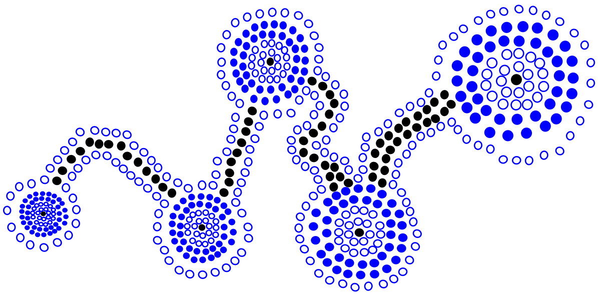
ADVANCED SORTING

- Review of Sorting
- Merge Sort
- Sets
- Quick Sort
- How Fast Can We Sort?



Sorting Algorithms

- Selection Sort uses a priority queue P implemented with an unsorted sequence:
  - **Phase 1**: the insertion of an item into P takes $O(1)$ time; overall $O(n)$
  - **Phase 2**: removing an item takes time proportional to the number of elements in P $O(n)$: overall $O(n^2)$
  - Time Complexity: $O(n^2)$

Sorting Algorithms (cont.)

- Insertion Sort is performed on a priority queue P which is a sorted sequence:
  - **Phase 1**: the first insertItem takes $O(1)$, the second $O(2)$, until the last insertItem takes $O(n)$: overall $O(n^2)$
  - **Phase 2**: removing an item takes $O(1)$ time; overall $O(n)$.
  - Time Complexity: $O(n^2)$
- Heap Sort uses a priority queue K which is a heap.
  - insertItem and removeMin each take $O(\log k)$, k being the number of elements in the heap at a given time.
  - **Phase 1**: n elements inserted: $O(n\log n)$ time
  - **Phase 2**: n elements removed: $O(n \log n)$ time.
  - Time Complexity: $O(n\log n)$

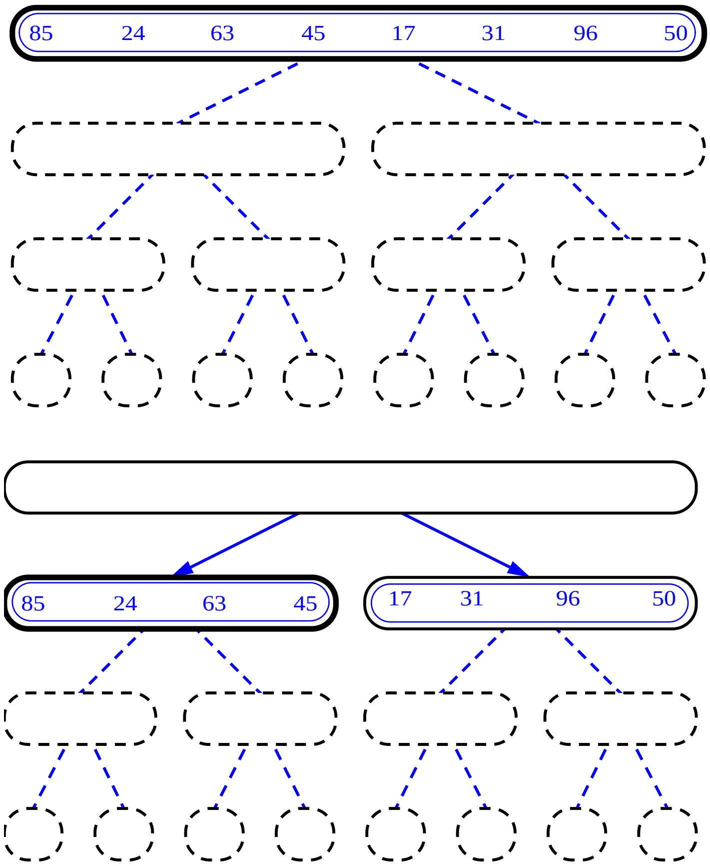
Divide-and-Conquer

- *Divide and Conquer* is more than just a military strategy, it is also a method of algorithm design that has created such efficient algorithms as Merge Sort.
- In terms or algorithms, this method has three distinct steps:
  - **Divide**: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets.
  - **Recur**: Use divide and conquer to solve the subproblems associated with the data subsets.
  - **Conquer**: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem.

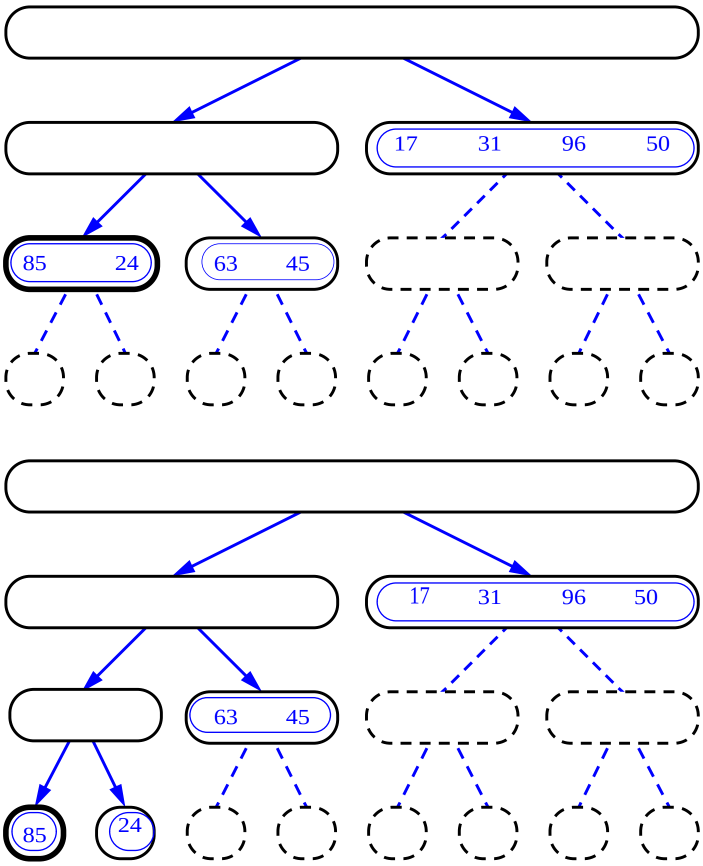
Merge-Sort

- Algorithm:
  - **Divide**: If S has at leas two elements (nothing needs to be done if S has zero or one elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.
  - **Recur**: Recursive sort sequences S_1 and S_2.
  - **Conquer**: Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.
- Merge Sort Tree:
  - Take a binary tree T
  - Each node of T represents a recursive call of the merge sort algorithm.
  - We assocoate with each node v of T a the set of input passed to the invocation v represents.
  - The external nodes are associated with individual elements of S, upon which no recursion is called.

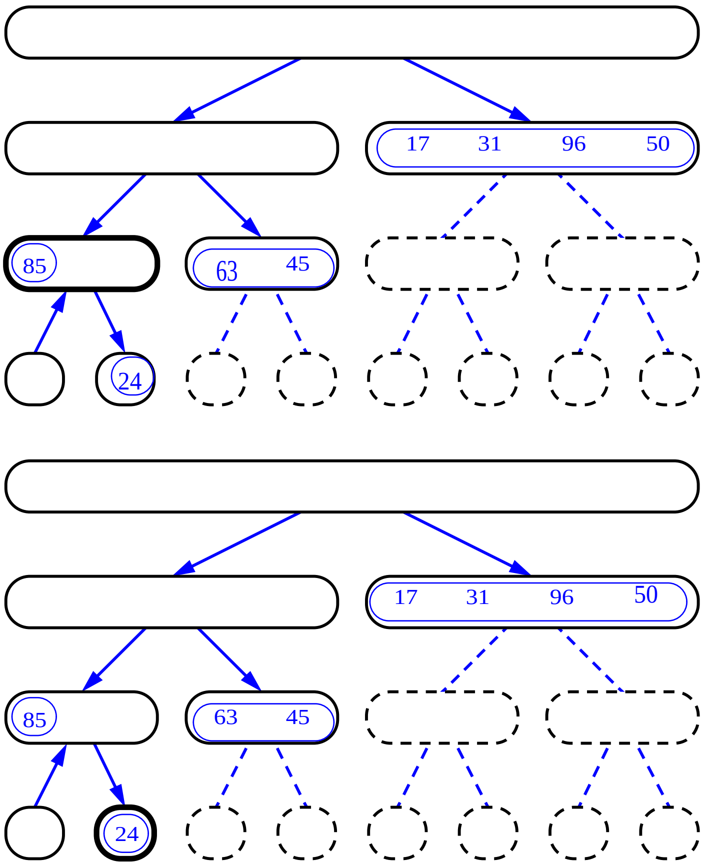
Merge-Sort



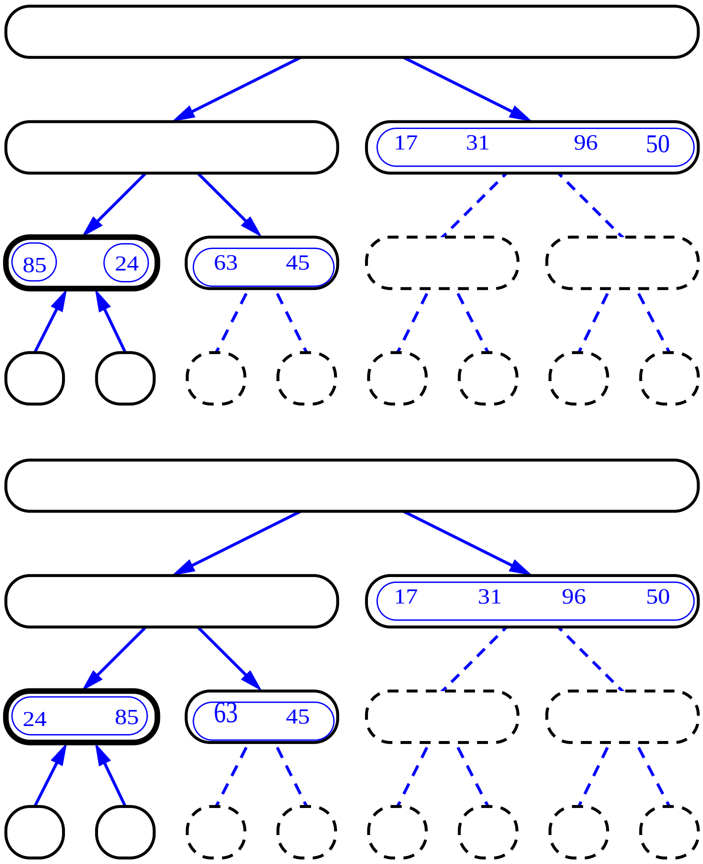
Merge-Sort(cont.)



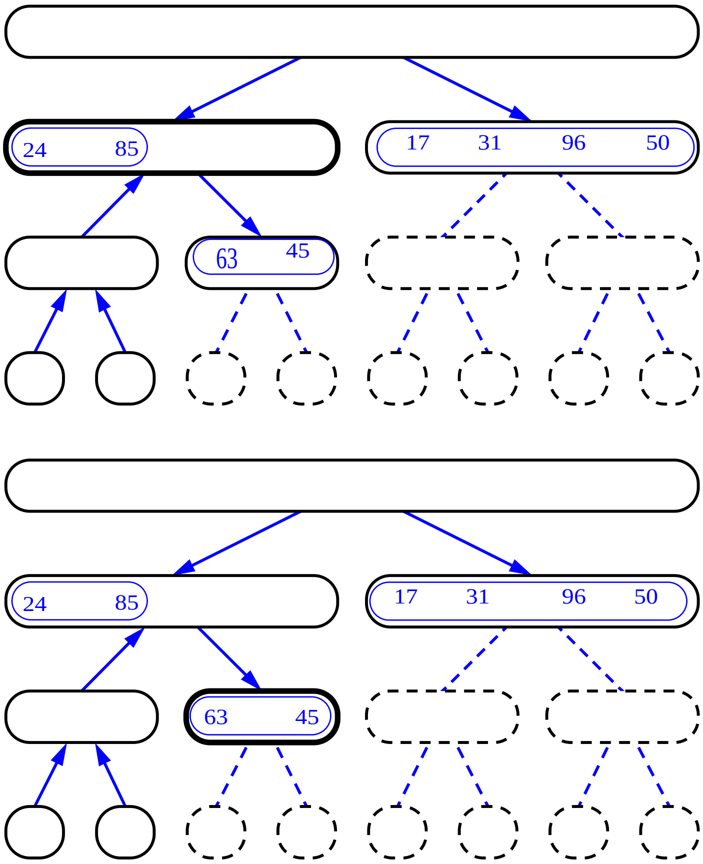
Merge-Sort (cont.)



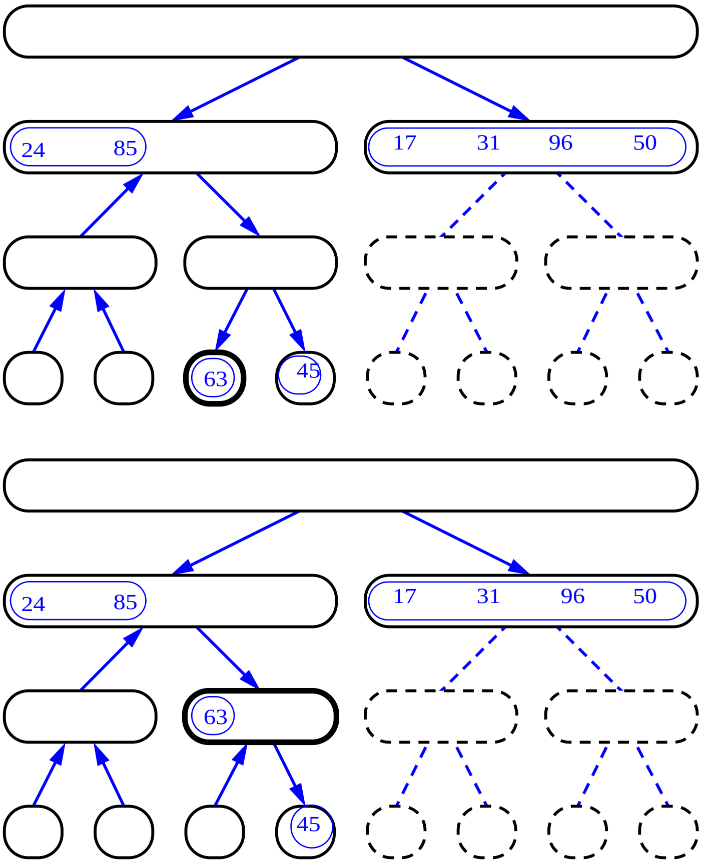
Merge-Sort (cont.)



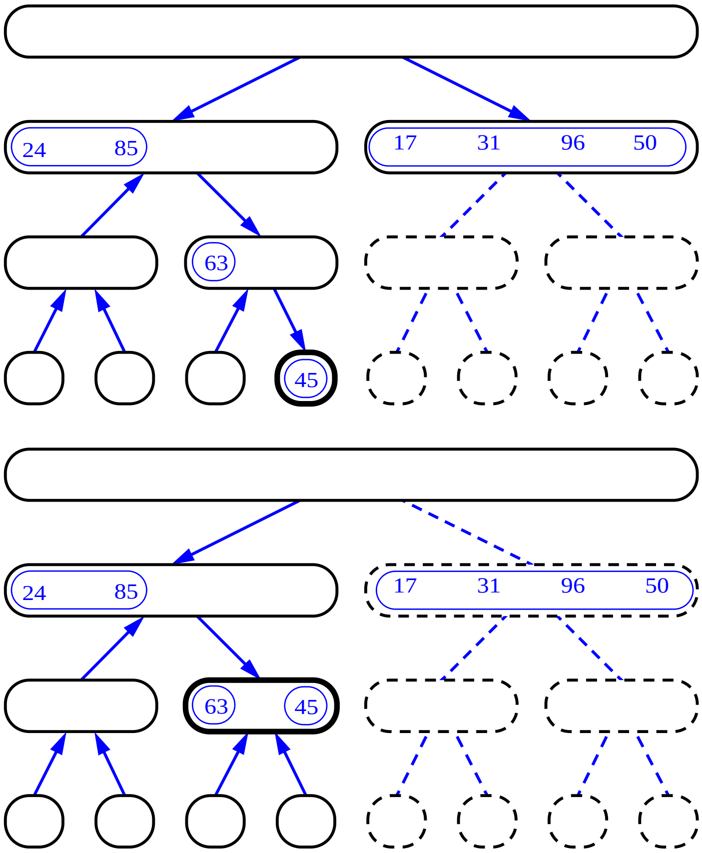
Merge-Sort (cont.)



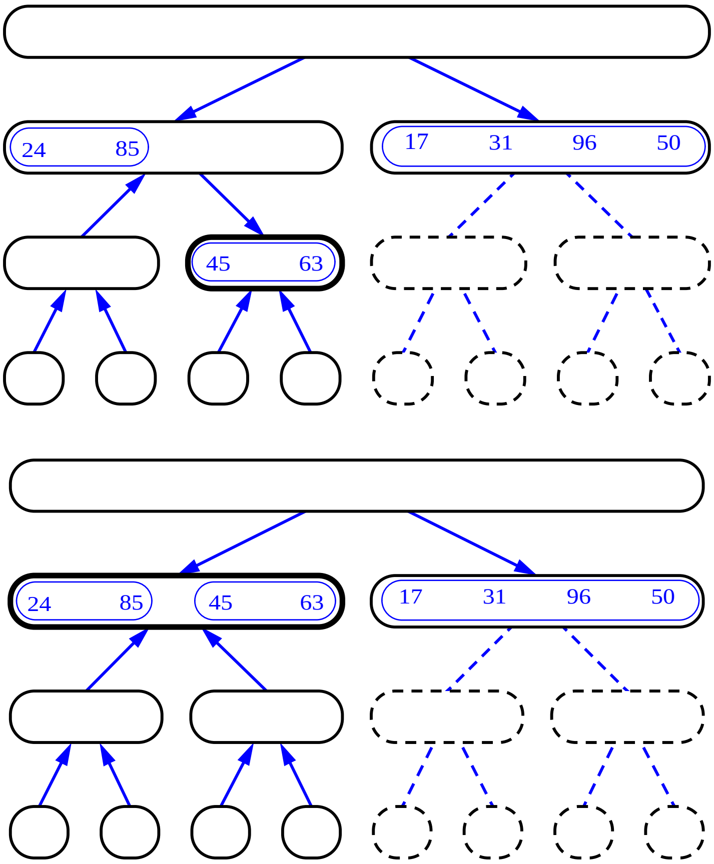
Merge-Sort (cont.)



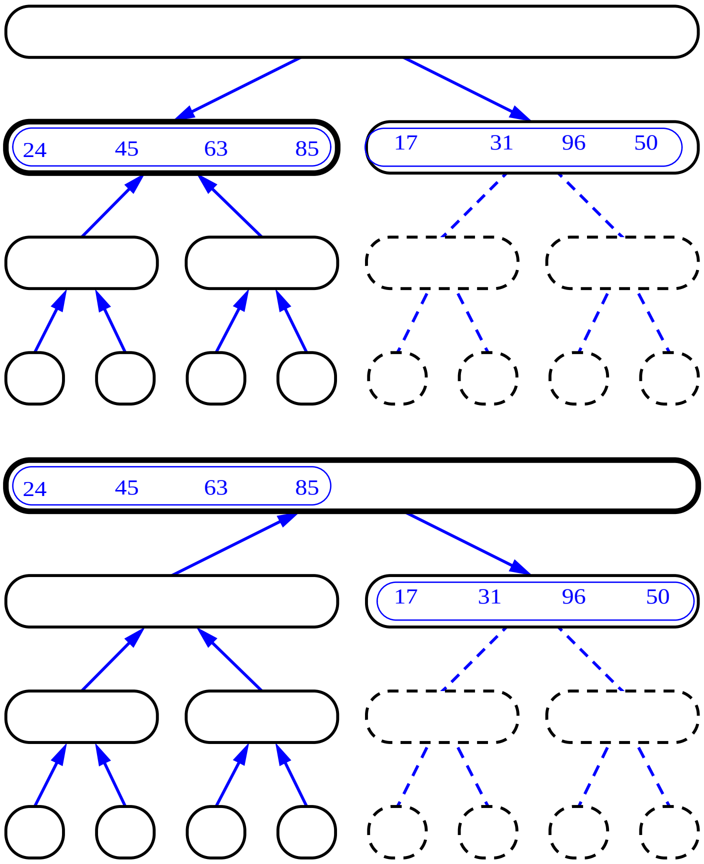
Merge-Sort (cont.)



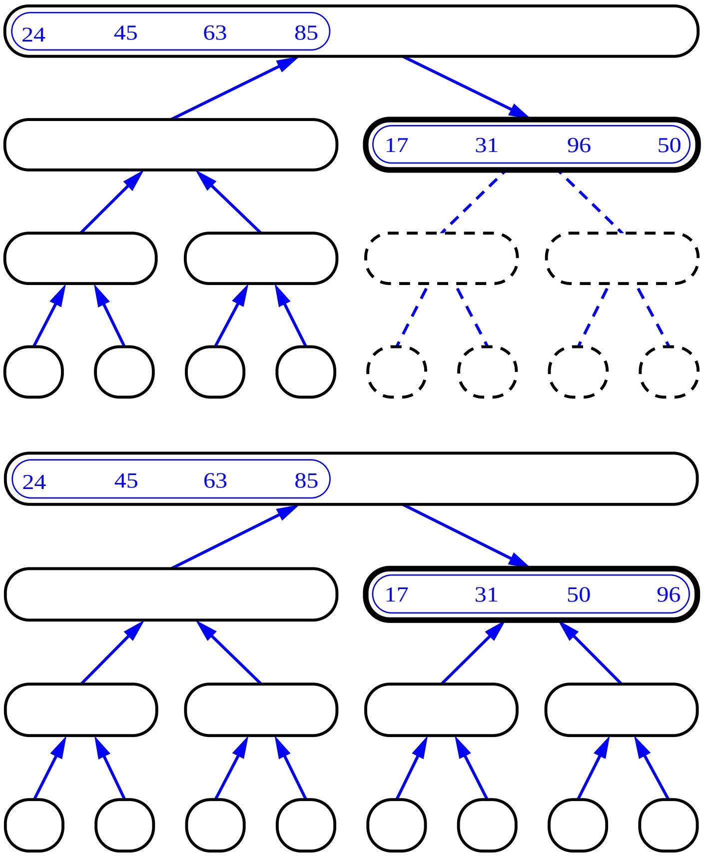
Merge-Sort(cont.)



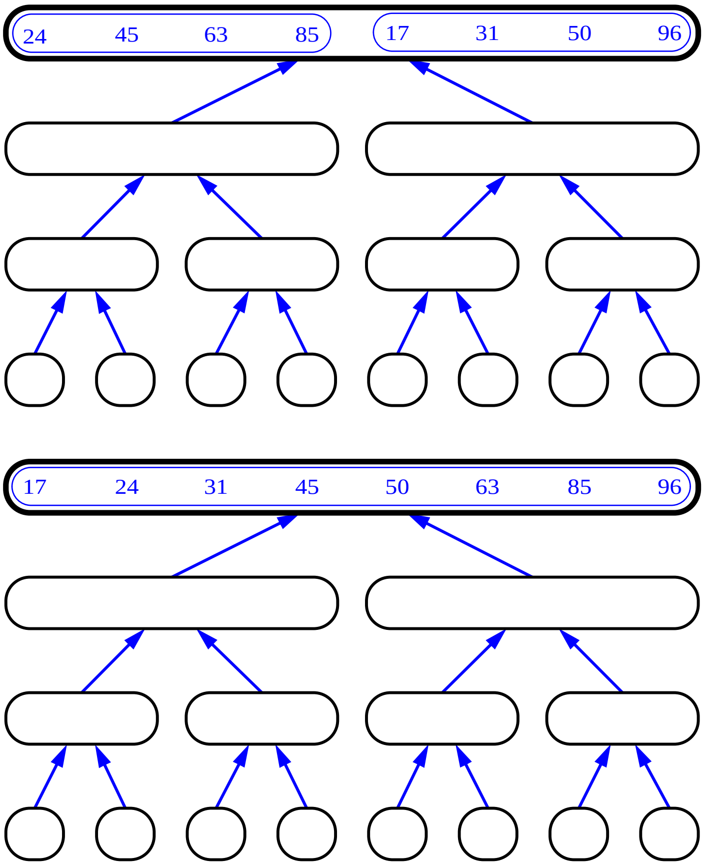
Merge-Sort (cont.)



Merge-Sort (cont.)



Merge-Sort (cont.)



Merging Two Sequences

- Pseudo-code for merging two sorted sequences into a unique sorted sequence

  Algorithm merge (S1, S2, S):

  Input: Sequence *S1* and *S2* (on whose elements a total order relation is defined) sorted in nondecreas ing order, and an empty sequence *S*.

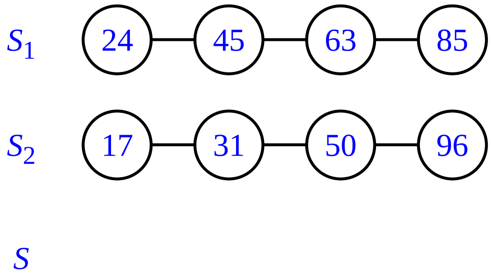
  Ouput: Sequence *S* containing the union of the ele ments from *S1* and *S2* sorted in nondecreasing order; sequence *S1* and *S2* become empty at the end of the execution

```
while S1 is not empty and S2 is not empty do
  if S1.first().element() ≤ S2.first().element() then
    {move the first element of S1 at the end of S}
    S.insertLast(S1.remove(S1.first()))
  else
    { move the first element of S2 at the end of S}
    S.insertLast(S2.remove(S2.first()))
while S1 is not empty do
  S.insertLast(S1.remove(S1.first()))
  {move the remaining elements of S2 to S}
while S2 is not empty do
  S.insertLast(S2.remove(S2.first()))
```

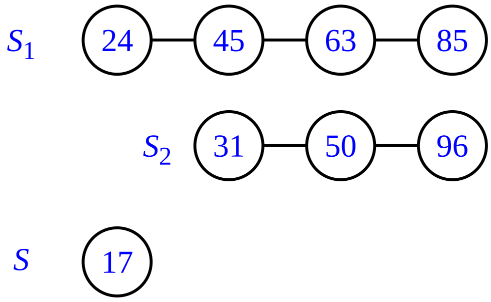

Merging Two Sequences (cont.)

- Some pictures:

  a)

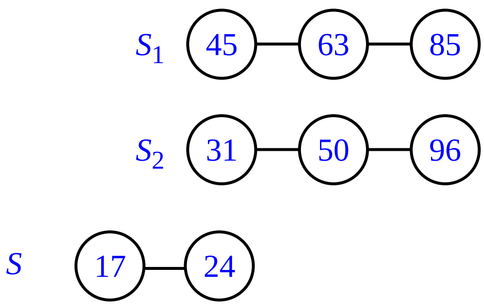


  b)

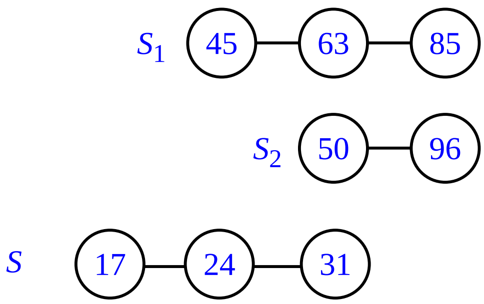


Merging Two Sequences (cont.)

c)

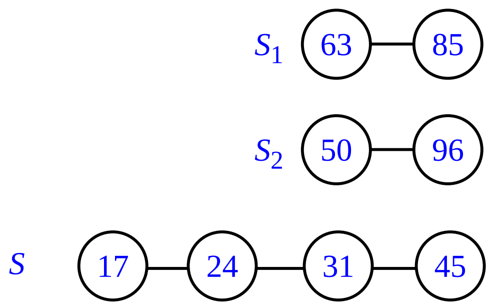


d)

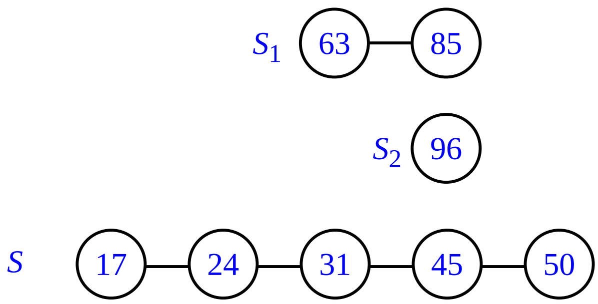


Merging Two Sequences (cont.)

e)

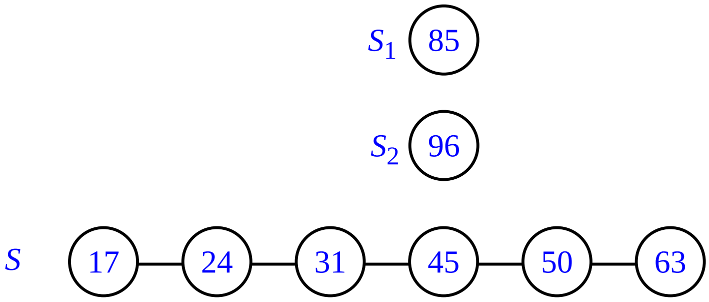


f)

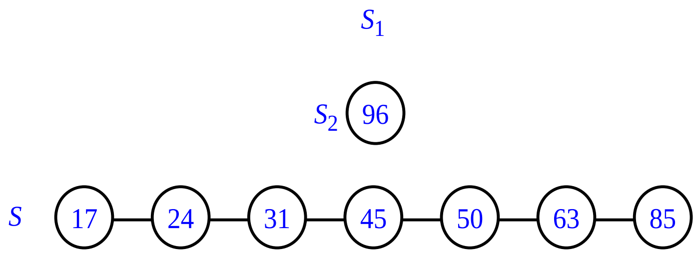


Merging Two Sequences (cont.)

g)

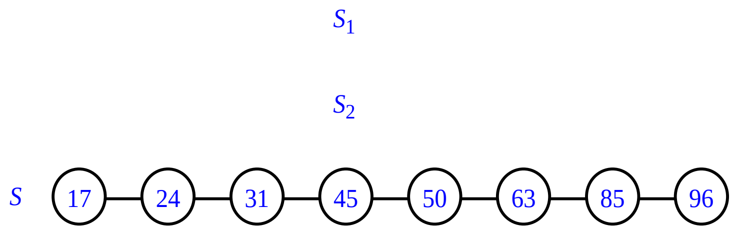


h)



Merging Two Sequences (cont.)

i)



Java Implementation of Merge-Sort

- Interface SortObject

```
public interface SortObject {
  //sort sequence S in nondecreasing order
   using compartor c
  public void sort (Sequence S, Comparator c);
}
```


Java Implementation of Merge-Sort(cont.)

```
public class ListMergeSort implements SortObject {
  public void sort(Sequence S, Comparator c) {
     int n = S.size();
    if (n < 2) return; // a sequence with 0 or
  1 element is already sorted.
    // divide
    Sequence S1 = (Sequence)S.newContainer();
    // put the first half of S into S1
    for (int i=1; i <= (n+1)/2; i++) {
     S1.insertLast(S.remove(S.first()));
    }
    Sequence S2 = (Sequence)S.newContainer();
    // put the second half of S into S2
    for (int i=1; i <= n/2; i++) {
      S2.insertLast(S.remove(S.first()));
    }
    sort(S1,c); // recur
    sort(S2,c);
    merge(S1,S2,c,S); // conquer
 }
```

Java Implementation of Merge-Sort(cont.)

```
public void merge(Sequence S1, Sequence S2,
    Comparator c, Sequence S) {
  while(!S1.isEmpty() && !S2.isEmpty()) {
    if(c.isLessThanOrEqualTo(S1.first().element(),
      S2.first().element())) {
      // S1's 1st elt <= S2's 1st elt
      S.insertLast(S1.remove(S1.first()));
    }
    else { // S2's 1st elt is the smaller one
      S.insertLast(S2.remove(S2.first()));
    }
  }

  if(S1.isEmpty()) {
    while(!S2.isEmpty()) {
      S.insertLast(S2.remove(S2.first()));
    }
  }
  if(S2.isEmpty()) {
    while(!S1.isEmpty()) {
      S.insertLast(S1.remove(S1.first()));
    }
  }
}
```


Running Time of Merge-Sort

- **Proposition 1**: The merge-sort tree associated with the execution of a merge-sort on a sequence of n elements has a height of $\lceil \log n \rceil$
- **Proposition 2**: A merge sort algorithm sorts a sequence of size n in $O(n\log n)$ time
- We assume only that the input sequence S and each of the sub-sequences created by each recursive call of the algorithm can access, insert to, and delete from the first and last nodes in $O(1)$ time.
- We call the time spent at node v of merge-sort tree T the running time of the recusive call associated with v, excluding the recursive calls sent to v's children.

Running Time of Merge-Sort (cont.)

- If we let i represent the depth of node v in the merge-sort tree, the time spent at node v is $O(n/2^i)$ since the size of the sequence associated with v is $n/2^i$.
- Observe that T has exactly 2^i nodes at depth i. The total time spent at depth i in the tree is then $O(2^i n/2^i)$, which is $O(n)$. We know the tree has height $\lceil \log n \rceil$

Therefore, the time complexity is $O(n\log n)$

Set ADT

- A Set is a data structure modeled after the mathematical notation of a set. The fundamaental set operations are *union*, *intersection*, and *subtraction*.
- A brief aside on mathemeatical set notation:
  - $A \cup B = \{ x: x \in A \text{ or } x \in B \}$
  - $A \cap B = \{ x: x \in A \text{ and } x \in B \}$
  - $A - B = \{ x: x \in A \text{ and } x \notin B \}$
- The specific methods for a Set A include the following:
  - union(B):
    Set A equal to $A \cup B$.
  - intersect(B):
    Set A equal to $A \cap B$.
  - subtract(B):
    Set A equal to $A - B$.

Generic Merging

Algorithm genericMerge(A, B):
  Input: Sorted sequences A and B
  Output: Sorted sequence C

```
let A' be a copy of A { We won't destroy A and B}
let B' be a copy of B
while A' and B' are not empty do
  a←A'.first()
  b←B'.first()
  if a<b then
    aIsLess(a, C)
    A'.removeFirst()
  else if a=b then
    bothAreEqual(a, b, C)
    A'.removeFirst()
    B'.removeFirst()
  else
    bIsLess(b, C)
    B'.removeFirst()
while A' is not empty do
  a←A'.first()
  aIsLess(a, C)
  A'.removeFirst()
while B' is not empty do
  b←B'.first()
  bIsLess(b, C)
  B'.removeFirst()
```


Set Operations

- We can specialize the generic merge algorithm to perform set operations like union, intersection, and subtraction.
- The generic merge algorithm examines and compare the current elements of A and B.
- Based upon the outcome of the comparision, it determines if it should copy one or none of the elements a and b into C.
- This decision is based upon the particular operation we are performing, i.e. union, intersection or subtraction.
- For example, if our operation is union, we copy the smaller of a and b to C and if a=b then it copies either one (say a).
- We define our copy actions in aIsLess, bothAreEqual, and bIsLess.
- Let's see how this is done ...

Set Operations (cont.)

- For union

```
public class UnionMerger extends Merger {
  protected void aIsLess(Object a, Object b, Sequence C) {
    C.insertLast(a);
  }
  protected void bothAreEqual(Object a, Object b,
                              Sequence C) {
    C.insertLast(a);
  }
  protected void bIsLess(Object b, Sequence C) {
    C.insertLast(b);
  }
}
```

- For intersect

```
public class IntersectMerger extends Merger {
  protected void aIsLess(Object a, Object b, SequenceC) {
  }
  protected void bothAreEqual(Object a, Object b,
                              Sequence C) {
    C.insertLast(a);
  }
  protected void bIsLess(Object b, Sequence C) { }
}
```


Set Operations (cont.)

- For subtraction

```
public class SubtractMerger extends Merger {
  protected void aIsLess(Object a, Object b,
                         Sequence C) {
    C.insertLast(a);
  }

  protected void bothAreEqual(Object a, Object b,
                              Sequence C) {
  }

  protected void bIsLess(Object b, Sequence C) {
  }
}
```

Quicksort

Quick-Sort

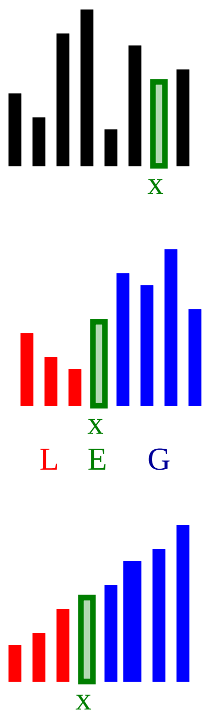
- To understand quick-sort, let's look at a high-level description of the algorithm
- 1) **Divide** : If the sequence *S* has 2 or more elements, select an element *x* from *S* to be your pivot. Any arbitrary element, like the last, will do. Remove all the elements of *S* and divide them into 3 sequences:
  - *L*, holds *S*'s elements less than *x*
  - *E*, holds *S*'s elements equal to *x*
  - *G*, holds *S*'s elements greater than *x*
- 2) **Recurse**: Recursively sort *L* and *G*
- 3) **Conquer**: Finally, to put elements back into *S* in order, first inserts the elements of *L*, then those of *E*, and those of *G*.
- Here are some pretty diagrams....

Idea of Quick Sort

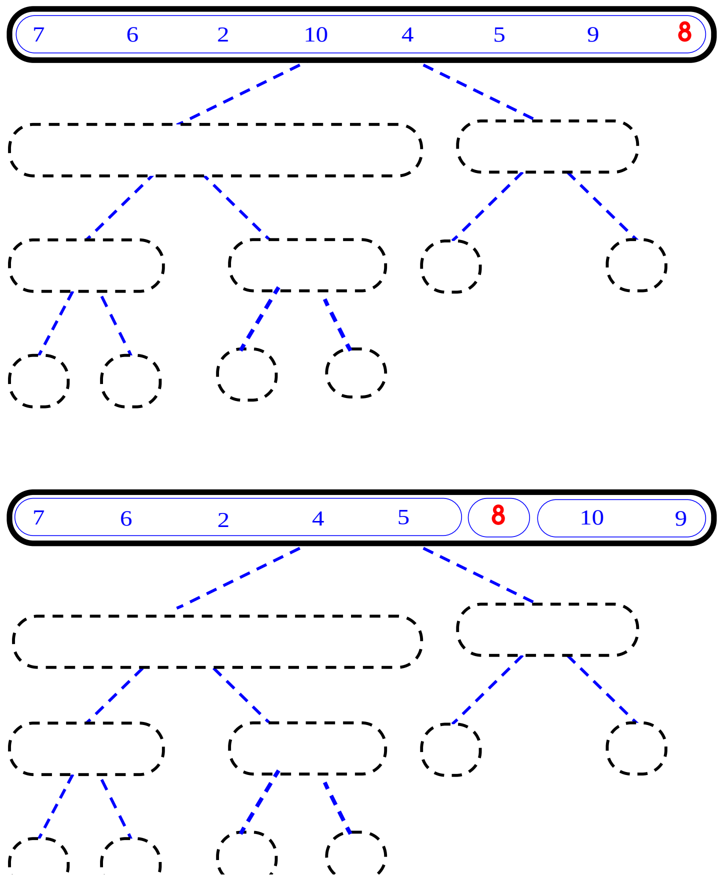
1. Select
pick *an* element

2.Devide
rearrange elements so that
- x goes to its final position E

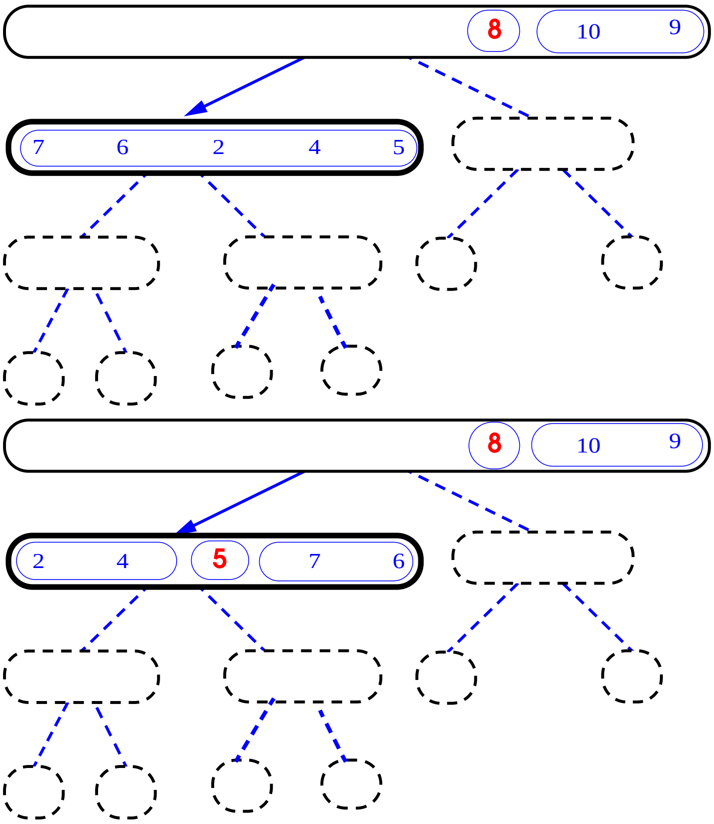
3. Recurse and Conquer
recursively sort



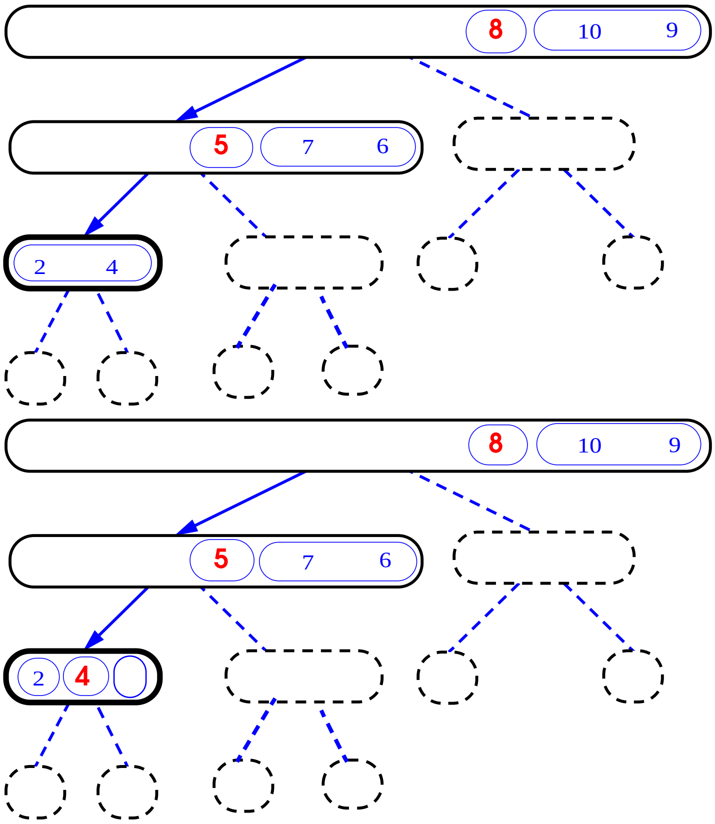
Quick-Sort Tree



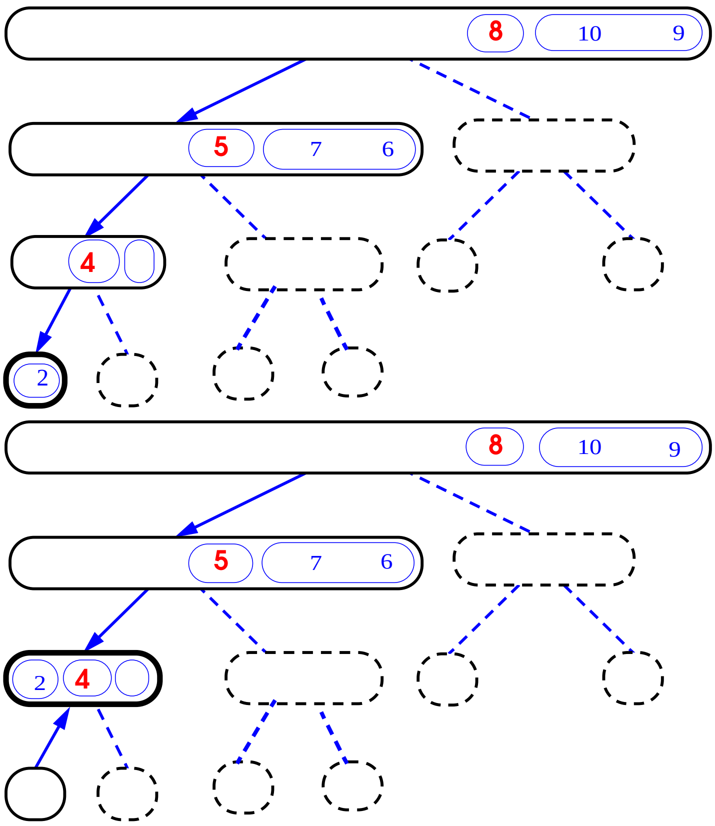
Quick-Sort Tree



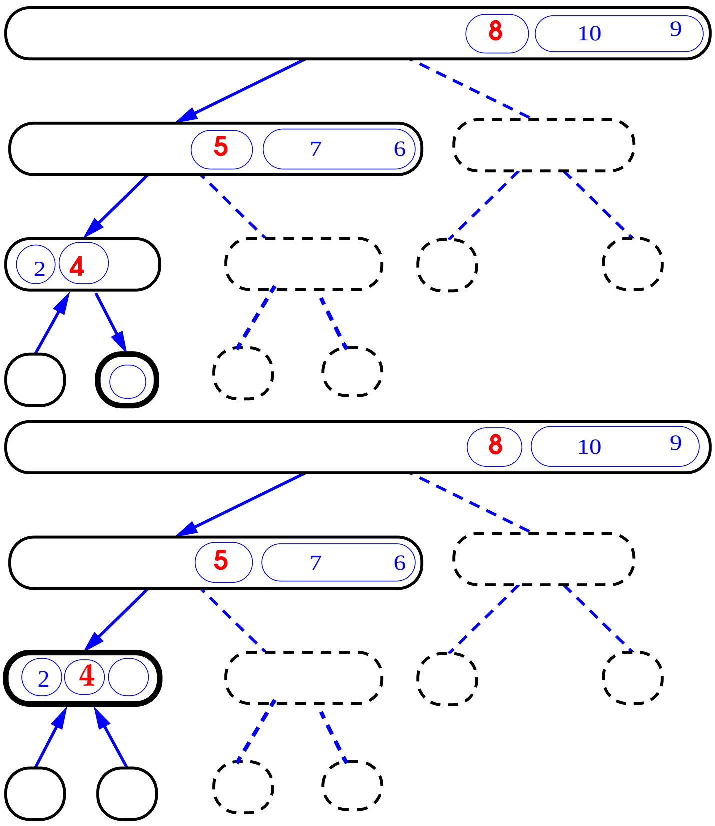
Quick-Sort Tree



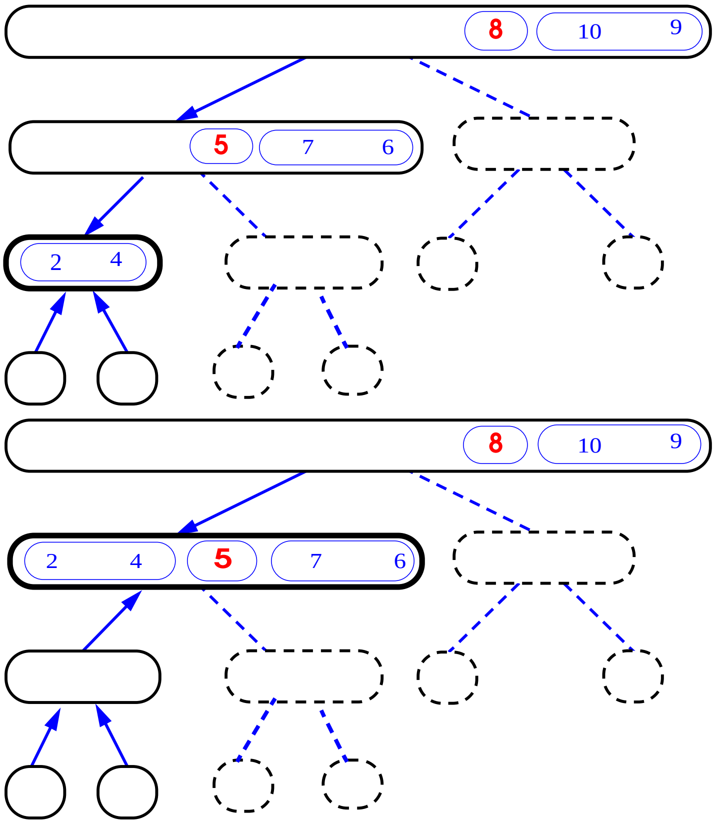
Quick-Sort Tree



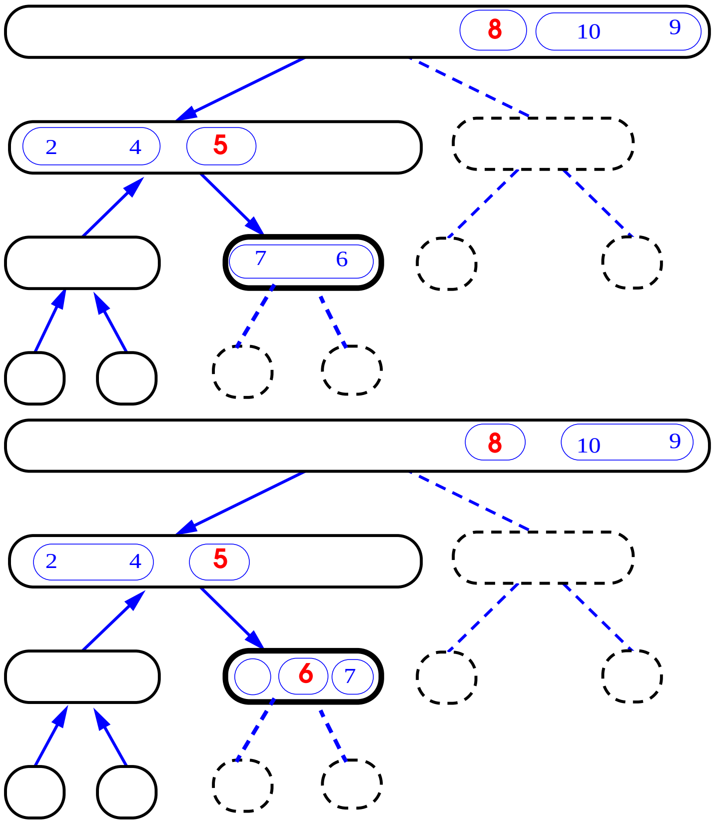
Quick-Sort Tree



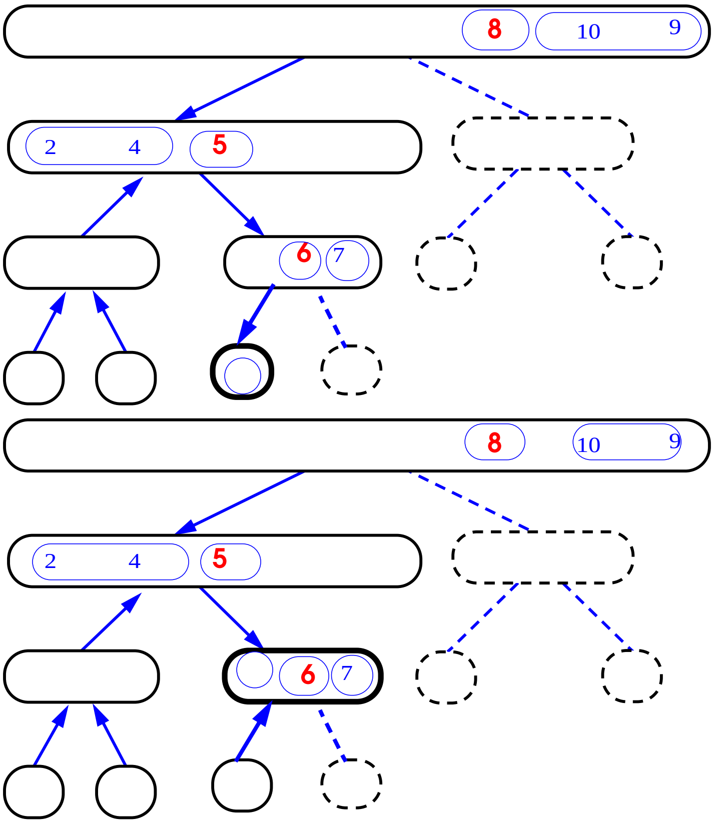
Quick-Sort Tree



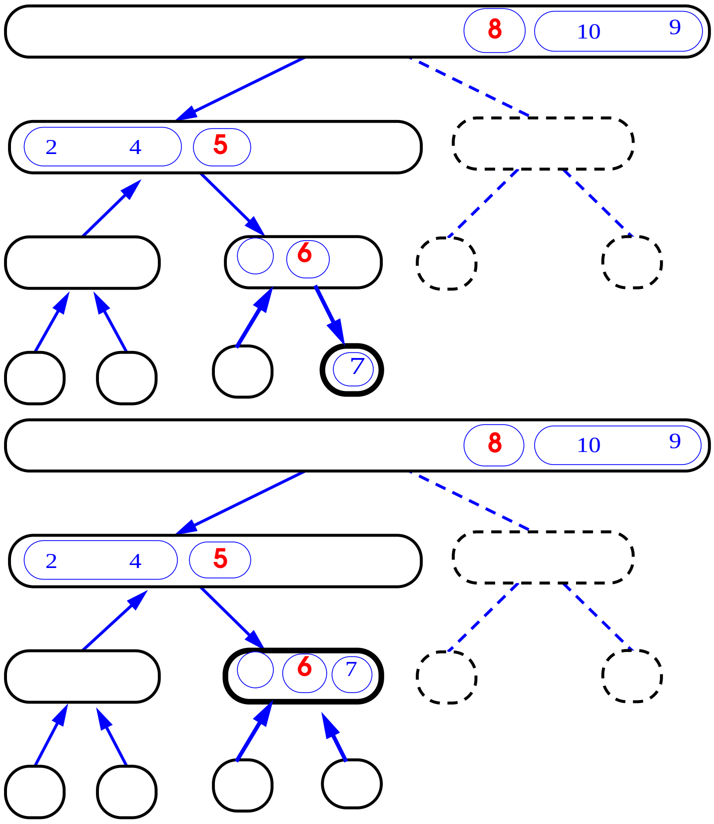
Quick-Sort Tree



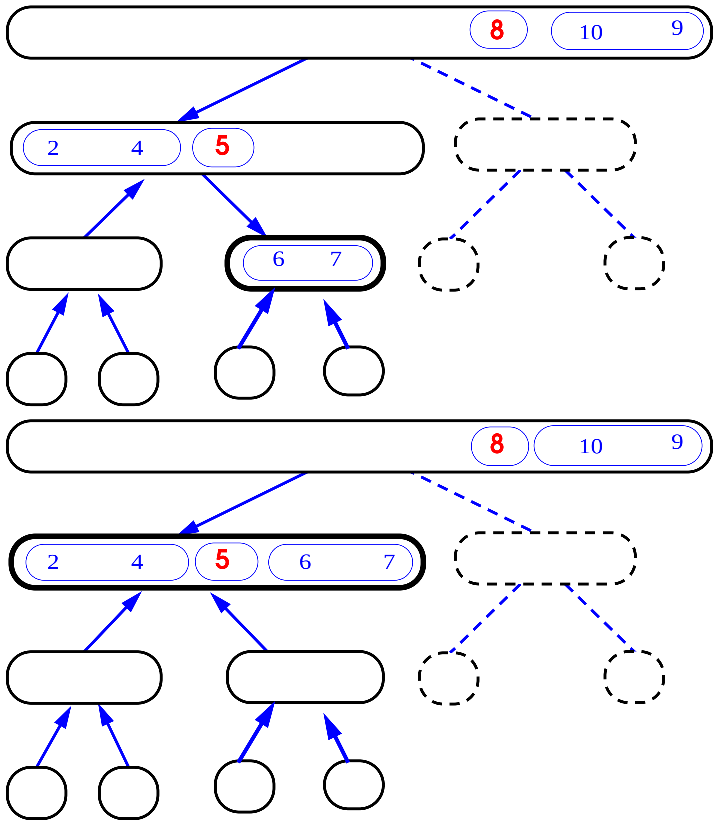
Quick-Sort Tree



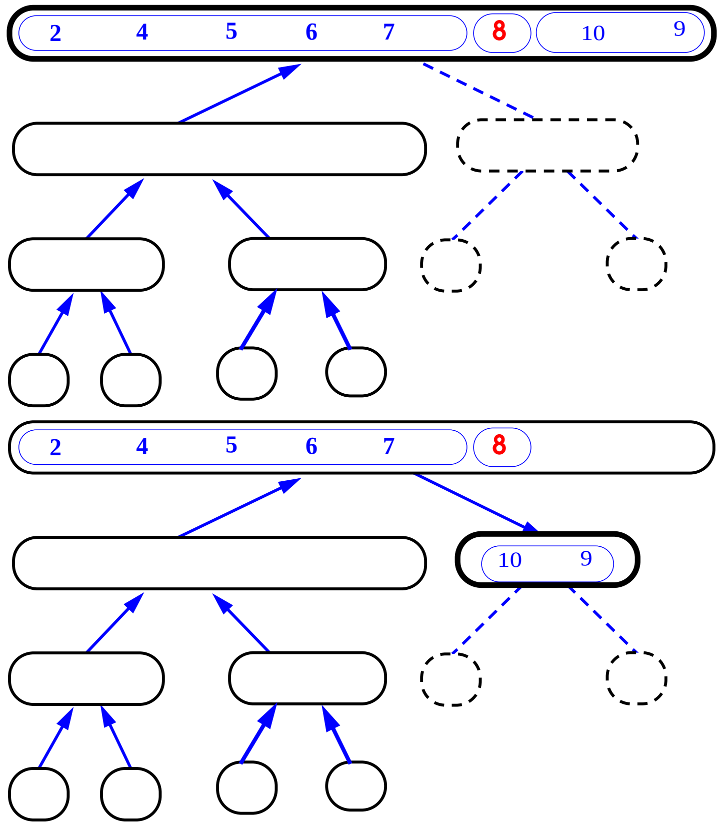
Quick-Sort Tree



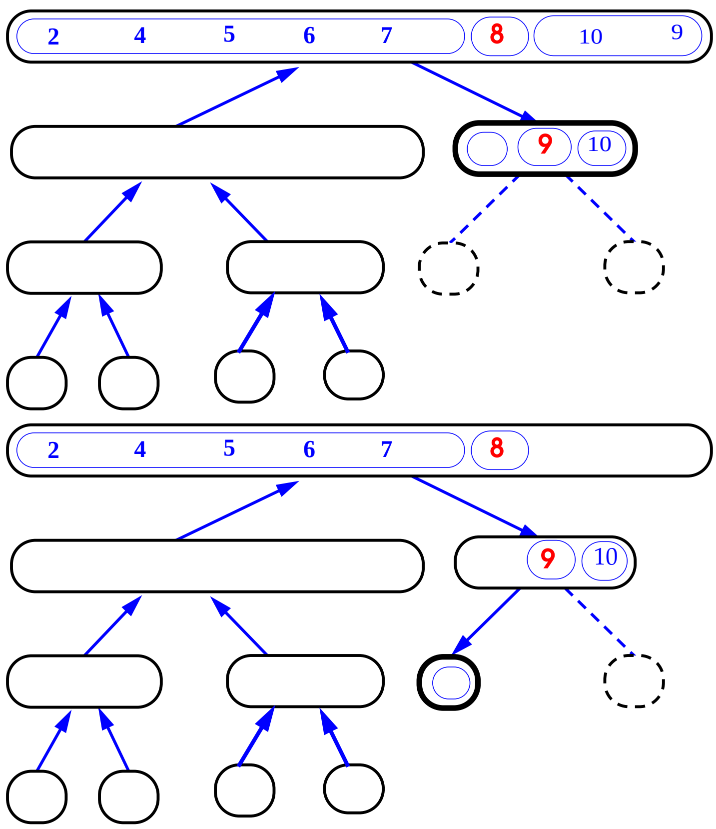
Quick-Sort Tree



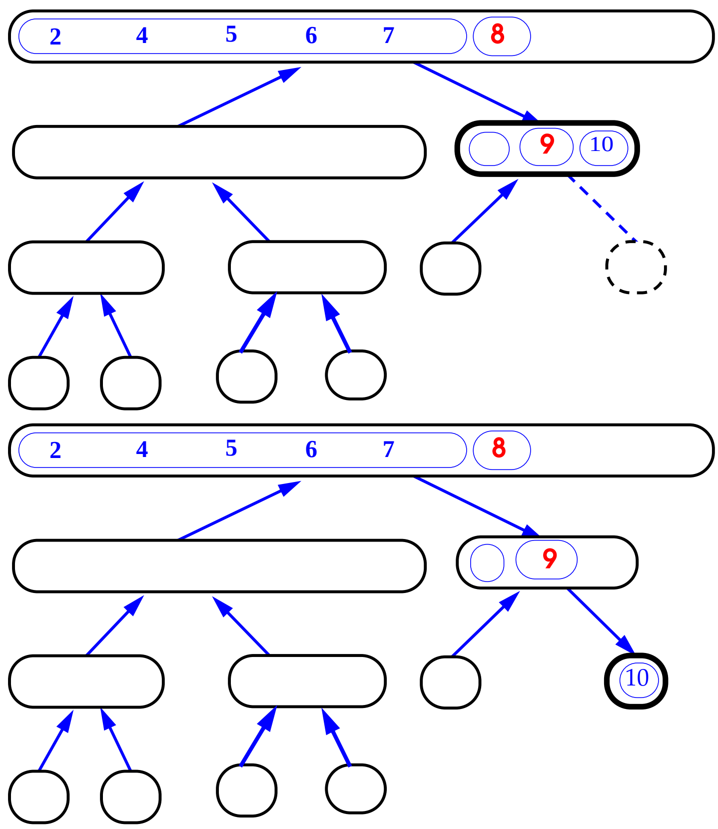
Quick-Sort Tree



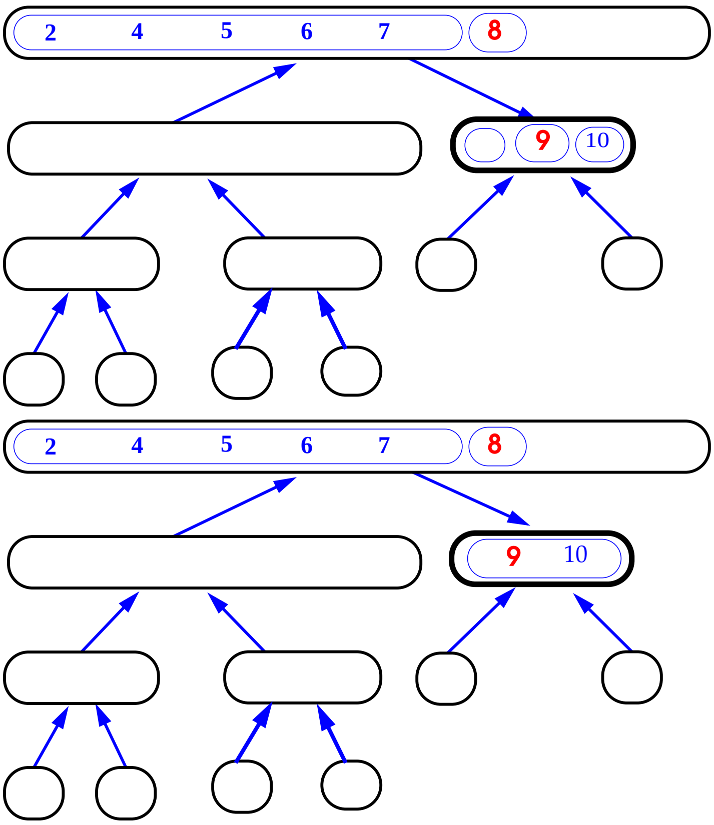
Quick-Sort Tree



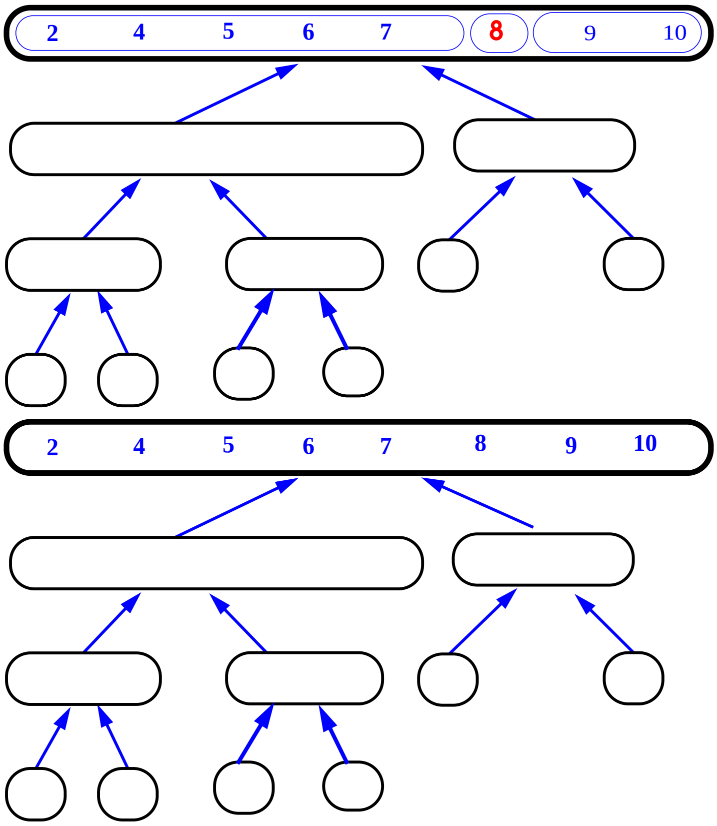
Quick-Sort Tree



Quick-Sort Tree

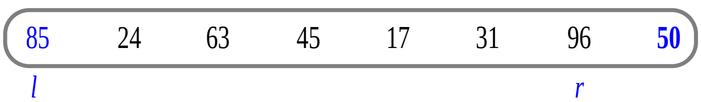


Quick-Sort Tree

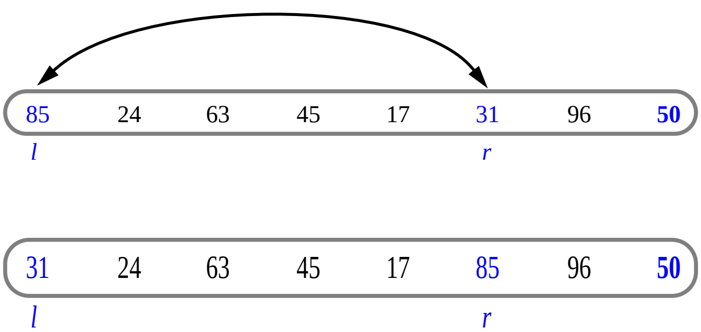


In-Place Quick-Sort

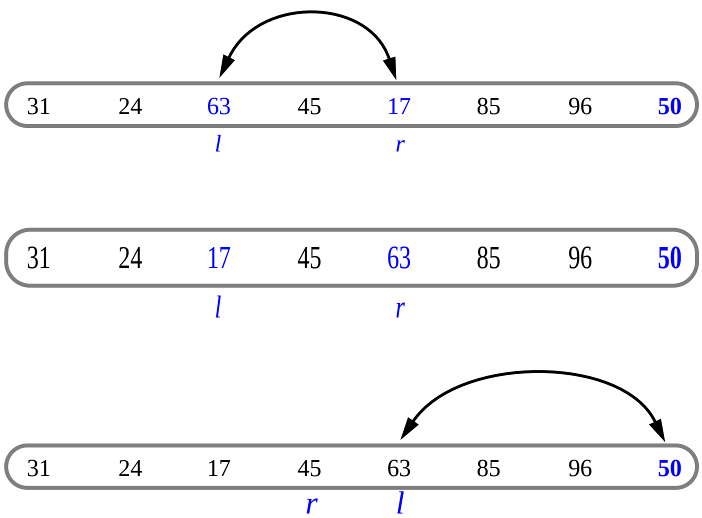
- **Divide step**: *l* scans the sequence from the left, and *r* from the right.



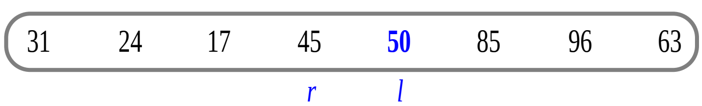
- A swap is performed when *l* is at an element larger than the pivot and *r* is at one smaller than the pivot.



In Place Quick Sort (contd.)



- A final swap with the pivot completes the divide step



In Place Quick Sort code

```
public class ArrayQuickSort implements SortObject {

  public void sort(Sequence S, Comparator c){
    quicksort(S, C, 0, S.size()-1);
  }

  private void quicksort (Sequence S, Comparator c,
                          int leftBound,
                          int rightBound) {
          // left and rightmost ranks of
             // sorting range
    if (S.size() < 2) return; //a sequence with 0 or
             // 1 elements is already sorted
    if (leftBound >= rightBound) return; //terminate
                      //recursion
    // pick the pivot as the current last
    // element in range
    Object pivot = S.atRank(rightBound).element();
   // indices used to scan the sorting range
    int leftIndex = leftBound; // will scan
                   // rightward
    int rightIndex = rightBound - 1; //will scan
                      // leftward
```


In Place Quick Sort code (contd.)

```
// outer loop
while (leftIndex <= rightIndex) {

  //scan rightward until an element larger than
  //the pivot is found or the indices cross
  while ((leftIndex <= rightIndex) &&
          (c.isLessThanOrEqualTo
          (S.atRank(leftIndex).element(),pivot))
    leftIndex++;

  //scan leftward until an element smaller than
  //the pivot is found or the indices cross
  while (rightIndex >= leftIndex) &&
          (c.isGreaterThanOrEqualTo
          (S.atRank(rightIndex).element(),pivot))
    rightIndex--;

  //if an element larger than the pivot and an
  //element smaller than the pivot have been
  //found, swap them
  if (leftIndex < rightIndex)
    S.swap(S.atRank(leftIndex),S.atRank(rightIndex));

} // the outer loop continues until
  // the indices cross. End of outer loop.
```


In Place Quick Sort code (contd.)

```
    //put the pivot in its place by swapping it
    //with the element at leftIndex
    S.swap(S.atRank(leftIndex),S.atRank(rightBound));

    // the pivot is now at leftIndex, so recur
    // on both sides
    quicksort (S, c, leftBound, leftIndex-1);
    quickSort (S, c, leftIndex+1, rightBound);
  } // end quicksort method
} // end ArrayQuickSort class
```


Analysis of Running Time

- Consider a quick-sort tree T:
  - Let $s_i(n)$ denote the sum of the input sizes of the nodes at depth i in T.
- We know that $s_0(n) = n$ since the root of T is associated with the entire input set.
- Also, $s_1(n) = n - 1$ since the pivot is not propagated.
- Thus: either $s_2(n) = n - 3$, or $n - 2$ (if one of the nodes has a zero input size).
- The worst case running time of a quick-sort is then:

$$O\left(\sum_{i=0}^{n-1} s_i(n)\right)$$

Which reduces to:

$$O\left(\sum_{i=0}^{n-1} (n-i)\right) = O\left(\sum_{i=1}^{n} i\right) = O(n^2)$$

- Thus quick-sort runs in time $O(n^2)$ in the worst case.

Analysis of Running Time (contd.)

- Now to look at the best case running time:
- We can see that quicksort behaves optimally if, whenever a sequence S is divided into subsequences L and G, they are of equal size.
- More precisely:
  - $s_0(n) = n$
  - $s_1(n) = n - 1$
  - $s_2(n) = n - (1 + 2) = n - 3$
  - $s_3(n) = n - (1 + 2 + 2^2) = n - 7$

    ...
  - $s_i(n) = n - (1 + 2 + 2^2 + ... + 2^i - 1) = n - 2^i + 1$

    ...
- This implies that T has height $\boldsymbol{O}(\log n)$
- Best Case Time Complexity: $\boldsymbol{O}(n\log n)$

Randomized Quick-Sort

- Select the pivot as a ***random*** element of the sequence
- The expected running time of randomized quick-sort on a sequence of size n is $O(n \log n)$
- The time spent at a level of the quick-sort tree is $O(n)$
- We show that the ***expected height*** of the quick-sort tree is $O(\log n)$
- good vs. bad pivots

  - ***good***: $1/4 \leq n_L/n \leq 3/4$
  - ***bad***: $n_L/n < 1/4$ **or** $n_L/n > 3/4$
- the probability of a good pivot is 1/2, thus we expect $k/2$ good pivots out of k pivots
- after a good pivot the size of each child sequence is at most 3/4 the size of the parent sequence
- After h pivots, we expect $(3/4)^{h/2}\, n$ elements
- the expected height h of the quick-sort tree is at most:

$$2 \log_{4/3} n$$