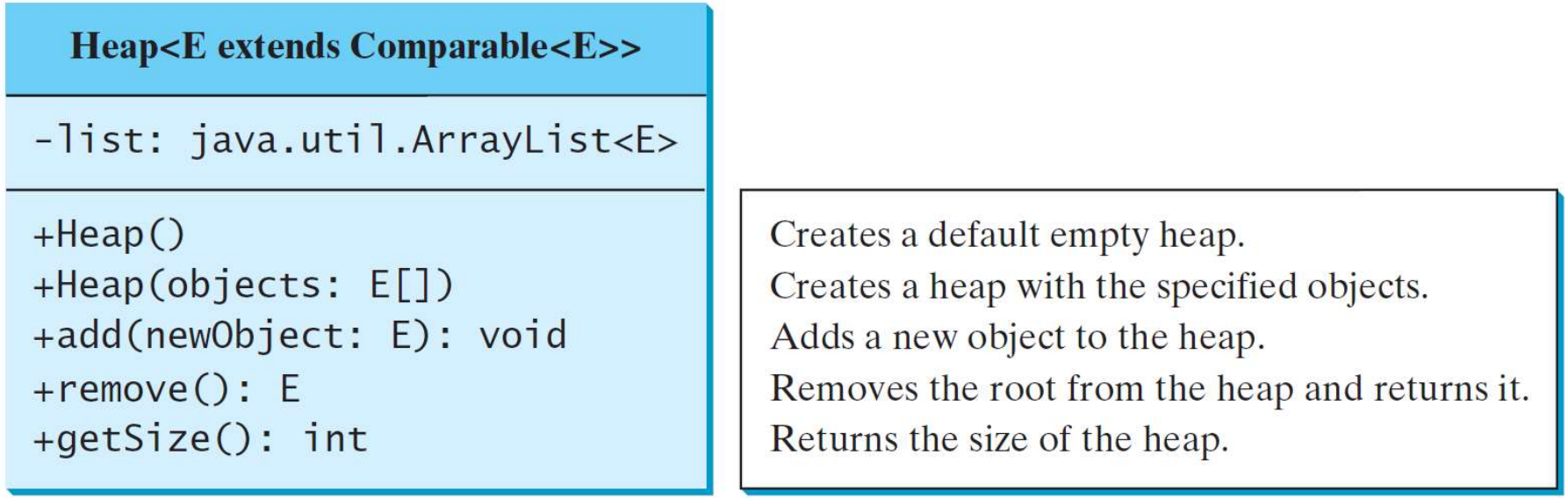


Removing the Root and Rebuild the Tree

Swap 9 with 30



The Heap Class



Heap

Heap Sort

HeapSort

Run

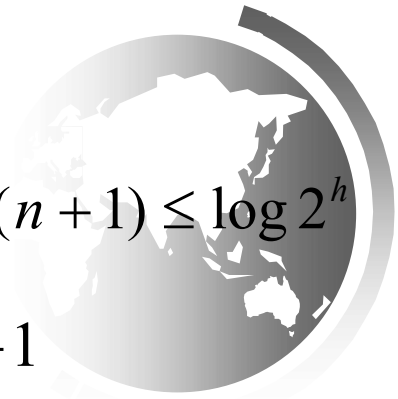
Heap Sort Time

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 hth level has at least one node and at most $2^{(h-1)}$ nodes. Therefore,

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

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

$$h-1 < \log(n+1) \le h \qquad \log(n+1) \le h < \log(n+1)+1$$

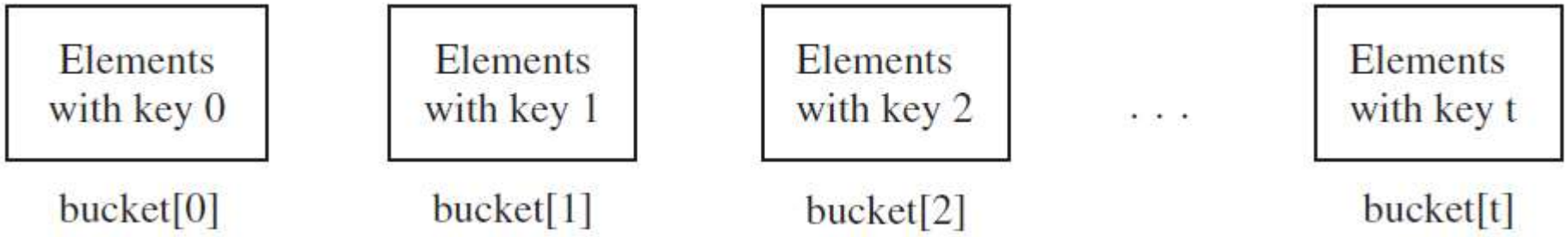


Bucket Sort and Radix Sort

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(nlogn). So, no sorting algorithms based on comparisons can perform better than O(nlogn). However, if the keys are small integers, you can use bucket sort without having to compare the keys.

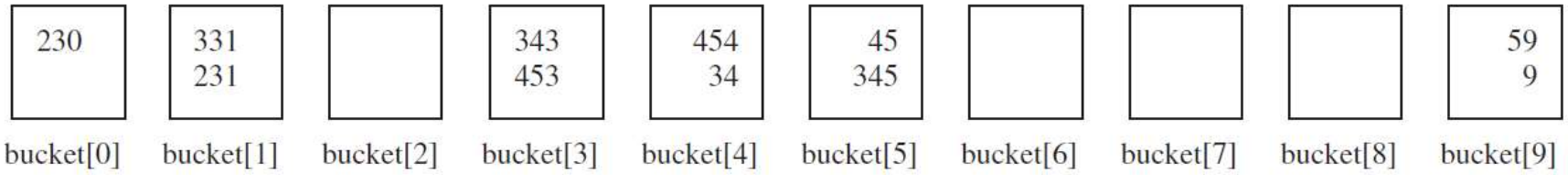
Bucket Sort

The bucket sort algorithm works as follows. Assume the keys are in the range from 0 to N-1. We need N buckets labeled 0, 1, ..., and N-1. 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.

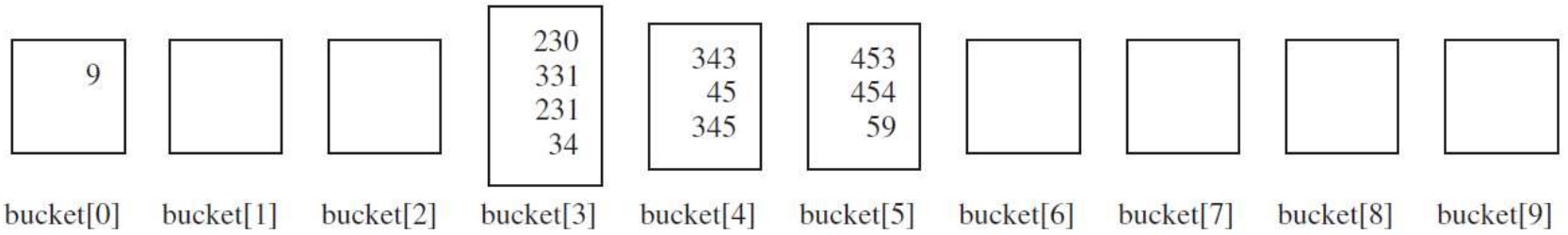


Radix Sort

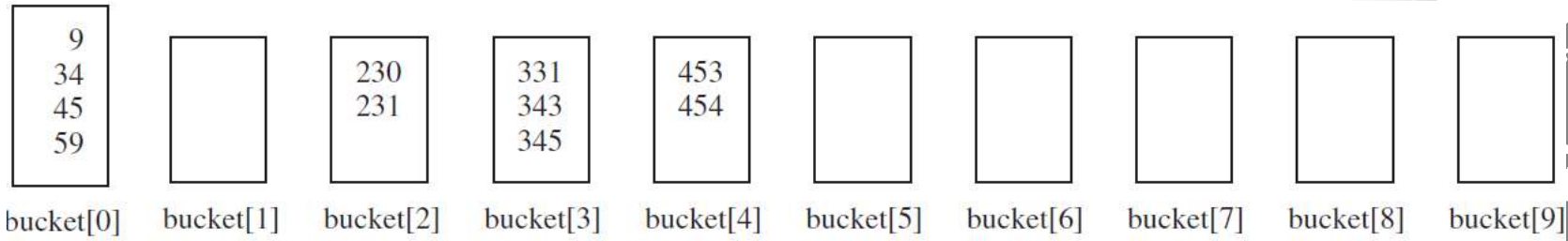
Sort 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9



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



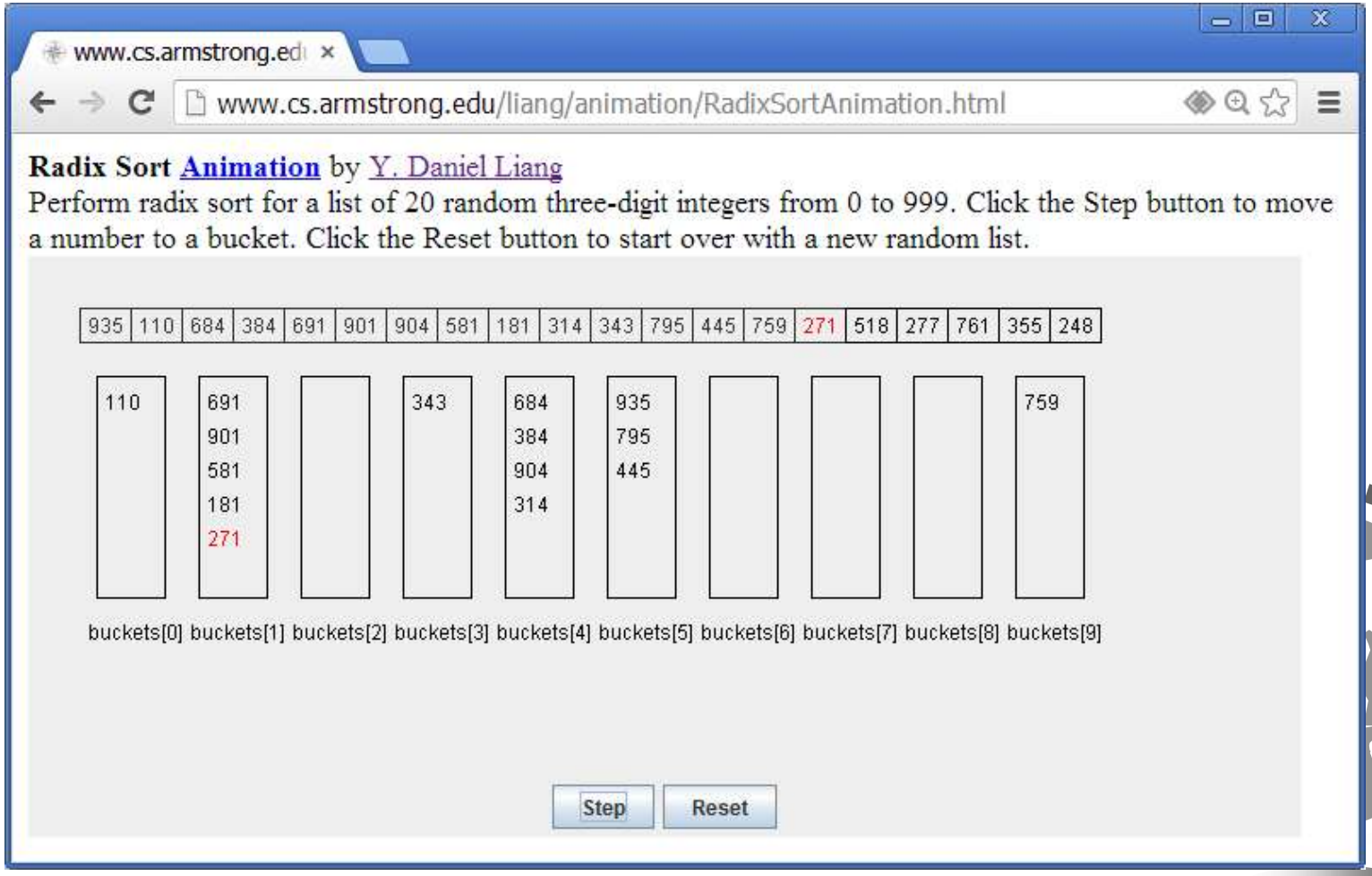
9, 230, 331, 231, 34, 343, 45, 345, 453, 454, 59



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

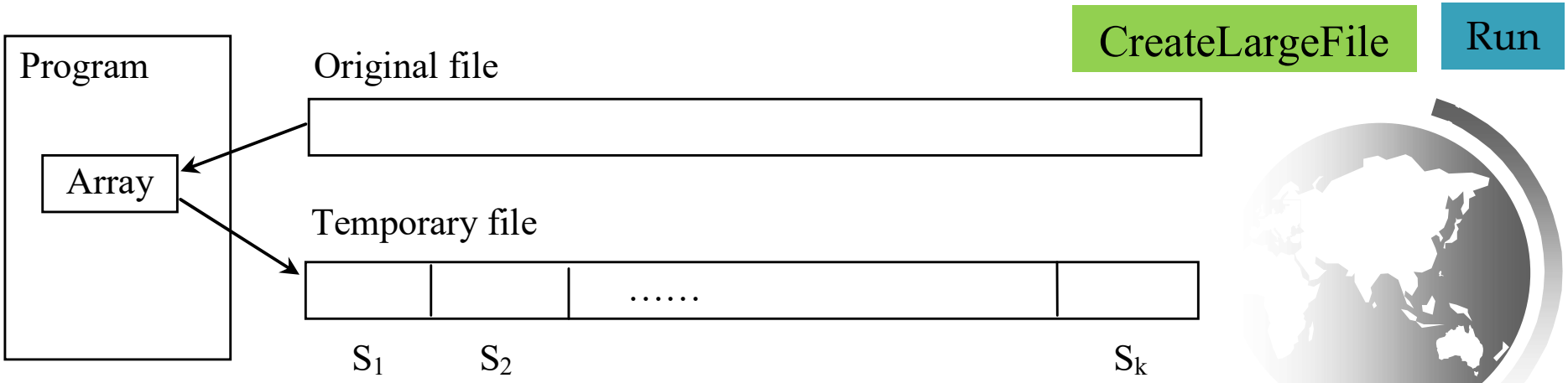
Radix Sort Animation

http://www.cs.armstrong.edu/liang/animation/web/RadixSort.html



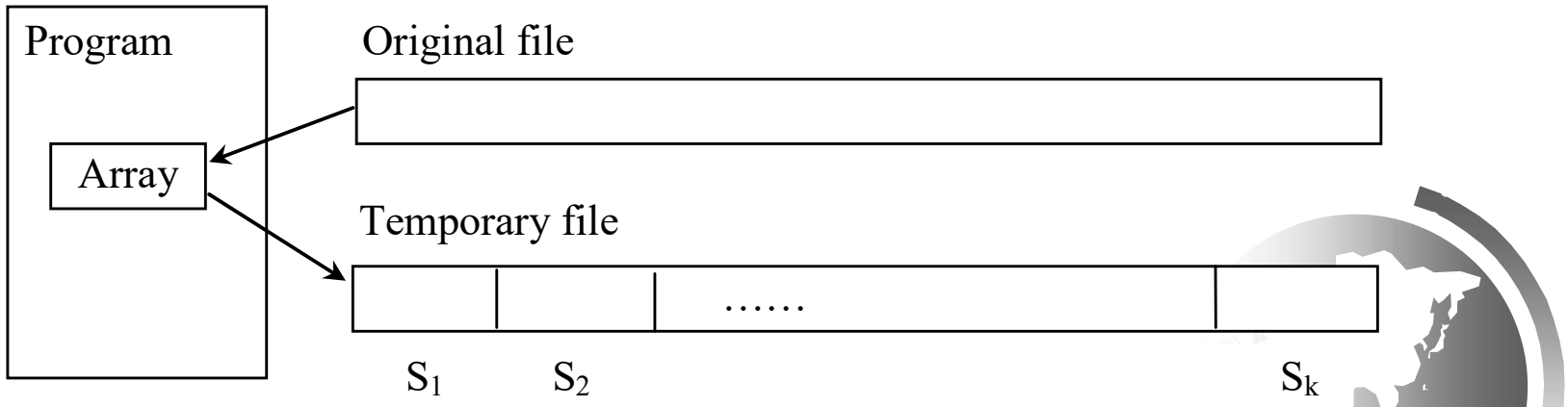
External Sort

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.



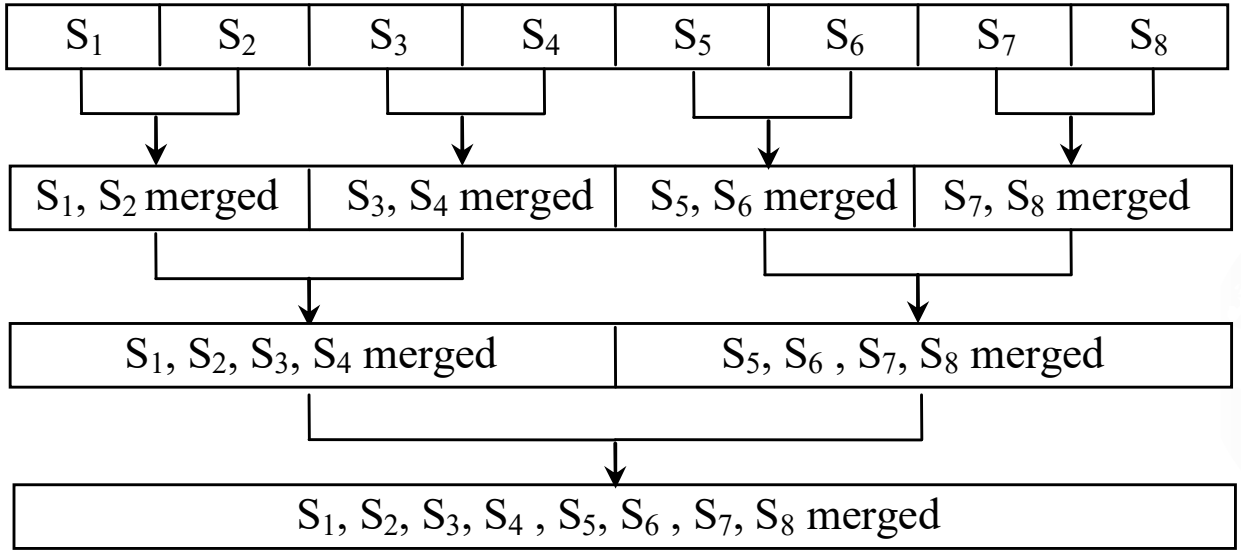
Phase I

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 a pair 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