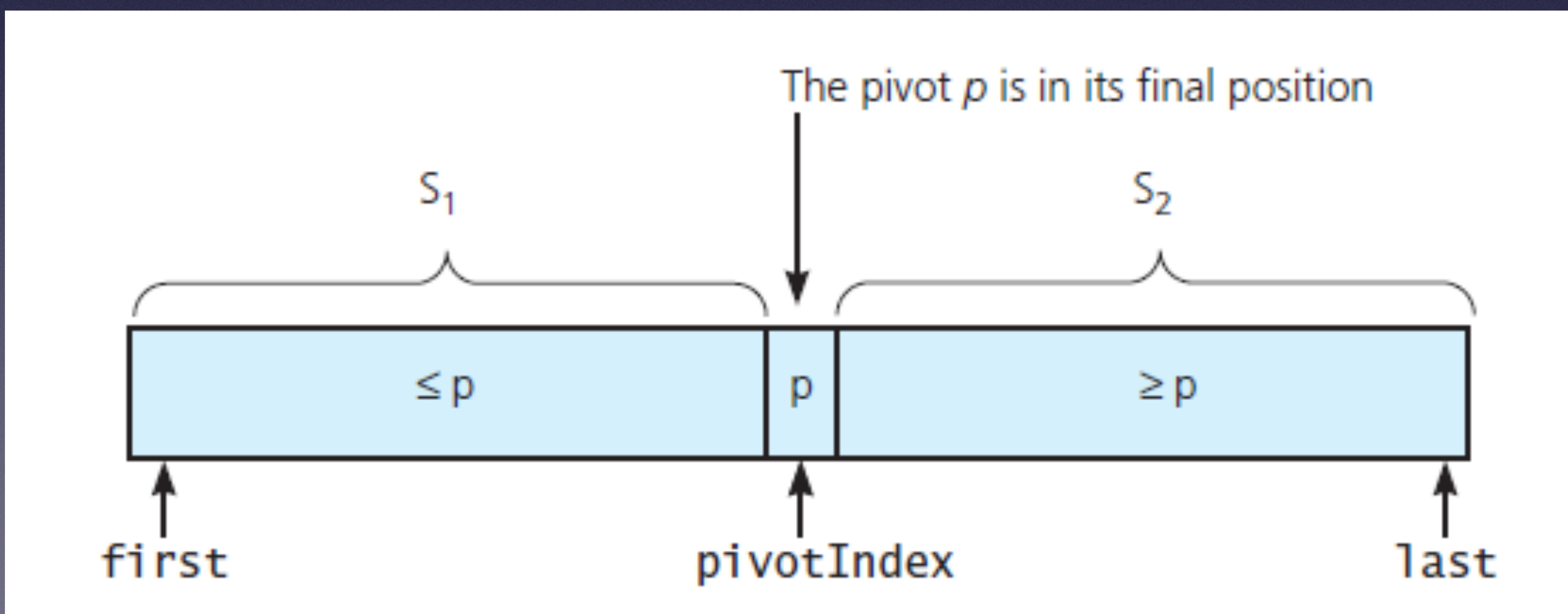
Quick Sort

CS110C

Max Luttrell, CCSF

quick sort

- Partition your array into two parts (S1 and S2), around a **pivot** p, such that:
  - all items in S1 <= p; all items in S2 >= p
- Note: this array isn't sorted, but we know that items in S1 remain in S1 after array is sorted, and likewise for S2. The pivot remains in its current position.



quick sort pseudocode

```
// perform quickSort on theArray between indices first and last
quickSort(theArray: ItemArray, first: integer, last: integer)
{
  if (first<last)
  {
    Choose a pivot p from theArray[first..last]
    Partition the items of theArray[first..last] about p, where
      p is at position pivotIndex

    // recursively sort first partition of array
    quickSort(theArray, first, pivotIndex-1)

    // recursively sort second partition of array
    quickSort(theArray, pivotIndex+1, last)
  }
}
```

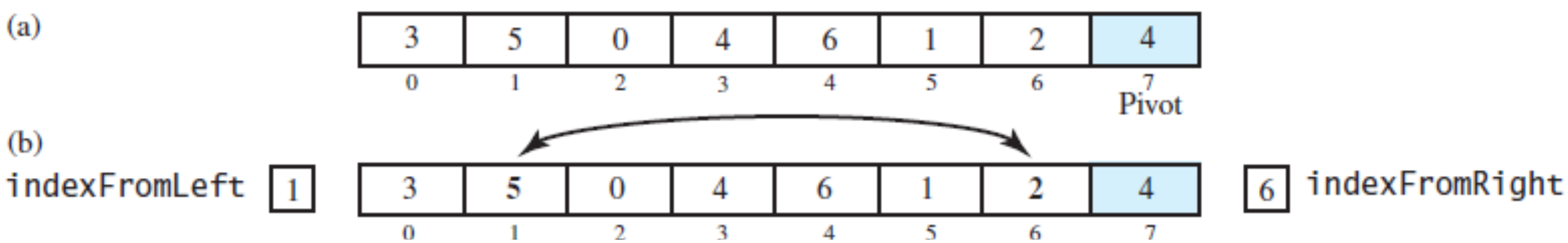
quick sort partitioning example

(a)

3	5	0	4	6	1	2	4
0	1	2	3	4	5	6	7

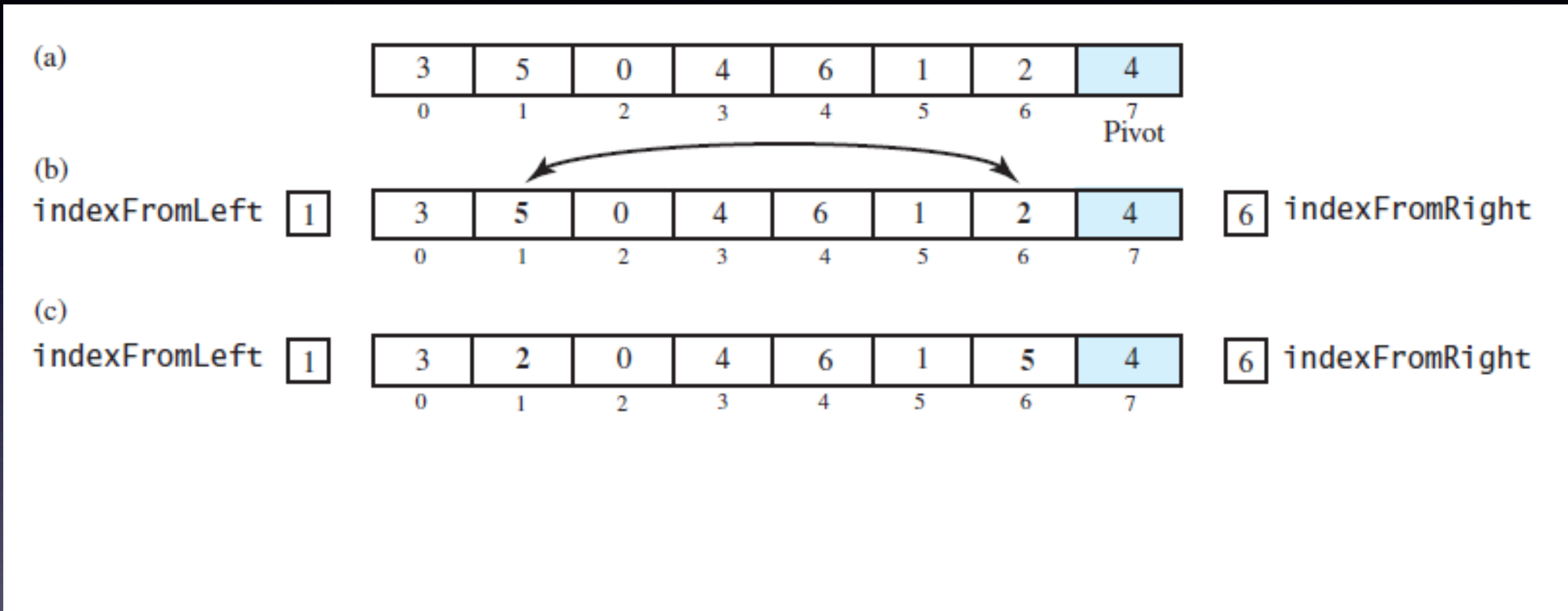
- Let's say we have chosen our pivot to be 4. Step (a) shows it swapped to the last position in array to get it out of the way while we do a partition.

quick sort partitioning example



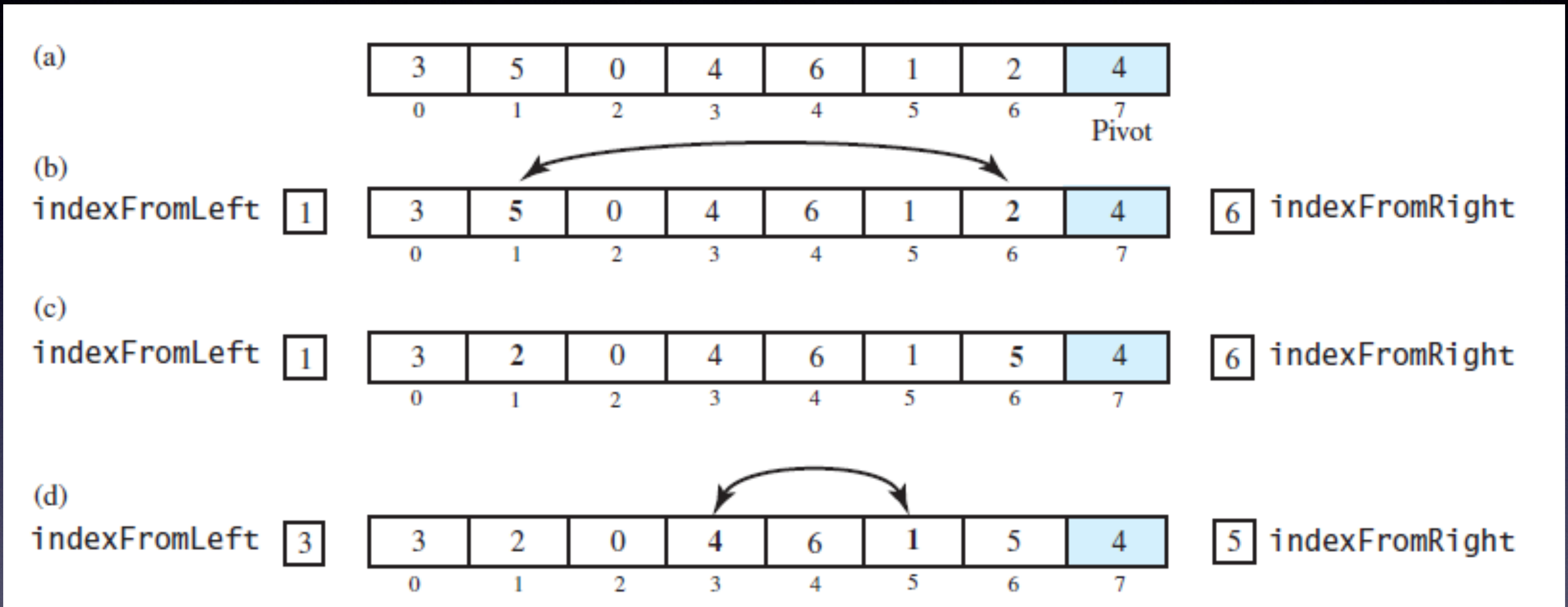
- Let's say we have chosen our pivot to be 4. Step (a) shows it swapped to the last position in array to get it out of the way while we do a partition.

quick sort partitioning example



- Let's say we have chosen our pivot to be 4. Step (a) shows it swapped to the last position in array to get it out of the way while we do a partition.

quick sort partitioning example



- Let's say we have chosen our pivot to be 4. Step (a) shows it swapped to the last position in array to get it out of the way while we do a partition.

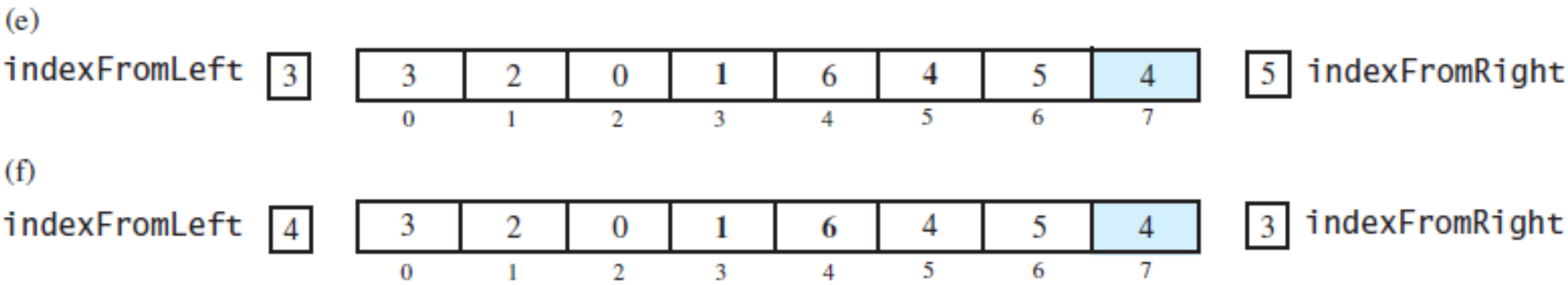
quick sort partitioning

(e)

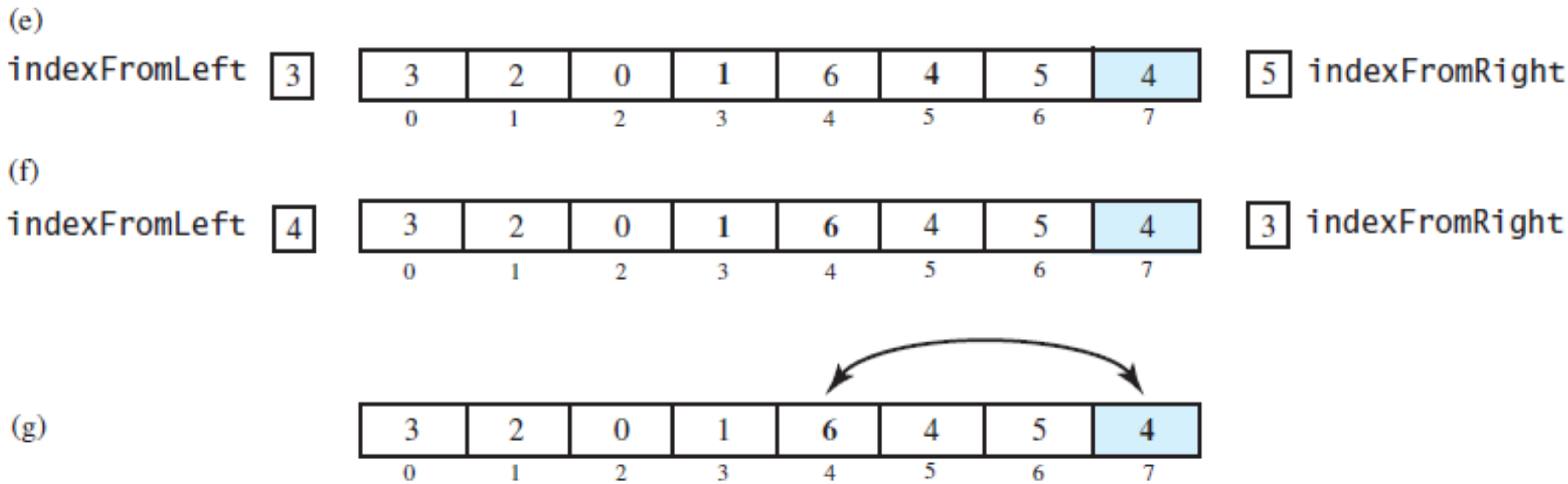
indexFromLeft $\boxed{3}$ | indexFromRight $\boxed{5}$

3	2	0	1	6	4	5	4
0	1	2	3	4	5	6	7

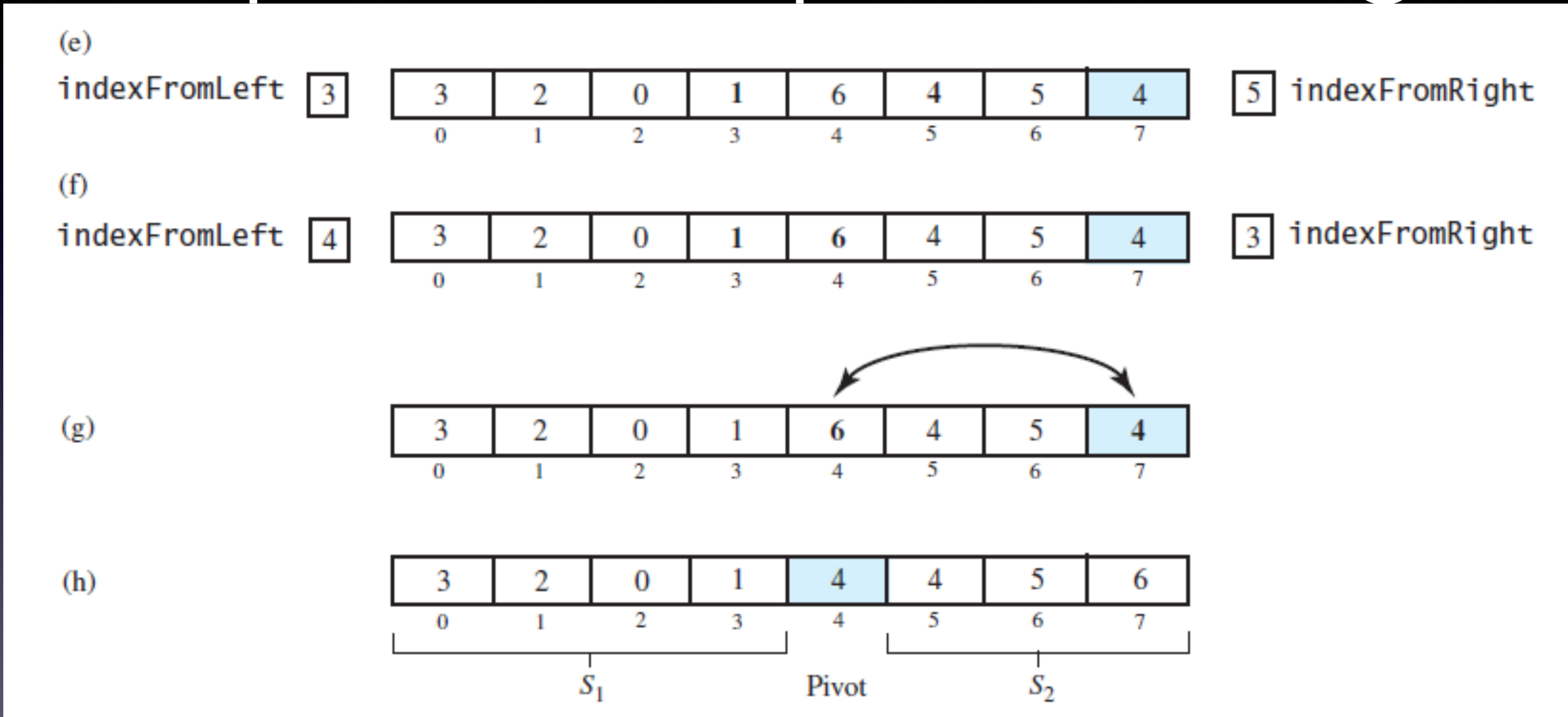
quick sort partitioning



quick sort partitioning



quick sort partitioning



quick sort partitioning

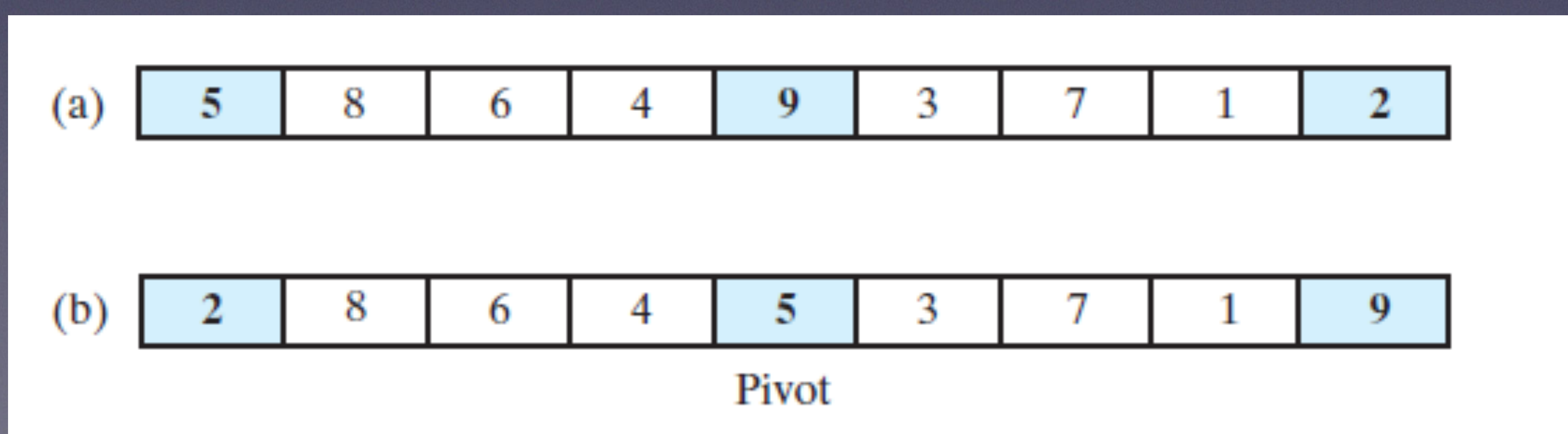
```
template<class ItemType>
int partition(ItemType theArray[], int first, int last)
{
   // Choose pivot using median-of-three selection
   int pivotIndex = sortFirstMiddleLast(theArray, first, last);

   // Reposition pivot so it is last in the array
   std::swap(theArray[pivotIndex], theArray[last - 1]);
   pivotIndex = last - 1;
   ItemType pivot = theArray[pivotIndex];

   // Determine the regions S1 and S2
   int indexFromLeft = first + 1;
   int indexFromRight = last - 2;

... continued
```

quick sort partitioning

```
... continued
   bool done = false;
   while (!done)
   {
      // Locate first entry on left that is >= pivot
      while (theArray[indexFromLeft] < pivot)
         indexFromLeft = indexFromLeft + 1;

      // Locate first entry on right that is <= pivot
      while (theArray[indexFromRight] > pivot)
         indexFromRight = indexFromRight - 1;

      if (indexFromLeft < indexFromRight)
      {
         std::swap(theArray[indexFromLeft], theArray[indexFromRight]);
         indexFromLeft = indexFromLeft + 1;
         indexFromRight = indexFromRight - 1;
      }
      else
         done = true;
   }  // end while

   // Place pivot in proper position between S1 and S2, and mark its new location
   std::swap(theArray[pivotIndex], theArray[indexFromLeft]);
   pivotIndex = indexFromLeft;

   return pivotIndex;
}  // end partition
```

quick sort - pivot

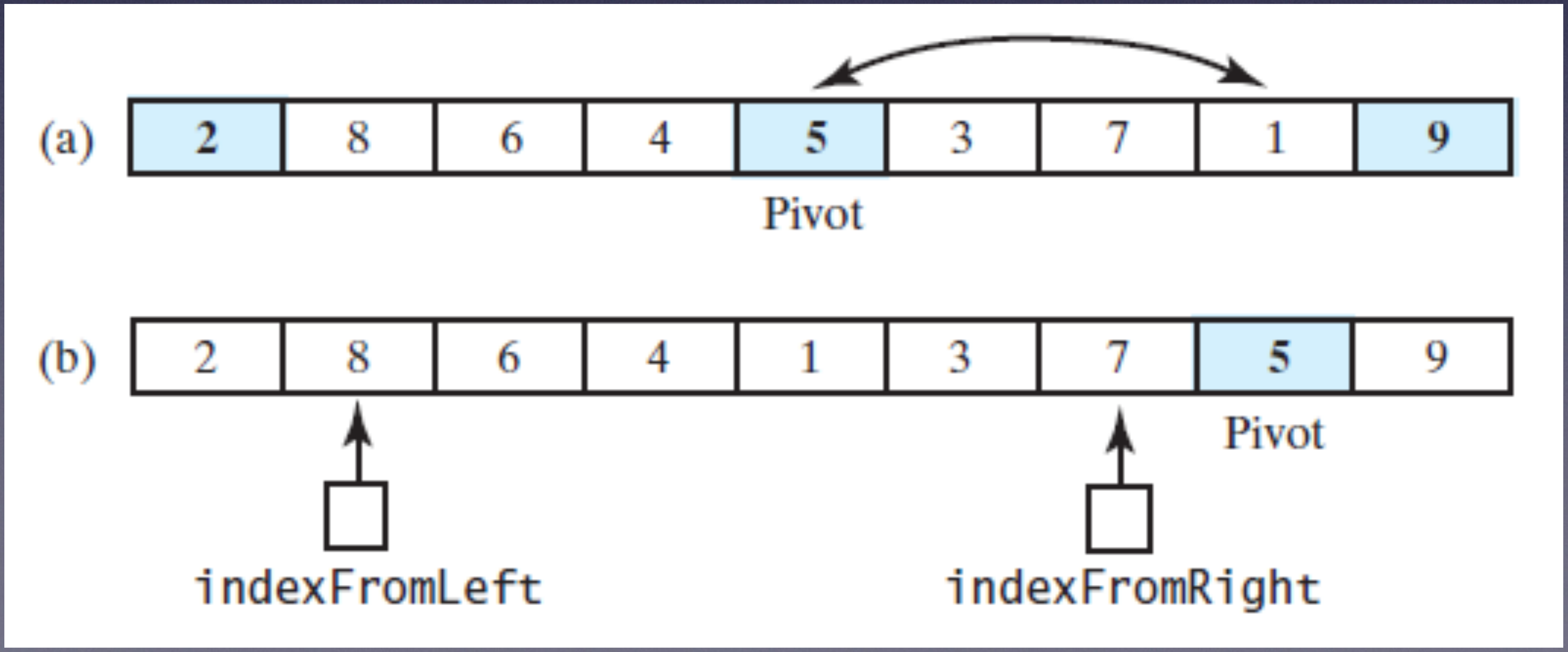
- Selecting a pivot -- how?
  - Simplistic approach: pick the first element in the partition. But this can lead to undesirable performance.
  - Ideal approach: pick the median value. But that's impractical.
  - In practice, we at least try to avoid a bad pivot. We can sort the first, middle, and last entries, and use the middle one for the pivot, here 5. This is called **median-of-three** pivot selection



quick sort - pivot

- Since we used median-of-three pivot selection, we know that the first item belongs in S1 and last item belongs in S2.

- Step b below shows the array right before partitioning (we've moved our pivot out of the way)



quick sort

```
quickSort(theArray: ItemArray, first: integer, last: integer)
{
  if (last-first + 1 < MIN_SIZE)
    insertionSort(theArray, first, last);
  else
  {
    // create partion: S1 | Pivot | S2
    int pivotIndex = partition(theArray, first, last);

    // Sort subarrays S1 and S2
    quickSort(theArray, first, pivotIndex - 1);
    quickSort(theArray, pivotIndex + 1, last);
  }
}
```

quick sort analysis

- The heavy lifting is the partitioning step. It requires at most n comparisons and is thus an O(n) step.
- In the best case, every recursive call partitions the array into equally sized S1 and S2, and the quick sort is thus similar to merge sort in that it is halving the array each level. Thus there are $\log_2 n$ recursive levels and the total complexity is O(n log n).
- In a worst case, we could have an empty S1 or empty S2 after partitioning each time! The nonempty subarray decreases in size by only 1, so we could potentially have n levels. So the total complexity becomes $O(n^2)$.
- In an average case, with generally randomly placed numbers, a formal analysis can show O(n log n) performance.

quick sort vs merge sort

criteria	merge sort	quick sort	advantage
worst case efficiency	O(n log n)	O(n^2)	merge sort
best / average case efficiency	O(n log n)	O(n log n)	none
needs extra storage	yes	no	quick sort

- Note: with a decent pivot selection algorithm like median-of-three, worst case performance for quick sort is rare