CHAPTER 7

Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

SORTING

- Assume entire array is in main memory, this kind of sorting is called internal sorting.

- Internal Sorting,
  - There are several easy algorithm to sort in $O(N^2)$, such as insertion sort
  - Complicated O(NlogN) sorting algorithms
  - Any general-purpose sorting algorithm requires Ω(NlogN) comparisons.

INSERTION SORT

- Insertion sort consist of N-1 passes, for pass p=1 through N-1, insertion sort ensures that the elements in positions 0 through p are in sorted order.
- it makes use of the fact that elements in position 0 through p -1 are already sorted.
- In pass p we move the element in position p left until its correct lace is found among the first p + 1 elements.

INSERTION SORT

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Figure 7.1 Insertion sort after each pass

INSERTION SORT

```
1       /**
2        * Simple insertion sort.
3        * @param a an array of Comparable items.
4        */
5       public static <AnyType extends Comparable<? super AnyType>>
6       void insertionSort( AnyType [ ] a )
7       {
8           int j;
9
10          for( int p = 1; p < a.length; p++ )
11          {
12              AnyType tmp = a[ p ];
13              for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
14                  a[ j ] = a[ j - 1 ];
15              a[ j ] = tmp;
16          }
17      }
```

Figure 7.2 Insertion sort routine

ANALYSIS OF INSERTION SORT

- Each of the two loop take N iterations, so $O(N^2)$. This bound is tight.
- A precise calculation shows that the number of tests in the inner loop is at most p+1 times, summing over all p

- $\sum_{i=0}^{N} i = 2 + 3 + 4 + \ \ldots + N = \ \theta(N^2)$

- On the other hand if the input is presorted the running time is O(N).

A LOWER BOUND FOR SIMPLE SORTING ALGORITHM

- An inversion in an array of numbers is any ordered pair(i,j) having the property i < j but a[i] > a[j].
- For example in list : 34,8,64,51,32,21 has nine inversions
- Swapping two adjacent elements that are out of place removes exactly one inversion, and a sorted array has no inversion.
- The running time of insertion sort is O(I + N) where I is the number of inversions.

Cont.

Theorem: The average number of inversions in an array of N distinct elements is N(N-1)/4.

Proof: For any list L of elements consider L_r, the list in reverse order. Consider any pair of two elements (x,y) . in one of the list the pair is an inversion. The total number of pairs in a list L and L_r is N(N-1) /2. So the average list has half the amount of inversions N(N-1)/ 4

Theorem: Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average.

Proof: Each swap removes one inversion we need $\Omega(N^2)$ swaps.

Shellsort

- It works by comparing elements by that are distant.
- Shellsort uses a sequence, h_1, h_2,……, h_t, called the increment sequence.
- After a phase, using some increment h_k, for every i, a[i] <= a[i + h_k]. The array is said to be h_k–sorted.

Shellsort

Example

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Figure 7.3 Shellsort after each pass, using {1, 3, 5} as the increment sequence

```
1       /**
2        * Shellsort, using Shell's (poor) increments.
3        * @param a an array of Comparable items.
4        */
5       public static <AnyType extends Comparable<? super AnyType>>
6       void shellsort( AnyType [ ] a )
7       {
8           int j;
9
10          for( int gap = a.length / 2; gap > 0; gap /= 2 )
11              for( int i = gap; i < a.length; i++ )
12              {
13                  AnyType tmp = a[ i ];
14                  for( j = i; j >= gap &&
15                              tmp.compareTo( a[ j - gap ] ) < 0; j -= gap )
16                      a[ j ] = a[ j - gap ];
17                  a[ j ] = tmp;
18              }
19      }
```

Figure 7.4 Shellsort routine using Shell's increments (better increments are possible)

Example of bad case

Start	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
After 1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 7.5 Bad case for Shellsort with Shell's increments (positions are numbered 1 to 16)

HEAPSORT

- Priority queues can be used to sort in O(NlogN) time. The algorithm based on this idea is known as heapsort.
- Algorithm:
  - buildHeap O(N)
  - perform N deleteMin operation O(NlogN)

If second array is used then memory requirement is doubled, instead use the same array.

- Use (max) heap and store the deleteMax element back into the array.
- For example if the array is 31, 41, 59,26, 53, 58, 97. The heap is

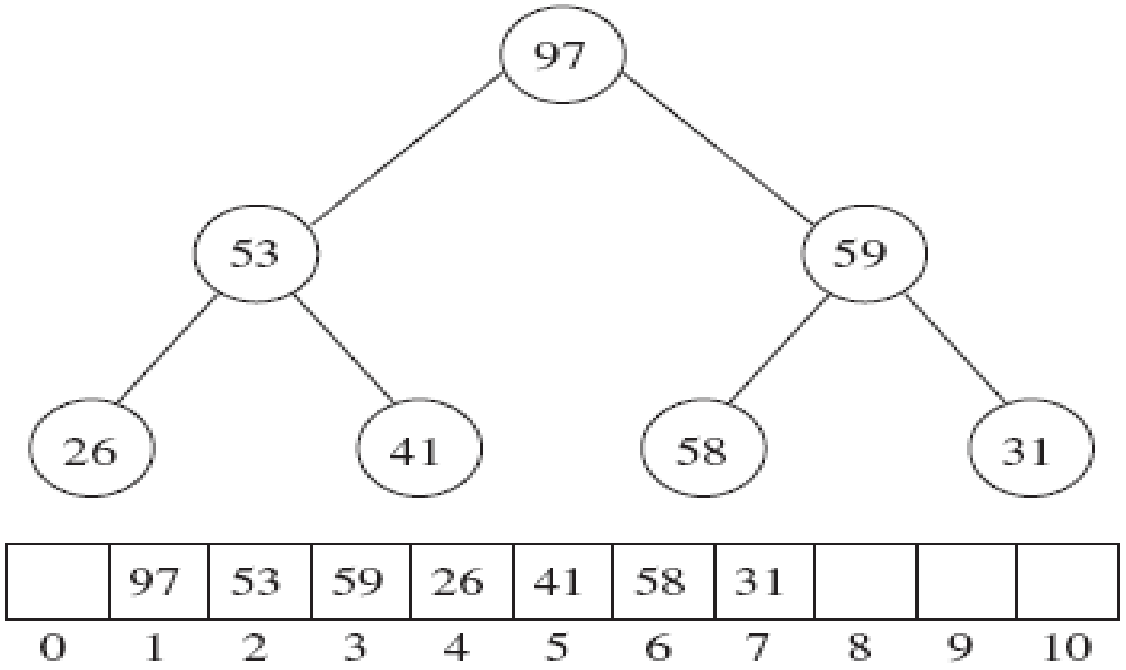


Figure 7.6 (*Max*) heap after `buildHeap` phase

After deleteMax, 97 has been placed in a part of the heap array that is technically no longer part of heap.

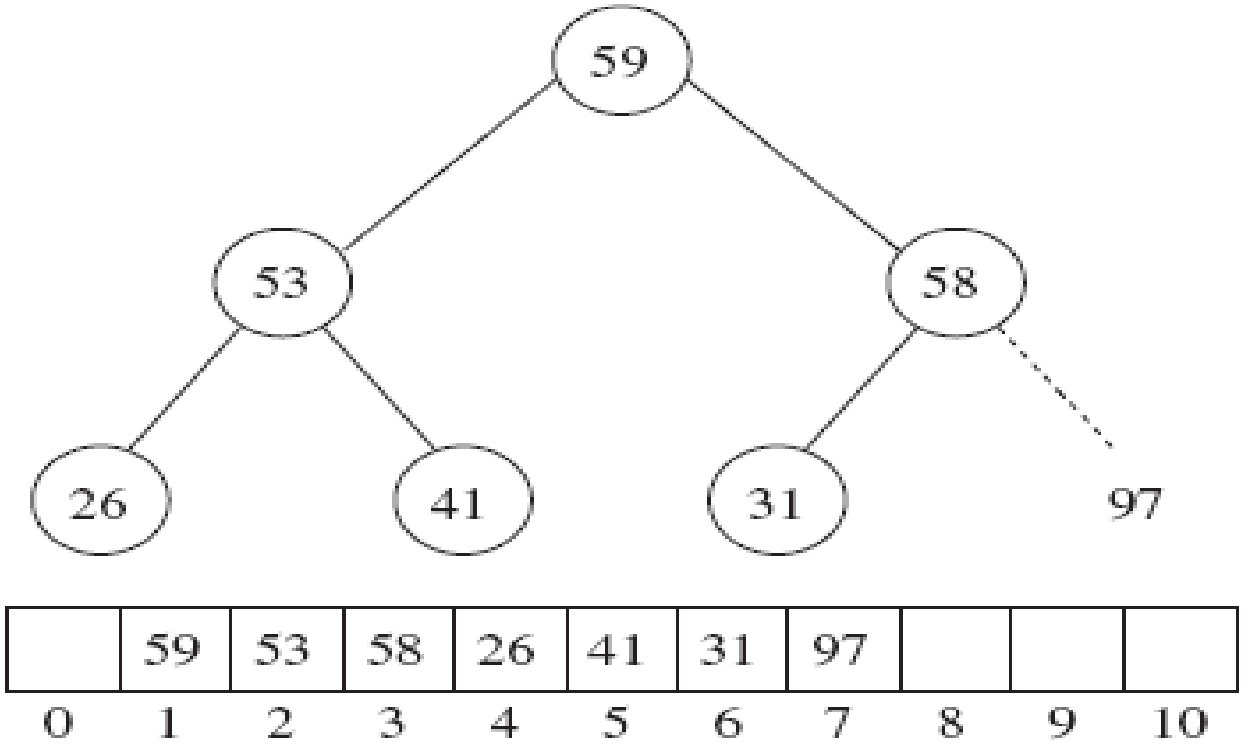


Figure 7.7 Heap after first `deleteMax`

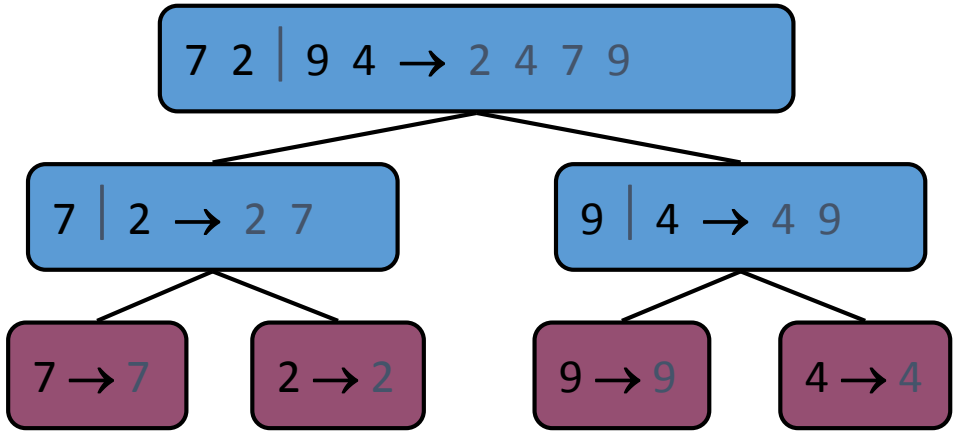
```
1       /**
2        * Internal method for heapsort.
3        * @param i the index of an item in the heap.
4        * @return the index of the left child.
5        */
6       private static int leftChild( int i )
7       {
8           return 2 * i + 1;
9       }
10
11      /**
12       * Internal method for heapsort that is used in deleteMax and buildHeap.
13       * @param a an array of Comparable items.
14       * @int i the position from which to percolate down.
15       * @int n the logical size of the binary heap.
16       */
17      private static <AnyType extends Comparable<? super AnyType>>
18      void percDown( AnyType [ ] a, int i, int n )
19      {
20          int child;
21          AnyType tmp;
22
23          for( tmp = a[ i ]; leftChild( i ) < n; i = child )
24          {
25              child = leftChild( i );
26              if( child != n - 1 && a[ child ].compareTo( a[ child + 1 ] ) < 0 )
27                  child++;
28              if( tmp.compareTo( a[ child ] ) < 0 )
29                  a[ i ] = a[ child ];
30              else
31                  break;
32          }
33          a[ i ] = tmp;
34      }
35
36      /**
37       * Standard heapsort.
38       * @param a an array of Comparable items.
39       */
40      public static <AnyType extends Comparable<? super AnyType>>
41      void heapsort( AnyType [ ] a )
42      {
43          for( int i = a.length / 2 - 1; i >= 0; i-- )  /* buildHeap */
44              percDown( a, i, a.length );
45          for( int i = a.length - 1; i > 0; i-- )
46          {
47              swapReferences( a, 0, i );                /* deleteMax */
48              percDown( a, 0, i );
49          }
50      }
```

Figure 7.8 Heapsort

ANALYSIS OF HEAPSORT

- Experiments have shown the performance of heapsort is extremely consistent: On average it uses only slightly fewer comparisons that the worst-case bound suggests.

Merge Sort



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S_1 and S_2
  - Recur: solve the subproblems associated with S_1 and S_2
  - Conquer: combine the solutions for S_1 and S_2 into a solution for S
- The base case for the recursion are subproblems of size 0 or 1

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1)$
    $mergeSort(S_2)$
    $S \leftarrow merge(S_1, S_2)$

Merging Two Sorted Sequences

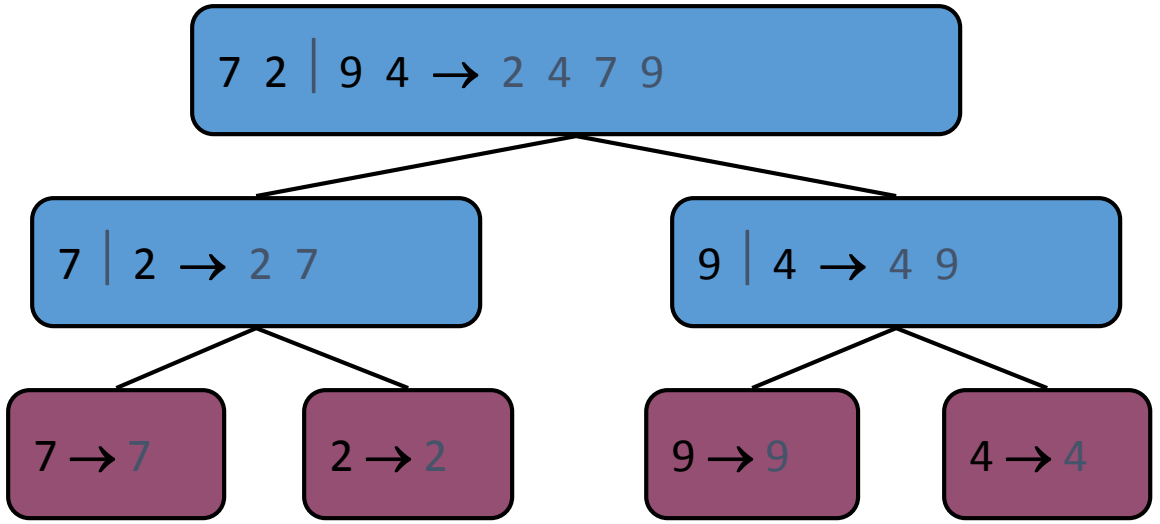
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

```
Algorithm merge(A, B)
    Input sequences A and B with
        n/2 elements each
    Output sorted sequence of A ∪ B

    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        if A.first().element() < B.first().element()
            S.addLast(A.remove(A.first()))
        else
            S.addLast(B.remove(B.first()))
    while ¬A.isEmpty()
        S.addLast(A.remove(A.first()))
    while ¬B.isEmpty()
        S.addLast(B.remove(B.first()))
    return S
```

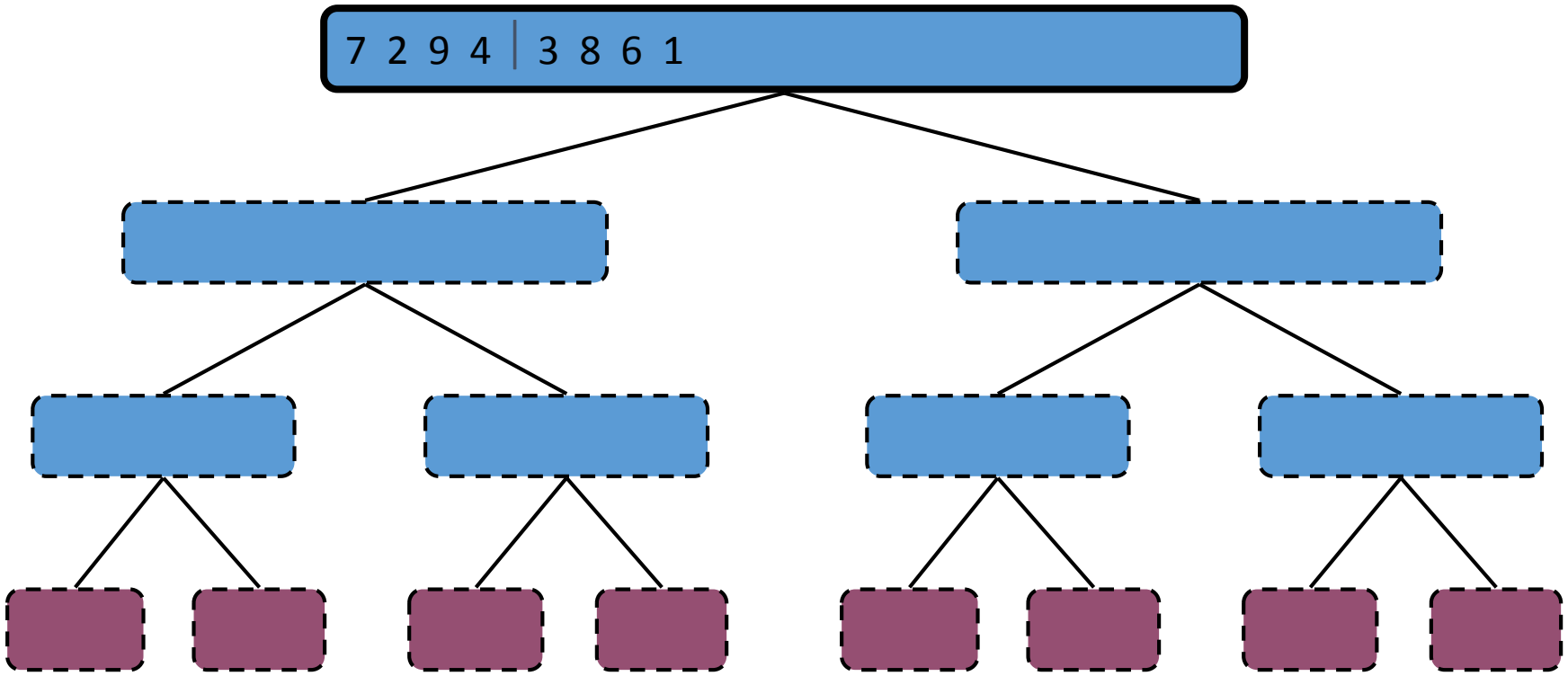
Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



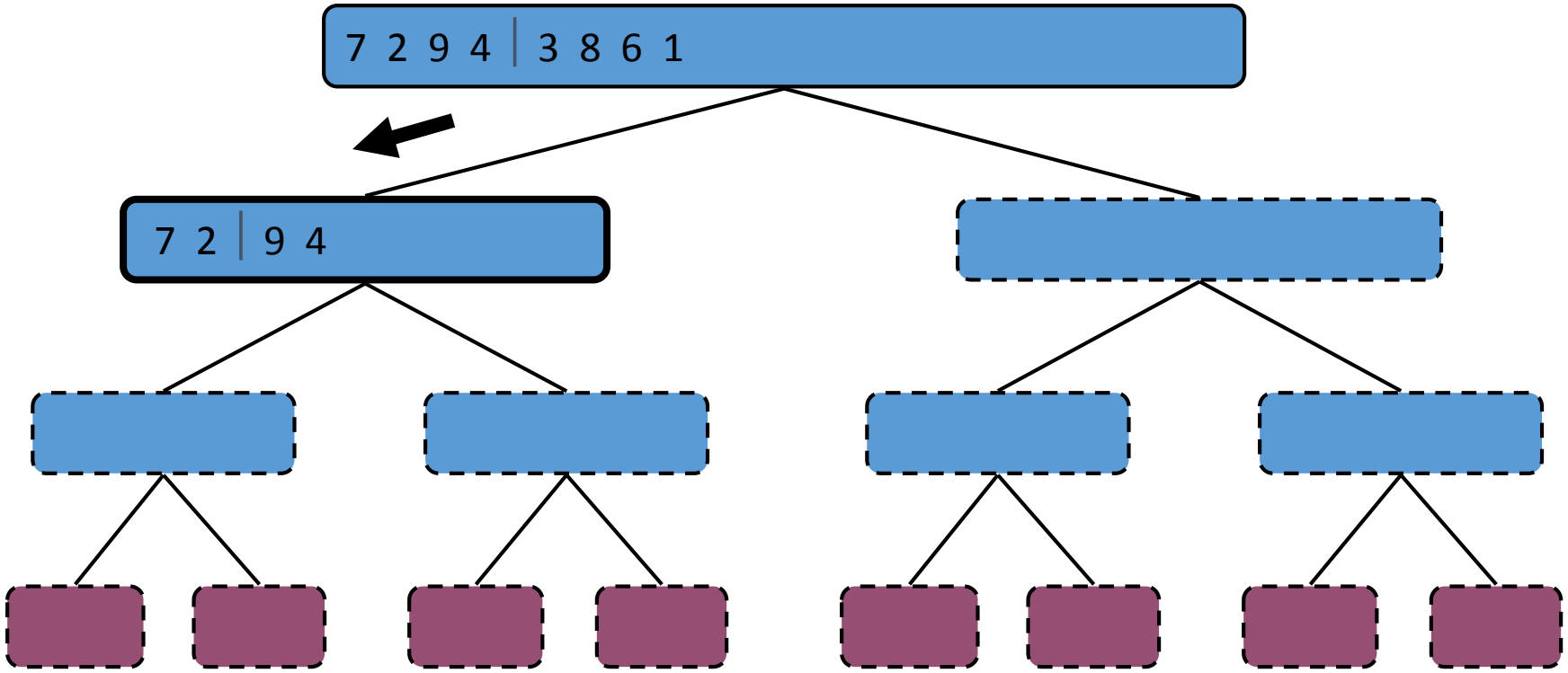
Execution Example

- Partition



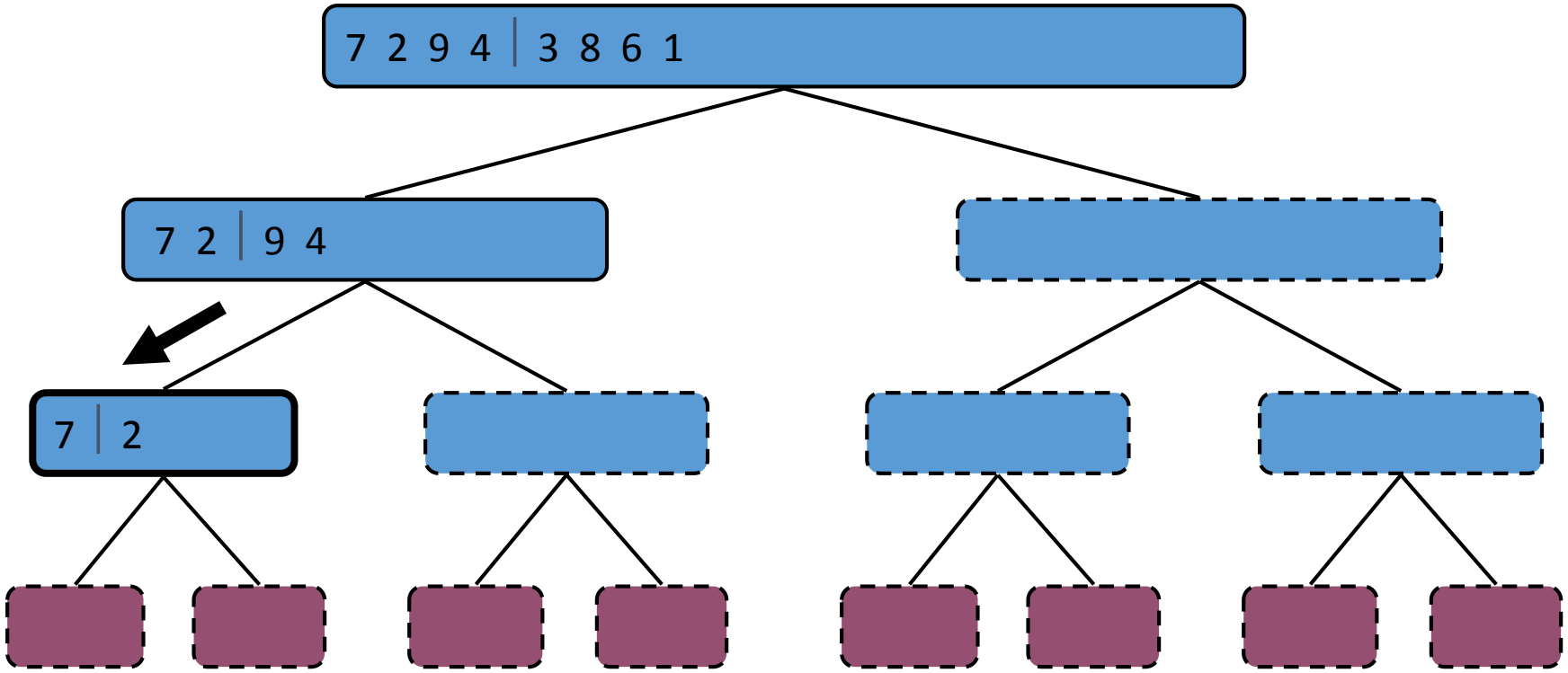
Execution Example (cont.)

- Recursive call, partition



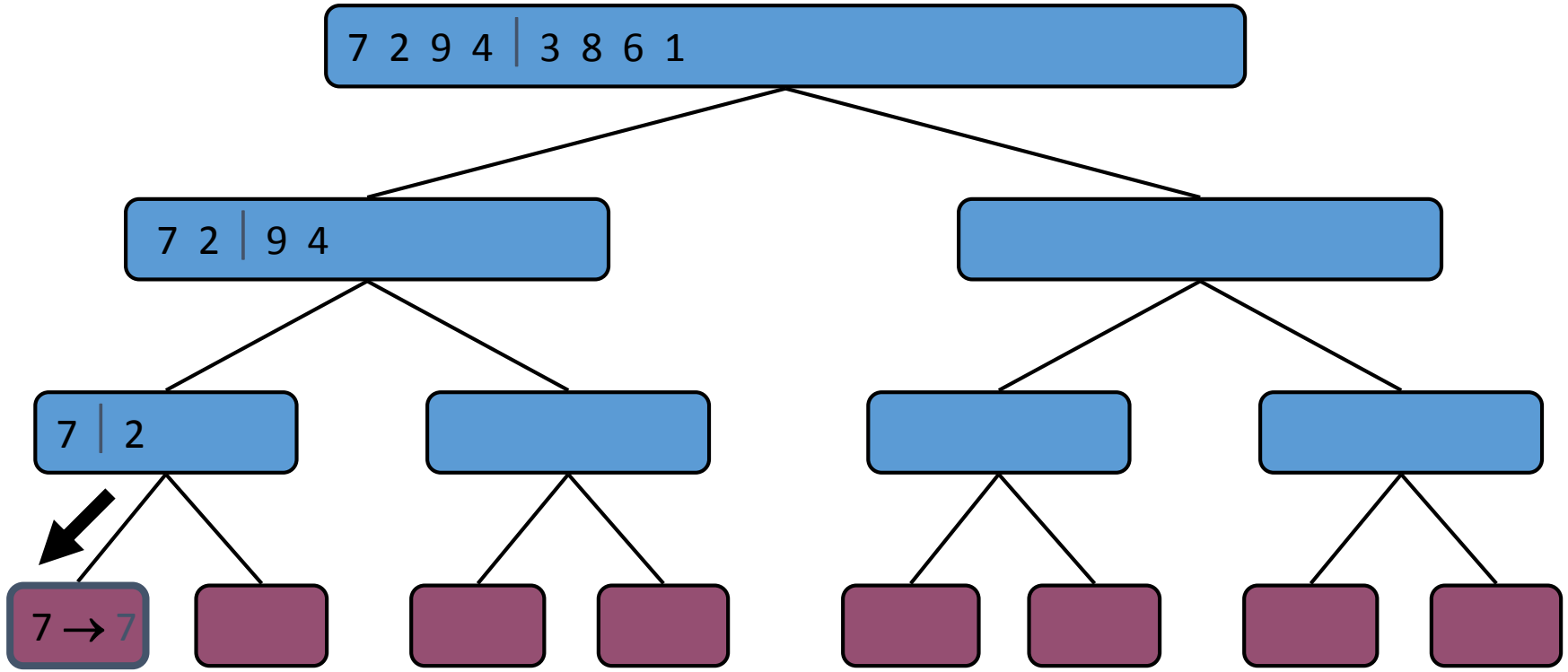
Execution Example (cont.)

- Recursive call, partition



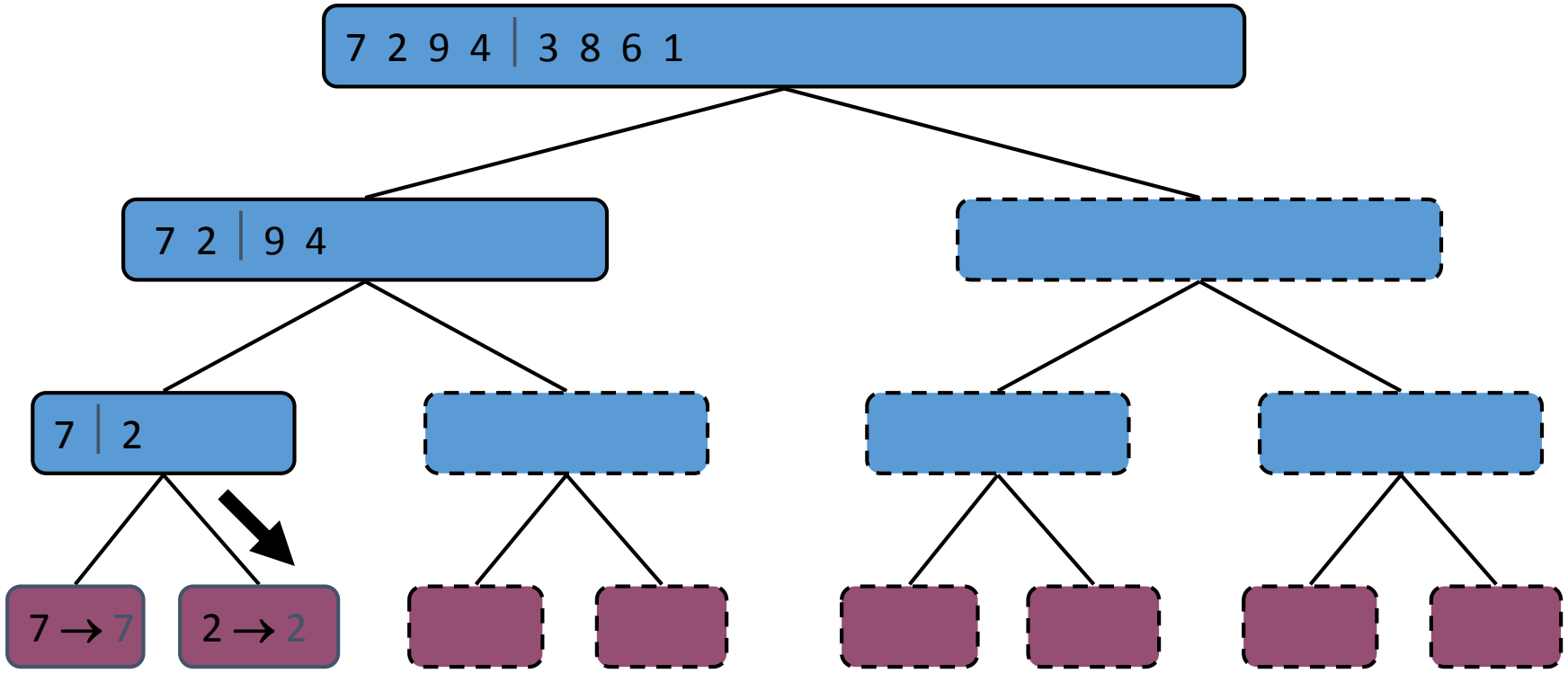
Execution Example (cont.)

- Recursive call, base case



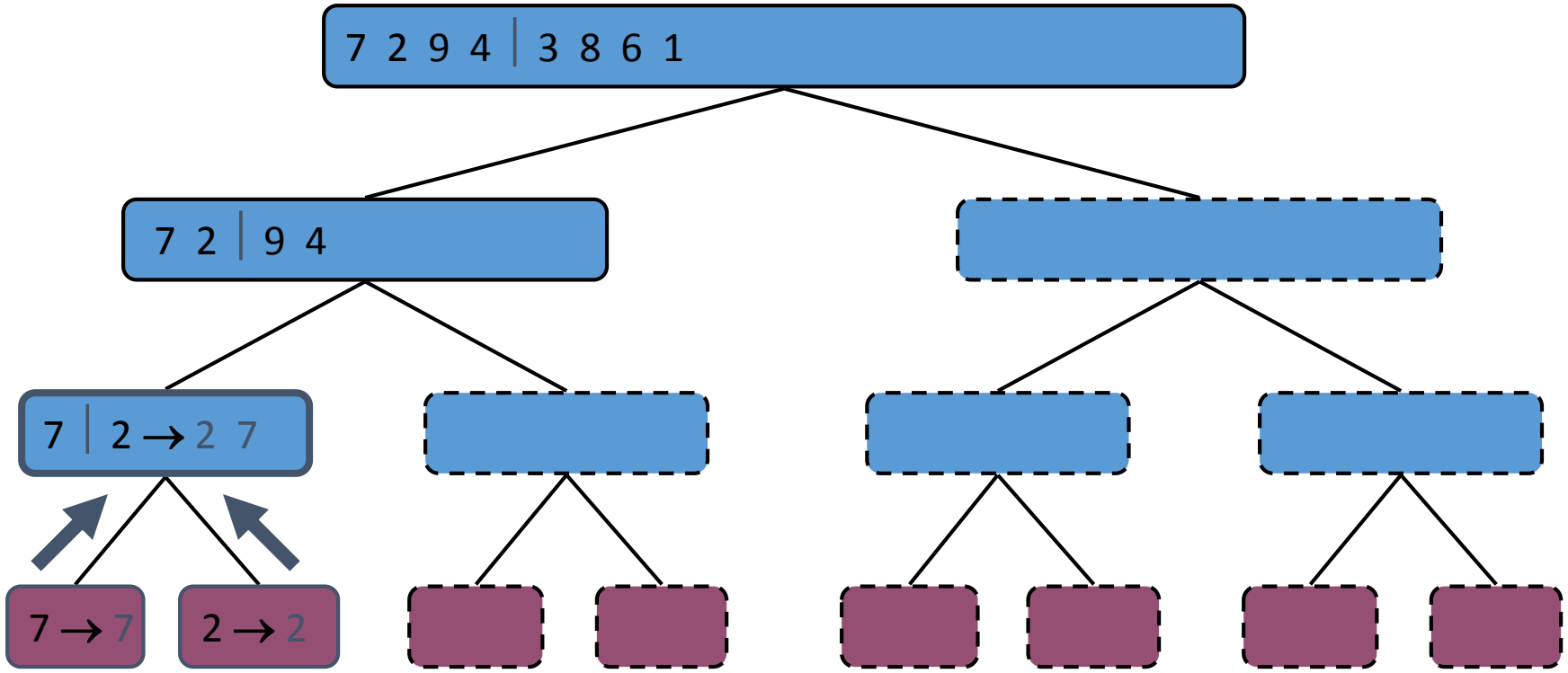
Execution Example (cont.)

- Recursive call, base case



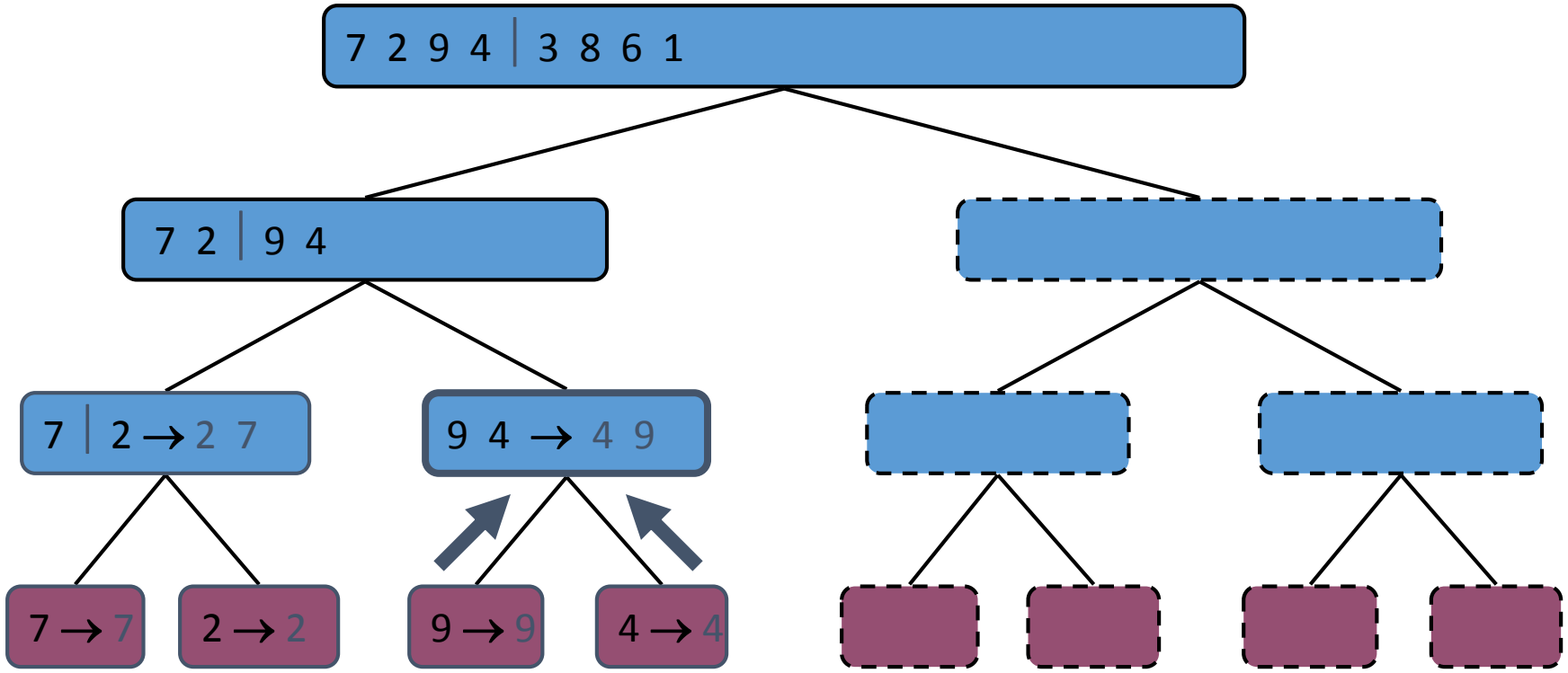
Execution Example (cont.)

- Merge



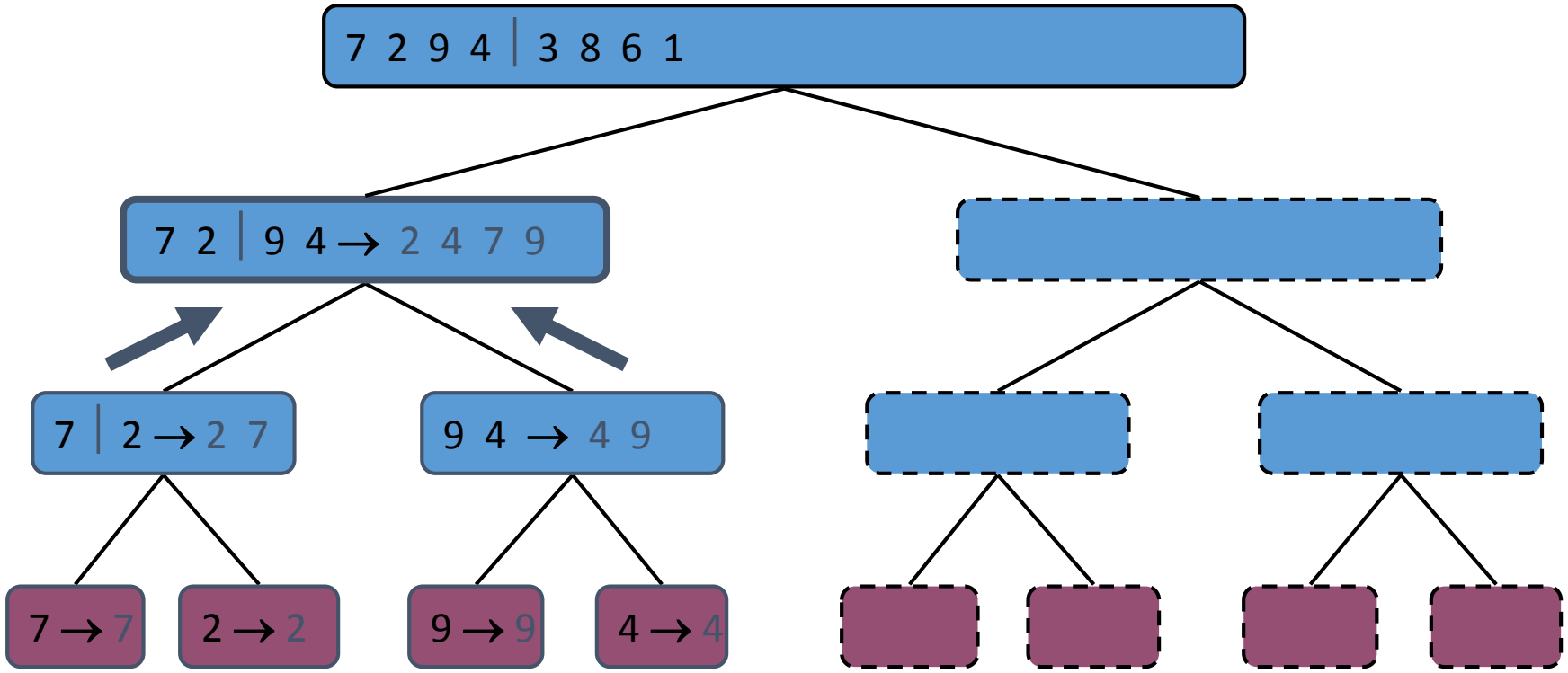
Execution Example (cont.)

- Recursive call, …, base case, merge



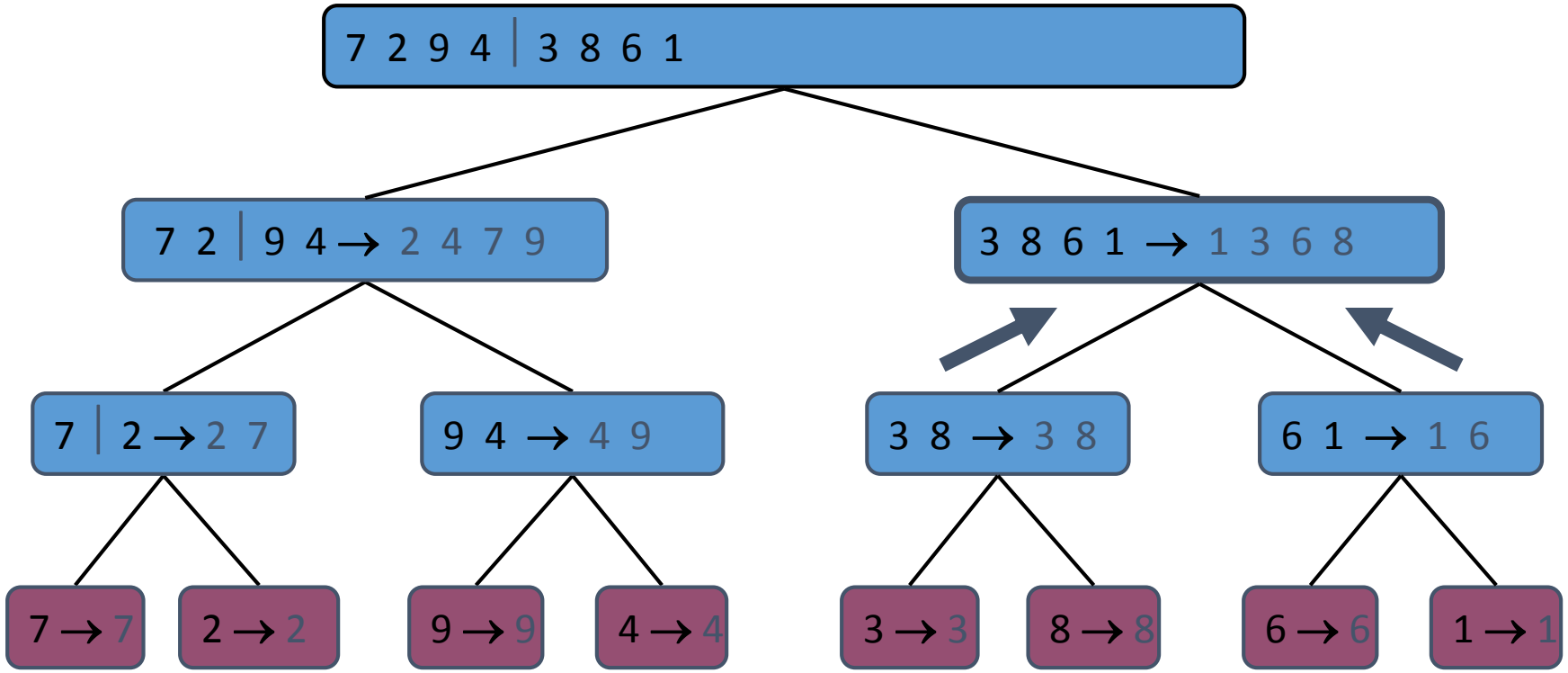
Execution Example (cont.)

- Merge



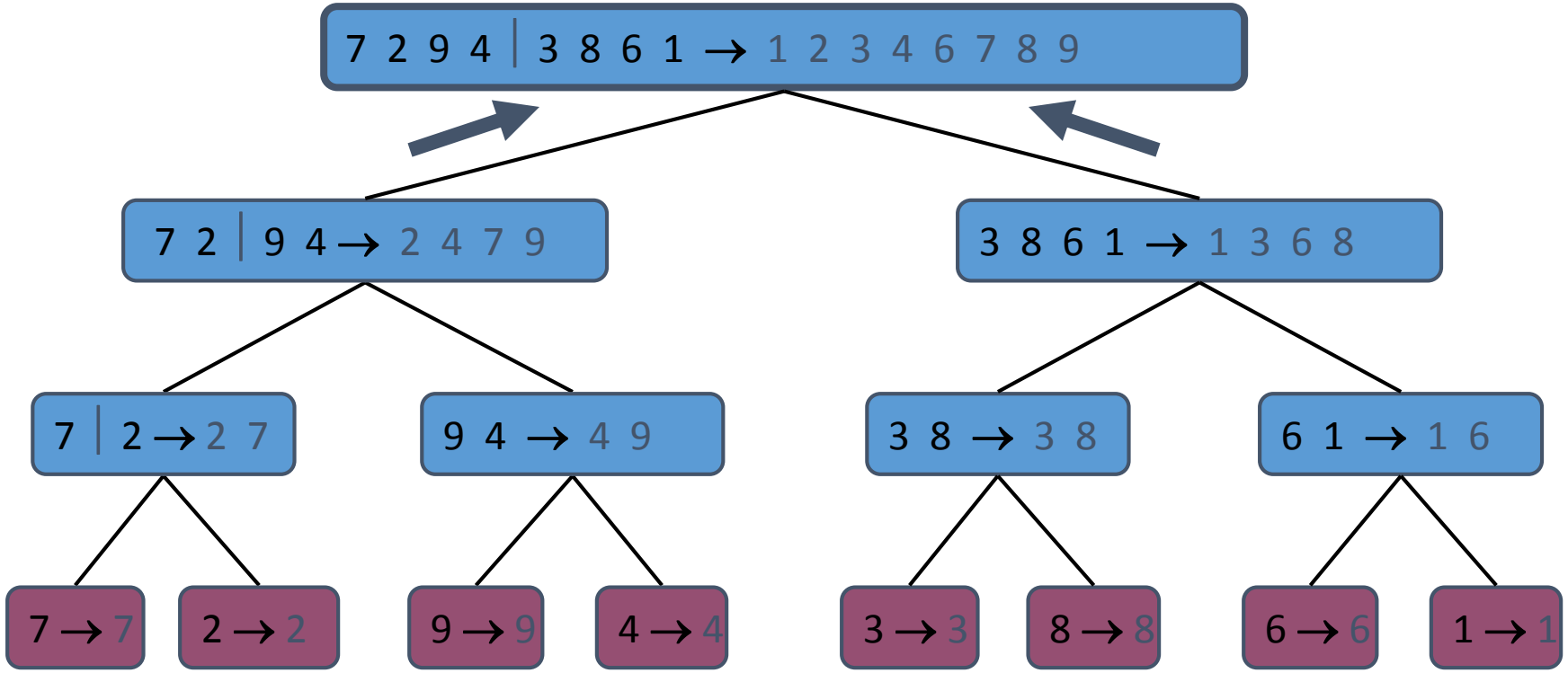
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



ANALYSIS OF MERGESORT

- Assume N is a power of 2
- For N =1 time for merge sort is constant

  T(1) = 1

  T(N) = 2T(N/2) + N
- This is a standard recurrent relation

$$\frac{T(N)}{N} = \frac{T(\frac{N}{2})}{N/2} + 1$$

$$\frac{T(\frac{N}{2})}{N/2} = \frac{T(\frac{N}{4})}{N/4} + 1$$

$$\frac{T(\frac{N}{4})}{N/4} = \frac{T(\frac{N}{8})}{N/8} + 1 \quad \ldots. \frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

- Now add all the equations

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

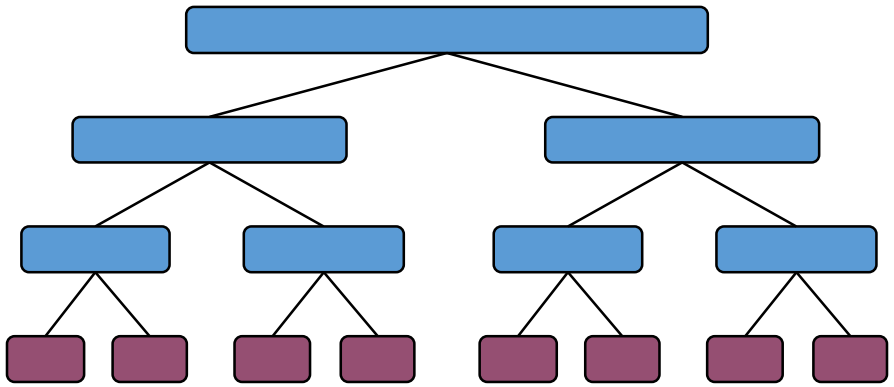
- Multiple through by N and we get

T(N) = N logN + N = O(NlogN)

Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount or work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



```
1       /**
2        * Internal method that makes recursive calls.
3        * @param a an array of Comparable items.
4        * @param tmpArray an array to place the merged result.
5        * @param left the left-most index of the subarray.
6        * @param right the right-most index of the subarray.
7        */
8       private static <AnyType extends Comparable<? super AnyType>>
9       void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray, int left, int right )
10      {
11          if( left < right )
12          {
13              int center = ( left + right ) / 2;
14              mergeSort( a, tmpArray, left, center );
15              mergeSort( a, tmpArray, center + 1, right );
16              merge( a, tmpArray, left, center + 1, right );
17          }
18      }
19
20      /**
21       * Mergesort algorithm.
22       * @param a an array of Comparable items.
23       */
24      public static <AnyType extends Comparable<? super AnyType>>
25      void mergeSort( AnyType [ ] a )
26      {
27          AnyType [ ] tmpArray = (AnyType[]) new Comparable[ a.length ];
28
29          mergeSort( a, tmpArray, 0, a.length - 1 );
30      }
```

Figure 7.9 Mergesort routines

```
1       /**
2        * Internal method that merges two sorted halves of a subarray.
3        * @param a an array of Comparable items.
4        * @param tmpArray an array to place the merged result.
5        * @param leftPos the left-most index of the subarray.
6        * @param rightPos the index of the start of the second half.
7        * @param rightEnd the right-most index of the subarray.
8        */
9       private static <AnyType extends Comparable<? super AnyType>>
10      void merge( AnyType [ ] a, AnyType [ ] tmpArray,
11                  int leftPos, int rightPos, int rightEnd )
12      {
13          int leftEnd = rightPos - 1;
14          int tmpPos = leftPos;
15          int numElements = rightEnd - leftPos + 1;
16
17          // Main loop
18          while( leftPos <= leftEnd && rightPos <= rightEnd )
19              if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
20                  tmpArray[ tmpPos++ ] = a[ leftPos++ ];
21              else
22                  tmpArray[ tmpPos++ ] = a[ rightPos++ ];
23
24          while( leftPos <= leftEnd )    // Copy rest of first half
25              tmpArray[ tmpPos++ ] = a[ leftPos++ ];
26
27          while( rightPos <= rightEnd )  // Copy rest of right half
28              tmpArray[ tmpPos++ ] = a[ rightPos++ ];
29
30          // Copy tmpArray back
31          for( int i = 0; i < numElements; i++, rightEnd-- )
32              a[ rightEnd ] = tmpArray[ rightEnd ];
33      }
```

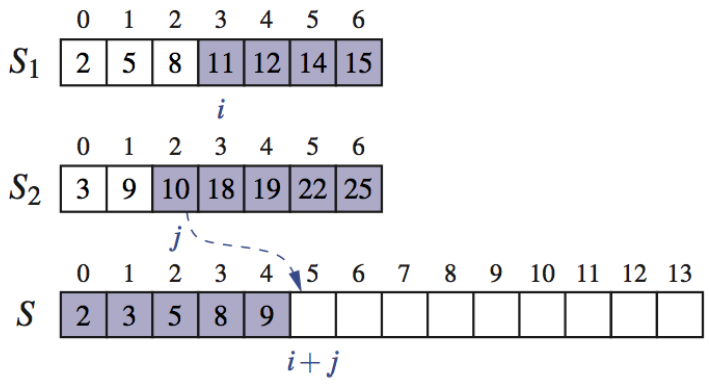
Figure 7.10 merge routine

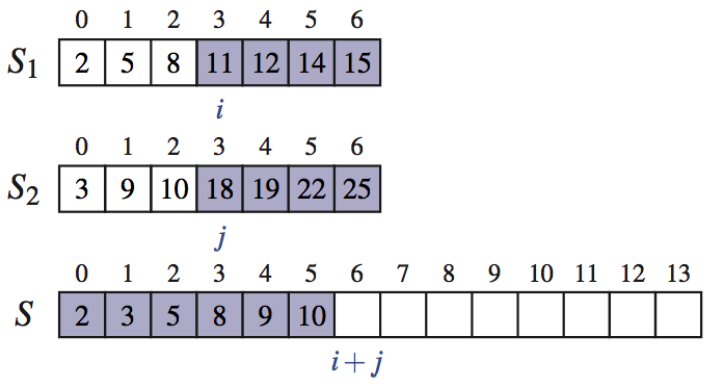
Java Merge-Sort Implementation

```
1   /** Merge-sort contents of array S. */
2   public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
3     int n = S.length;
4     if (n < 2) return;                          // array is trivially sorted
5     // divide
6     int mid = n/2;
7     K[ ] S1 = Arrays.copyOfRange(S, 0, mid);    // copy of first half
8     K[ ] S2 = Arrays.copyOfRange(S, mid, n);    // copy of second half
9     // conquer (with recursion)
10    mergeSort(S1, comp);                        // sort copy of first half
11    mergeSort(S2, comp);                        // sort copy of second half
12    // merge results
13    merge(S1, S2, S, comp);                     // merge sorted halves back into original
14  }
```

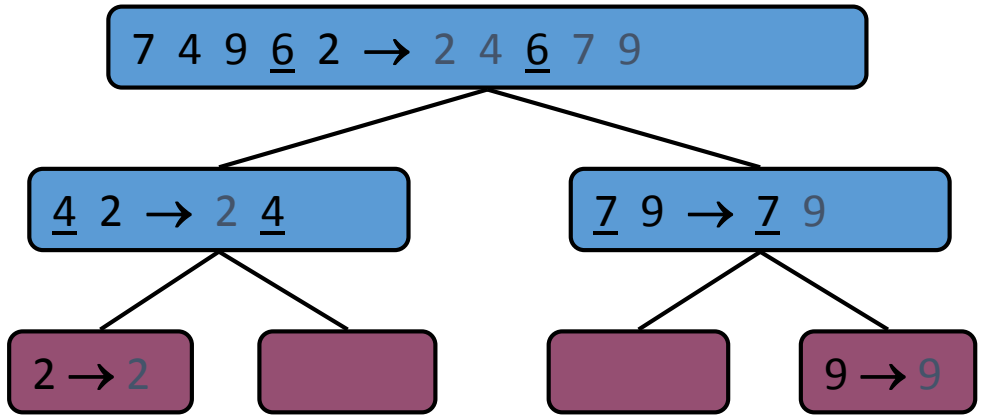
Java Merge Implementation

```
1   /** Merge contents of arrays S1 and S2 into properly sized array S. */
2   public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {
3     int i = 0, j = 0;
4     while (i + j < S.length) {
5       if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
6         S[i+j] = S1[i++];                    // copy ith element of S1 and increment i
7       else
8         S[i+j] = S2[j++];                    // copy jth element of S2 and increment j
9     }
10  }
```



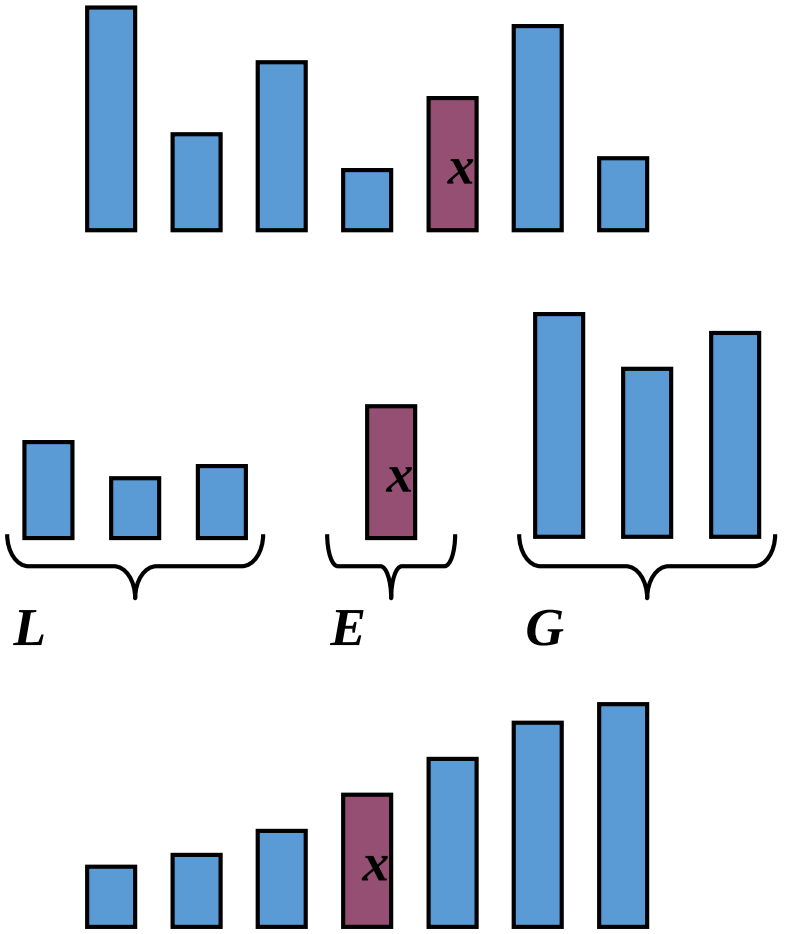


Quick-Sort

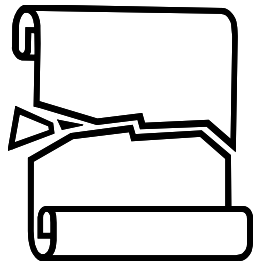


Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: join L, E and G



Partition

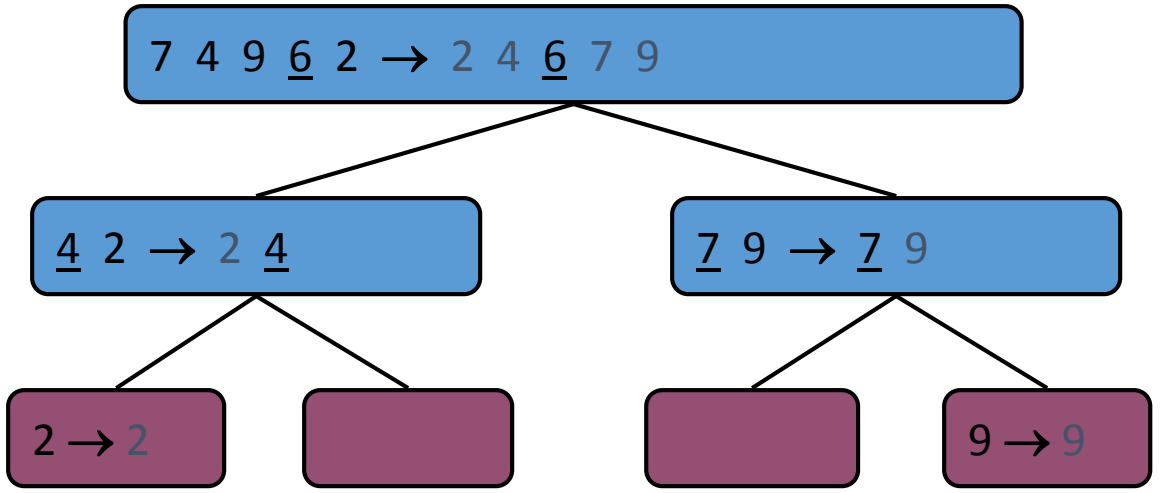


- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-sort takes $O(n)$ time

```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else { y > x }
            G.addLast(y)
    return L, E, G
```

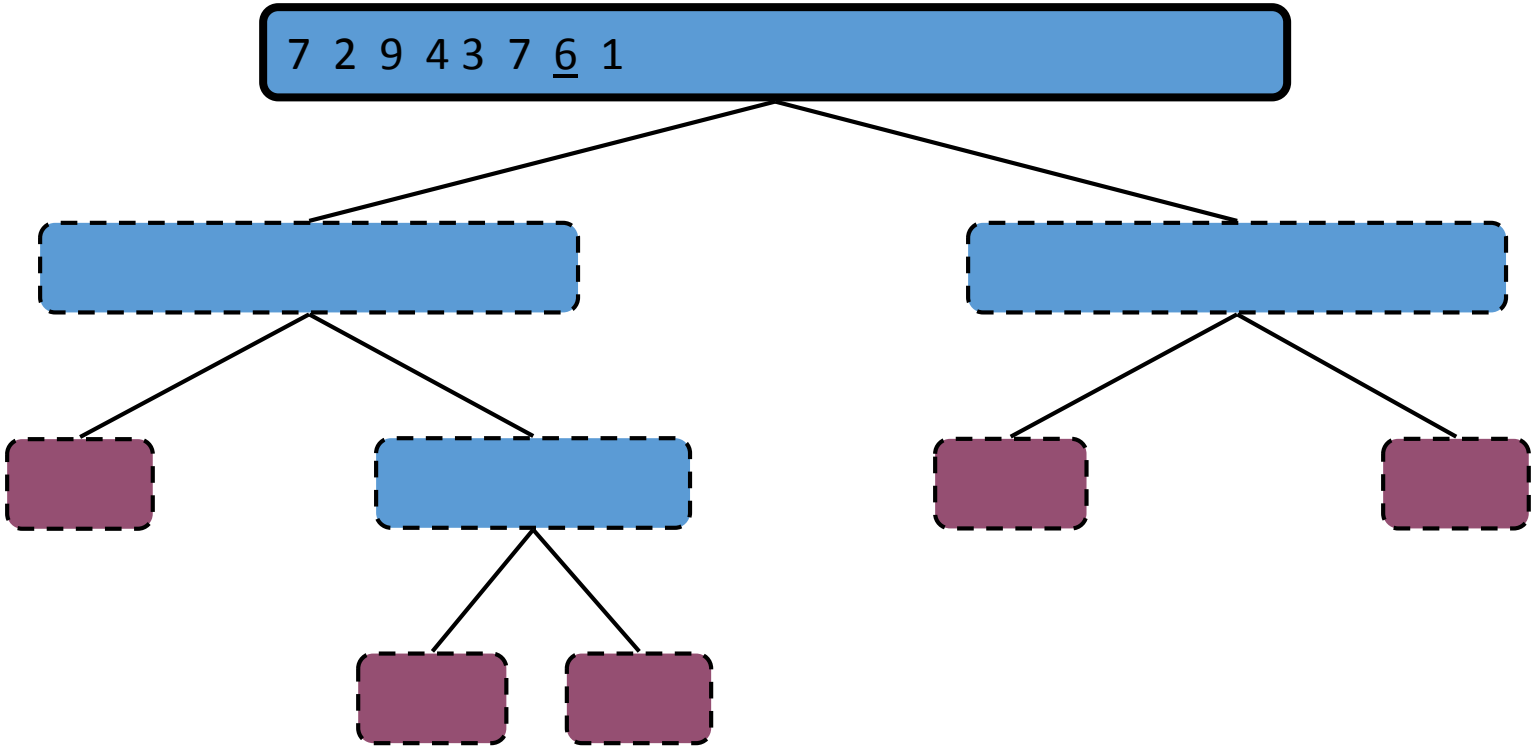
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - Sorted sequence at the end of the execution
  - The root is the initial call
  - The leaves are calls on subsequences of size 0 or 1



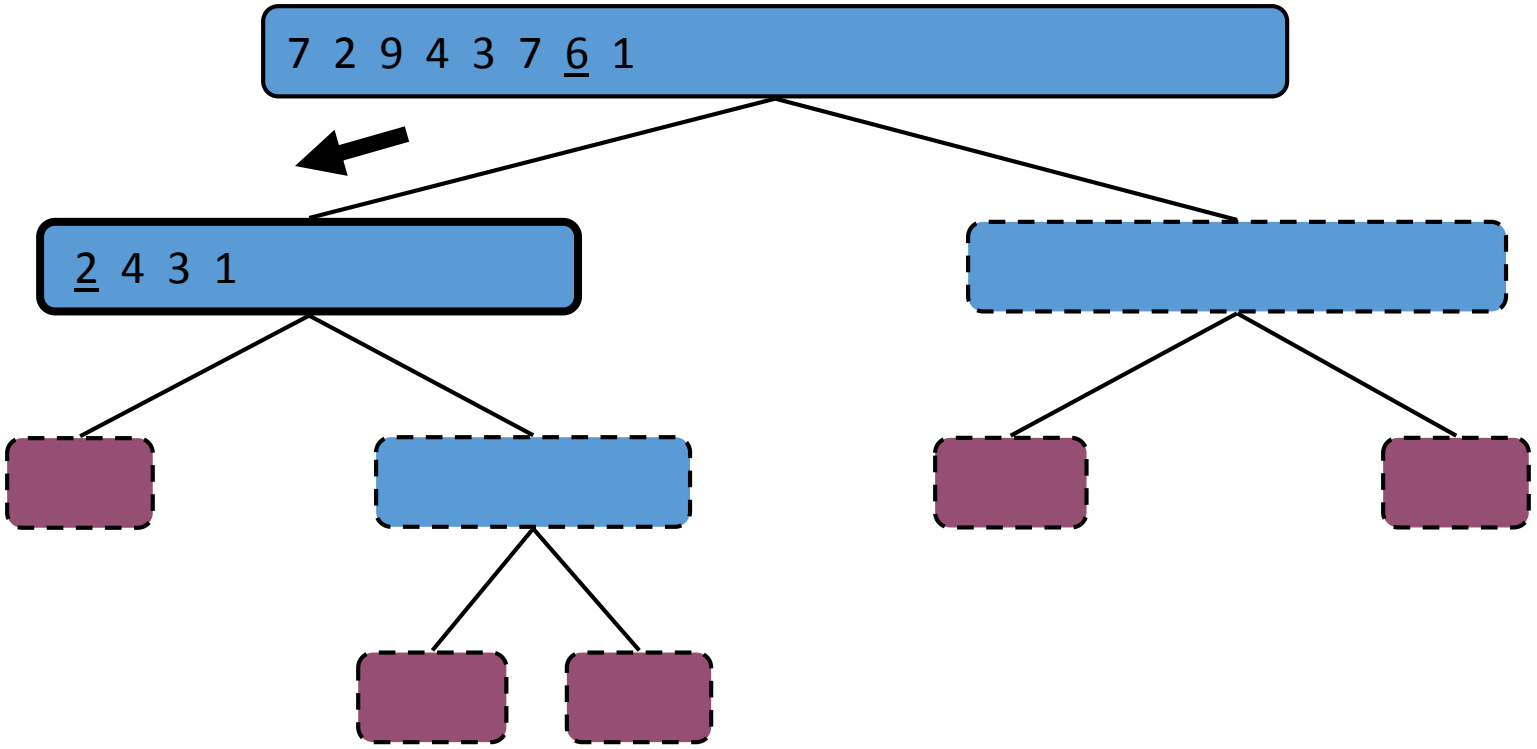
Execution Example

- Pivot selection



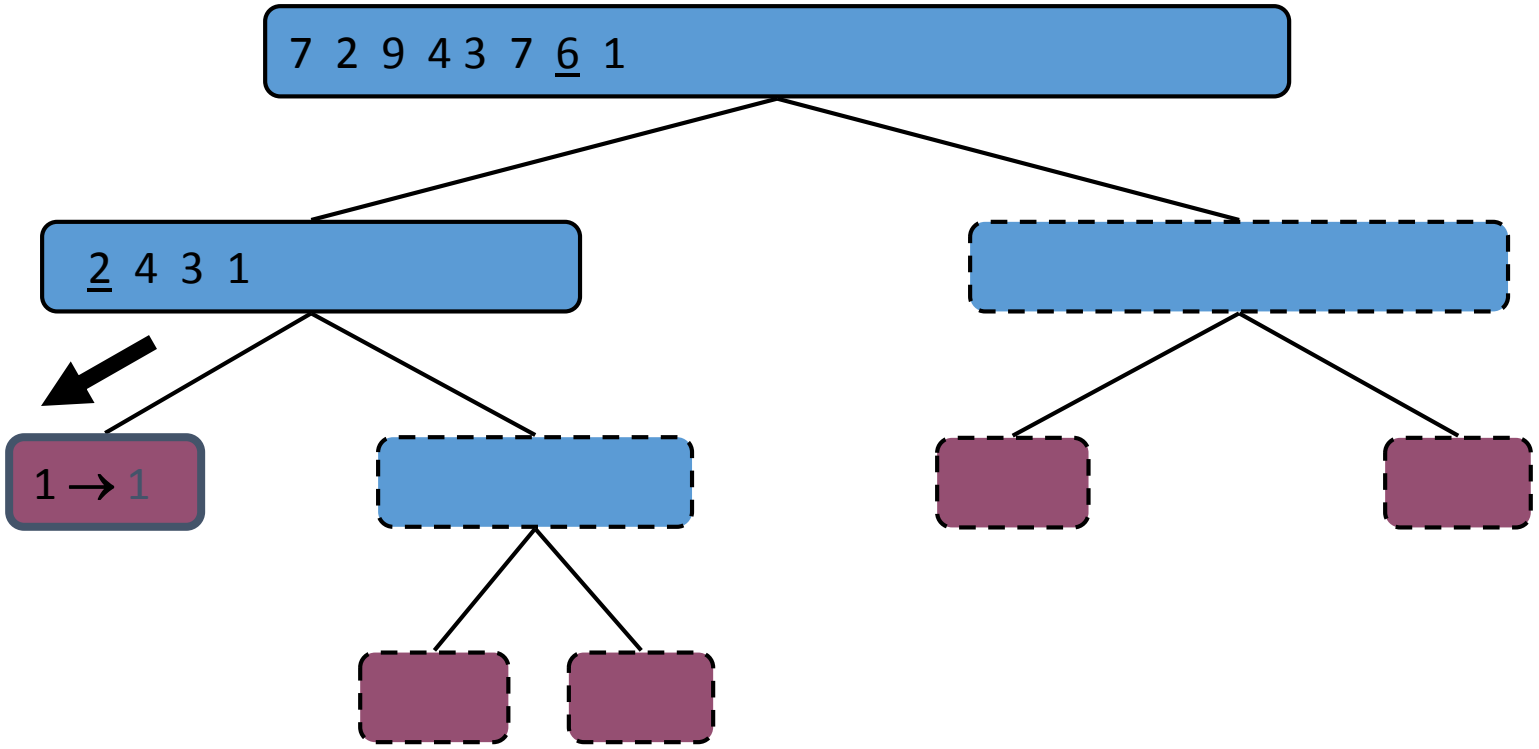
Execution Example (cont.)

- Partition, recursive call, pivot selection



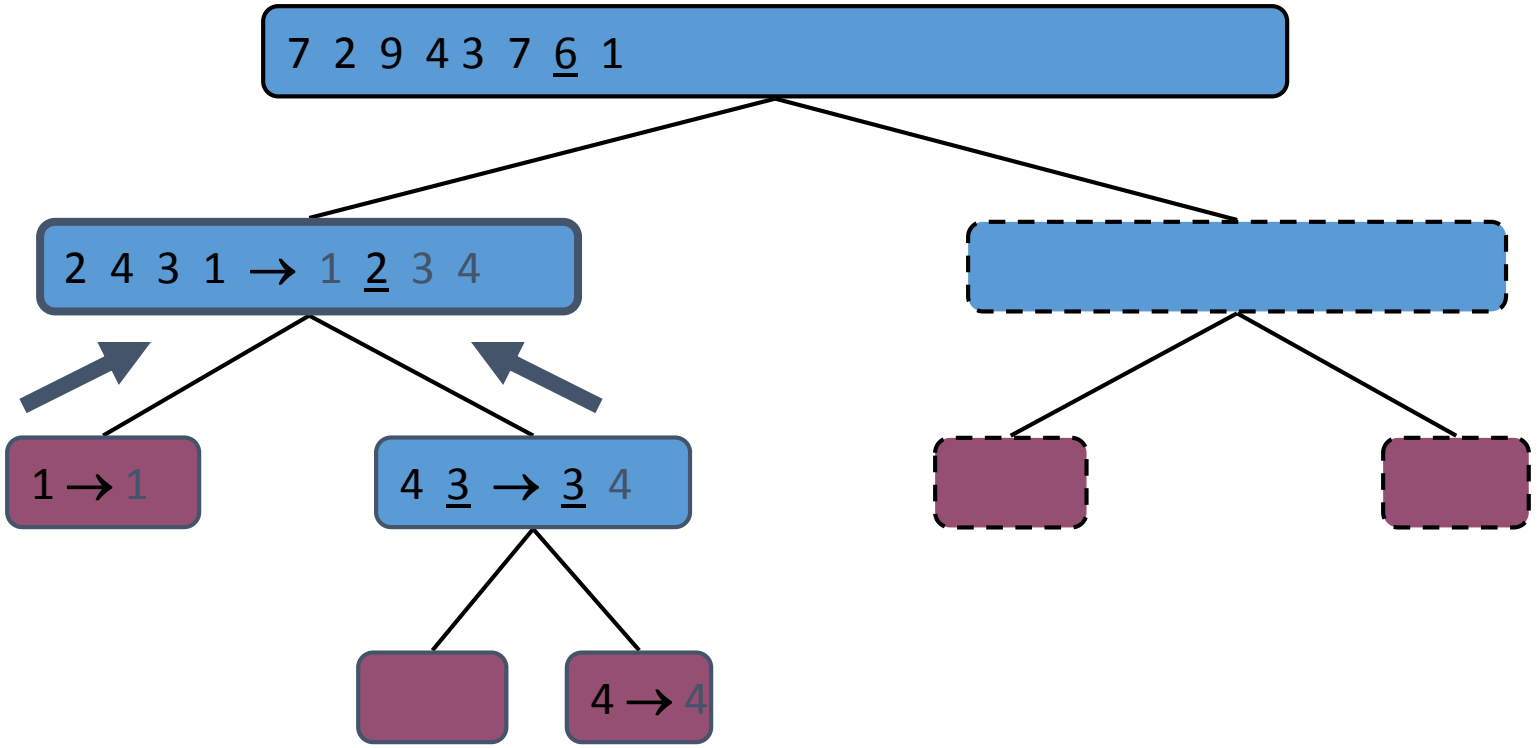
Execution Example (cont.)

- Partition, recursive call, base case



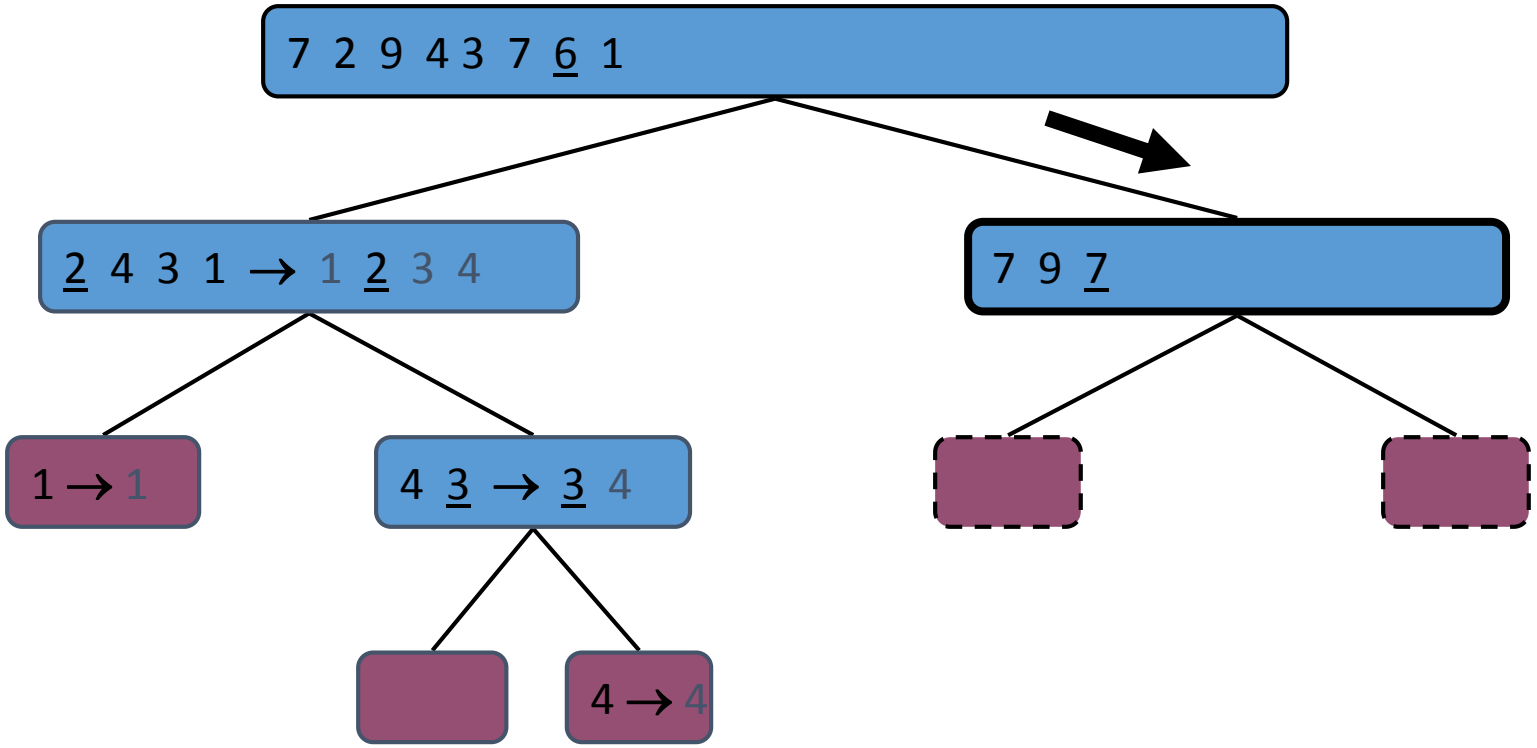
Execution Example (cont.)

- Recursive call, …, base case, join



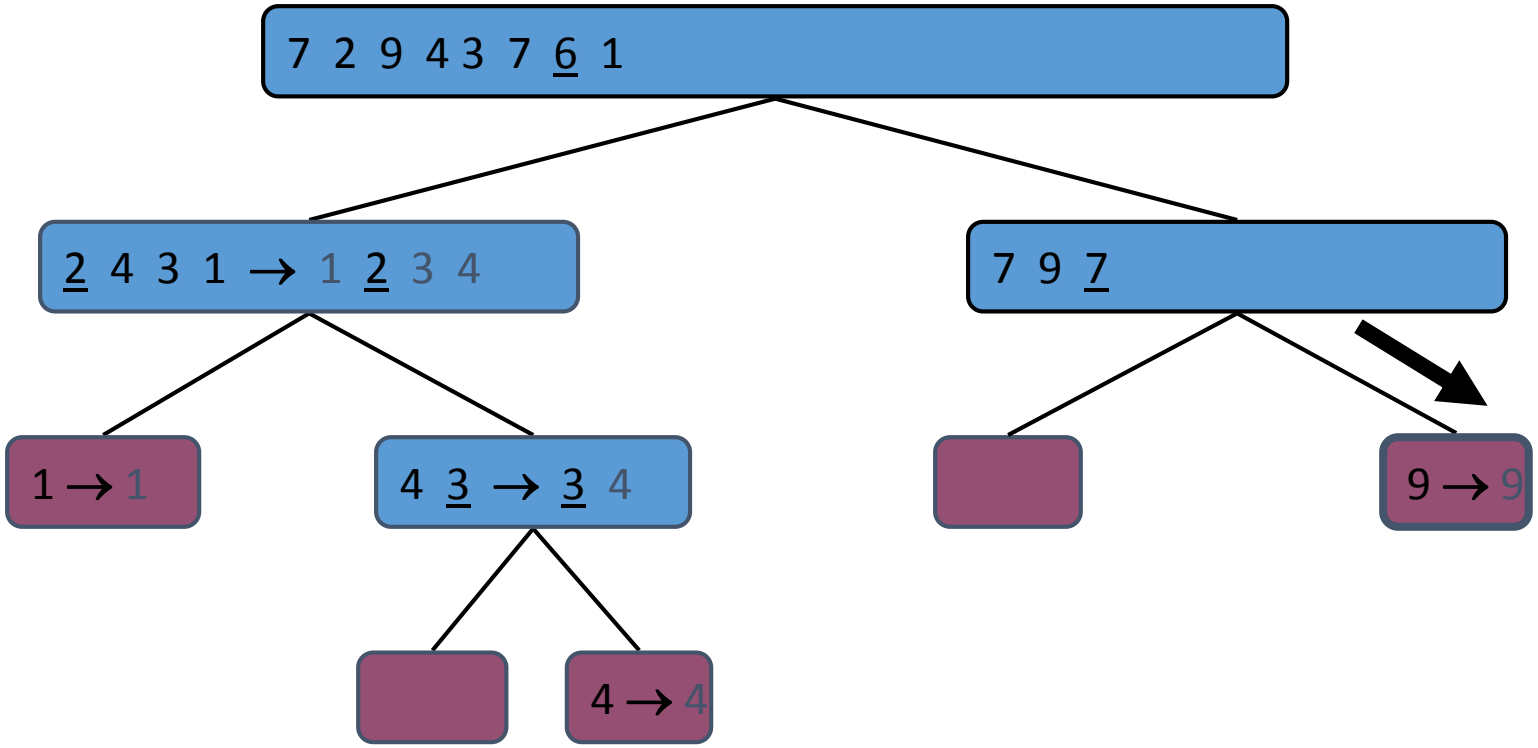
Execution Example (cont.)

- Recursive call, pivot selection



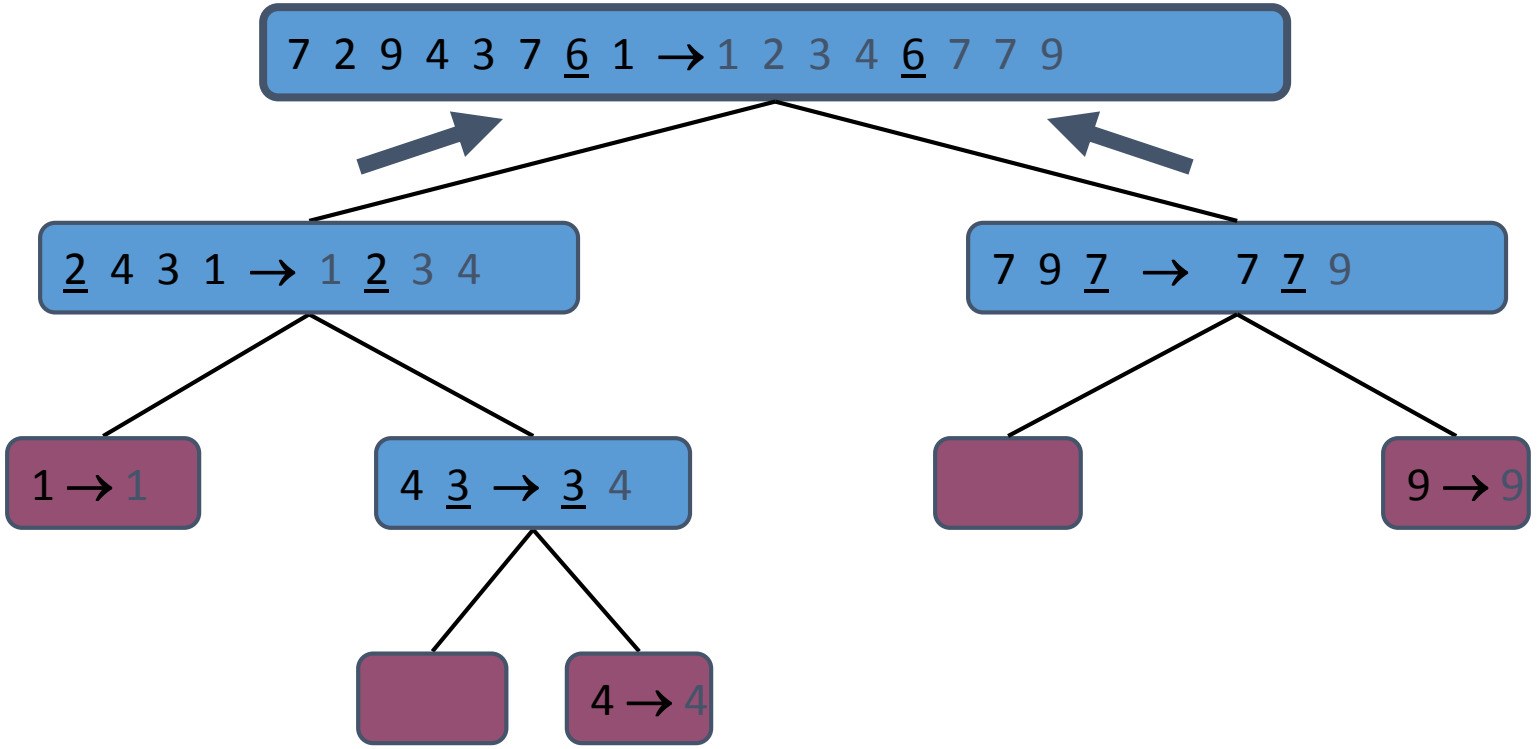
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



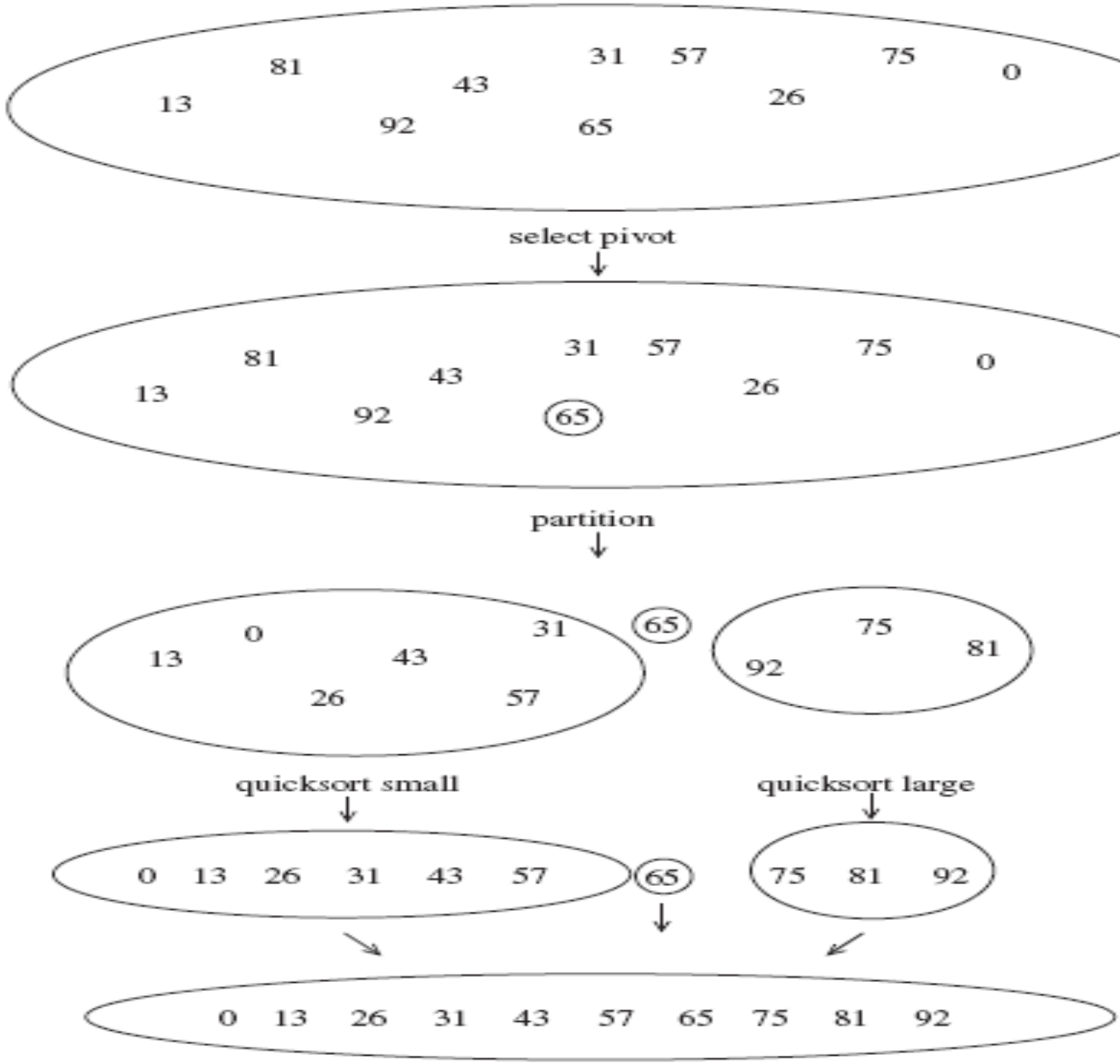


Figure 7.12 The steps of quicksort illustrated by example

```
1   public static void sort( List<Integer> items )
2   {
3       if( items.size( ) > 1 )
4       {
5           List<Integer> smaller = new ArrayList<>( );
6           List<Integer> same    = new ArrayList<>( );
7           List<Integer> larger  = new ArrayList<>( );
8
9           Integer chosenItem = items.get( items.size( ) / 2 );
10          for( Integer i : items )
11          {
12              if( i < chosenItem )
13                  smaller.add( i );
14              else if( i > chosenItem )
15                  larger.add( i );
16              else
17                  same.add( i );
18          }
19
20          sort( smaller );   // Recursive call!
21          sort( larger );    // Recursive call!
22
23          items.clear( );
24          items.addAll( smaller );
25          items.addAll( same );
26          items.addAll( larger );
27      }
28  }
```

Figure 7.11 Simple recursive sorting algorithm

CLASSIC QUICKSORT

- The classic quicksort algorithm to sort an array S consists of the following four easy steps:
  1. if the number of elements in S is 0 or 1 , then return
  2. Pick any element v in S. This is called the pivot
  3. Partition S –{v} into two disjoint groups S1 = { x ϵ S – {v} | x<= v}, and S2 = {x ϵ S – {v} | x>= v}
  4. Return quicksort(S1) followed by v followed by quicksort (S2)

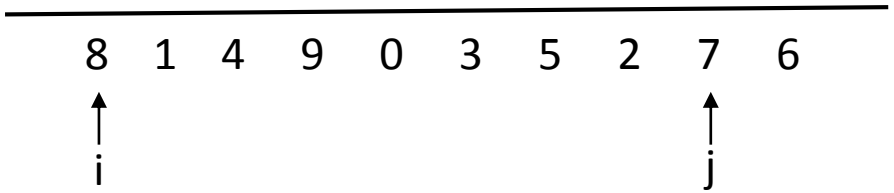
PICKING THE PIVOT

- Wrong Way : To use the first element as pivot.
- Safe choice is merely to choose the pivot randomly.
- Median of Three Partitioning : Pick three elements randomly and choose the median of these three as pivot.

- Selecting random might be expensive as just choose the median of left, right and center.

PARTITIONING STARTEGY

- For instance if the array is 8,1,4,9,6,3,5,2,7,0 then the pivot is 6
- put the pivot at the end of the array, i starts from the first element and j from the next to last



- While i is to the left of j, we move i right, skipping over elements that are smaller than the pivot. We move j left, skipping over elements that are larger than the pivot.
- When i and j have stopped, i is pointing at a large element and j is pointing at a small element, so we swap them

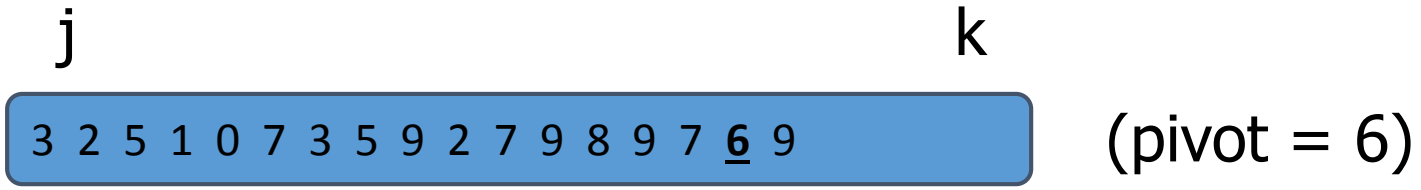
In-Place Quick-Sort

- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than k

Algorithm *inPlaceQuickSort*(S, l, r)
  Input sequence S, ranks l and r
  Output sequence S with the elements of rank between l and r rearranged in increasing order
  if $l \geq r$
    return
  $i \leftarrow$ a random integer between l and r
  $x \leftarrow S.elemAtRank(i)$
  $(h, k) \leftarrow inPlacePartition(x)$
  inPlaceQuickSort$(S, l, h-1)$
  inPlaceQuickSort$(S, k+1, r)$

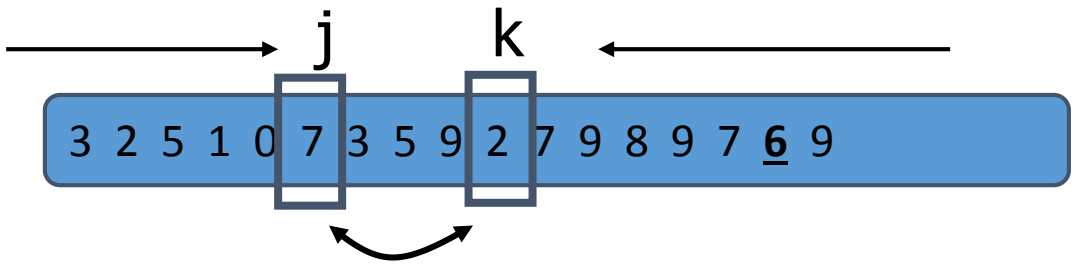
In-Place Partitioning

- Perform the partition using two indices to split S into E and E U G (a similar method can split E U G into E and G).

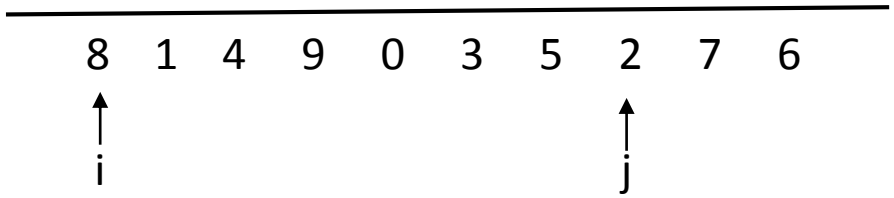


(pivot = 6)

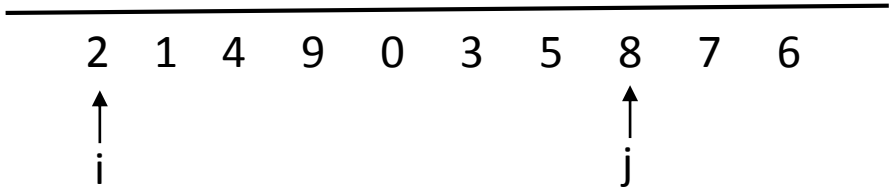
- Repeat until j and k cross:
  - Scan j to the right until finding an element ≥ x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k



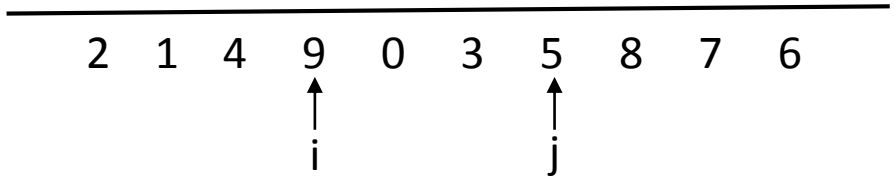
Before First Swap



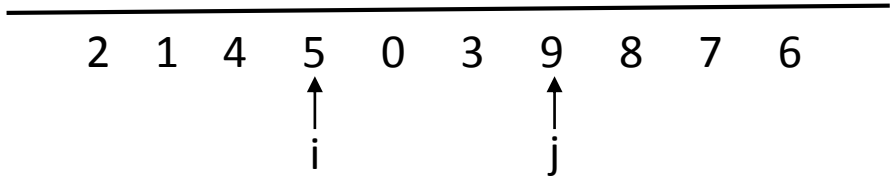
After First Swap



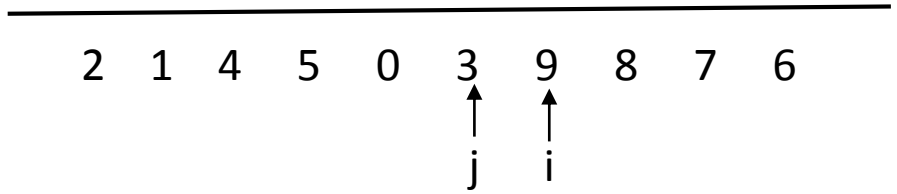
Before Second Swap



After Second Swap

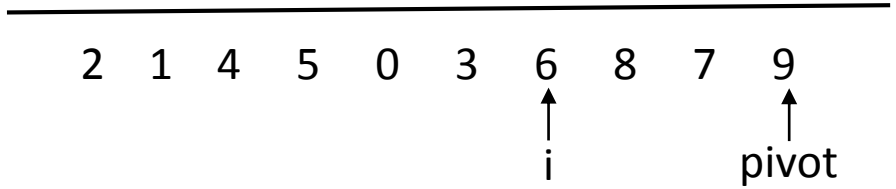


Before Third Swap



At this stage i and j have crossed, so no swap is performed. The final part of the partitioning is to swap the pivot element with the element pointed to by i

After Swap with Pivot



```
1       /**
2        * Quicksort algorithm.
3        * @param a an array of Comparable items.
4        */
5       public static <AnyType extends Comparable<? super AnyType>>
6       void quicksort( AnyType [ ] a )
7       {
8           quicksort( a, 0, a.length - 1 );
9       }
```

Figure 7.13 Driver for quicksort

```
1       /**
2        * Return median of left, center, and right.
3        * Order these and hide the pivot.
4        */
5       private static <AnyType extends Comparable<? super AnyType>>
6       AnyType median3( AnyType [ ] a, int left, int right )
7       {
8           int center = ( left + right ) / 2;
9           if( a[ center ].compareTo( a[ left ] ) < 0 )
10              swapReferences( a, left, center );
11          if( a[ right ].compareTo( a[ left ] ) < 0 )
12              swapReferences( a, left, right );
13          if( a[ right ].compareTo( a[ center ] ) < 0 )
14              swapReferences( a, center, right );
15
16              // Place pivot at position right - 1
17          swapReferences( a, center, right - 1 );
18          return a[ right - 1 ];
19      }
```

Figure 7.14 Code to perform median-of-three partitioning

```
1       /**
2        * Internal quicksort method that makes recursive calls.
3        * Uses median-of-three partitioning and a cutoff of 10.
4        * @param a an array of Comparable items.
5        * @param left the left-most index of the subarray.
6        * @param right the right-most index of the subarray.
7        */
8       private static <AnyType extends Comparable<? super AnyType>>
9       void quicksort( AnyType [ ] a, int left, int right )
10      {
11          if( left + CUTOFF <= right )
12          {
13              AnyType pivot = median3( a, left, right );
14
15                  // Begin partitioning
16              int i = left, j = right - 1;
17              for( ; ; )
18              {
19                  while( a[ ++i ].compareTo( pivot ) < 0 ) { }
20                  while( a[ --j ].compareTo( pivot ) > 0 ) { }
21                  if( i < j )
22                      swapReferences( a, i, j );
23                  else
24                      break;
25              }
26
27              swapReferences( a, i, right - 1 );   // Restore pivot
28
29              quicksort( a, left, i - 1 );    // Sort small elements
30              quicksort( a, i + 1, right );   // Sort large elements
31          }
32          else  // Do an insertion sort on the subarray
33              insertionSort( a, left, right );
34      }
```

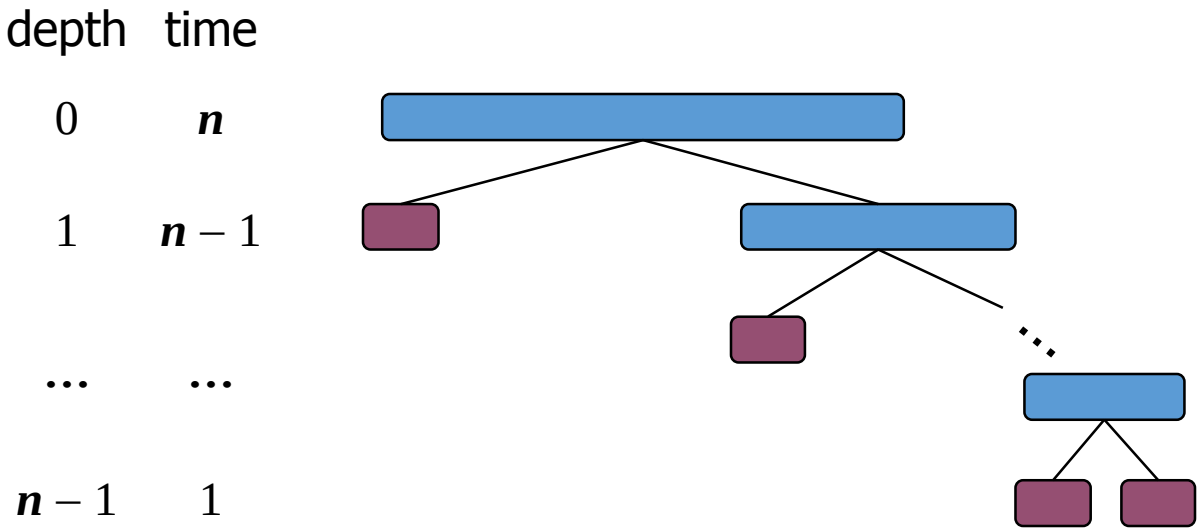
Figure 7.15 Main quicksort routine

Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

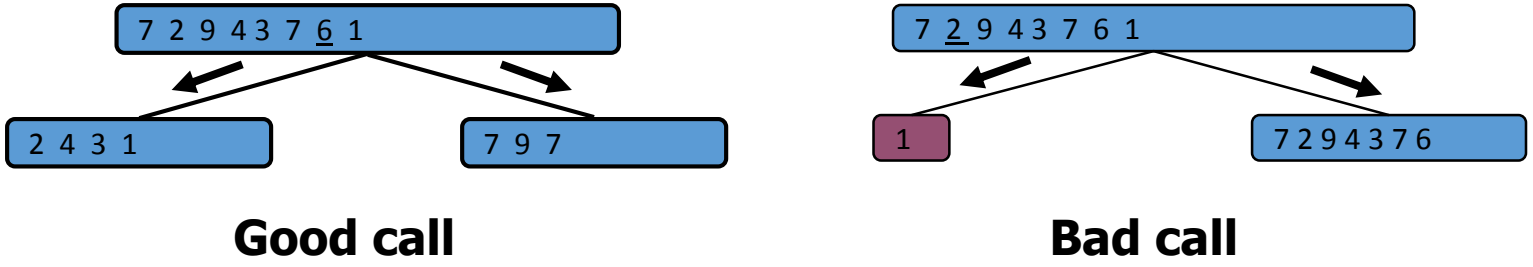
$$n + (n - 1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$

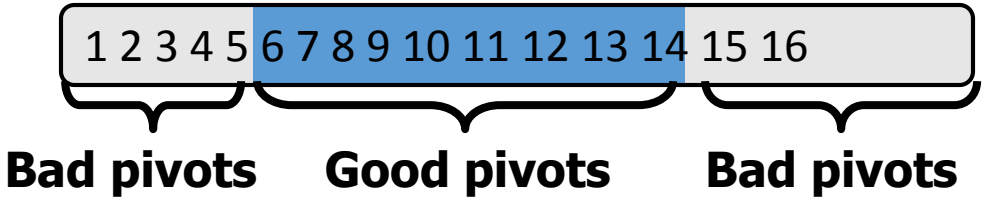


Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$



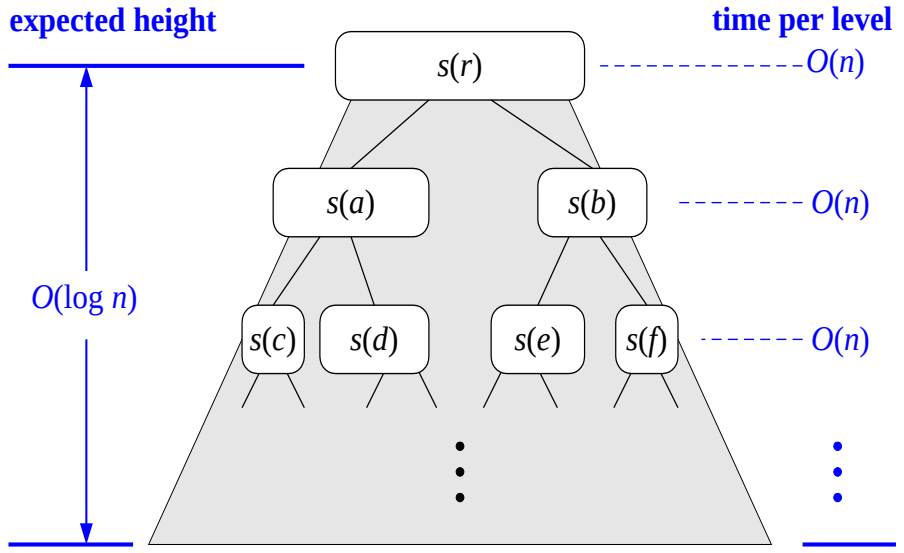
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$

- Therefore, we have
  - For a node of depth $2\log(3/4\ n)$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



total expected time: $O(n \log n)$

Java Implementation

```
1   /** Quick-sort contents of a queue. */
2   public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
3     int n = S.size( );
4     if (n < 2) return;                          // queue is trivially sorted
5     // divide
6     K pivot = S.first( );                       // using first as arbitrary pivot
7     Queue<K> L = new LinkedQueue<>( );
8     Queue<K> E = new LinkedQueue<>( );
9     Queue<K> G = new LinkedQueue<>( );
10    while (!S.isEmpty( )) {                     // divide original into L, E, and G
11      K element = S.dequeue( );
12      int c = comp.compare(element, pivot);
13      if (c < 0)                                // element is less than pivot
14        L.enqueue(element);
15      else if (c == 0)                          // element is equal to pivot
16        E.enqueue(element);
17      else                                      // element is greater than pivot
18        G.enqueue(element);
19    }
20    // conquer
21    quickSort(L, comp);                         // sort elements less than pivot
22    quickSort(G, comp);                         // sort elements greater than pivot
23    // concatenate results
24    while (!L.isEmpty( ))
25      S.enqueue(L.dequeue( ));
26    while (!E.isEmpty( ))
27      S.enqueue(E.dequeue( ));
28    while (!G.isEmpty( ))
29      S.enqueue(G.dequeue( ));
30  }
```

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

Java Implementation

```
1   /** Sort the subarray S[a..b] inclusive. */
2   private static <K> void quickSortInPlace(K[ ] S, Comparator<K> comp,
3                                                              int a, int b) {
4     if (a >= b) return;          // subarray is trivially sorted
5     int left = a;
6     int right = b-1;
7     K pivot = S[b];
8     K temp;                      // temp object used for swapping
9     while (left <= right) {
10      // scan until reaching value equal or larger than pivot (or right marker)
11      while (left <= right && comp.compare(S[left], pivot) < 0) left++;
12      // scan until reaching value equal or smaller than pivot (or left marker)
13      while (left <= right && comp.compare(S[right], pivot) > 0) right--;
14      if (left <= right) {         // indices did not strictly cross
15        // so swap values and shrink range
16        temp = S[left]; S[left] = S[right]; S[right] = temp;
17        left++; right--;
18      }
19    }
20    // put pivot into its final place (currently marked by left index)
21    temp = S[left]; S[left] = S[b]; S[b] = temp;
22    // make recursive calls
23    quickSortInPlace(S, comp, a, left - 1);
24    quickSortInPlace(S, comp, left + 1, b);
25  }
```

QUICKSELECT

- We can use quick sort algorithm for the selection problem.
  1. if |S| = 1 then k =1 and return the element in S as the answer.
  2. Pick a pivot element v ϵ S.
  3. Partition S – {v} into S1 and S2 as in quicksort
  4. if k < = |S1| then the kth smallest element must be in S1. In this case return quickselect(S1, k). if k = 1+|S1| , then pivot is the kth smallest element and we return it as the answer. Otherwise kth smallest element lies in S2 and it is the (k-|S1|-1) th element in S2. We make a recursive call and return (S2, k-|S1|-1).

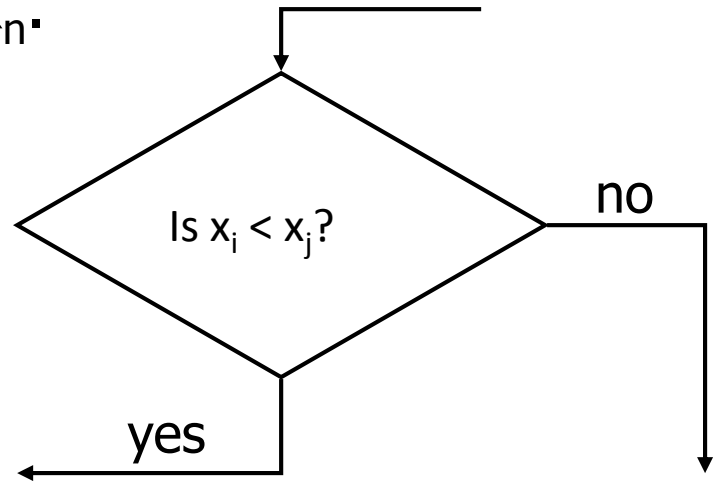
```
1       /**
2        * Internal selection method that makes recursive calls.
3        * Uses median-of-three partitioning and a cutoff of 10.
4        * Places the kth smallest item in a[k-1].
5        * @param a an array of Comparable items.
6        * @param left the left-most index of the subarray.
7        * @param right the right-most index of the subarray.
8        * @param k the desired index (1 is minimum) in the entire array.
9        */
10      private static <AnyType extends Comparable<? super AnyType>>
11      void quickSelect( AnyType [ ] a, int left, int right, int k )
12      {
13          if( left + CUTOFF <= right )
14          {
15              AnyType pivot = median3( a, left, right );
16
17                  // Begin partitioning
18              int i = left, j = right - 1;
19              for( ; ; )
20              {
21                  while( a[ ++i ].compareTo( pivot ) < 0 ) { }
22                  while( a[ --j ].compareTo( pivot ) > 0 ) { }
23                  if( i < j )
24                      swapReferences( a, i, j );
25                  else
26                      break;
27              }
28
29              swapReferences( a, i, right - 1 );   // Restore pivot
30
31              if( k <= i )
32                  quickSelect( a, left, i - 1, k );
33              else if( k > i + 1 )
34                  quickSelect( a, i + 1, right, k );
35          }
36          else  // Do an insertion sort on the subarray
37              insertionSort( a, left, right );
38      }
```

Figure 7.17 Main quickselect routine

Sorting Lower Bound

Comparison-Based Sorting

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, …
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort n elements, $x_1, x_2, \ldots, x_n$.



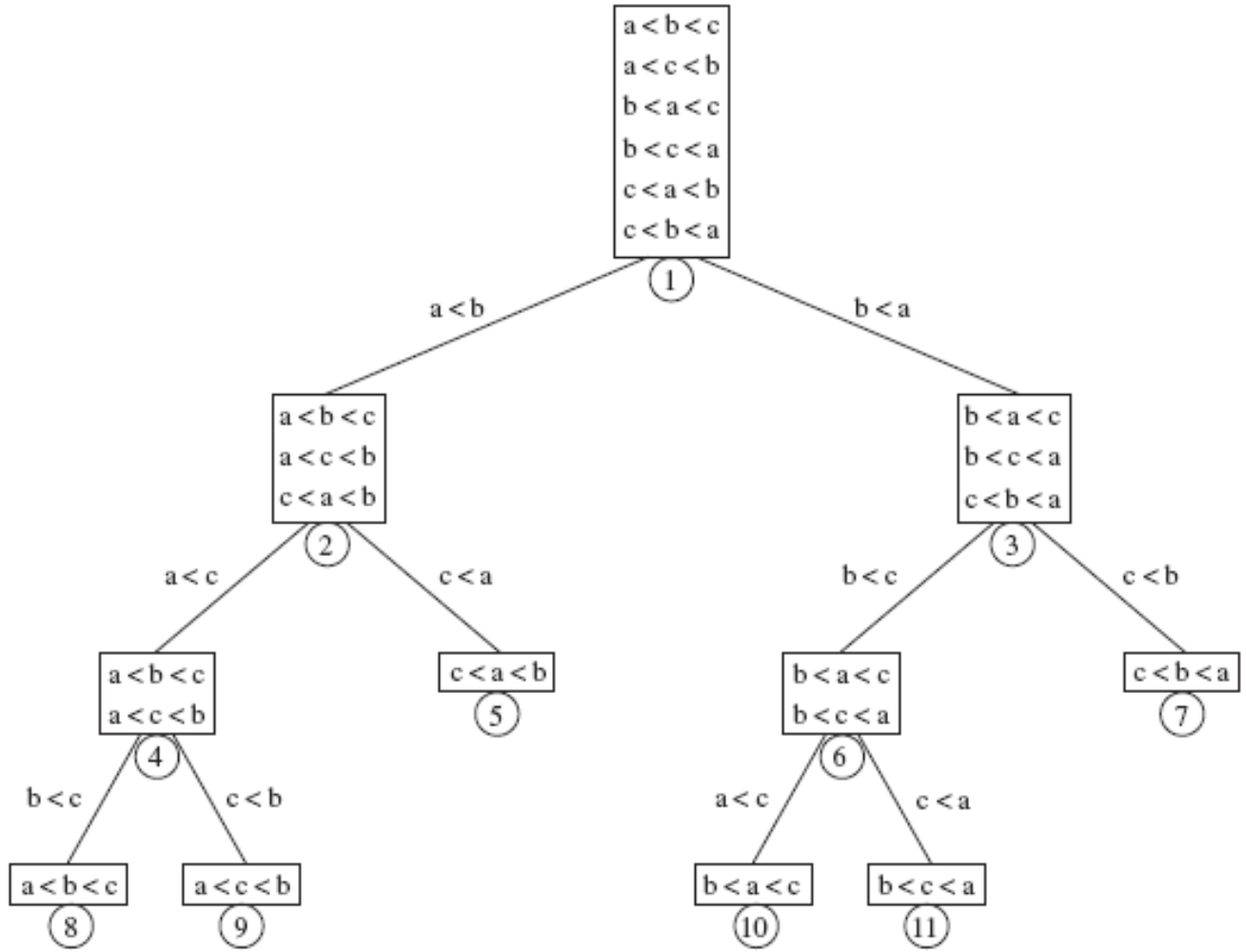
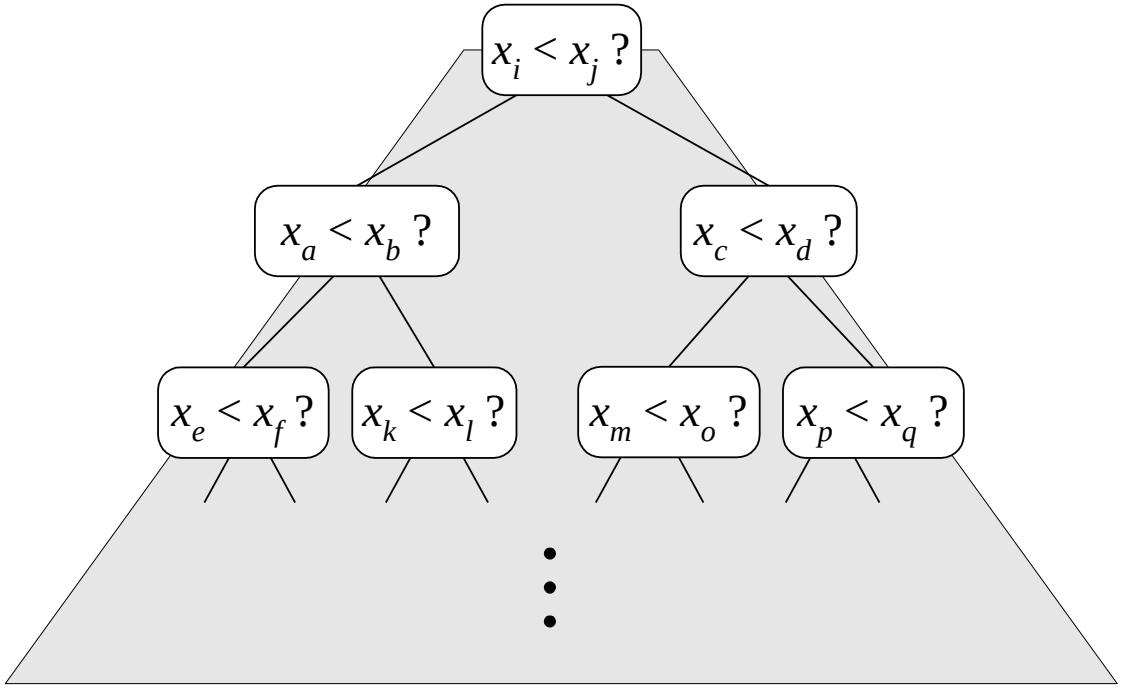


Figure 7.18 A decision tree for three-element sort

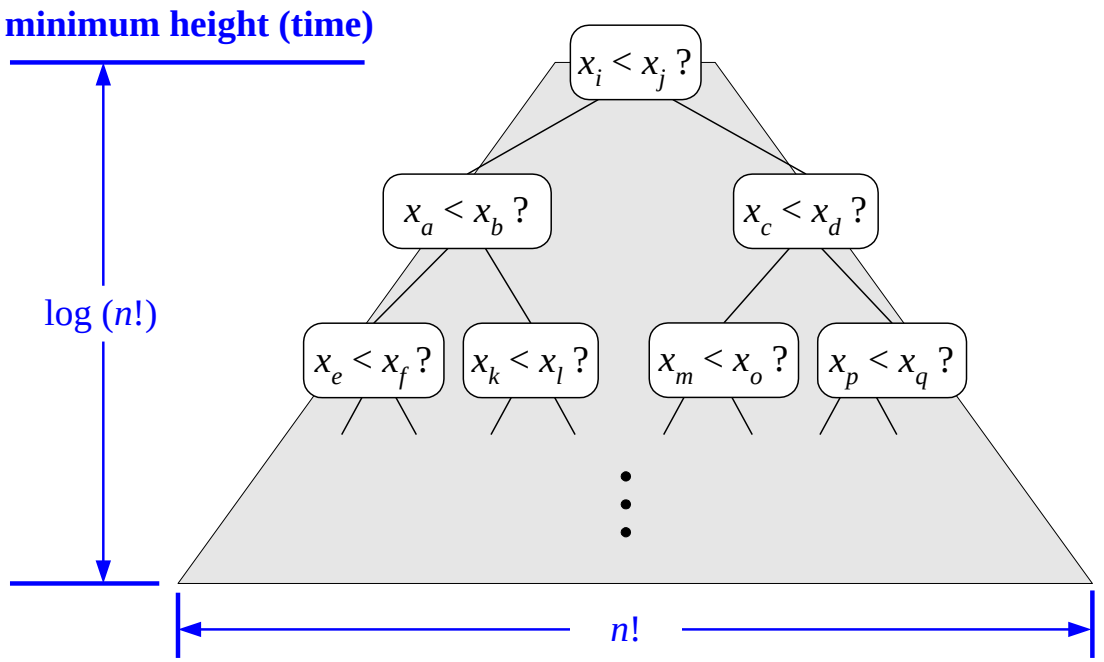
Counting Comparisons

- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree

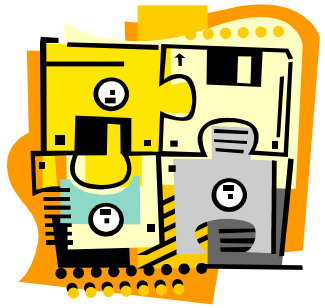


Decision Tree Height

- The height of the decision tree is a lower bound on the running time
- Every input permutation must lead to a separate leaf output
- If not, some input …4…5… would have same output ordering as …5…4…, which would be wrong
- Since there are n!=1·2 · … ·n leaves, the height is at least log (n!)



The Lower Bound

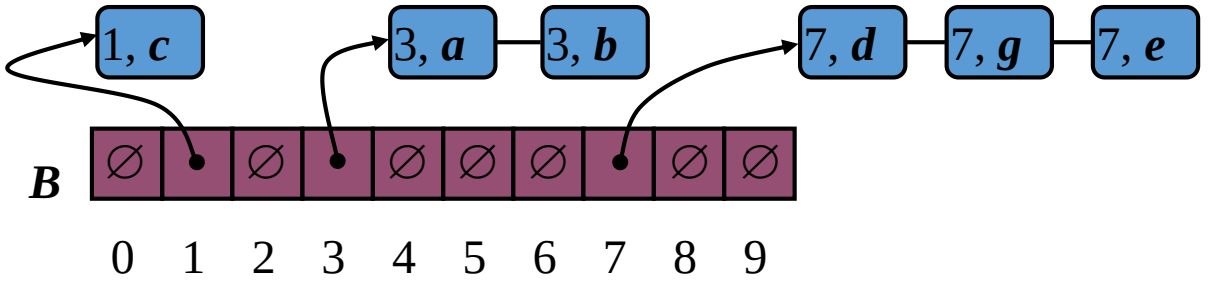


- Any comparison-based sorting algorithms takes at least log (n!) time
- Therefore, any such algorithm takes time at least

$$\log\,(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2).$$

- That is, any comparison-based sorting algorithm must run in Ω(n log n) time.

Bucket-Sort and Radix-Sort



Bucket-Sort

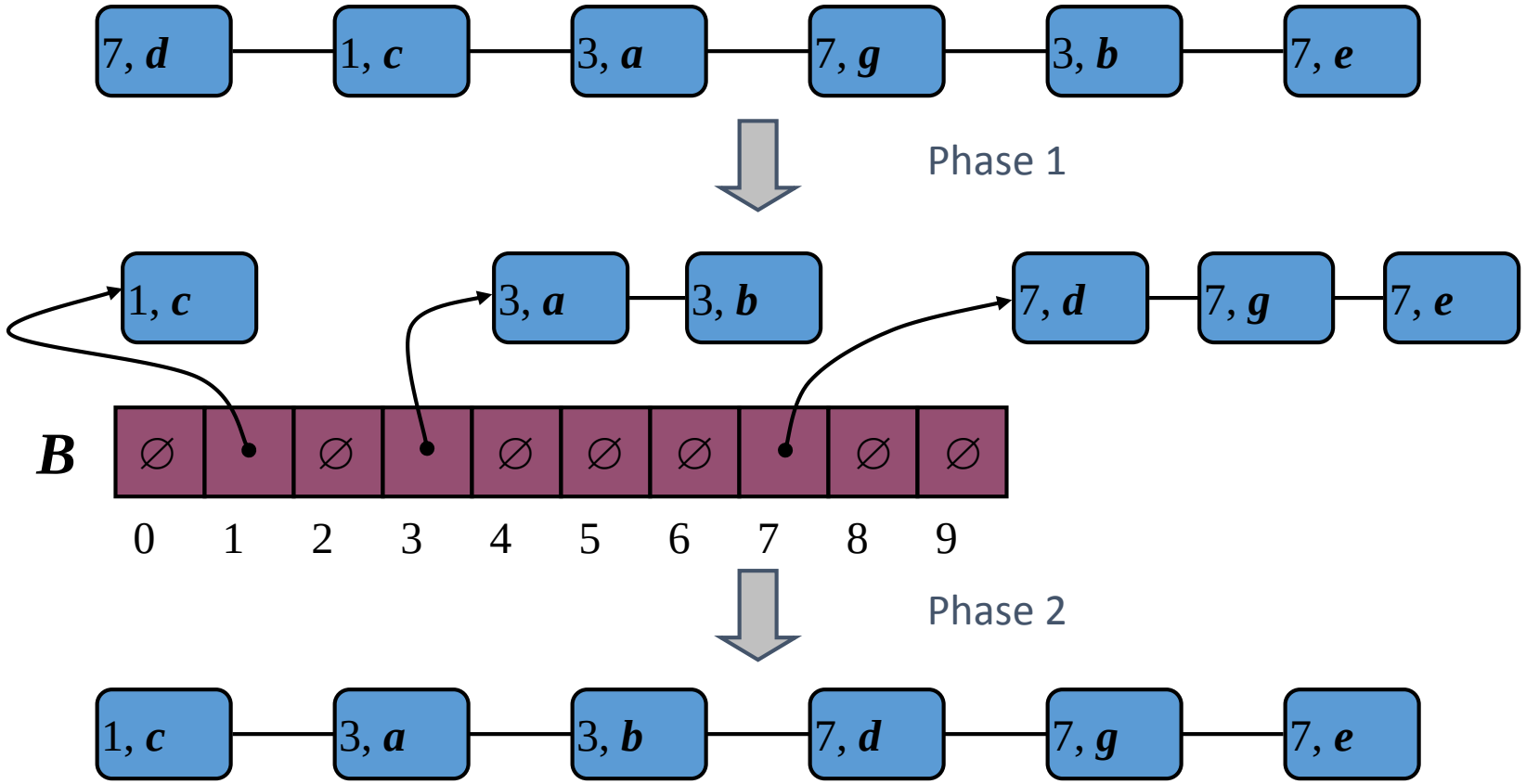
- Let be S be a sequence of n (key, element) items with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ to the end of sequence S
- Analysis:
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time

  Bucket-sort takes $O(n + N)$ time

Algorithm bucketSort(S):
Input: Sequence S of entries with integer keys in the range [0, N − 1]
Output: Sequence S sorted in nondecreasing order of the keys
let B be an array of N sequences, each of which is initially empty
for each entry e in S **do**
  k = the key of e
  remove e from S
  insert e at the end of bucket B[k]
for i = 0 to N−1 **do**
  for each entry e in B[i] **do**
    remove e from B[i]
    insert e at the end of S

Example

- Key range [0, 9]



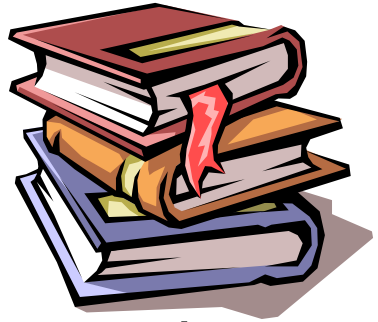
Properties and Extensions

- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator
- Stable Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

Extensions

- Integer keys in the range $[a, b]$
  - Put entry (k, o) into bucket $B[k - a]$
- String keys from a set D of possible strings, where D has constant size (e.g., names of the 50 U.S. states)
  - Sort D and compute the rank $r(k)$ of each string k of D in the sorted sequence
  - Put entry (k, o) into bucket $B[r(k)]$

Lexicographic Order



- A d-tuple is a sequence of d keys $(k_1, k_2, \ldots, k_d)$, where key k_i is said to be the i-th dimension of the tuple
- Example:
  - The Cartesian coordinates of a point in space are a 3-tuple
- The lexicographic order of two d-tuples is recursively defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$
$$\Leftrightarrow$$
$$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

Lexicographic-Sort

- Let C_i be the comparator that compares two tuples by their i-th dimension
- Let *stableSort*(S, C) be a stable sorting algorithm that uses comparator C
- Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm *stableSort*, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of *stableSort*

Algorithm *lexicographicSort*(S)
  Input sequence S of d-tuples
  Output sequence S sorted in lexicographic order

  for $i \leftarrow d$ **downto** 1
    stableSort(S, C_i)

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

Radix-Sort

- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, N-1]$
- Radix-sort runs in time $O(d(n+N))$

Algorithm *radixSort*(S, N)
  Input sequence S of d-tuples such that $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \leq (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S
  Output sequence S sorted in lexicographic order
  for $i \leftarrow d$ **downto** 1
    bucketSort(S, N)

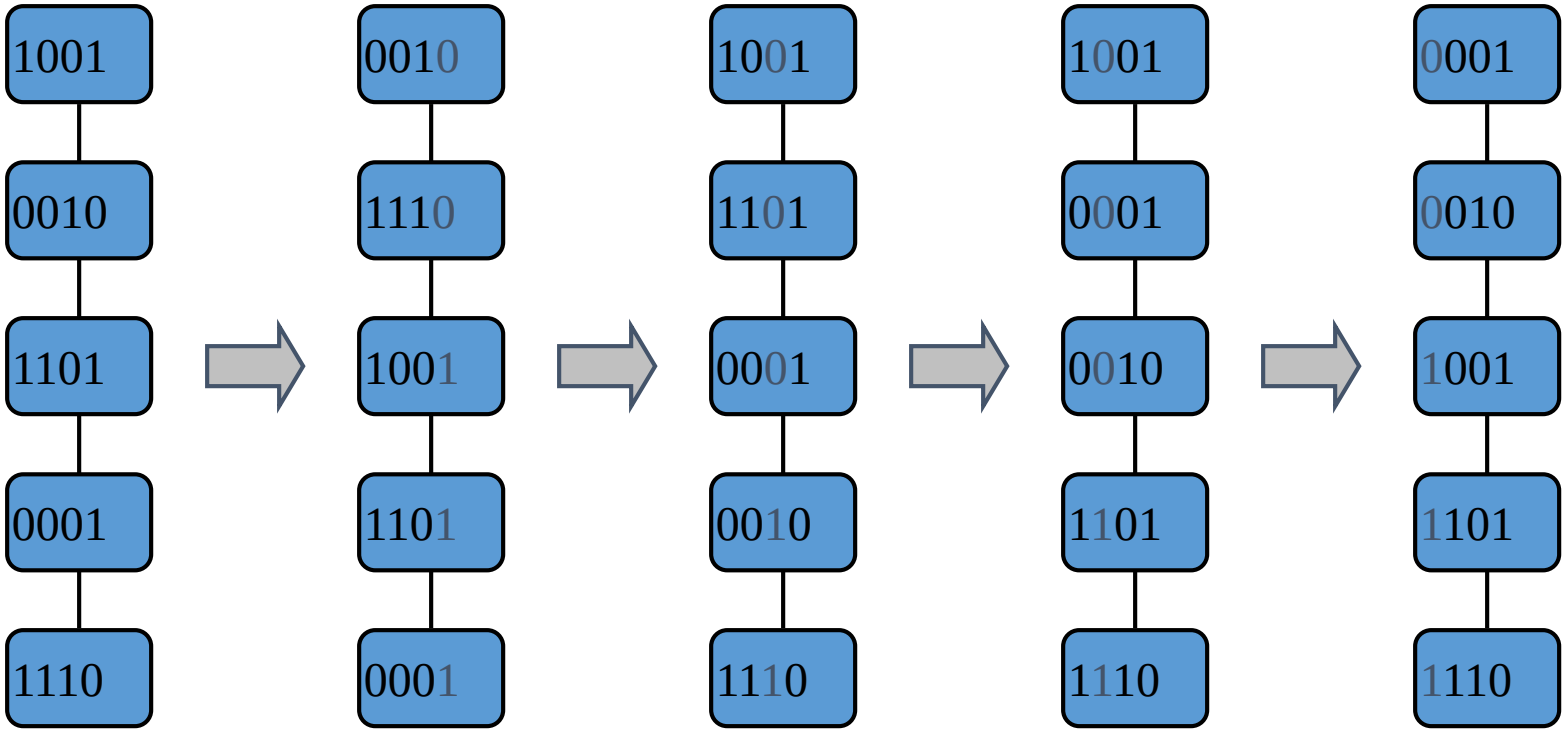
Radix-Sort for Binary Numbers

- Consider a sequence of n b-bit integers
$$x = x_{b-1} \ldots x_1 x_0$$
- We represent each element as a b-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$
- This application of the radix-sort algorithm runs in $O(bn)$ time
- For example, we can sort a sequence of 32-bit integers in linear time

Algorithm *binaryRadixSort*(S)
  Input sequence S of b-bit integers
  Output sequence S sorted
  replace each element x of S with the item $(0, x)$
  for $i \leftarrow 0$ **to** $b - 1$
    replace the key k of each item (k, x) of S with bit x_i of x
    bucketSort(S, 2)

Example

- Sorting a sequence of 4-bit integers



Radix Sort

- The running time is O(p(N+b)) where p is the number of passes, N is number of items and b is number of buckets.

INITIAL ITEMS:	064, 008, 216, 512, 027, 729, 000, 001, 343, 125
SORTED BY 1's digit:	000, 001, 512, 343, 064, 125, 216, 027, 008, 729
SORTED BY 10's digit:	000, 001, 008, 512, 216, 125, 027, 729, 343, 064
SORTED BY 100's digit:	000, 001, 008, 027, 064, 125, 216, 343, 512, 729

Figure 7.22 Radix sort trace

```
1       /*
2        * Radix sort an array of Strings
3        * Assume all are all ASCII
4        * Assume all have same length
5        */
6       public static void radixSortA( String [ ] arr, int stringLen )
7       {
8           final int BUCKETS = 256;
9           ArrayList<String> [ ] buckets = new ArrayList<>[ BUCKETS ];
10
11          for( int i = 0; i < BUCKETS; i++ )
12              buckets[ i ] = new ArrayList<>( );
13
14          for( int pos = stringLen - 1; pos >= 0; pos-- )
15          {
16              for( String s : arr )
17                  buckets[ s.charAt( pos ) ].add( s );
18
19              int idx = 0;
20              for( ArrayList<String> thisBucket : buckets )
21              {
22                  for( String s : thisBucket )
23                      arr[ idx++ ] = s;
24
25                  thisBucket.clear( );
26              }
27          }
28      }
```

Figure 7.23 Simple implementation of radix sort for strings, using an `ArrayList` of buckets

```
1         /*
2          * Counting radix sort an array of Strings
3          * Assume all are all ASCII
4          * Assume all have same length
5          */
6         public static void countingRadixSort( String [ ] arr, int stringLen )
7         {
8             final int BUCKETS = 256;
9
10            int N = arr.length;
11            String [ ] buffer = new String [ N ];
12
13            String [ ] in = arr;
14            String [ ] out = buffer;
15
16            for( int pos = stringLen - 1; pos >= 0; pos-- )
17            {
18                int [ ] count = new int [ BUCKETS + 1 ];
19
20                for( int i = 0; i < N; i++ )
21                    count[ in[ i ].charAt( pos ) + 1 ]++;
22
23                for( int b = 1; b <= BUCKETS; b++ )
24                    count[ b ] += count[ b - 1 ];
25
26                for( int i = 0; i < N; i++ )
27                    out[ count[ in[ i ].charAt( pos ) ]++ ] = in[ i ];
28
29                  // swap in and out roles
30                String [ ] tmp = in;
31                in = out;
32                out = tmp;
33            }
34
35               // if odd number of passes, in is buffer, out is arr; so copy back
36            if( stringLen % 2 == 1 )
37                for( int i = 0; i < arr.length; i++ )
38                   out[ i ] = in[ i ];
39        }
```

Figure 7.24 Counting radix sort for fixed-length strings

```
1   /*
2    * Radix sort an array of Strings
3    * Assume all are all ASCII
4    * Assume all have length bounded by maxLen
5    */
6   public static void radixSort( String [ ] arr, int maxLen )
7   {
8       final int BUCKETS = 256;
9
10      ArrayList<String> [ ] wordsByLength = new ArrayList<>[ maxLen + 1 ];
11      ArrayList<String> [ ] buckets = new ArrayList<>[ BUCKETS ];
12
13      for( int i = 0; i < wordsByLength.length; i++ )
14          wordsByLength[ i ] = new ArrayList<>( );
15
16      for( int i = 0; i < BUCKETS; i++ )
17          buckets[ i ] = new ArrayList<>( );
18
19      for( String s : arr )
20          wordsByLength[ s.length( ) ].add( s );
21
22      int idx = 0;
23      for( ArrayList<String> wordList : wordsByLength )
24          for( String s : wordList )
25              arr[ idx++ ] = s;
26
27      int startingIndex = arr.length;
28      for( int pos = maxLen - 1; pos >= 0; pos-- )
29      {
30          startingIndex -= wordsByLength[ pos + 1 ].size( );
31
32          for( int i = startingIndex; i < arr.length; i++ )
33              buckets[ arr[ i ].charAt( pos ) ].add( arr[ i ] );
34
35          idx = startingIndex;
36          for( ArrayList<String> thisBucket : buckets )
37          {
38              for( String s : thisBucket )
39                  arr[ idx++ ] = s;
40
41              thisBucket.clear( );
42          }
43      }
44  }
```

Figure 7.25 Radix sort for variable length strings

External Sorting

- Algorithms designed to handle very large input, too big to be in the main memory
- Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable.
- Much more device dependent , subsequent algorithms are considered for tapes one of restrictive storage devices

Simple Algorithm

- The basic external sorting algorithm uses the merging algorithm from merge sort.
- Suppose we have four tapes $T_{a1}, T_{a2}, T_{b1}, T_{b2}$, which are two input tabes and two output tapes.

T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

Simple Algorithm

- Suppose the internal memory can hold (and sort) M records at a time.
- First step is to read M records and sort the records internally and write the sorted records alternatively to T_{b1} and T_{b2}.
- Each sorted records is called a run. Now tapes T_{b1} and T_{b2} contains groups of runs.

T_{a1}							
T_{a2}							
T_{b1}	11	81	94	17	28	99	15
T_{b2}	12	35	96	41	58	75	

Simple Algorithm

- Take first run from each tapes and merge them, writing the results which a run twice as long onto T_{a1}.
- Merging is performed as T_{b1} and T_{b2} advance
- Then take next run from each tape, merge these, and write the results to T_{a2}.

T_{a1}	11	12	35	81	94	96	15
T_{a2}	17	28	41	58	75	99	
T_{b1}							
T_{b2}							

- Do this until both T_{b1} and T_{b2} become empty.
- The algorithm will require $\left\lceil \log(\frac{N}{M}) \right\rceil$ passes, plus the initial run constructing pass.

T_{a1}												
T_{a2}												
T_{b1}	11	12	17	28	41	58	75	81	94	96	99	
T_{b2}	15											

T_{a1}	11	12	15	17	28	41	58	75	81	94	96	99
T_{a2}												
T_{b1}												
T_{b2}												

Multiway Merge

- If we have extra tapes, then we can except to reduce the number of passes required to sort our input. We do this by extending the basic (two-way) merge to a k-way merge.

- If there are k input tapes, we need to find the smallest of k elements. This can be done using priority queue.

- To obtain the next element to write on the output tape , perform deleteMin operation.

Multiway Merge

- Using the same example as before, distribute the input onto three tapes.
- We then need two more passes of three-way merging to complete the sort.

T_{a1}						
T_{a2}						
T_{a3}						
T_{b1}	11	81	94	41	58	75
T_{b2}	12	35	96	15		
T_{b3}	17	28	99			

- After the initial run constructing phase the number of passes required using k-way merging is $\left\lceil log_3(\frac{N}{M}) \right\rceil$, because the runs get k times as large in each pas

T_{a1}	11	12	17	28	35	81	94	96	99				
T_{a2}	15	41	58	75									
T_{a3}													
T_{b1}													
T_{b2}													
T_{b3}													

T_{a1}													
T_{a2}													
T_{a3}													
T_{b1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{b2}													
T_{b3}													

Polyphase Merge

- The k-way merging strategy developed in the last section requires the use of 2k tapes. It is possible to get by using only k +1 tapes.
- This can be done by splitting the original runs unevenly.
- The original distribution of runs makes great deal of difference.
- If the number of runs is a Fibonacci number F_N, then the best way to distribute them is to split them into two Fibonacci numbers F_{N-1} and F_{N-2}.

	Run Const.	After $T_3 + T_2$	After $T_1 + T_2$	After $T_1 + T_3$	After $T_2 + T_3$	After $T_1 + T_2$	After $T_1 + T_3$	After $T_2 + T_3$
T_1	0	13	5	0	3	1	0	1
T_2	21	8	0	5	2	0	1	0
T_3	13	0	8	3	0	2	1	0

Replacement Selection

- As soon as the first record is written to an output tape, the memory it used becomes available for another record. if the next record on the tape is larger than the record we have just output, then it can be included in the run.

- This technique is commonly referred as replacement selection.

- if the input is randomly distributed, replacement selection can be shown to produce runs of average length 2M.

Replacement Selection

	3 Elements in Heap Array h[1]	h[2]	h[3]	Output	Next Element Read
Run 1	11	94	81	11	96
	81	94	96	81	12*
	94	96	12*	94	35*
	96	35*	12*	96	17*
	17*	35*	12*	End of Run	Rebuild Heap
Run 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	15*
	58	99	15*	58	End of Tape
	99		15*	99	
			15*	End of Run	Rebuild Heap
Run 3	15			15	

Figure 7.26 Example of run construction