CSE332: Data Abstractions

Section 6

HyeIn Kim

Spring 2013

Section Agenda

- **Sorting Algorithms**
  - Insertion & Selection Sort
  - Heap & AVL Sort
  - Merge & Quick Sort
  - Bucket & Radix Sort

- **Homework 4**
  - Q & A

Sorting Algorithms

Comparison & Non-comparison based sorting

Sorting

- **Sorting**

  Rearranging elements in collection into a specific order

- Can be solved in many ways

- **Comparison-based sorting**

  Determining order by comparing pairs of elements

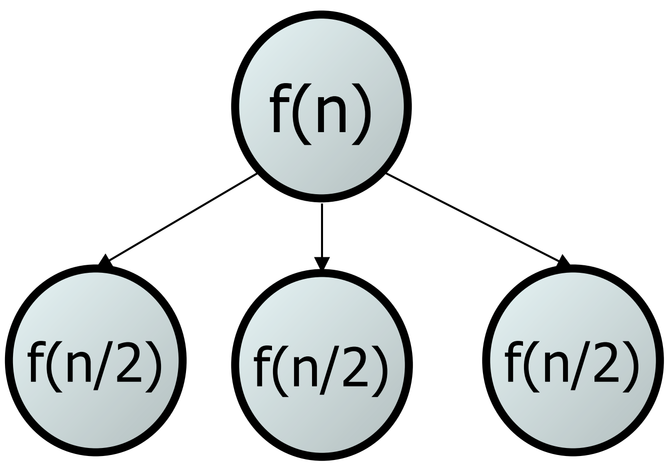
  Insertion sort, selection sort, quick sort ...

Recurrence Relations

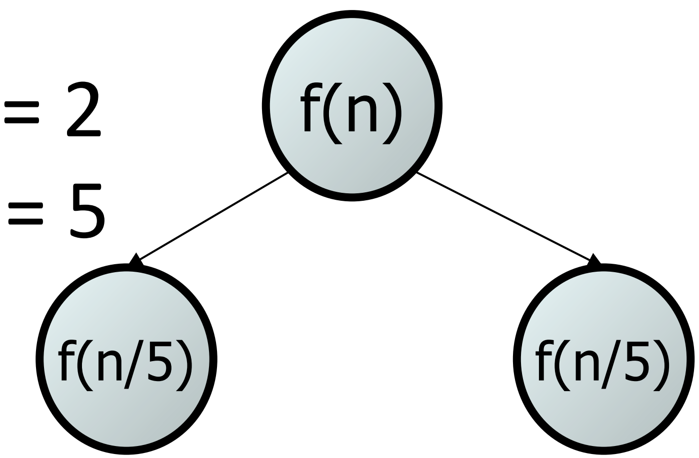
- **T(n) = a * T(n / b) + f(n)**

  - a: Branching factor
  - b: Work reduction
  - f(n): Work

a = 3
b = 2



a = 2
b = 5



Insertion Sort

- At k^{th} step, insert k^{th} element in correct position among the first k elements
- At k^{th} step, the first k elements are sorted
- Works well when input is mostly sorted

Insertion sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42

Insert 18

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42

Insert 12

index	0	1	2	3	4	5	6	7
value	18	22	12	-4	58	7	31	42

Insert -4

index	0	1	2	3	4	5	6	7
value	12	18	22	-4	58	7	31	42

Insert 58

index	0	1	2	3	4	5	6	7
value	-4	12	18	22	58	7	31	42

Insertion sort example

Insert 7

index	0	1	2	3	4	5	6	7
value	-4	12	18	22	58	7	31	42

Insert 31

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	58	31	42

Insert 42

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	31	58	42

Sorted!

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	31	42	58

Insertion Sort Runtime

- **Base Case:** T(1) = c

  - Sorting 1 element take constant time

- **Recurrence Relation**

  - When input is sorted:

    Time for Sorting n elements

    = (Time for sorting n – 1 elements) + (1 comparisons)

$$T(n) = T(n-1) + 1, \qquad T(n) \in O(n)$$

Insertion Sort Runtime

- **Recurrence Relation**
  - When input is unsorted:

    Time for Sorting n elements

    = (Time for sorting n – 1 elements) + (n – 1 comparisons)

    $T(n) = T(n-1) + (n-1),$ $T(n) \in O(n^2)$

Selection Sort

- At k^{th} step, find smallest value from unsorted items and place it in position k
- At k^{th} step, the first k elements are sorted, and are the smallest elements

Selection sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42

Min: -4

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42

Min: 7

index	0	1	2	3	4	5	6	7
value	-4	18	12	22	58	7	31	42

Min: 12

index	0	1	2	3	4	5	6	7
value	-4	7	12	22	58	18	31	42

Min: 18

index	0	1	2	3	4	5	6	7
value	-4	7	12	22	58	18	31	42

Selection sort example

Min: 22

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	58	22	31	42

Min: 31

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	58	31	42

Min: 42

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	31	58	42

Sorted!

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	31	42	58

Selection Sort Runtime

- **Base Case:** $T(1) = c$

  - Sorting 1 element take constant time

- **Recurrence Relation**

  - At each step, work decrease by 1

  - At each step, need to compare all remaining elements to find the minimum

$T(n) = T(n-1) + n,$ $\quad T(n) \in O(n^2)$

AVL Sort

- **Use AVL tree to sort**

  - Insert each element into AVL Tree

  - Find and delete smallest element from AVL Tree

- **Space Requirement**

  - Need another data structure (AVL Tree)

  - Tree needs to store things other than data (Pointer to child nodes)

AVL Sort Runtime

- **Base Case:** $T(1) = c$

  - Sorting 1 element take constant time

- **Recurrence Relation**

  - At each step, work decrease by 1

  - At each step, need log n work

$$T(n) = T(n-1) + \log n, \qquad T(n) \in O(n \log n)$$

$$n \log n + n \log n \in O(n \log n)$$

Heap Sort

- **Use Heap to sort**

 - Insert each element into Heap

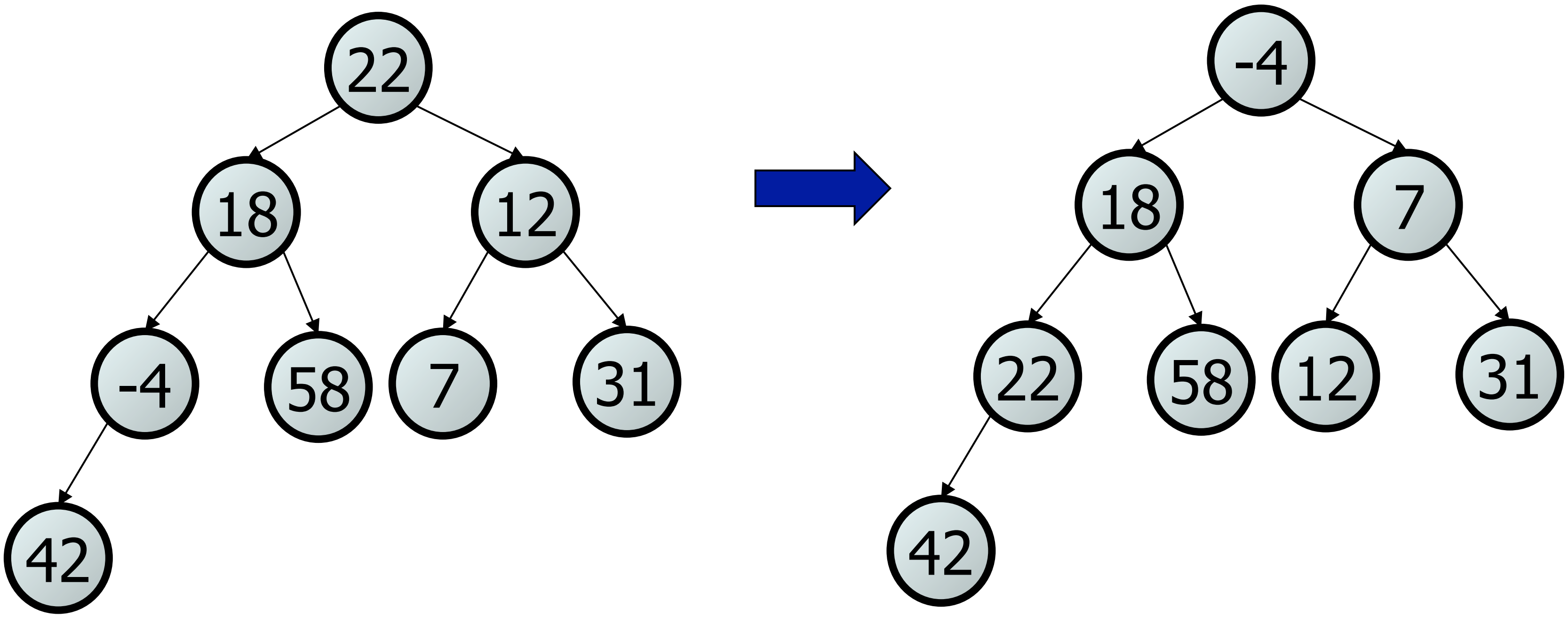
 - Call deleteMin() to get minimum element

- **Space Requirement**

 - Can do in-place sorting, using buildHeap()

Heap sort example

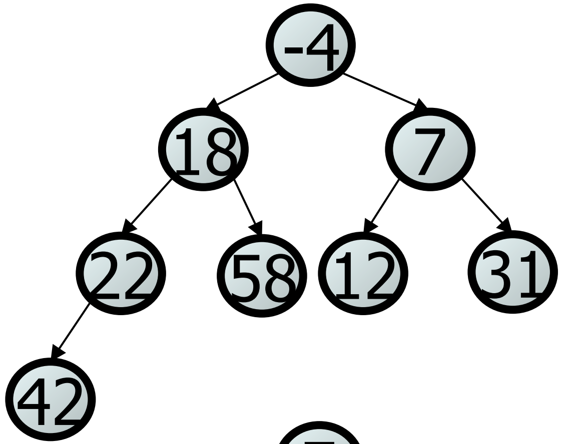
index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



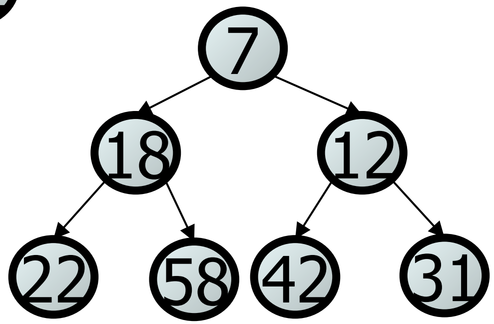
Build Heap

index	0	1	2	3	4	5	6	7
value	-4	18	7	22	58	12	31	42

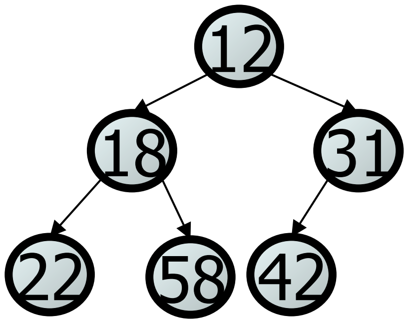
Heap sort example



index	0	1	2	3	4	5	6	7
value	-4	18	7	22	58	12	31	42

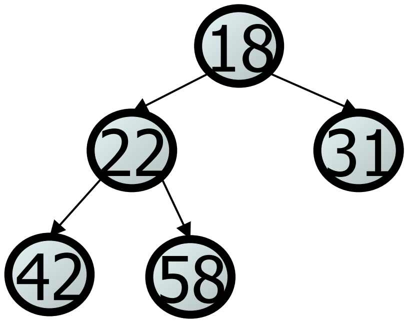


index	0	1	2	3	4	5	6	7
value	7	18	12	22	58	42	31	-4

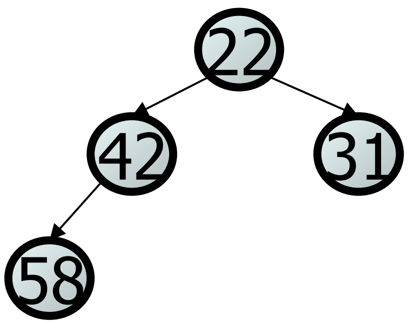


index	0	1	2	3	4	5	6	7
value	12	18	31	22	58	42	7	-4

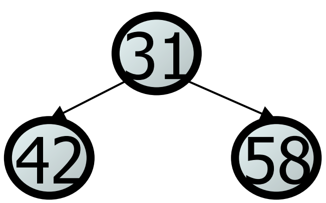
Heap sort example



index	0	1	2	3	4	5	6	7
value	18	22	31	42	58	12	7	-4

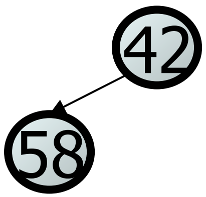


index	0	1	2	3	4	5	6	7
value	22	42	31	58	18	12	7	-4



index	0	1	2	3	4	5	6	7
value	31	42	58	22	18	12	7	-4

Heap sort example



index	0	1	2	3	4	5	6	7
value	42	58	31	22	18	12	7	-4

index	0	1	2	3	4	5	6	7
value	58	42	31	22	18	12	7	-4

Reverse & Sorted!

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	31	42	58

Heap Sort Runtime

- **Base Case:** $T(1) = c$

  - Sorting 1 element take constant time

- **Recurrence Relation**

  - Build heap takes O(n) work
  - deleteMin: At each step, work decrease by 1
  - deleteMin: At each step, need log n work

$T(n) = T(n-1) + \log n, \qquad T(n) \in O(n \log n)$

$n + n \log n \in O(n \log n)$

Merge Sort

- **Divide & Conquer**

- Divide into two roughly equal halves.

- Sort each halves

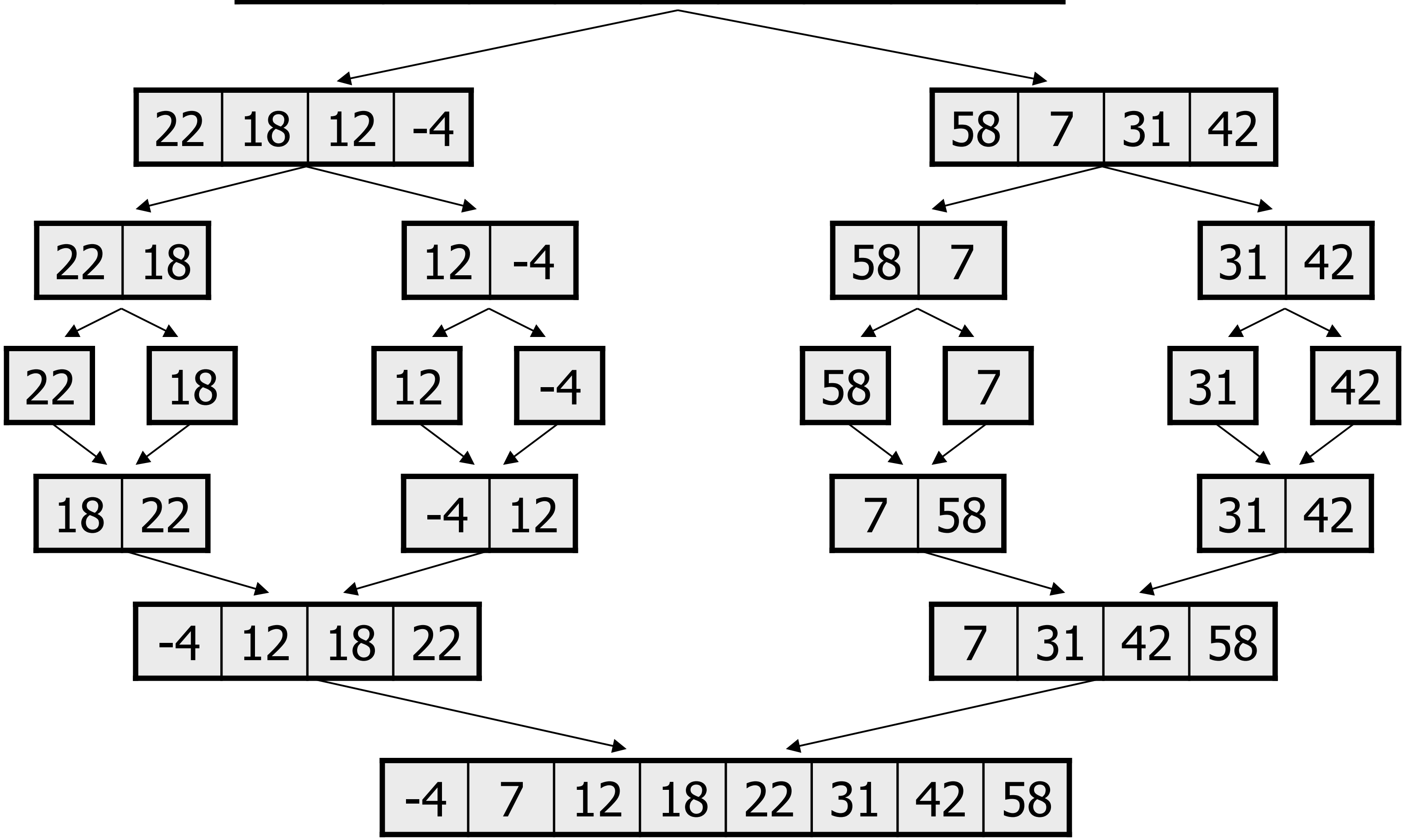
- Merge two sorted halves

- **Parallelizes Well**

- Multiple processors can work on different parts of array

Merge sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



Merging sorted halves

Subarrays								Next include	Merged array							
0	1	2	3	0	1	2	3		0	1	2	3	4	5	6	7
14	32	67	76	23	41	58	85	14 from left	14							
i1				i2					i							
14	32	67	76	23	41	58	85	23 from right	14	23						
	i1			i2						i						
14	32	67	76	23	41	58	85	32 from left	14	23	32					
	i1				i2						i					
14	32	67	76	23	41	58	85	41 from right	14	23	32	41				
		i1			i2							i				
14	32	67	76	23	41	58	85	58 from right	14	23	32	41	58			
		i1				i2							i			
14	32	67	76	23	41	58	85	67 from left	14	23	32	41	58	67		
		i1					i2							i		
14	32	67	76	23	41	58	85	76 from left	14	23	32	41	58	67	76	
			i1				i2								i	
14	32	67	76	23	41	58	85	85 from right	14	23	32	41	58	67	76	85
							i2									i

Merge Sort Runtime

- **Base Case:** $T(1) = c$

  - Sorting 1 element take constant time

- **Recurrence Relation**

  - At each step, branch into two
  - At each step, work decrease by half
  - At each step, need to visit all elements

$$T(n) = 2*T(n/2) + n, \qquad T(n) \in O(n \log n)$$

Quick Sort

- **Divide & Conquer**

- Divide into two pieces

- Sort each piece

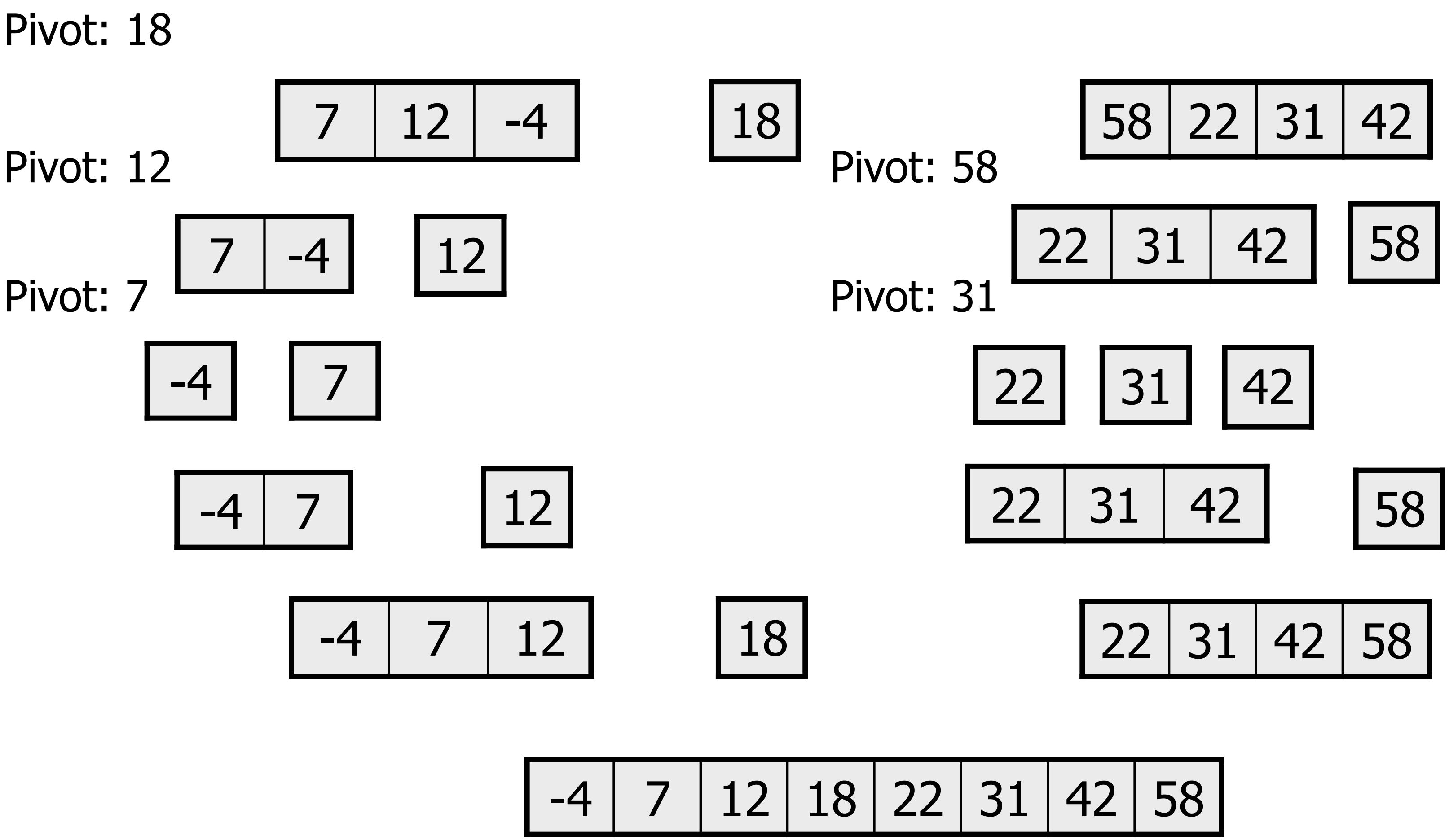
- Merge two sorted piece

- Pick pivot, partition into < pivot & > pivot

- Less copying & more comparisons compared to merge sort

Quick sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



Quick Sort Runtime

- **Base Case:** $T(1) = c$

  - Sorting 1 element take constant time

- **Recurrence Relation**

  - When pivot is the best:

    At each step, work decrease by half

    At each step, need to visit all elements

$$T(n) = 2*T(n/2) + n, \qquad T(n) \in O(n \log n)$$

Quick Sort Runtime

- **Recurrence Relation**
  - When pivot is the worst:

    At each step, work decrease by 1

    At each step, need to visit all elements

$$T(n) = T(n-1) + n, \qquad T(n) \in O(n^2)$$

Bucket Sort

- **No Comparisons**

  - Create a bucket for every possible elements in input
  - Store counts for occurrence in corresponding bucket

- **Runtime:** O(n + k)

  - k = range of possible values (size of bucket)
  - Good for small k
  - When k >>> n, space can be wasted

Bucket sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42

First pass, find range (K): Min = -4, Max = 58

K = Max – Min + 1 = 63

Bucket	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
count	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bucket	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
count	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bucket	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
count	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bucket	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	
count	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Bucket sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42

Second pass, Count occurrences

Bucket	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
count	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Bucket	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
count	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
Bucket	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
count	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
Bucket	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	
count	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Bucket sort example

Bucket	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
count	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Bucket	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
count	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
Bucket	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
count	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
Bucket	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	
count	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	

Third pass, Print occurrences

Sorted!

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	31	42	58

Radix Sort

- **No Comparisons**

- Bucket sort on 1 digit at a time

- After k passes, last k digits are sorted

- **Runtime:** O(d*(n + k))

- k = radix (number of buckets)

- d = max number of digit = $\log_k$ (Max element)

Radix sort example

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42

First pass, Sort by 1's digit:

Digit	0	1	2	3	4	5	6	7	8	9
values					-4					
Digit	0	1	2	3	4	5	6	7	8	9
values		31	12 22 42					7	18 58	

Radix sort example

Digit	0	1	2	3	4	5	6	7	8	9
values					-4					
Digit	0	1	2	3	4	5	6	7	8	9
values		31	12 22 42					7	18 58	

Second pass, Sort by 10's digit:

Digit	0	1	2	3	4	5	6	7	8	9
values	-4									
Digit	0	1	2	3	4	5	6	7	8	9
values	7	12 18	22	31	42	58				

Radix sort example

Digit	0	1	2	3	4	5	6	7	8	9
values	-4									
Digit	0	1	2	3	4	5	6	7	8	9
values	7	12 18	22	31	42	58				

Write out the values:

Sorted!

index	0	1	2	3	4	5	6	7
value	-4	7	12	18	22	31	42	58