Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Advance Sorting Algorithms

- We have seen sorting techniques in lecture 04
  - Bubble Sort - $O(n^2)$
  - Selection Sort - $O(n^2)$
  - Insertion Sort - $O(n^2)$ [best case $O(n)$]

- Now we will see advanced sorting techniques
  - Heap Sort
  - Merge Sort
  - Quick Sort

Heap Sort

- The efficiency of the heap data structure lends itself to a surprisingly simple and very efficient sorting algorithm called ***heapsort***
- We know that realizing a priority queue with a heap has the advantage that *all the methods in the priority queue ADT* run in ***logarithmic time or better***
- Therefore, it is suitable for applications where fast running time
- The basic idea is to
  - Insert all the unordered items into a heap using the normal insert() routine
  - Repeated application of the remove() routine will then remove the items in sorted order

Proposition 9.4: *The heap-sort algorithm sorts a sequence S of n elements in $O(n \log n)$ time, assuming two elements of S can be compared in $O(1)$ time.*

Heap Sort

- The algorithm for sorting a sequence ***S*** with a priority queue ***P*** is quite simple and consists of the following two phases:
  1. In the first phase, we insert the elements of ***S*** as keys into an initially empty priority queue ***P*** by means of a series of ***n*** insert operations, one for each element
  2. In the second phase, we extract the elements from ***P*** in nondecreasing order by means of a series of ***n*** **removeMin** operations, putting them back into ***S*** in that order

Implementing Heap Sort

```
/** Sorts sequence S, using initially empty priority queue P to produce the order. */
public static <E> void pqSort(PositionalList<E> S, PriorityQueue<E,?> P) {
  int n = S.size();
  for (int j=0; j < n; j++) {
    E element = S.remove(S.first());
    P.insert(element, null);          // element is key; null value
  }
  for (int j=0; j < n; j++) {
    E element = P.removeMin().getKey();
    S.addLast(element);               // the smallest key in P is next placed in S
  }
}
```

Divide-and-Conquer

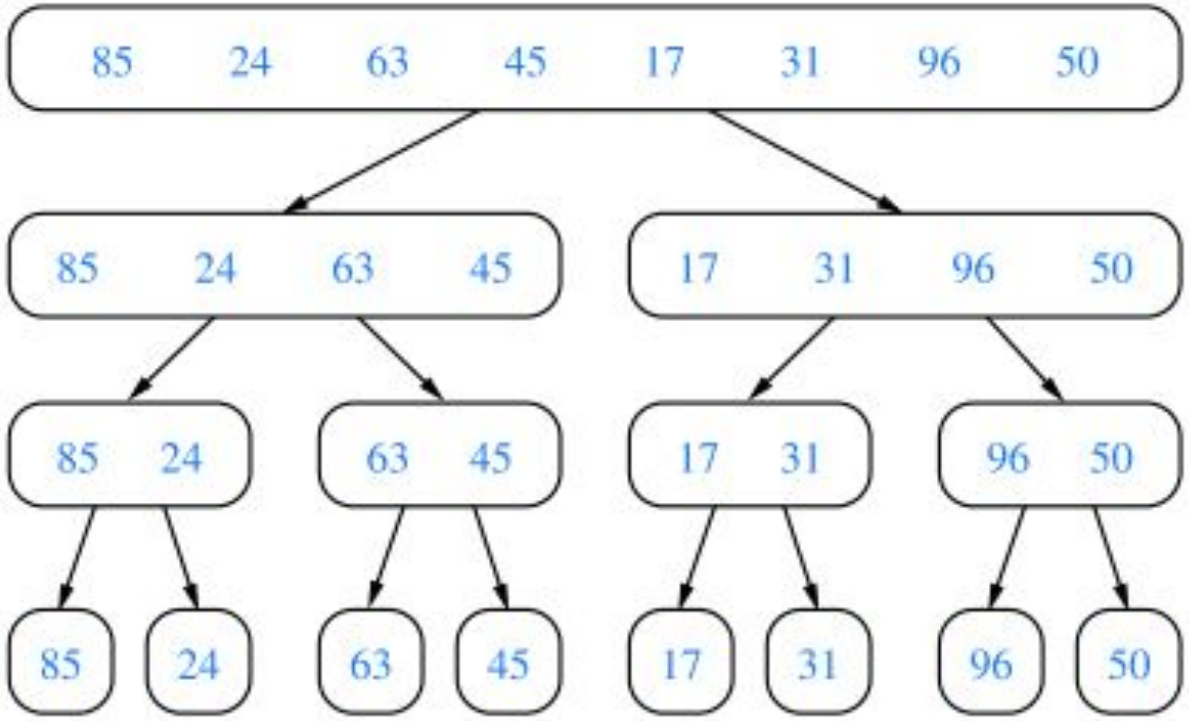
- Merge-sort and quick-sort, use recursion in an algorithmic design pattern called ***divide-and-conquer***
- The divide-and-conquer pattern consists of the following three steps:
  1. **Divide:** If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution so obtained. Otherwise, divide the input data into two or more disjoint subsets.
  2. **Conquer:** Recursively solve the subproblems associated with the subsets.
  3. **Combine:** Take the solutions to the subproblems and merge them into a solution to the original problem.

Merge Sort

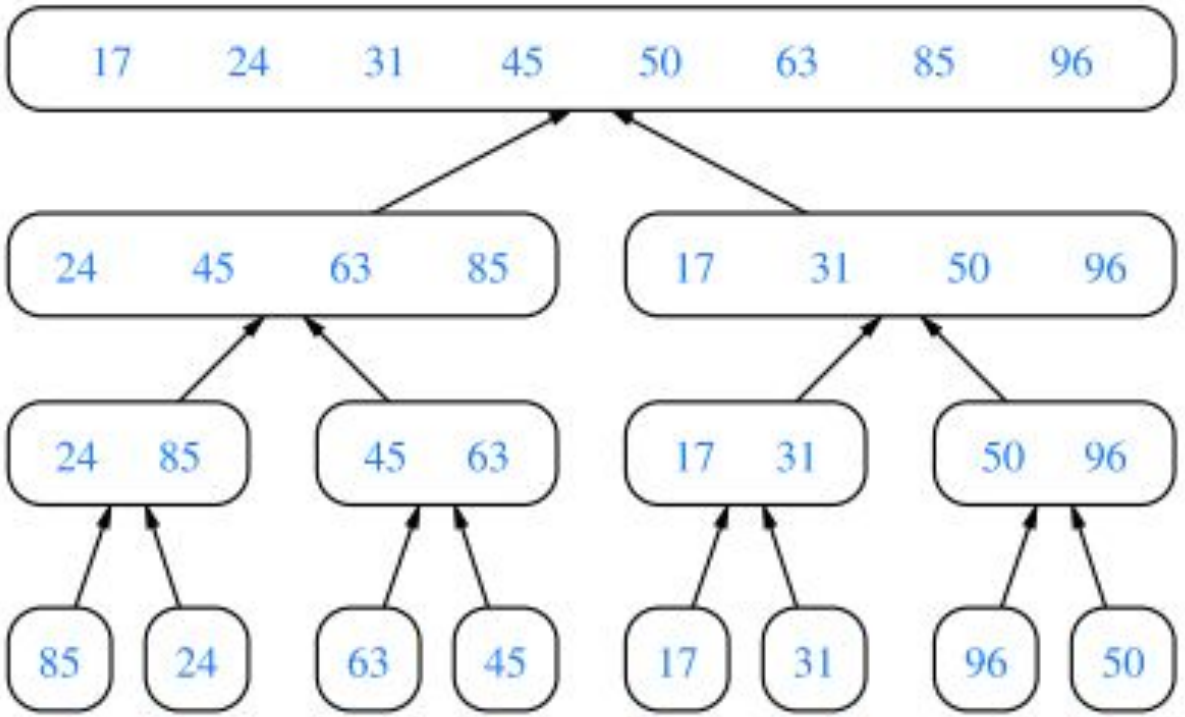
- To sort a sequence ***S*** with ***n*** elements using the three divide-and-conquer steps, the merge-sort algorithm proceeds as follows:
  - **Divide:** If ***S*** has zero or one element, return ***S*** immediately; it is already sorted. Otherwise, remove all the elements from ***S*** and put them into two sequences, ***S1*** and ***S2***, each containing about half of the elements of ***S***; that is, ***S1*** contains the first $\lfloor n/2 \rfloor$ elements of ***S***, and ***S2*** contains the remaining $\lceil n/2 \rceil$ elements.
  - **Conquer:** Recursively sort sequences ***S1*** and ***S2***.
  - **Combine:** Put the elements back into ***S*** by merging the sorted sequences ***S1*** and ***S2*** into a sorted sequence.

Merge Sort

- We can visualize an execution of the merge-sort algorithm by means of a binary tree T, called the merge-sort tree



(a)



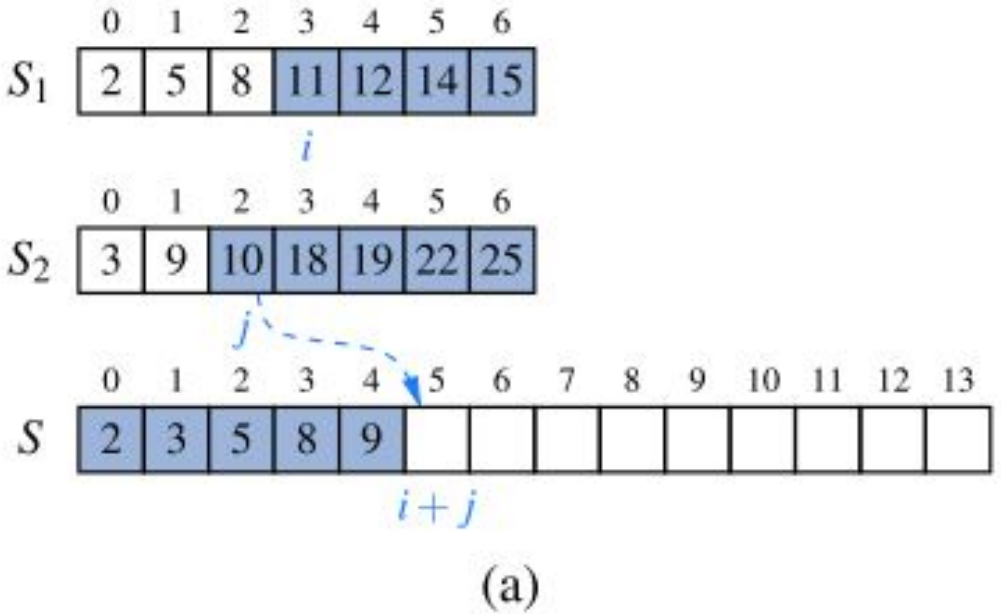
(b)

Array-Based Implementation of Merge-Sort

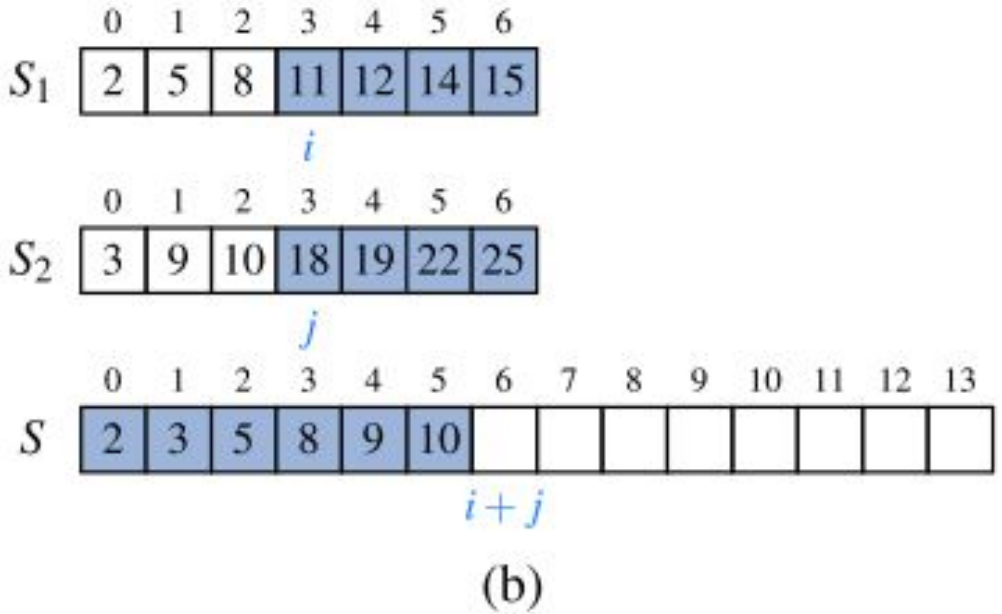
```
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
  int n = S.length;
  if (n < 2) return;                          // array is trivially sorted
  // divide
  int mid = n/2;
  K[ ] S1 = Arrays.copyOfRange(S, 0, mid);    // copy of first half
  K[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

Array-Based Implementation of Merge-Sort

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



(a)



(b)

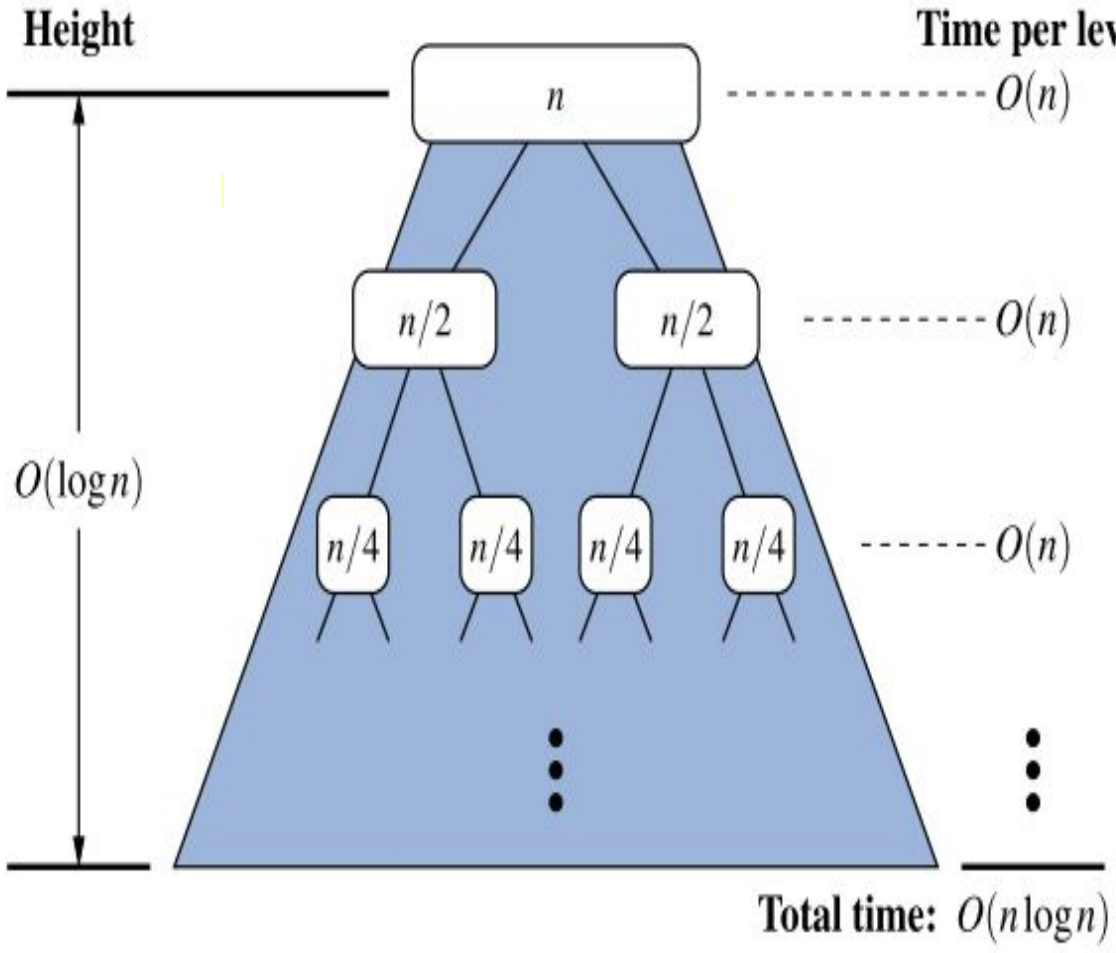
The Running Time of Merge-Sort

- The running time of the merge algorithm
  - Let n1 and n2 be the number of elements of S1 and S2, respectively
  - The operations performed inside each pass of the while loop take O(1) time
  - The number of iterations of the loop is n1 +n2
  - Thus, the running time of algorithm merge is O(n1 +n2)
- The running time of the entire merge-sort algorithm
  - Let an input sequence S of n elements (for simplicity consider n is a power of 2)
  - In the case of our mergeSort method, we account for the time to divide the sequence into two subsequences, and the call to merge to combine the two sorted sequences

Proposition 12.2: *Algorithm merge-sort sorts a sequence S of size n in $O(n \log n)$ time, assuming two elements of S can be compared in $O(1)$ time.*

The Running Time of Merge-Sort

- A merge-sort tree T can guide our analysis
- Consider a recursive call associated with a node v of the merge-sort tree T
- The divide step at node v is straight forward; this step runs in time proportional to the size of the sequence for v, based on the use of slicing to create copies of the two list halves
- The merging step also takes time that is linear in the size of the merged sequence
- Therefore, the time spent at node v is $O(n/2^i)$, if we let i denote the depth of node v (the size of the sequence handled by the recursive call associated with v is equal to $n/2^i$)
- The running time of merge-sort is equal to the sum of the times spent at the nodes of T
- The overall time spent at all the nodes of T at depth i is $O(2^i \cdot n/2^i)$, which is O(n)
- The height of T is $\lceil \log n \rceil$. Thus, since the time spent at each of the $\lceil \log n \rceil + 1$ levels of T is O(n)



Proposition 12.2: *Algorithm merge-sort sorts a sequence S of size n in $O(n \log n)$ time, assuming two elements of S can be compared in $O(1)$ time.*