Chapter 17 Sorting

Objectives

- To study and analyze time efficiency of various sorting algorithms (§§17.2–17.7).
- To design, implement, and analyze bubble sort (§17.2).
- To design, implement, and analyze merge sort (§17.3).
- To design, implement, and analyze quick sort (§17.4).
- To design and implement a heap (§17.5).
- To design, implement, and analyze heap sort (§17.5).
- To design, implement, and analyze bucket sort and radix sort (§17.6).
- To design, implement, and analyze external sort for large data in a file (§17.7).

why study sorting?

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

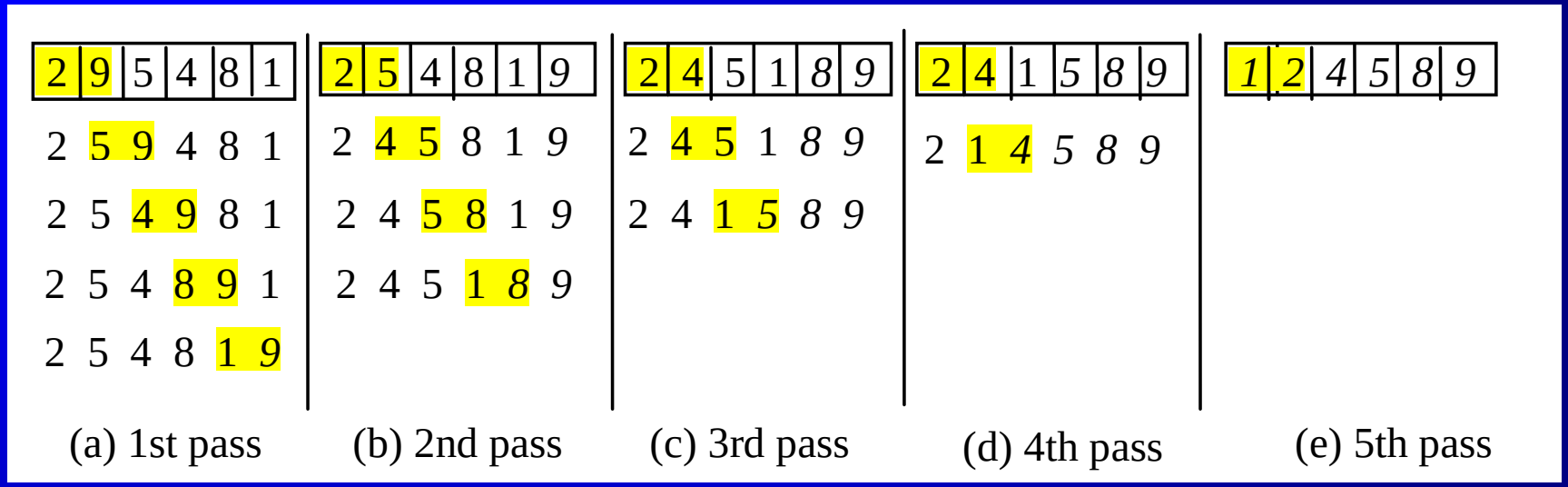
- First, sorting algorithms illustrate many creative approaches to problem solving and these approaches can be applied to solve other problems.
- Second, sorting algorithms are good for practicing fundamental programming techniques using selection statements, loops, methods, and arrays.
- Third, sorting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

what data to sort?

The data to be sorted might be integers, doubles, characters, or objects. For simplicity, this chapter assumes:

- data to be sorted are integers,
- data are stored in a list, and
- data are sorted in ascending order

Bubble Sort



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

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

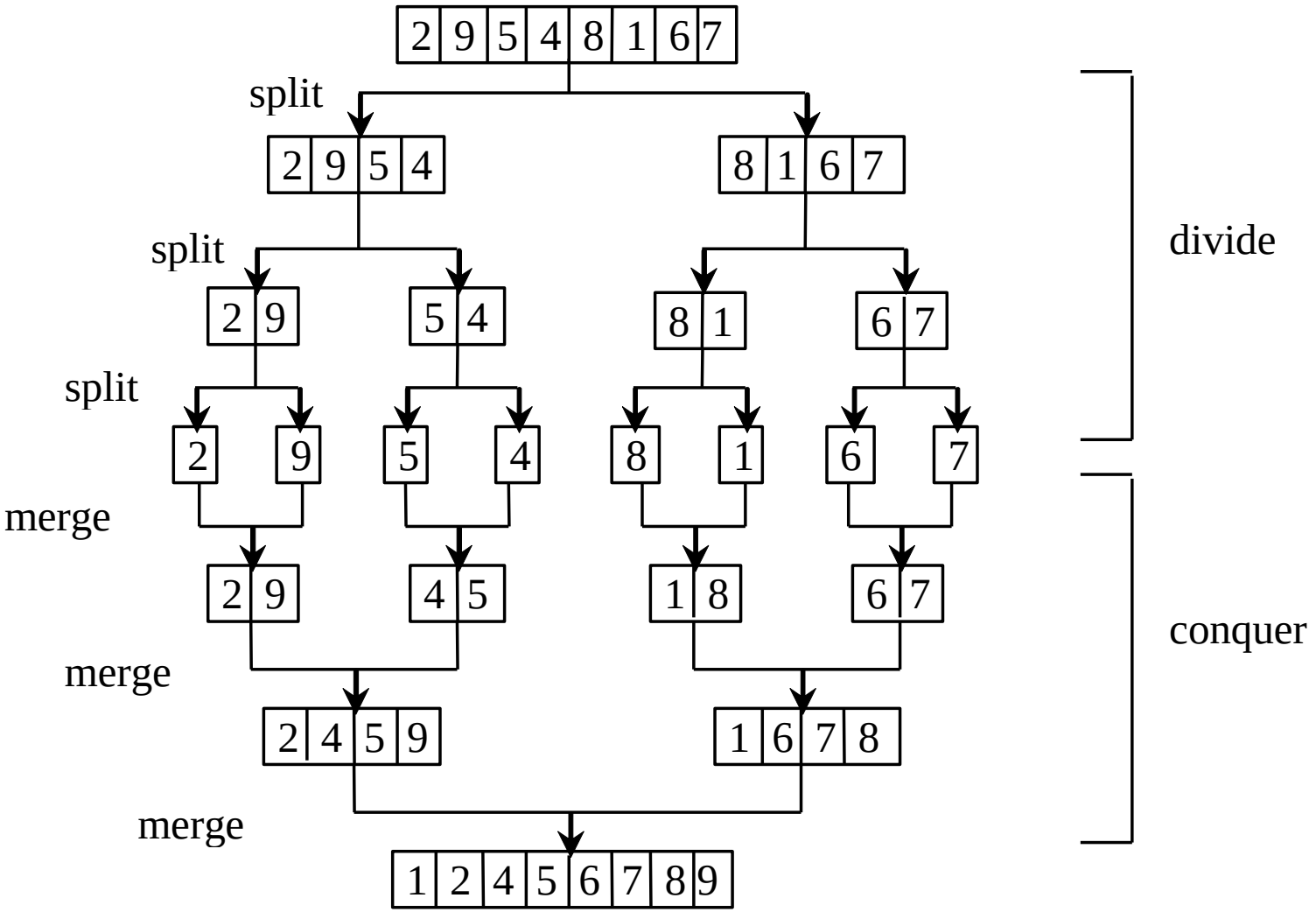
BubbleSort

Run

Bubble Sort Animation

http://www.cs.armstrong.edu/liang/animation/BubbleSortAnimation.html

Merge Sort



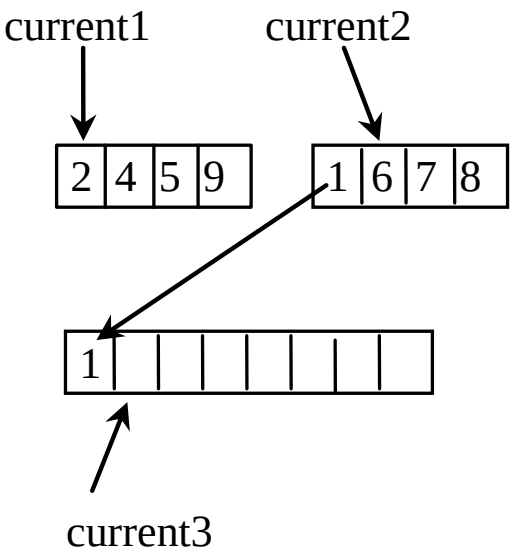
MergeSort

Run

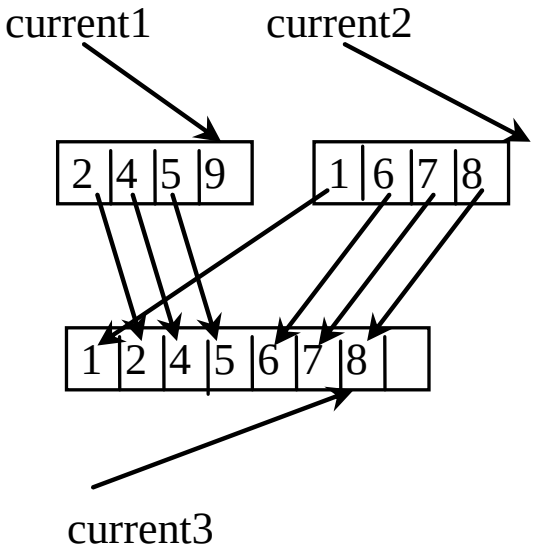
Merge Sort

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

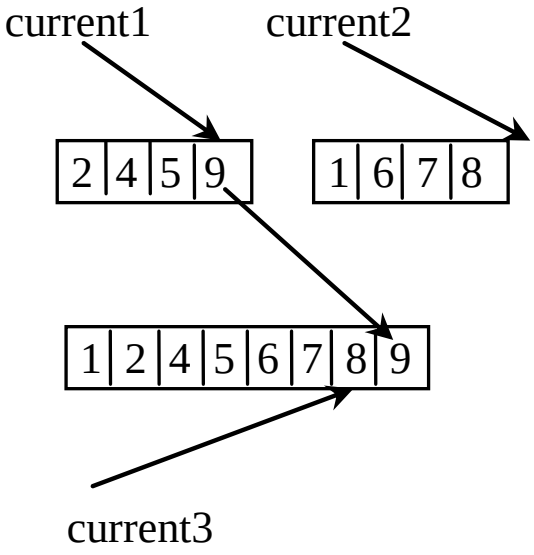
Merge Two Sorted Lists



(a) After moving 1 to temp



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



(c) After moving 9 to temp

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,

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

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,

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

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

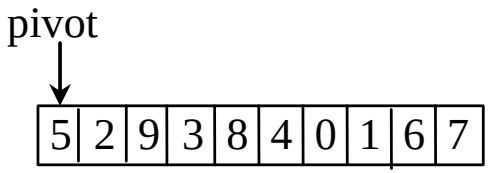
$$= 2^{\log n} T(\frac{n}{2^{\log n}}) + 2n - 2^{\log n - 1} + \ldots + 2n - 2 + 2n - 1$$

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

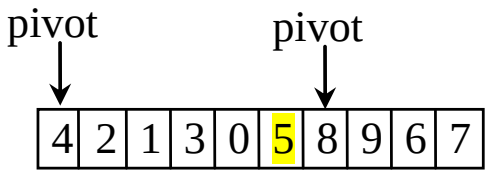
Quick Sort

- 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.

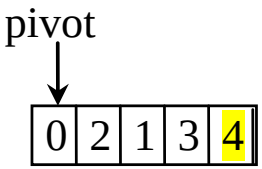
Quick Sort



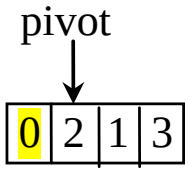
(a) The original array



(b)The original array is partitioned



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



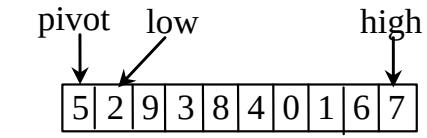
(d) The partial array (0 2 1 3) is partitioned

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

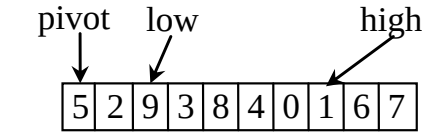
Partition

QuickSort

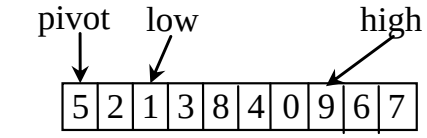
Run



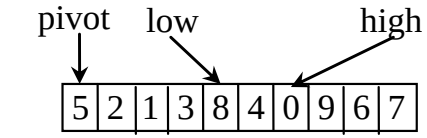
(a) Initialize pivot, low, and high



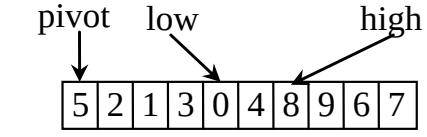
(b) Search forward and backward



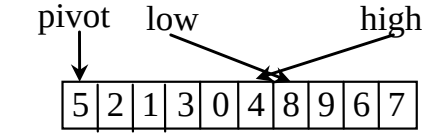
(c) 9 is swapped with 1



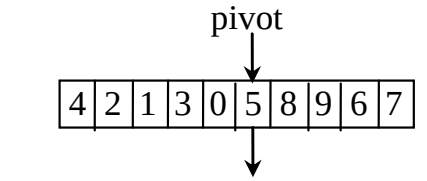
(d) Continue search



(e) 8 is swapped with 0



(f) when high < low, search is over



(g) pivot is in the right place

The index of the pivot is returned

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.

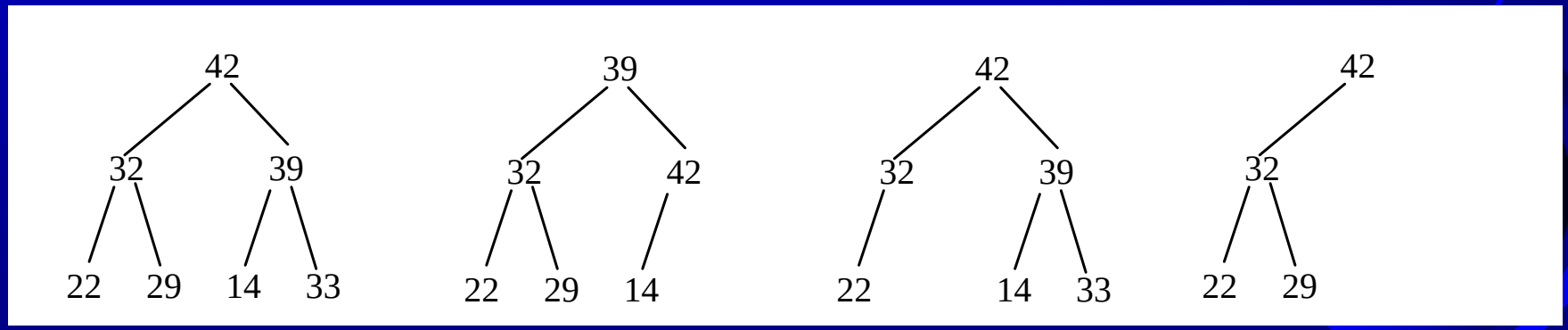
Heap

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).

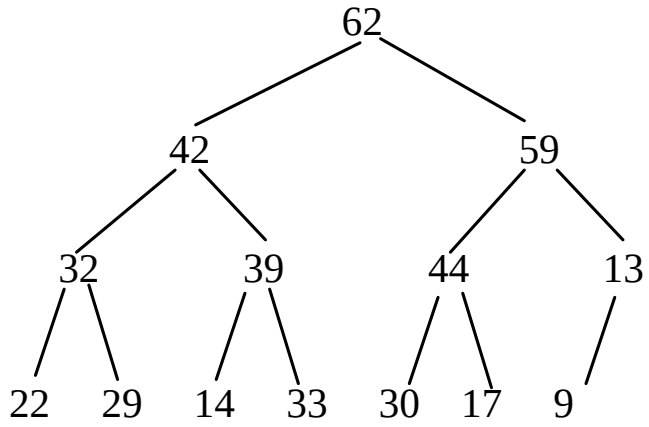


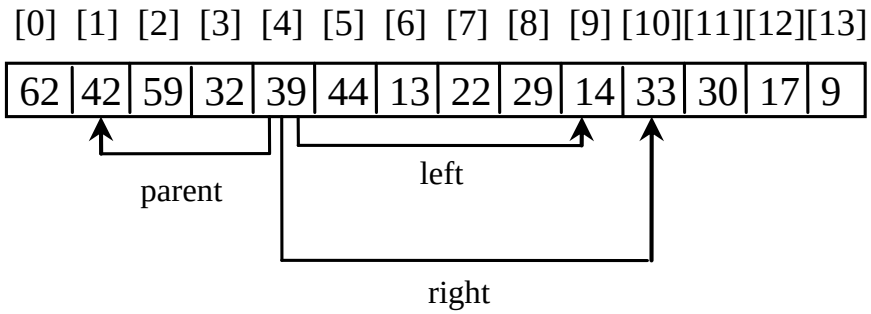
See How a Heap Works

http://www.cs.armstrong.edu/liang/animation/HeapAnimation.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 (*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*).

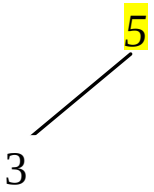




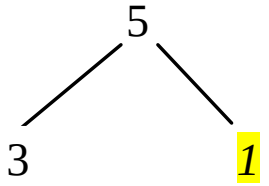
Adding Elements to the Heap

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

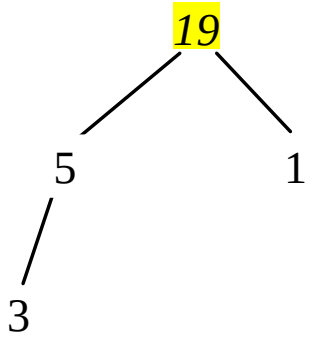
(a) After adding 3



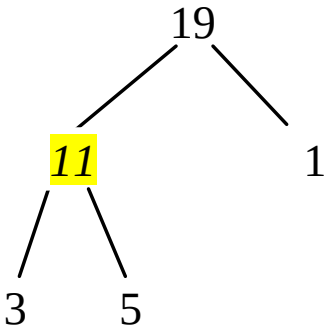
(b) After adding 5



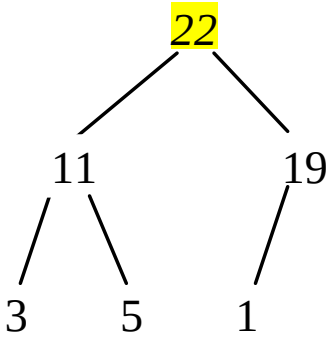
(c) After adding 1



(d) After adding 19



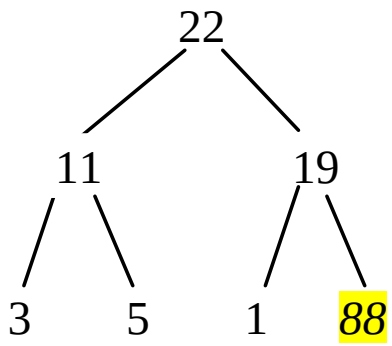
(e) After adding 11



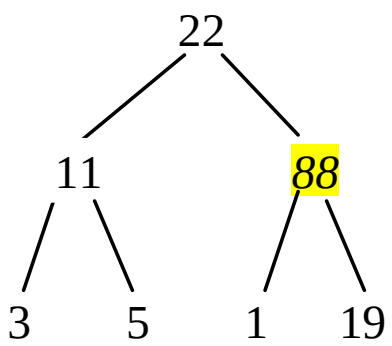
(f) After adding 22

Rebuild the heap after adding a new node

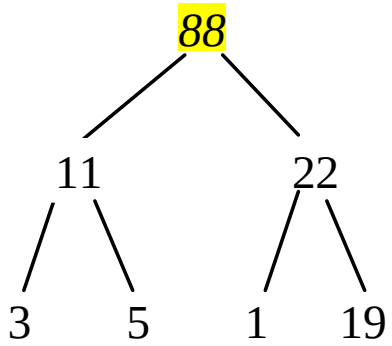
Adding 88 to the heap



(a) Add 88 to a heap



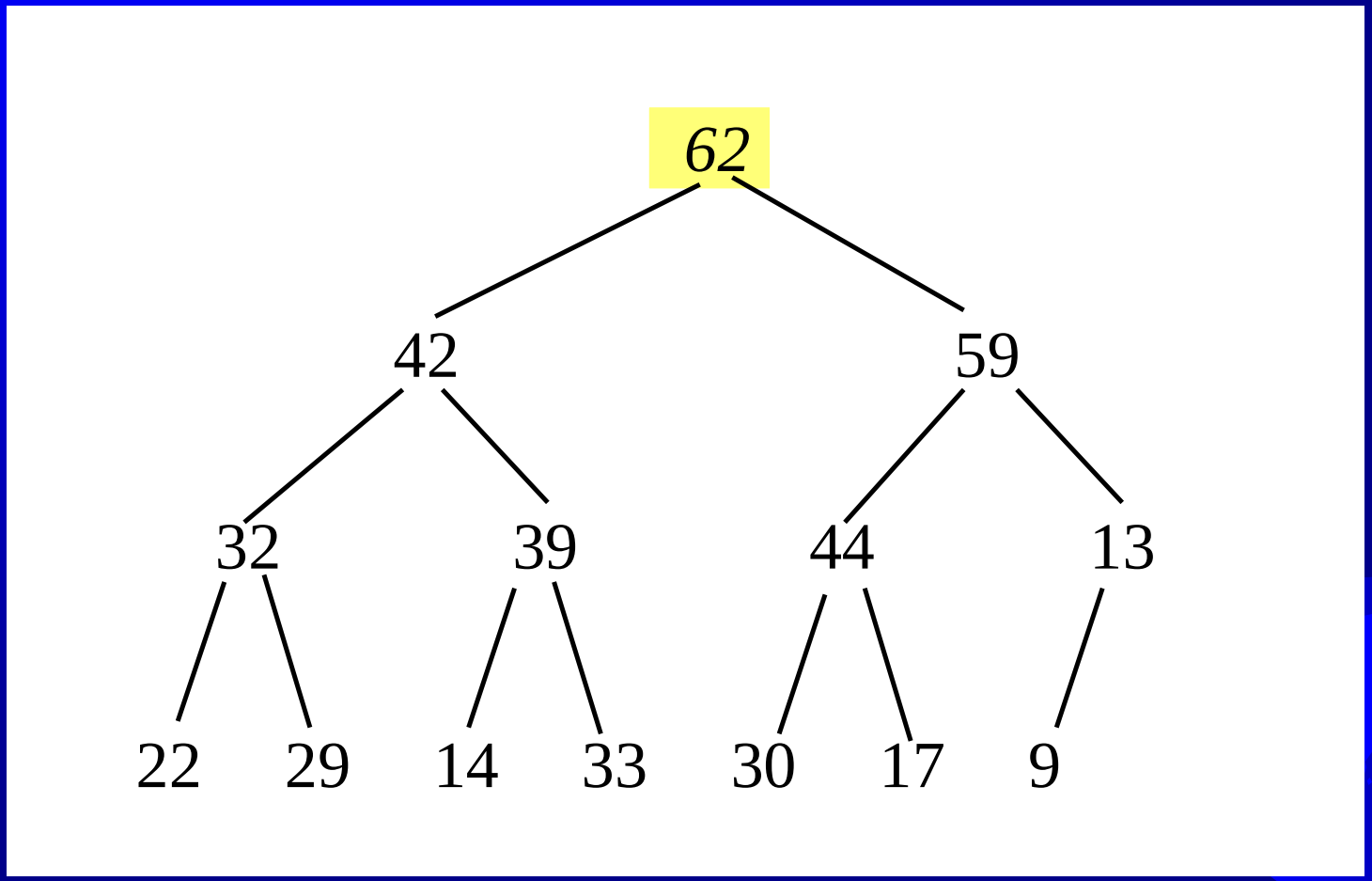
(b) After swapping 88 with 19



(b) After swapping 88 with 22

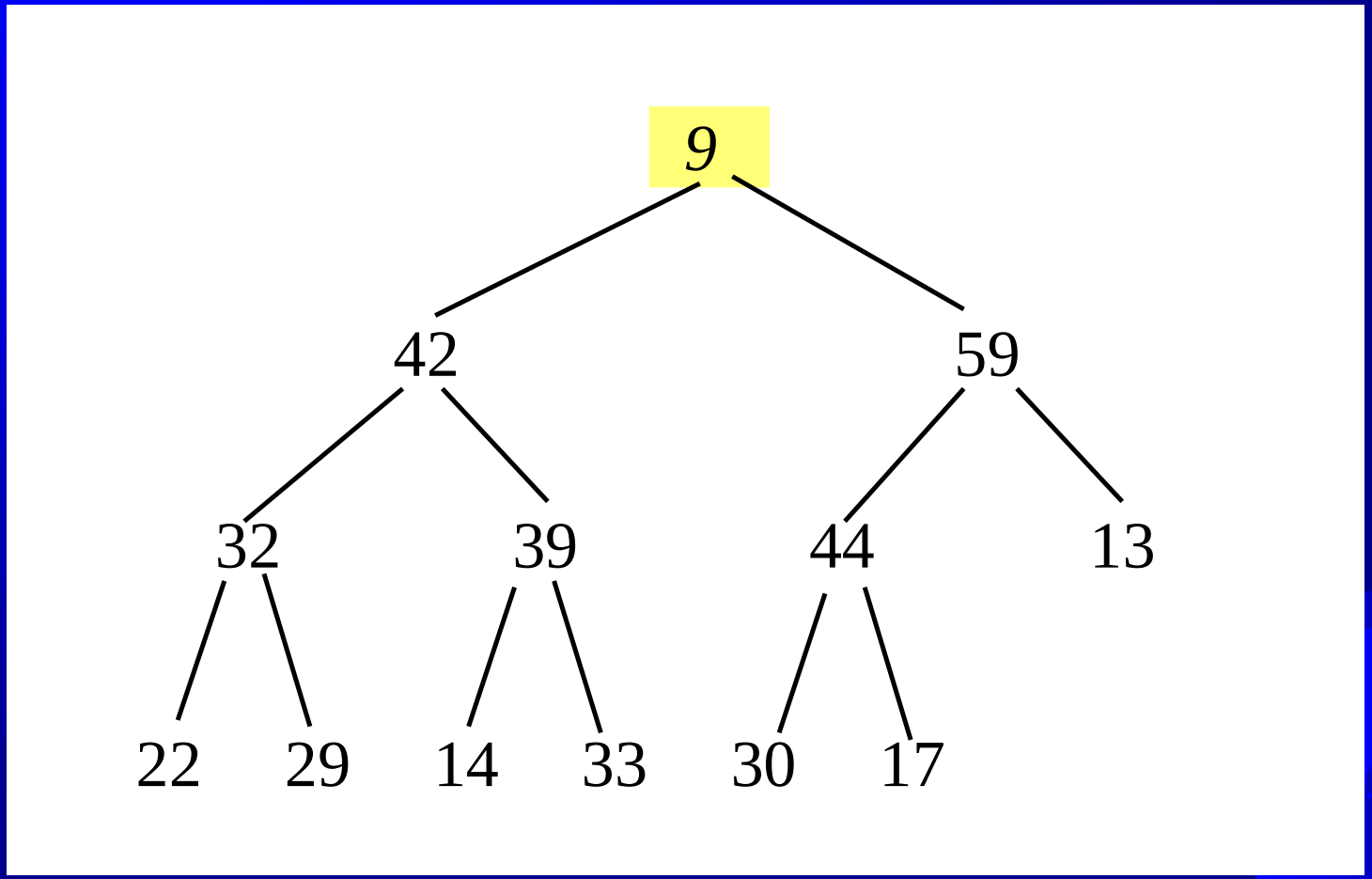
Removing the Root and Rebuild the Tree

Removing root 62 from the heap



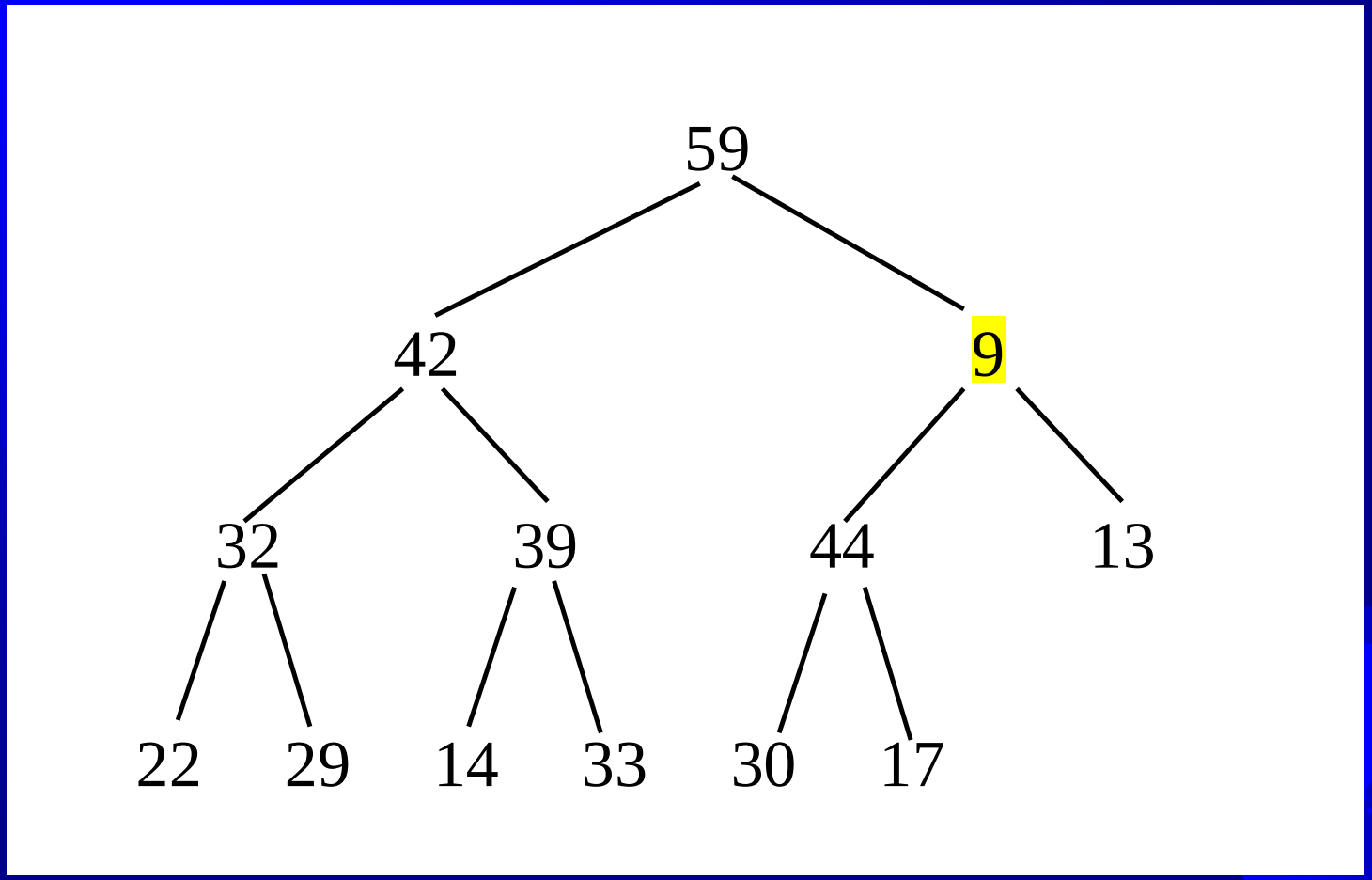
Removing the Root and Rebuild the Tree

Move 9 to root



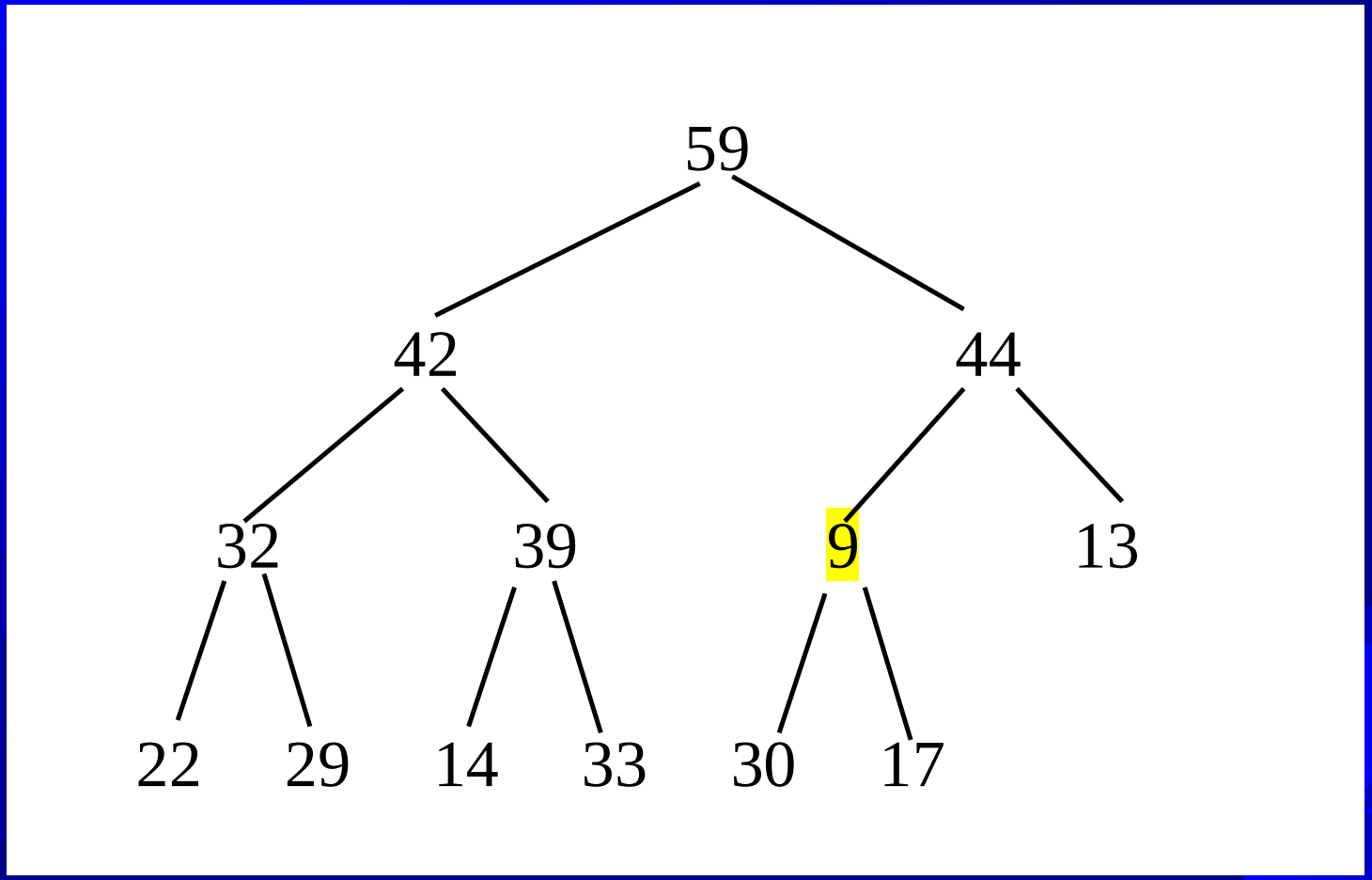
Removing the Root and Rebuild the Tree

Swap 9 with 59



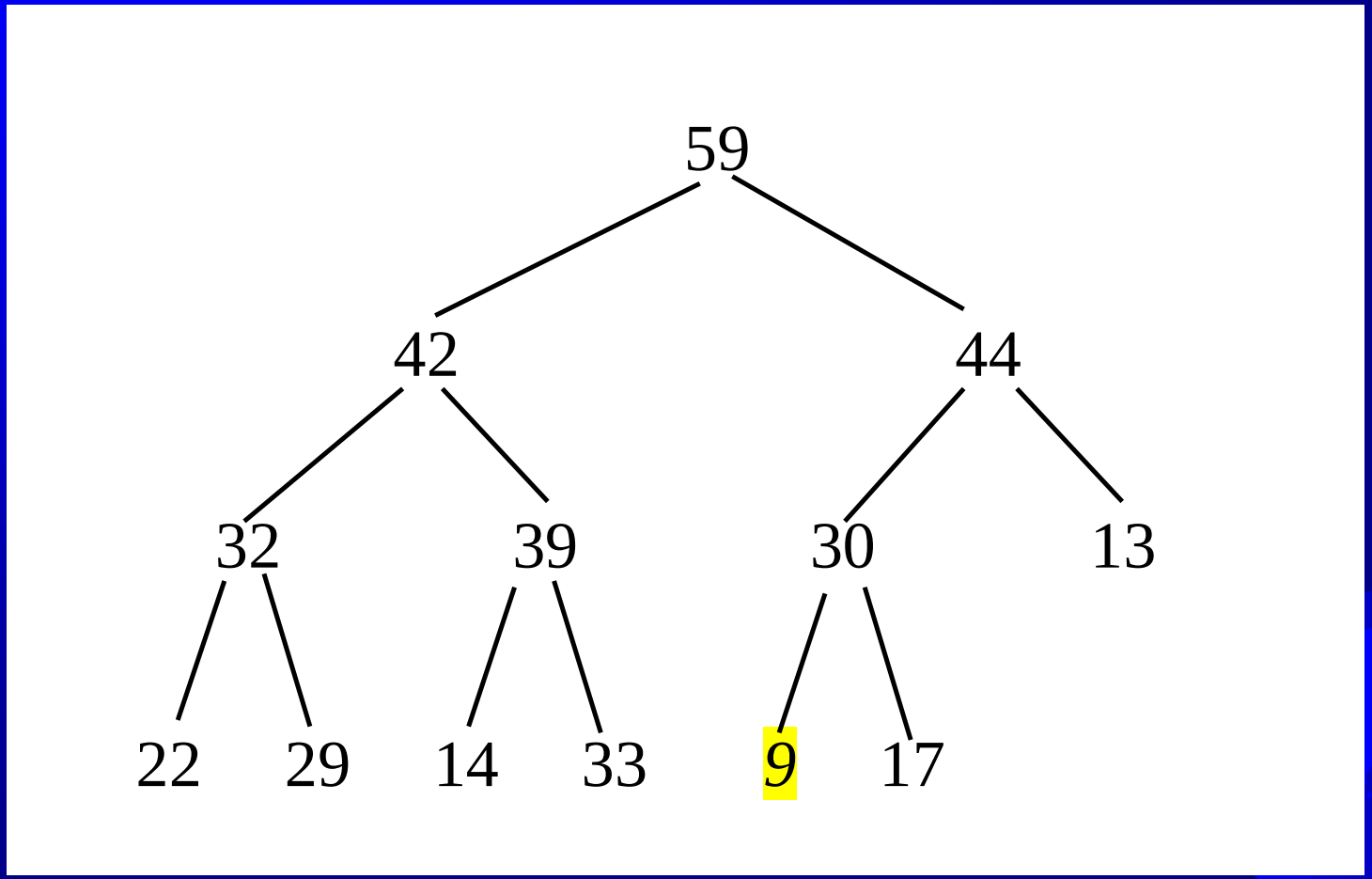
Removing the Root and Rebuild the Tree

Swap 9 with 44



Removing the Root and Rebuild the Tree

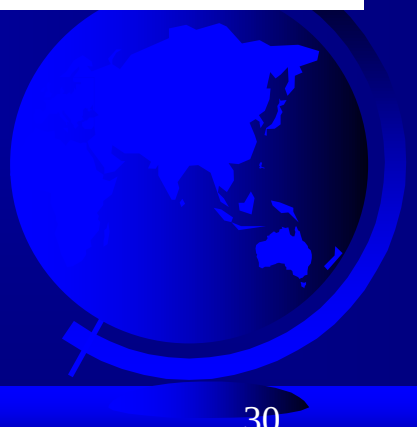
Swap 9 with 30



The Heap Class

Heap	
-lst: list	Values are stored in a list internally.
Heap()	Creates an empty heap.
add(e: object): None	Adds a new element to the heap.
remove(): object	Removes the root from the heap and returns it.
getSize(): int	Returns the size of the heap.
isEmpty(): bool	Returns True if the list is empty.
peek(): object	Returns the largest element in the heap without removing it.
getLst(): list	Returns the list for the heap.

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,

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

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

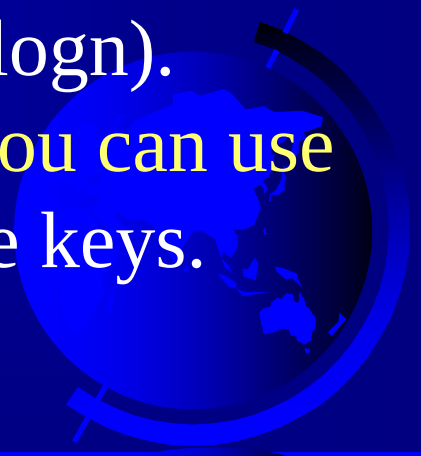
$$h-1 < \log(n+1) \le h \qquad \log(n+1) \le h < \log(n+1)+1$$

Heap Sort Time Complexity

- More precisely, you can prove that $h = \lfloor \log n \rfloor$ for a non-empty tree.
- Since the **add** method traces a path from a leaf to a root, it takes at most h 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 steps to rebuild a heap after removing the root from the heap. Since the **remove** method is invoked n times, the total time for producing a sorted array from a heap is $O(n \log n)$.
- Both merge and heap sorts require $O(n \log n)$ time. A 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

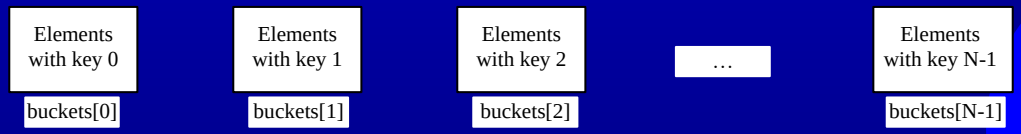
- All sort algorithms discussed so far are general sorting algorithms that work for any types of keys (e.g., integers, strings, and any comparable objects).
- These algorithms sort the elements by comparing their keys.
- The lower bound for general sorting algorithms is O(nlogn). So, no sorting algorithms 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.

Bucket Sort



Bucket Sort

- Bucket Sort considers that the input is generated by a random process that distributes elements uniformly over the interval μ=[0,1].
- To sort n input numbers, Bucket Sort
  1. Partition μ into n non-overlapping intervals called buckets.
  2. Puts each input number into its buckets
  3. Sort each bucket using a simple algorithm, e.g. Insertion Sort and then
  4. Concatenates the sorted lists.
- Bucket Sort considers that the input is an n element array A and that each element A [i] in the array satisfies $0 \leq A[i] < 1$. The code depends upon an auxiliary array B [0....n-1] of linked lists (buckets) and considers that there is a mechanism for maintaining such lists.

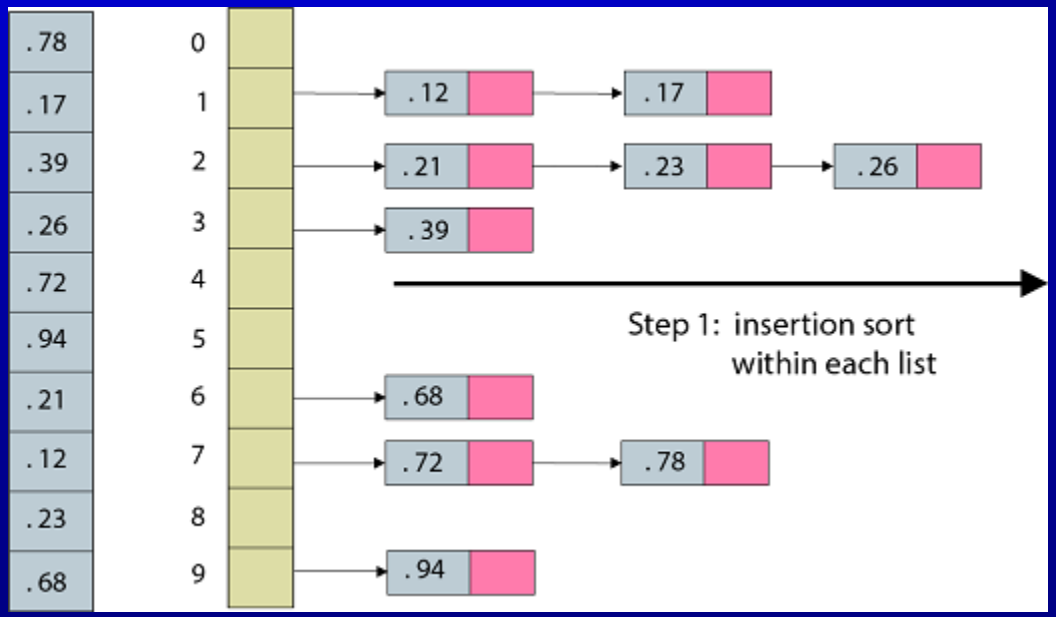
Bucket Sort

BUCKET-SORT (A)

1. n ← length [A]
2. for i ← 1 to n
3. do insert A [i] into list B [n A[i]]
4. for i ← 0 to n-1
5. do sort list B [i] with insertion sort.
6. Concatenate the lists B [0], B [1] ...B [n-1] together in order.

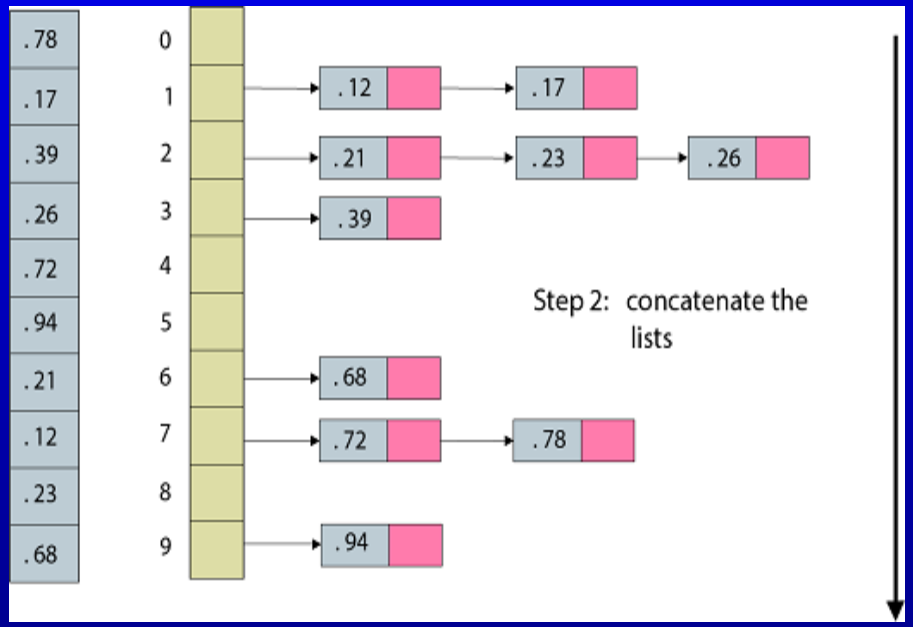
Bucket Sort

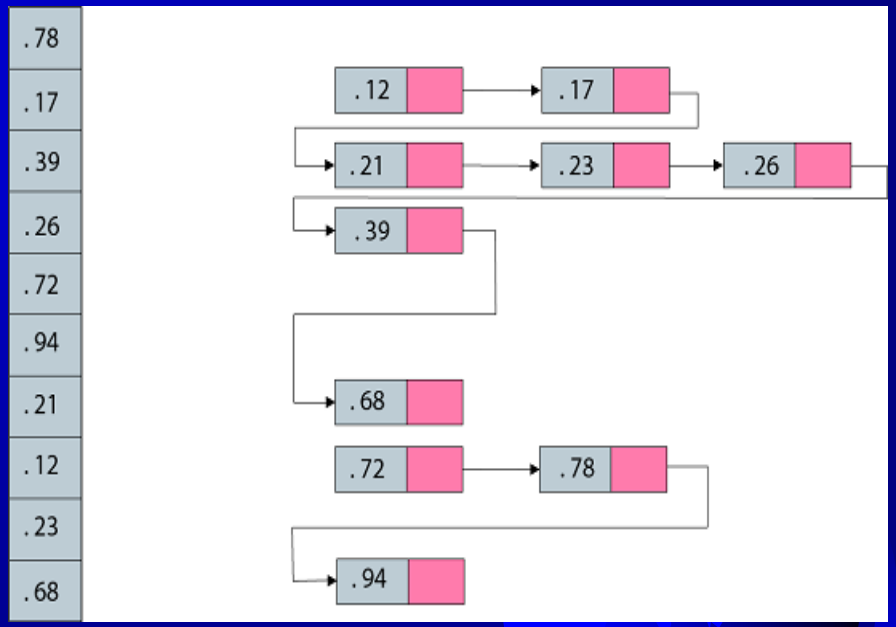
- A = (0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 068)
- Step 1: placing keys in bins in sorted order



Bucket Sort

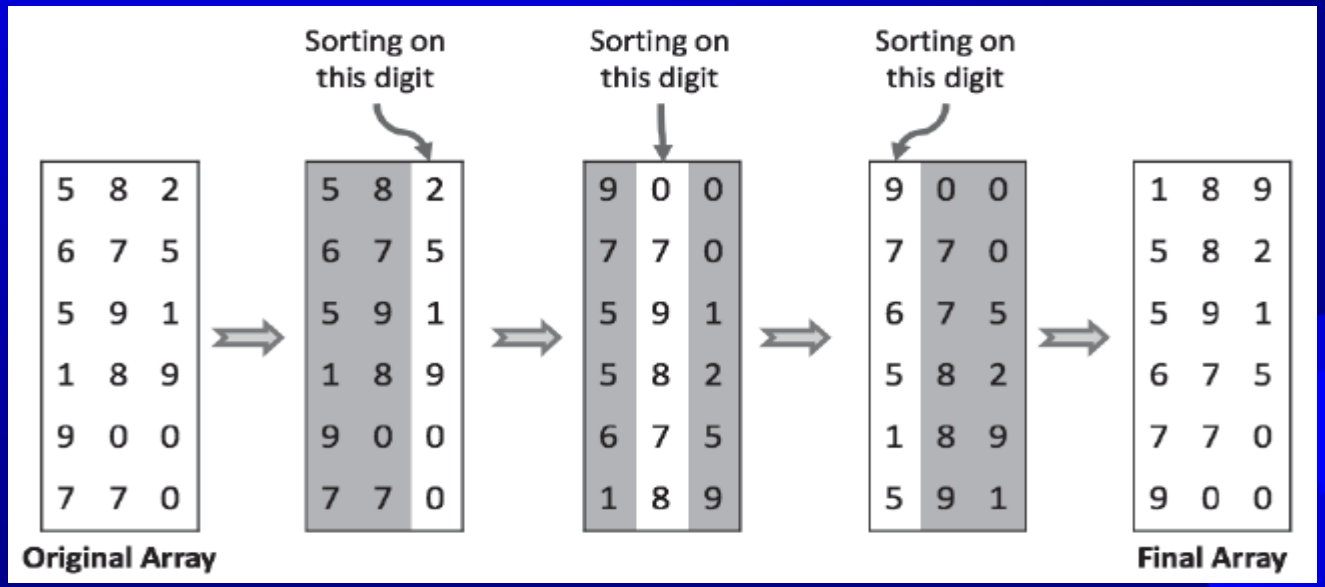
- Step 2: concatenate the lists





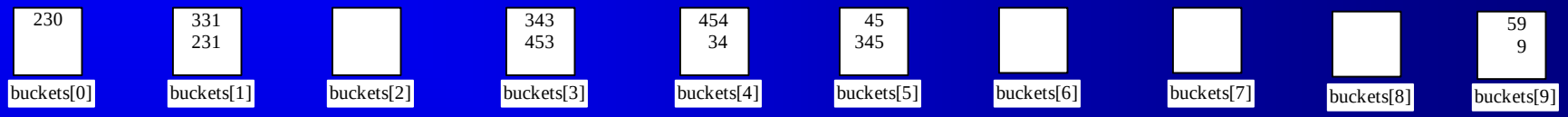
Radix Sort

- Assume 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.
- Example: sort a = [582, 675, 591, 189, 900, 770]

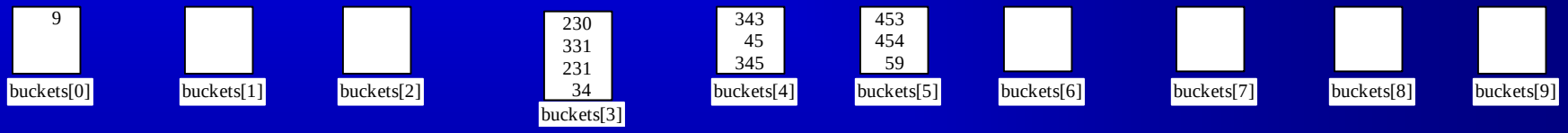


Radix Sort

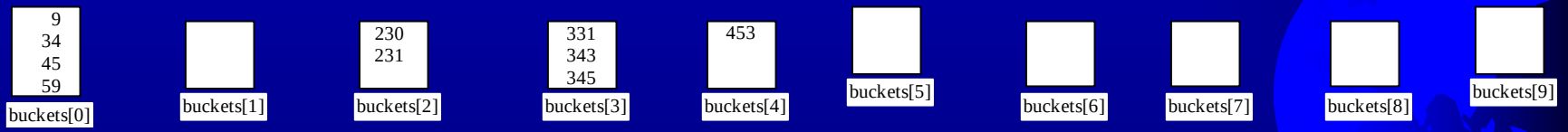
Sort 331, 454, 230, 34, 343, 45, 59, 453, 345, 231, 9



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

Radix Sort Animation

http://www.cs.armstrong.edu/liang/animation/RadixSortAnimation.html