7. Cole, R., and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3(3):329-346, 1988.
8. Cole, R., and U. Vishkin. Approximate parallel scheduling, Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Computing*, 17(1):128-142, 1988.
9. Cole, R., and U. Vishkin. Faster optimal prefix sums and list ranking. *Information and Computation*, 81(3):344-352, 1989.
10. Eppstein, S., and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.*, 3:233-283, Annual Reviews Inc., Palo Alto, CA, 1988.
11. Gabow, H. N., J. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings Sixteenth Annual ACM Symposium on Theory of Computing*, Washington, DC, 1984, pages 135-143, ACM Press, New York.
12. Gibbons, A., and W. Rytter. An optimal parallel algorithm for dynamic evaluation and its applications. In *Proceedings Sixth Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 241*, New Delhi, India, 1986, pages 453-469. Springer-Verlag, New York.
13. He, X., and Y. Yesha. Binary tree algebraic computations and parallel algorithms for simple graphs. *Journal of Algorithms*, 9(1):92-113, 1988.
14. Kedem, Z. M., and K. V. Palem. Optimal parallel algorithms for forest and term matching. *Theoretical Computer Science* (in press).
15. Kosaraju, S. R., and A. Delcher. Optimal parallel evaluation of tree-structured computations by raking. In *Proceedings of AWOC88*, Corfu, Greece, 1988, pages 101-110. Springer-Verlag, New York.
16. Kruskal, C., L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10):965-968, 1985.
17. Miller, G. L., V. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Computing*, 17(4):687-695, 1988.
18. Miller, G. L., and J. H. Reif. Parallel tree contraction and its applications. In *Proceedings Twenty-Sixth Annual IEEE Symposium on Foundations of Computer Science*, Portland, OR, 1985, pages 478-489. IEEE Press, Piscataway, NJ.
19. Schieber, B., and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Computing*, 17(6):1253-1262, 1988.
20. Stout, Q. F. Topological matching. In *Proceedings Fifteenth Annual ACM Symposium on Theory of Computing*, Boston, MA, 1983, pages 24-31. ACM Press, New York.
21. Tarjan, R. E., and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. *SIAM J. Computing*, 14(4):862-874, 1985.
22. Valiant, L. G., S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. Computing*, 12(4):641-644, 1983.
23. Vishkin, U. Randomized speed-ups in parallel computation. In *Proceedings Sixteenth ACM Symposium on Theory of Computing*, Washington, D.C., 1984, pages 230-239. ACM Press, New York.
24. Wagner, W., and Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem. In *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, 1986, pages 924-930.
25. Wyllie, J. C. *The Complexity of Parallel Computations.* PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, 1979.

4 Searching, Merging, and Sorting

The task of processing a table consisting of records whose keys come from a linearly ordered set arises in many practical situations. Examples of specific processing tasks include searching for a key in the table and sorting the keys of the table. Such tasks typically involve repeated applications of comparison and data-movement operations. We assume that each key is an atomic unit that cannot be manipulated as an integer or as a string of bits. This class of problems has a rich mathematical theory that still attracts a substantial research interest.

In this chapter, optimal algorithms for basic problems in searching, merging, and sorting are presented. Several algorithmic techniques are explored, including *parallel searching* (Section 4.1), *partitioning* (Section 4.2), *pipelined* or *cascading divide-and-conquer* (Section 4.3), and *bitonic sorting* (Section 4.4). These techniques are used to develop efficient parallel algorithms for searching, merging, selection, and sorting. In addition, we introduce for the first time in this book lower-bound proofs on the parallel complexity of certain problems (Section 4.6). These proofs are especially tailored for the class of comparison problems considered in this chapter.

4.1 Searching

Let $X = (x_1, x_2, \ldots, x_n)$ be n distinct elements drawn from a linearly ordered set $(S, \leq)$ such that $x_1 < x_2 < \cdots < x_n$. Given an element $y \in S$, we are interested in solving the following **search problem:** identify the index i for which $x_i \leq y < x_{i+1}$, where $x_0 = -\infty$ and $x_{n+1} = +\infty$, and $-\infty$ and $+\infty$ are two elements that are added to S and that satisfy $-\infty < x < +\infty$, for all $x \in S$.

The well-known binary search method solves this problem in $O(\log n)$ optimal sequential time. This method consists of comparing y with the middle element of X. Based on the outcome of this comparison, the search either is terminated with a success or it can be restricted to the upper or the lower half of X. The process is repeated until either an element x_i is encountered such that $y = x_i$ or the size of the subarray under consideration is equal to 1. Hence, in either case, the solution is determined.

A natural extension of the binary search method to parallel processing is **parallel searching,** in which we compare y concurrently with several elements of X, say p elements of X, which split X into $p + 1$ segments of almost equal lengths. The outcome of this parallel comparison step is either to identify an element x_i that is equal to y, or to restrict the search to one of the $p + 1$ segments. We repeat this process until either an element x_i is encountered such that $y = x_i$ or the number t of elements in the subarray under consideration is no more than p. In the former case, the solution is found; in the latter case, the solution can be determined by t comparisons being made in parallel.

In this section, we shall not use the WT framework to present and analyze our parallel-search algorithm. Instead, the algorithm is couched within the alternative time-processors framework. The parameter p that we introduced can be viewed as the number of processors available on our PRAM model.

The parallel-search algorithm for processor P_j is given next, where $1 \leq j \leq p$. Processor P_1 is responsible for various initializations and for taking care of the boundary cases. The variable c_j is used to indicate the outcome of the comparison performed in a given step; the indices l and r are pointers to the (left and right) boundaries of the current subarray under consideration.

ALGORITHM 4.1

(Parallel Search for Processor P_j)

Input: *(1) An array $X = (x_1, x_2, \ldots, x_n)$ such that $x_1 < x_2 < \ldots < x_n$; (2) an element y; (3) the number p of processors, where $p \leq n$; (4) the processor number j, where $1 \leq j \leq p$.*

Output: *An index i such that $x_i \leq y < x_{i+1}$.*

```
begin
  1. if $(j = 1)$ then do
        1.1. Set $l := 0; r := n + 1; x_0 := -\infty; x_{n+1} := +\infty$
        1.2. Set $c_0 := 0; c_{p+1} := 1$
  2. while $(r - l > p)$ do
        2.1. if $(j = 1)$ then {set $q_0 := l; q_{p+1} := r$}
        2.2. Set $q_j := l + j \lfloor \frac{r-l}{p+1} \rfloor$
        2.3. if $(y = x_{q_j})$ then {return$(q_j)$; exit}
                  else {set $c_j := 0$ if $y > x_{q_j}$ and $c_j := 1$
                  if $y < x_{q_j}$}
        2.4. if $(c_j < c_{j+1})$ then {set $l := q_j; r := q_{j+1}$}
        2.5. if $(j = 1$ and $c_0 < c_1)$ then {set $l := q_0; r := q_1$}
  3. if $(j \leq r - l)$ then do
        3.1. Case statement:
                $y = x_{l+j}$ : {return $(l + j)$; exit}
                $y > x_{l+j}$ : set $c_j := 0$
                $y < x_{l+j}$ : set $c_j := 1$
        3.2. if $(c_{j-1} < c_j)$ then return $(l + j - 1)$
end
```

EXAMPLE 4.1:

Let X be the array $X = (2, 4, 6, \ldots, 30)$ consisting of all even integers between 2 and 30, and let $y = 19$. Suppose that $p = 2$. After the execution of step 1, P_1 will set $l = 0, r = 16, c_0 = 0, c_3 = 1, x_0 = -\infty$, and $x_{16} = +\infty$. The **while** loop runs for three iterations; the effect of each such iteration is shown in the following table:

iteration	1	2	3
q_0	0	5	7
q_1	5	6	8
q_2	10	7	9
q_3	16	10	10
c_0	0	0	0
c_1	0	0	0
c_2	1	0	0
c_3	1	1	1
l	5	7	9
r	10	10	10

During the execution of step 3.1, P_1 sets $c_1 = 1$. Hence, at step 3.2, P_1 verifies that $c_0 < c_1$ and returns the index 9. □

We are ready for the following theorem.

Theorem 4.1: *Given an array $X = (x_1, x_2, \ldots, x_n)$ with $x_1 < x_2 < \ldots < x_n$ and an element y, Algorithm 4.1 determines the index i such that $x_i \le y < x_{i+1}$. The parallel time required is $O\left(\frac{\log(n+1)}{\log(p+1)}\right)$, where p is the number of processors used.*

Proof: The correctness proof is simple and is left to the reader (see Exercise 4.2).

As for the number of steps, notice that, after the ith iteration of the **while** loop, the size of the subarray to be searched is reduced from $s_i = r - l$ to $s_{i+1} \le \frac{r-l}{p+1} + p = \frac{s_i}{p+1} + p$, which is the maximum possible length of the $(p+1)$st segment. Setting $s_0 = n + 1$, it is straightforward to check that $s_i \le \frac{n+1}{(p+1)^i} + p + 1$ satisfies this recurrence. Therefore, the number of iterations needed is $O\left(\frac{\log(n+1)}{\log(p+1)}\right)$, and each iteration takes $O(1)$ time. Hence, the **while** loop requires $O(\log(n+1)/\log(p+1))$ time. Since step 3 takes $O(1)$ time, the time bound stated in the theorem follows. □

PRAM Model: Algorithm 4.1 can be implemented on the CREW PRAM with p processors. A concurrent-read capability is needed, since l, r and the search key y need to be accessed by all the processors. Surprisingly, we can show that $\Omega(\log n - \log p)$ parallel steps are required on the EREW PRAM model even if the search key is made available to all the processors (see Section 10.3). Hence, no significant speedup can be achieved on the EREW PRAM model for any number p of processors such that $p \le n^c$, where c is any constant satisfying $0 < c < 1$. □

Remark 4.1: According to our notion of optimality, a parallel-search algorithm is optimal if its total number of operations is asymptotically equal to the sequential complexity $\Theta(\log n)$ of the search problem. Hence, Theorem 4.1 shows that optimality on the CREW PRAM is achieved only when p is constant and hence $T(n) = \Theta(\log n)$. However, we show in Section 4.6 that the performance of Algorithm 4.1 cannot be improved. □

4.2 Merging

We have already considered the problem of merging two sorted sequences in Section 2.4. Based on the **partitioning strategy**, an $O(\log n)$-time parallel algorithm was presented there. The corresponding total work is $O(n)$; hence, the algorithm is optimal.

In this section, we refine the previous partitioning strategy to obtain an optimal $O(\log\log n)$ time algorithm that runs on the CREW PRAM. This problem is one of the few known to have such a fast algorithm on the CREW PRAM. Compare the merging problem with the simple problem of computing the maximum of n elements, which requires $\Omega(\log n)$ parallel steps on the CREW PRAM, regardless of the number of the processors available. Therefore, the existence of an optimal $O(\log\log n)$ time merging algorithm that runs on the CREW PRAM is perhaps surprising.

We start by recalling a few definitions presented in Section 2.4. The **rank** of an element x in a given sequence X, denoted by $rank(x : X)$, is the number of elements of X that are less than or equal to x. If X is sorted, it is useful to define the **predecessor** of an arbitrary element x to be the element x_r of X such that $r = rank(x : X)$. **Ranking** a sequence $Y = (y_1, y_2, \ldots, y_m)$ in X amounts to computing the integer array $rank(Y : X) = (r_1, r_2, \ldots, r_m)$, where $r_i = rank(y_i : X)$.

4.2.1 RANKING A SHORT SEQUENCE IN A SORTED SEQUENCE

Let X be a sorted sequence with n distinct elements, and let Y be an arbitrary sequence of size m such that $m = O(n^s)$, where s is a constant that satisfies $0 < s < 1$. The parallel-search algorithm (Algorithm 4.1) can be used to rank each element of Y in X separately. We set the number p of processors of this algorithm to $p = \lfloor n/m \rfloor = \Omega(n^{1-s})$. Then, each element of Y can be ranked in X in $O(\log(n+1)/\log(p+1)) = O(1)$ time. The total number of operations used to rank each such element is $O(n/m)$, since $p = \lfloor n/m \rfloor$, and the running time is $O(1)$. We therefore have the following lemma.

Lemma 4.1: *Let Y be an arbitrary sequence with m elements, and let X be a sorted sequence with n distinct elements such that $m = O(n^s)$ for some constant $0 < s < 1$. Then, all the elements of Y can be ranked in X in $O(1)$ time using a total of $O(n)$ operations.* □

PRAM Model: The parallel-search algorithm (Algorithm 4.1) requires a concurrent-read capability. Hence, the procedure referred to in Lemma 4.1 requires the CREW PRAM model. □

4.2.2 A FAST MERGING ALGORITHM

Consider the problem of determining $rank(B : A)$ for sorted sequences A and B of lengths n and m, respectively. Assume that all the elements of A and B are distinct and hence that no element of A appears in B.

As in Section 2.4, we use the partitioning strategy to merge the two sequences A and B. We rank a set of $\sqrt{m}$ elements of B that partition B into blocks of almost equal lengths in the sorted sequence A. The computed ranks of the chosen elements will induce a partition on A into blocks such that each block of A has to fit between two of the chosen elements of B. Hence, the overall problem is now reduced to ranking the elements of each block of B (excluding already ranked elements) into a corresponding block of A.

The algorithm is given next. Figure 4.1 illustrates the partitions introduced in the algorithm.

ALGORITHM 4.2

(Ranking a Sorted Sequence in Another Sorted Sequence)

Input: *Two arrays* $A = (a_1, \ldots, a_n)$ *and* $B = (b_1, \ldots, b_m)$ *in increasing order. Assume that* $\sqrt{m}$ *is an integer; otherwise, replace* $\sqrt{m}$ *whenever it occurs by* $\lfloor \sqrt{m} \rfloor$.

Output: *The array* $rank(B : A)$.

```
begin
  1. If $m < 4$, then rank the elements of $B$ by applying the
     parallel-search algorithm (Algorithm 4.1) with $p = n$, and exit.
  2. Concurrently rank the elements $b_{\sqrt{m}}, b_{2\sqrt{m}}, \ldots, b_{i\sqrt{m}}, \ldots, b_m$ in
     $A$ by using the parallel-search algorithm (Algorithm 4.1) with $p =$
     $\sqrt{n}$. Let $rank(b_{i\sqrt{m}} : A) = j(i)$, for $1 \le i \le \sqrt{m}$. Set $j(0) = 0$.
  3. For $0 \le i \le \sqrt{m} - 1$, let $B_i = (b_{i\sqrt{m}+1}, \ldots, b_{(i+1)\sqrt{m}-1})$
     and let $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$; if $j(i) = j(i+1)$, then set
     $rank(B_i : A_i) = (0, \ldots, 0)$, else recursively compute $rank(B_i : A_i)$.
  4. Let $1 \le k \le m$ be an arbitrary index that is not a multiple of
     $\sqrt{m}$, and let $i = \lfloor \frac{k}{\sqrt{m}} \rfloor$. Then $rank(b_k : A) = j(i) + rank(b_k : A_i)$.
end
```

EXAMPLE 4.2:

Let $A = (-5, 0, 3, 4, 17, 18, 24, 28)$ and $B = (1, 2, 15, 21)$. At the termination of step 2, we obtain $j(0) = 0$, $j(1) = 2$ and $j(2) = 6$, where 2 and 6 are the ranks of $b_2 = 2$ and $b_4 = 21$, respectively. During the execution of step 3, we introduce $B_0 = (1)$ and $A_0 = (-5, 0)$, $B_1 = (15)$ and $A_1 = (3, 4, 17, 18)$. In this case, $rank(1 : A_0) = 2$ and $rank(15 : A_1) = 2$. At step 4, we adjust the ranks of $b_1 = 1$ and $b_3 = 15$ as follows: $rank(1 : A) = j(0) + rank(1 : A_0) = 2$ and $rank(15 : A) = j(1) + rank(15, A_1) = 4$. Therefore, we get the array $rank(B : A) = (2, 2, 4, 6)$. □

Lemma 4.2: *Let A and B be two sorted sequences such that $|A| = n$ and $|B| = m$. Then, Algorithm 4.2 computes the array $rank(B : A)$ in $O(\log \log m)$ time using $O((n + m) \log \log m)$ operations.*

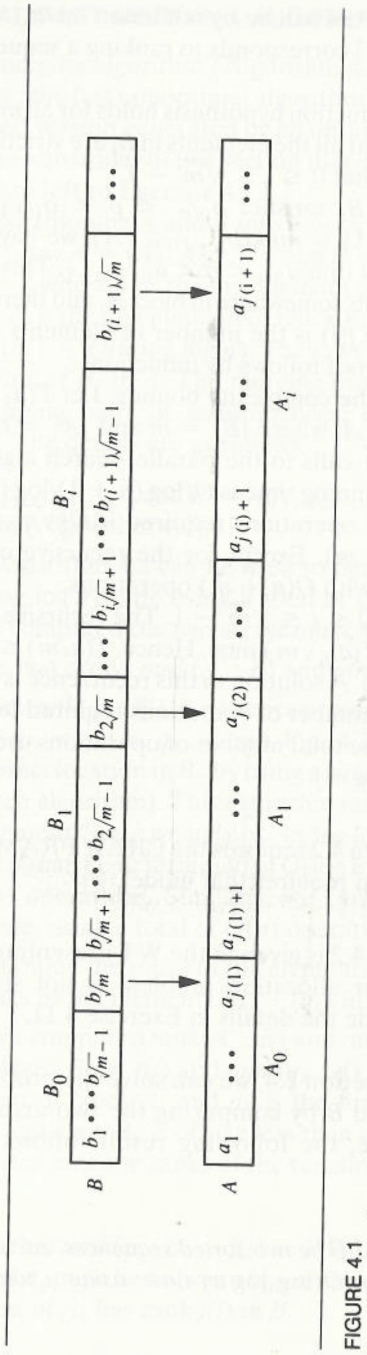


FIGURE 4.1
Partitions induced by Algorithm 4.2. Each block B_i is of size $\sqrt{m}$ but the size of each A_i block may vary. The index $j(i)$ is defined by $j(i) = rank(b_{i\sqrt{m}} : A)$.

Proof: The correctness proof will be by induction on m.

The base case $m = 3$ corresponds to ranking a sequence (b_1, b_2, b_3) in A. Step 1 settles this case.

Suppose that the induction hypothesis holds for all $m' < m$, where $m \geq 4$. We establish the fact that all the elements in B_i are strictly between $a_{j(i)}$ and $a_{j(i+1)+1}$, for any i such that $0 \leq i \leq \sqrt{m} - 1$.

Any element p in B_i satisfies $b_{i\sqrt{m}} < p < b_{(i+1)\sqrt{m}}$. Since $j(i) = rank(b_{i\sqrt{m}} : A)$ and $j(i + 1) = rank(b_{(i+1)\sqrt{m}} : A)$, we have $a_{j(i)} < b_{i\sqrt{m}}$ and $b_{(i+1)\sqrt{m}} < a_{j(i+1)+1}$, and thus $a_{j(i)} < p < a_{j(i+1)+1}$. This result implies that any element p of block B_i fits somewhere in block A_i and therefore $rank(p : A) = j(i) + rank(p : A_i)$, since $j(i)$ is the number of elements in A preceding A_i. Hence the correctness proof follows by induction.

We now establish the complexity bounds. Let $T(n, m)$ be the parallel time it takes to rank B in A, where $|B| = m$ and $|A| = n$.

Step 2 invokes $\sqrt{m}$ calls to the parallel-search algorithm (Algorithm 4.1) with $p = \sqrt{n}$. The running time is $O(\log(n + 1)/\log(\sqrt{n} + 1)) = O(1)$, and the total number of operations required is $O(\sqrt{m}.\sqrt{n}) = O(n + m)$ (since $2\sqrt{m}.\sqrt{n} \leq n + m$). Except for the recursive calls, steps 3 and 4 trivially take $O(1)$ time, with $O(n + m)$ operations.

Let $|A_i| = n_i$, for $0 \leq i \leq \sqrt{m} - 1$. The recursive call corresponding to the pair (B_i, A_i) takes $T(n_i, \sqrt{m})$ time. Hence, $T(n, m) \leq \max_i T(n_i, \sqrt{m}) + O(1)$, and $T(n, 3) = O(1)$. A solution to this recurrence is given by $T(n, m) = O(\log\log m)$. Since the number of operations required for each set of recursive calls is $O(n + m)$, the total number of operations used by Algorithm 4.2 is thus $O((n + m)\log\log m)$. □

PRAM Model: Algorithm 4.2 requires the CREW PRAM model because the parallel-search algorithm requires that model. □

Remark 4.2: Algorithm 4.2 is given in the WT presentation framework. The corresponding processor allocation problem is not straightforward. The reader is asked to provide the details in Exercise 4.11. □

As indicated in Section 2.4, we can solve the problem of merging two sorted sequences A and B by computing the two arrays $rank(A : B)$ and $rank(B : A)$. Therefore, the following result follows immediately from Lemma 4.3.

Corollary 4.1: *Let A and B be two sorted sequences, each of length n. Merging A and B can be done in $O(\log\log n)$ time, using a total of $O(n \log\log n)$ operations.* □

4.2.3 AN OPTIMAL FAST MERGING ALGORITHM

The previous fast merging algorithm (Algorithm 4.2) is clearly nonoptimal. As is usually the case, the fast nonoptimal algorithm is used to solve a suitably chosen reduced-sized version generated by an optimal (and slow) algorithm.

Assume for the remainder of this section that $m = n$. The details for the more general case are left to Exercise 4.12.

We begin by partitioning A and B into blocks, each of size less than or equal to $\lceil \log\log n \rceil$, say $A = (A_1, A_2, \ldots)$ and $B = (B_1, B_2, \ldots)$. We then let $A' = (p_1, p_2, \ldots)$, where p_i is the first element of block A_i of A and $B' = (q_1, q_2, \ldots)$, where q_i is the first element of block B_i of B. As a result, $|A'| = O(n/\log\log n)$ and $|B'| = O(n/\log\log n)$. Then, we rank the elements of A' in B and the elements of B' in A. The merging problem is now reduced to merging nonoverlapping pairs of subsequences; each subsequence is of length $O(\log\log n)$. The details are as follows:

1. Merge $A' = (p_1, p_2, \ldots)$ and $B' = (q_1, q_2, \ldots)$ using the fast nonoptimal algorithm (Algorithm 4.2).

   Explanation: Algorithm 4.2 can be used to perform the merging of A' and B' in $O(\log\log n)$ time using a total of $O(n)$ operations. At this point, we have computed the two arrays $rank(A' : B')$ and $rank(B' : A')$.
2. Determine the two arrays $rank(A' : B)$ and $rank(B' : A)$.

   Explanation: Let $rank(p_i : B') = r_i$; hence p_i must fit somewhere in block B_{r_i} of B since $q_{r_i} < p_i < q_{r_i+1}$ (see Fig. 4.2). For each p_i, we can determine its exact location in B_{r_i} by using a simple sequential algorithm (or binary search algorithm). This algorithm takes $O(\log\log n)$ sequential time per element. Since we have $\leq \lceil n/\log\log n \rceil$ such elements, the array $rank(A' : B)$ can be determined in $O(\log\log n)$ parallel time, using a total of $O(n)$ operations. Similarly, we can obtain $rank(B' : A)$ in $O(\log\log n)$ time, using a total of $O(n)$ operations.
3. For each i, determine the ranks of the elements $A_i - \{p_i\}$ in B, and, for each k, the ranks of the elements $B_k - \{q_k\}$ in A.

   Explanation: We computed $rank(A' : B)$ and $rank(B' : A)$ by the end of step 2. Let $rank(p_i : B) = j(i)$ and $rank(q_k : A) = \bar{j}(k)$. Recall that p_i is the first element of block A_i and q_k is the first element of block B_k, where each A_i and each B_k is of size less than or equal to $\lceil \log\log n \rceil$. In step 3, we determine the ranks of the remaining elements in each A_i and B_k.

Given the facts that $rank(p_i : B) = j(i)$ and that p_i is the first element of A_i, all the elements of A_i must lie between $b_{j(i)}$ and $b_{j(i+1)+1}$. Hence, if $j(i) = j(i + 1)$, each element of A_i has rank $j(i)$ in B.

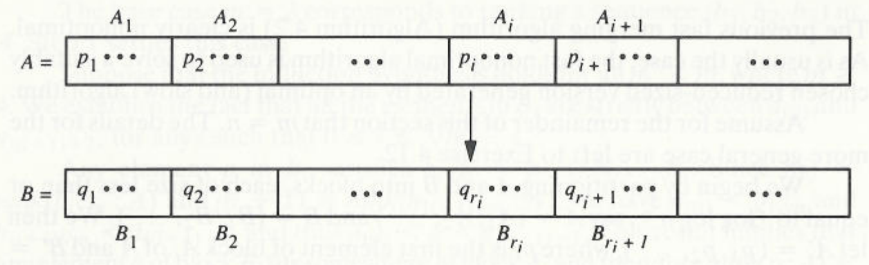


FIGURE 4.2
Step 2 of the optimal merging procedure. Since $rank(p_i : B') = r_i$, where $B' = (q_1, q_2, \ldots)$, we have that $q_{r_i} < p_i < q_{r_i+1}$; hence, p_i must fit somewhere in block B_{r_i}.

Suppose that $j(i+1) > j(i)$. In this case, all the elements $b_{j(i)+1}, \ldots, b_{j(i+1)}$ fit between p_i and p_{i+1}. If the number of such elements (that is, the $j(i+1) - j(i)$ elements $b_{j(i)+1}, \ldots, b_{j(i+1)}$) is less than $\log\log n$, then we can merge A_i with the corresponding subarray of B in $O(\log\log n)$ sequential time. Otherwise, there exist elements of B'—say, $q_k, q_{k+1}, \ldots, q_{k+s}$—that lie between $b_{j(i)+1}$ and $b_{j(i+1)}$ (see Fig. 4.3). Since the ranks $\tilde{j}(k), \tilde{j}(k+1), \ldots, \tilde{j}(k+s)$ of $q_k, q_{k+1}, \ldots, q_{k+s}$ are already known, our problem is reduced to merging no more than $s+2$ pairs of subsequences, each pair containing $O(\log\log n)$ elements (see Fig. 4.3). Note that the subsequences are completely disjoint. Hence, this merging can be done in $O(\log\log n)$ time, using a linear number of operations (in the total size of the subsequences involved). This process can be performed concurrently for all blocks A_i.

In summary, the optimal merging algorithm consists of three main steps:

1. Partition A and B into blocks $A = (A_1, A_2, \cdots)$ and $B = (B_1, B_2, \cdots)$ such that each block is of size $\leq \lceil \log\log n \rceil$. Use the fast but nonoptimal algorithm (Algorithm 4.2) to merge $A' = (p_1, p_2, \ldots)$ and $B' = (q_1, q_2, \ldots)$, where p_i and q_i are the first elements of A_i and B_i respectively.
2. Each p_i can now be located in a block B_{r_i}; hence, its exact rank in B can be determined easily, since $|B_{r_i}| \leq \lceil \log\log n \rceil$. The ranks of all the p_i's in B are determined concurrently. Similarly the ranks of the q_i's in A are determined concurrently.
3. Since the exact ranks of the elements of A' in B and the exact ranks of B' in A are known, the merging problem is reduced to a set of nonoverlapping merging subproblems; each of the corresponding pairs of subsequences involves $O(\log\log n)$ elements. All the merging subproblems can now be solved concurrently in $O(\log\log n)$ time, using a linear number of operations.

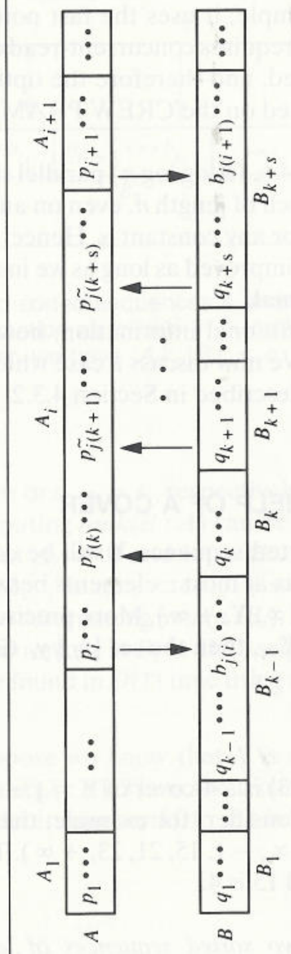


FIGURE 4.3
Step 3 of the optimal merging algorithm. Each A_i and B_i block is of size approximately $\log\log n$. The case depicted corresponds to $j(i+1) - j(i)$ being much larger than $\log\log n$. An arrow, say from p_i to $b_{j(i)}$, indicates that p_i must fit after $b_{j(i)}$ in the sequence B (more precisely, $j(i) = rank(p_i : B)$). Any consecutive pair of arrows determines two subsequences that have to be merged.

Therefore we have the following theorem.

Theorem 4.2: *The problem of merging two sorted sequences each of length n can be done in $O(\log\log n)$ time, using a total of $O(n)$ operations.*

PRAM Model: A concurrent-read capability is required by the optimal fast merging algorithm since, for example, it uses the fast nonoptimal merging algorithm (Algorithm 4.2), which requires concurrent-read operations. However, no concurrent write was used, and therefore the optimal $O(\log\log n)$ time algorithm can be implemented on the CREW PRAM model. □

As we shall show in Section 4.6, $\Omega(\log\log n)$ parallel steps are required to merge two sorted sequences each of length n, even on any CRCW PRAM model with $n\log^{\alpha} n$ processors, for any constant α. Hence, the optimal time fast merging algorithm cannot be improved as long as we insist on optimality; that is, it is **work-time (WT) optimal.**

If we are provided with additional information, however, we may be able to perform merging faster. We now discuss a case which pertains to the $O(\log n)$ time sorting algorithm described in Section 4.3.2.

4.2.4 *MERGING WITH THE HELP OF A COVER

Let c be a positive integer. A sorted sequence X will be called a ***c*-cover** of another sorted sequence Y if Y has at most c elements between each pair of *consecutive* elements in $X_\infty = (-\infty, X, +\infty)$. More precisely, given any two consecutive elements α and β of X_∞, then the set $\{y_i \mid y_i \in Y \text{ and } \alpha < y_i \le \beta\}$ has at most c elements.

EXAMPLE 4.3:

The sequence $X = (-1, 15, 21, 23)$ is a 4-cover of $Y = (-10, -5, -2, -1, 4, 5, 10, 12, 20, 22, 26, 31, 50)$. Consider, for example, the two consecutive elements -1 and 15 of $X_\infty = (-\infty, -1, 15, 21, 23, +\infty)$. The total number of elements of Y between -1 and 15 is 4. □

Lemma 4.3: *Let A and B be two sorted sequences of lengths n and m, respectively, and let X be a c-cover of A and B for some constant c. If $rank(X : A)$ and $rank(X : B)$ are known, then the problem of merging A and B can be solved in $O(1)$ time, using $O(|X|)$ operations.*

Proof: Let $X = (x_1, \ldots, x_s)$, $rank(X : A) = (r_1, r_2, \ldots, r_s)$ and $rank(X, B) = (t_1, t_2, \ldots, t_s)$. For each i, $1 \le i \le s+1$, let $A_i = (a_{r_{i-1}+1}, \ldots, a_{r_i})$, and $B_i = (b_{t_{i-1}+1}, \ldots, b_{t_i})$, where $r_0 = t_0 = 0$, and $r_{s+1} = n$, $t_{s+1} = m$. Figure 4.4 illustrates the partitions of A and B. Note that A_i or B_i could be

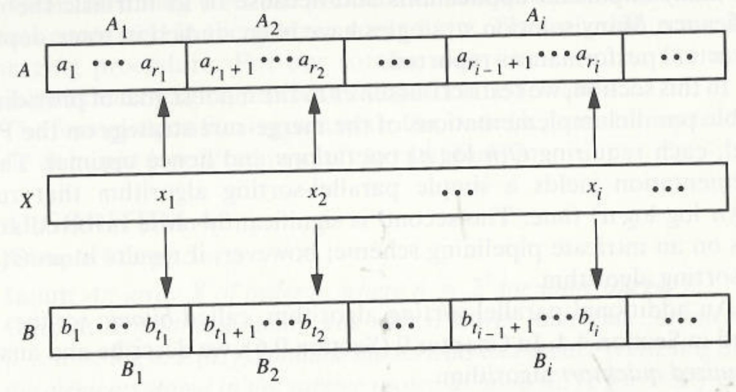


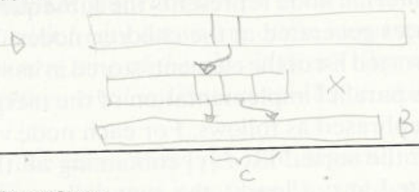
FIGURE 4.4
A merge of the two sorted sequences A and B with the cover X. The arrows indicate rank information as follows: $r_i = rank(x_i : A)$ and $t_i = rank(x_i : B)$. Since X is a c-cover, we have $|A_i|, |B_i| \le c$.

empty (if $r_{i-1} = r_i$ or $t_{i-1} = t_i$, respectively). We now show how to compute $rank(A : B)$. Computing $rank(B : A)$ can be done in a similar fashion.

Suppose that $A_i \ne \emptyset$ and let $a \in A_i$. Then, $rank(a : B) = t_{i-1} + rank(a : B_i)$, since $b_{t_{i-1}} \le x_{i-1} < a_{r_{i-1}+1} \le a \le a_{r_i} \le x_i < b_{t_i+1}$. Hence, the problem reduces to determining $rank(a : B_i)$. But $|B_i| \le c$, since X is a c-cover of B. Hence, a can be ranked in B_i in $O(1)$ sequential time. Therefore, the array $rank(A : B)$ can be found in $O(1)$ time using a linear number of operations. □

Remark 4.3: Suppose we know that X is a c-cover of B, and we are given $rank(A : X)$ and $rank(X : B)$. Then, the technique used in the proof of Lemma 4.3 allows us to determine $rank(A : B)$ in $O(1)$ time, using $O(|A| + |X|)$ operations. □

trivi



4.3 Sorting

The problem of **sorting** a sequence X is the process of rearranging the elements of X such that they appear in nondecreasing or nonincreasing order. The problem of sorting has been studied extensively in the literature because

assumption: a knows the value i for which $a \in A_i$
or can compute it in $O(1)$ time.
Ok because X is a c-cover of
make the "covered" elements in A
and then scan within A up to

of its many important applications and because of its intrinsic theoretical significance. Many solution strategies have been studied in some depth and their actual performances reported.

In this section, we restrict ourselves to the modest goal of providing two possible parallel implementations of the **merge-sort strategy** on the PRAM model, each requiring $O(n \log n)$ operations and hence optimal. The first implementation yields a simple parallel-sorting algorithm that runs in $O(\log n \log \log n)$ time. The second is significantly more involved and depends on an intricate pipelining scheme; however, it results in an $O(\log n)$ time sorting algorithm.

An additional parallel-sorting algorithm, called *bitonic sorting*, is described in Section 4.4. In Chapter 9 (Section 9.6), we describe and analyze a *randomized quicksort* algorithm.

4.3.1 A SIMPLE OPTIMAL SORTING ALGORITHM

As its name indicates, the *merge-sort strategy* is based on a merging procedure that is used to sort successively a number of larger and larger nonoverlapping subsequences until the whole sequence is sorted. One possible way to implement this strategy, referred to as **two-way merge sort,** is to start by sorting pairs of elements of the given sequence X, and then to sort every pair of consecutive pairs, and so on, until X is sorted.

The two-way merge-sort algorithm can be also viewed as an application of the divide-and-conquer strategy that consists of (1) dividing the input sequence X into two subsequences, X_1 and X_2, of approximately the same size; (2) sorting X_1 and X_2 separately; and finally (3) merging the two sorted sequences.

The sequence of operations required by the two-way merge-sort algorithm can be represented by a binary tree as follows. Let T be a balanced binary tree with n leaves. The elements of X are distributed among the leaves, one per leaf. The nodes at height 1 represent the lists we obtain by merging the pairs of consecutive elements contained in the children nodes (leaves, in this case). More generally, each internal node represents the subsequence we obtain by merging the subsequences generated at the children nodes. Hence, each internal node represents the sorted list of the elements stored in its subtree.

Since we are interested in a parallel implementation of the merge-sort algorithm, our problem can be rephrased as follows. For each node v of the balanced binary tree T, compute the sorted list $L[v]$ containing all the elements stored in the subtree rooted at v. Clearly, the root will contain the sorted list.

This process can be achieved in a fashion similar to that used in Algorithm 3.8 (the basic range-minima algorithm), which computes, for each node v of a balanced binary tree, the prefix minima and the suffix minima of the elements contained in the subtree rooted at v. The only difference lies in the merging procedure. For our sorting algorithm, we use the optimal $O(\log \log n)$ time merging procedure described in Section 4.2.

The formal algorithm is given next. The node (h,j) of a binary tree is the jth node at height h ordered in a left to right fashion.

ALGORITHM 4.3
(Simple Merge Sort)

Input: *An array X of order n, where $n = 2^l$ for some integer l.*
Output: *A balanced binary tree with n leaves such that, for each $0 \le h \le \log n$, $L(h,j)$ contains the sorted subsequence consisting of the elements stored in the subtree rooted at node (h,j), for $1 \le j \le n/2^h$. That is, node (h,j) contains the sorted list of the elements $X(2^h(j-1)+1), X(2^h(j-1)+2), \ldots, X(2^hj)$.*

```
begin
   1. for $1 \le j \le n$ pardo
         Set $L(0,j) := X(j)$
   2. for $h = 1$ to $\log n$ do
         for $1 \le j \le n/2^h$ pardo
            Merge $L(h-1, 2j-1)$ and $L(h-1, 2j)$ into the sorted
            list $L(h,j)$
end
```

EXAMPLE 4.4:

Consider the sequence $X = (12, -5, -7, 51, 6, 28, 3, -8)$. Fig. 4.5 shows the binary tree with the initial contents of the leaves. During iteration $h = 1$, we get $L(1,1) = (-5, 12), L(1,2) = (7, 51), L(1,3) = (6, 28)$ and $L(1,4) = (-8, 3)$. The next iteration causes the following two lists to be created: $L(2,1) = (-7, -5, 12, 51)$ and $L(2,2) = (-8, 3, 6, 28)$. Finally, the list at the root is generated and is given by $L(3,1) = (-8, -7, -5, 3, 6, 12, 28, 51)$. □

Theorem 4.3: *For each node v of the balanced tree T, Algorithm 4.3 generates the sorted list $L[v]$ consisting of the elements stored in the subtree rooted at v. The running time of the algorithm is $O(\log n \log \log n)$, and the total number of operations used is $O(n \log n)$. Hence, this algorithm is optimal.*

Proof: The correctness proof of Algorithm 4.3 is straightforward.

The number of iterations executed at step 2 is $O(\log n)$. Since the total number of elements involved at each level is n, each iteration takes $O(\log \log n)$ time, using a total of $O(n)$ operations if we use the optimal fast merging algorithm presented in Section 4.2. Hence, the theorem follows. □

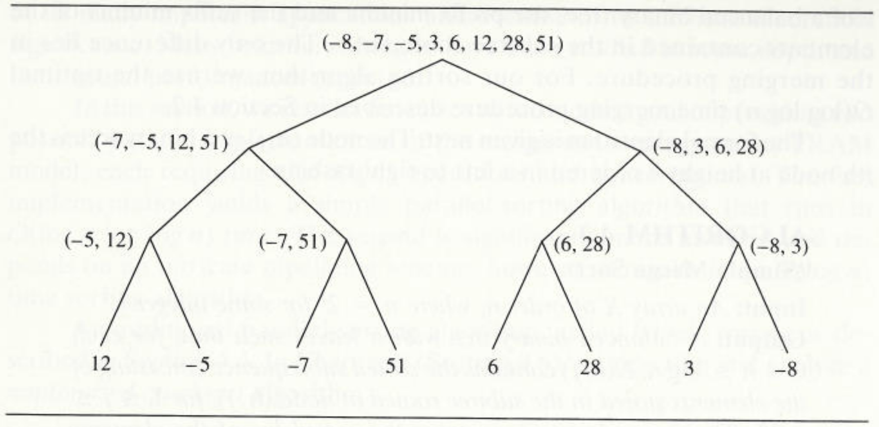


FIGURE 4.5
A merge-sort tree for eight elements. Each node contains the sorted list containing the elements stored in its subtree.

Remark 4.4: The space used by Algorithm 4.3 can be made $O(n)$ if we are interested in only the final sorted list. Once iteration h is completed, the nodes at height $h - 1$ will not be needed; hence, their space can be used to store the nodes at height h. □

Corollary 4.2: *Sorting a sequence of n elements can be done optimally in* $O(\log n \log\log n)$ *time.*

PRAM Model: The only nontrivial operation used in Algorithm 4.3 is the $O(\log\log n)$ time merging procedure of Section 4.2. Hence, this algorithm requires the CREW PRAM model. Had we used an EREW PRAM merging procedure, the corresponding sorting algorithm would have been an EREW PRAM algorithm. □

4.3.2 *AN OPTIMAL O(LOG *n*) TIME SORTING ALGORITHM

This section is devoted to the derivation of an $O(\log n)$ time optimal sorting algorithm. Actually, we address a slightly more general problem; its solution will provide us with an $O(\log n)$ time optimal sorting algorithm. In addition, several applications will make use of the general version.

The general formulation is as follows. Let T be a binary tree such that each leaf u contains an unsorted list $A(u)$ drawn from a linearly ordered set. We consider the problem of determining, for each internal node v, the sorted list $L[v]$ that contains all the elements stored in the subtree rooted at v. Note that the initial list $A(u)$ of a leaf u could be empty.

EXAMPLE 4.5:

Consider the tree T shown in Fig. 4.6(a). The list that should be generated at the indicated node is given by $(-9, -7, -6, 2, 5)$. □

We start by making a couple of transformations. We replace each leaf u with a balanced binary tree with $|A(u)|$ leaves such that each element of $A(u)$ is stored in one of the leaves. The height of T has increased by $O(\log(\max_u |A(u)|))$, but each leaf of T now contains at most one element.

The second transformation is to force each internal node to have two children. If this is not the case, a leaf containing no elements can be inserted.

EXAMPLE 4.6:

Applying the two transformations to the tree T given in Fig. 4.6(a), we obtain the tree T' shown in Fig. 4.6(b). □

Therefore, we assume for the remainder of this section that T is a binary tree such that at most one element is stored in a leaf node and every internal node has exactly two children.

Pipelined Merge-Sort Algorithm. We introduced in Section 4.2 the notion of a c-cover, and the integer array $rank(A : B)$ corresponding to ranking a sorted list A in a sorted list B. Before proceeding, we need to make the following additional definition.

Given a sorted list L, the ***c*-sample** of L, denoted by $sample_c\ (L)$, is the sorted sublist of L consisting of every cth element of L; that is, if $L = (l_1, l_2, \dots)$, then $sample_c\ (L) = (l_c, l_{2c}, \dots)$.

EXAMPLE 4.7:

Let $L = (4, 7, 8, 9, 11, 15, 38)$. Then $sample_3\ (L)$ is given by $sample_3\ (L) = (8, 15)$. □

The parallel merge-sort strategy presented earlier is based on a forward traversal of the binary tree such that, for all vertices at a given height h, the lists $L[v]$ are *completely determined* before processing of the nodes at height $h + 1$ begins.

The **pipelined (or cascading) divide-and-conquer strategy** consists of determining $L[v]$ over a number of stages such that, at stage s, $L_s[v]$ is an approximation of $L[v]$ that will be improved at the next stage $s + 1$. At the

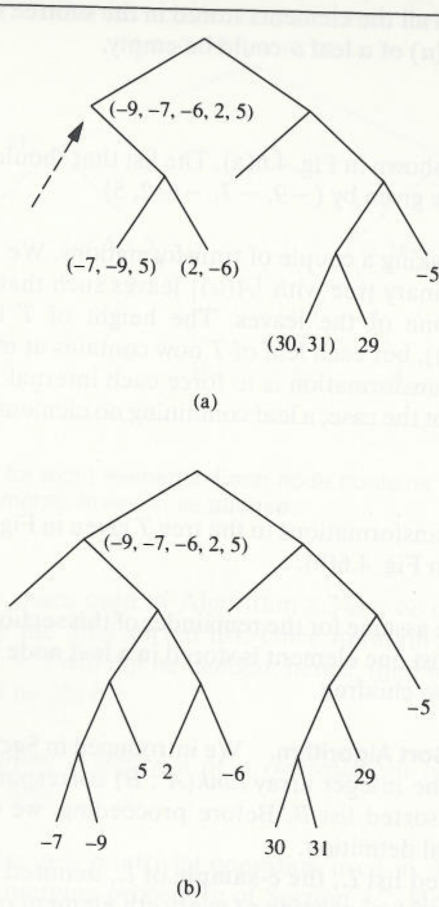


FIGURE 4.6
The tree for Examples 4.5 and 4.6. (a) An input tree for the general sorting problem; the arrow points to a node with its expected sorted list. (b) The tree T' after application of the two transformations: we replace each leaf containing a list by a balanced binary tree, and force each internal node to have exactly two children.

same time, a *sample* of $L_s[v]$ is propagated upward to be used for obtaining approximations of the lists to be generated at higher heights. The success of this method is due to the intricate combination of pipelining and the efficient merging of sample lists.

We now describe the procedure for determining $L_s[v]$ precisely; later, we provide a correctness proof and an analysis of the resources required.

Let $L_0[v] = \emptyset$ if v is an internal node; otherwise, $L_0[v]$ consists of the item (if any) stored at the leaf v. Let the **altitude** of a node v be defined as $alt(v) = h(T) - level(v)$, where $h(T)$ is the height of T, and, as usual, $level(v)$ is the length of the path from the root to v. The list stored at an internal node v will be updated over the stages s satisfying $alt(v) \le s \le 3alt(v)$.

We say that v is **active** during stage s if $alt(v) \le s \le 3alt(v)$. The algorithm will update the list $L_s[v]$ such that node v will be **full**—that is, $L_s[v] = L[v]$—when $s \ge 3alt(v)$. It is clear that, if this invariant can be maintained, then, after $3h(T)$ stages, the node at the root will be full, and all the nodes will contain their sorted lists.

An additional notation is needed before a description of the algorithm is given. Define $Sample(L_s[x])$ for an arbitrary node x as follows.

$$Sample(L_s[x]) = \begin{cases} sample_4\ (L_s[x])\ \textit{if}\ s \le 3alt(x); \\ sample_2\ (L_s[x])\ \textit{if}\ s = 3alt(x) + 1; \\ sample_1\ (L_s[x])\ \textit{if}\ s = 3alt(x) + 2. \end{cases}$$

Therefore, $Sample(L_s[x])$ is the sublist consisting of every fourth element of $L_s[x]$ until it becomes full; then $Sample(L_s[x])$ is every other element in the following stage (that is, stage $3alt(x) + 1$), and every element in stage $3alt(x) + 2$.

We next give a description of a general stage of the pipelined merge-sort algorithm that maintains the stated invariant. That is, for each node v, $L_s[v]$ will be full when $s \ge 3alt(v)$.

ALGORITHM 4.4
(Pipelined Merge Sort)

Input: *For each node v of a binary tree, a sorted list $L_s[v]$ such that v is full whenever $s \ge 3alt(v)$.*

Output: *For each node v, a sorted list $L_{s+1}[v]$ such that v is full whenever $s \ge 3alt(v) - 1$.*

Algorithm for $(s + 1)$ at stage:

```
begin
  for all active nodes v pardo
      1. Let u and w be the children of v. Set $L'_{s+1}[u] = Sample(L_s[u])$
         and $L'_{s+1}[w] = Sample(L_s[w])$.
      2. Merge the two lists $L'_{s+1}[u]$ and $L'_{s+1}[w]$ into the sorted list
         $L_{s+1}[v]$.
end
```

EXAMPLE 4.8:

Let T be the binary tree shown in Fig. 4.7(a). The lists corresponding to a set of selected stages are shown in the table of Fig. 4.7(b). Note that, initially, no changes occur until stage $s = 3$. At the end of stage $s = 3$, all the nodes of altitude 1 (nodes v_5 and v_6, in this case) become full. Consider, for example, node v_5. Since $alt(v_5) = 1$, v_5 is active during stage $s = 3$. In this case, $L_3^1[v_1] = sample_1(L_2[v_1]) = (7)$, and similarly $L_3^1[v_2] = (8)$. Hence, $L_3[v_5] = (7, 8)$. During this stage, we also obtain $L_3[v_6] = (1, 6)$.

The lists generated at several later stages are shown in the figure. Notice that, at the end of stage $s = 6$, the nodes at altitude 2 become full; at the end of stage $s = 9$, the nodes at altitude 3 become full. The root v_{21} is active for all stages s, $5 \le s \le 15$. However, $L_s[v_{21}]$ remains empty until stage $s = 13$ since, at each of the previous stages, the lists of the children nodes v_{19} and v_{20} contain less than four elements. At the end of stage $s = 12$, nodes v_{19} and v_{20} become full and each contain at least four elements. Hence, at stage $s = 13$, $L_s[v_{21}] = (5, 15)$, which results from the merging of $sample_4(L[v_{19}])$ and $sample_4(L[v_{20}])$. At the end of stage $s = 15$, $L_{15}[v_{21}]$ consists of the sorted list of all the items stored in the tree. □

We are ready to show that Algorithm 4.4 works correctly.

Lemma 4.4: *Let v be an arbitrary node of the binary tree T. Then, at the end of stage $s = 3alt(v)$ of Algorithm 4.4, v becomes full; that is, $L_s[v] = L[v]$.*

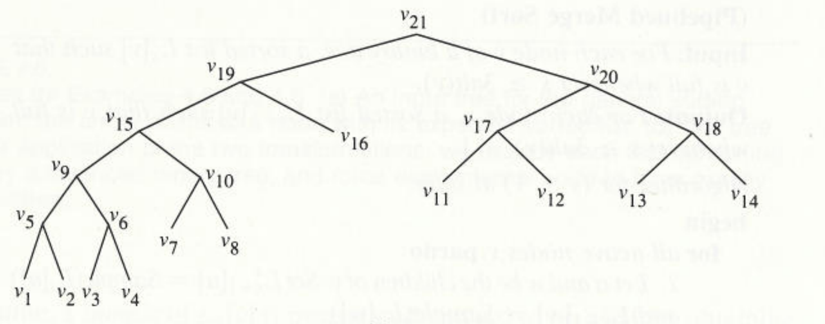


FIGURE 4.7
The tree for Example 4.8. (a) A binary tree.

v	$s = 0$	$s = 3$	$s = 5$	$s = 6$	$s = 8$	$s = 9$	$s = 11$	$s = 13$
1	(7)	(7)	(7)	(7)	(7)	(7)	(7)	(7)
2	(8)	(8)	(8)	(8)	(8)	(8)	(8)	(8)
3	(6)	(6)	(6)	(6)	(6)	(6)	(6)	(6)
4	(1)	(1)	(1)	(1)	(1)	(1)	(1)	(1)
5	0	(7, 8)	(7, 8)	(7, 8)	(7, 8)	(7, 8)	(7, 8)	(7, 8)
6	0	(1, 6)	(1, 6)	(1, 6)	(1, 6)	(1, 6)	(1, 6)	(1, 6)
7	(5)	(5)	(5)	(5)	(5)	(5)	(5)	(5)
8	(3)	(3)	(3)	(3)	(3)	(3)	(3)	(3)
9	0	0	(6, 8)	(1, 6, 7, 8)	(1, 6, 7, 8)	(1, 6, 7, 8)	(1, 6, 7, 8)	(1, 6, 7, 8)
10	0	0	0	(3, 5)	(3, 5)	(3, 5)	(3, 5)	(3, 5)
11	(4)	(4)	(4)	(4)	(4)	(4)	(4)	(4)
12	(10)	(10)	(10)	(10)	(10)	(10)	(10)	(10)
13	(9)	(9)	(9)	(9)	(9)	(9)	(9)	(9)
14	(15)	(15)	(15)	(15)	(15)	(15)	(15)	(15)
15	0	0	0	0	(5, 6, 8)	(1, 3, 5, 6, 7, 8)	(1, 3, 5, 6, 7, 8)	(1, 3, 5, 6, 7, 8)
16	(2)	(2)	(2)	(2)	(2)	(2)	(2)	(2)
17	0	0	0	0	0	(4, 10)	(4, 10)	(4, 10)
18	0	0	0	0	0	(9, 15)	(9, 15)	(9, 15)
19	0	0	0	0	0	0	(3, 6, 8)	(1, 2, 3, 5, 6, 7, 8)
20	0	0	0	0	0	0	(10, 15)	(4, 9, 10, 15)
21	0	0	0	0	0	0	0	(5, 15)

(b)

FIGURE 4.7 (*continued*)
(b) The lists arising during the execution of the indicated stages of the pipelined merge-sort algorithm.

Proof: The proof is by induction on $alt(v)$. The claim is obviously true for all nodes satisfying $alt(v) = 0$, since they all are leaves satisfying initially $L_0[v] = L[v]$.

Let v be a node with $alt(v) = k > 0$. If v is a leaf, then $L_0[v] = L[v]$, and there is nothing to prove. Assume that v is an internal node with children u and w. Clearly, $alt(u) = alt(w) = k - 1$. By the induction hypothesis, u and w will become full at stage s' such that $s' = 3(k - 1)$. During stage $s' + 1$, $Sample(L_{s'}[u])$ and $Sample(L_{s'}[w])$ will be merged to form $L_{s'+1}[v]$. But $Sample(L_{s'}[u]) = sample_4\ (L_{s'}[u]) = sample_4\ (L[u])$, and similarly for w. During stage $s' + 2$, $Sample(L_{s'+1}[u])$ will consist of every other element of $L[u]$; during stage $s' + 3$, $Sample(L_{s'+2}[u]) = L[u]$. Similar arguments hold for node w. Hence, at stage $s' + 3 = 3k = 3alt(v)$, $L_{s'+3}[v] = L[v]$, and the lemma follows by induction. □

The following lemma states that the size of each list can grow at most by roughly a factor of 2 after each stage. The proof is a simple induction on s and is left to the reader in Exercise 4.22.

Lemma 4.5: *Let v be an arbitrary node of T and let $s \geq 1$. Then, $|L_{s+1}[v]| \leq 2|L_s[v]| + 4$.* □

Remark 4.5: For a given stage s, the total number of elements stored in all the active nodes of T is given by $n_s = \sum_{v\ active} |L_s[v]| = \sum_{\lfloor \frac{s}{3} \rfloor \leq alt(v) \leq s} |L_s[v]|$. Note that, if a node v is full, where $alt(v) = \lfloor s/3 \rfloor$, none of its ancestors can be full. Thus, $\sum_{alt(v) = \lfloor \frac{s}{3} \rfloor} |L_s[v]| \leq n$, where n is the number of leaves in T. Consider the active nodes at the level just above the level of these full nodes. There can be at most $n/2$ elements stored in these nodes, and at most $n/4$ elements stored at the level above them, and so on. Therefore, $n_s = O(n)$. □

Implementation Detail and Analysis. The only essential details left are to show how to perform step 2 of Algorithm 4.4 in $O(1)$ time, using $O(n_s) = O(n)$ operations.

We have already seen (Lemma 4.3) that the merging of two sorted lists A and B can be done optimally in $O(1)$ time if we are given a c-cover X for A and B and if $rank(X:A)$ and $rank(X:B)$ are also given as a part of the input. Guided by this observation, we next show that, for each node v, the list $L_s[v]$ is a 4-cover for $L'_{s+1}[u]$ and for $L'_{s+1}[w]$, where u and w are the children of v. We later show how the two arrays $rank(L_s[v] : L'_{s+1}[u])$ and $rank(L_s[v] : L'_{s+1}[w])$ can be generated efficiently.

Covers for the Lists to be Merged. We start by establishing the fact that $Sample(L_{s-1}[v]) = L'_s[v]$ is a 4-cover of $Sample(L_s[v]) = L'_{s+1}[v]$.

Lemma 4.6: *Let v be an arbitrary node of T and let $s \geq 1$. Then, $L'_s[v]$ is a 4-cover of $L'_{s+1}[v]$.*

Proof: We prove a slightly stronger result. Let $[a, b]$ be an interval with $a, b \in (-\infty, L'_s[v], +\infty)$. We say that $[a, b]$ **intersects** $(-\infty, L'_s[v], +\infty)$ in k items (or that there are k items **in common**) if the number of elements $x \in (-\infty, L'_s[v], +\infty)$ such that $a \leq x \leq b$ is equal to k. We use induction on s to establish the following claim.

Claim: If $[a, b]$ intersects $(-\infty, L'_s[v], +\infty)$ in $k \geq 2$ items, then $[a, b]$ intersects $L'_{s+1}[v]$ in at most $2k$ items.

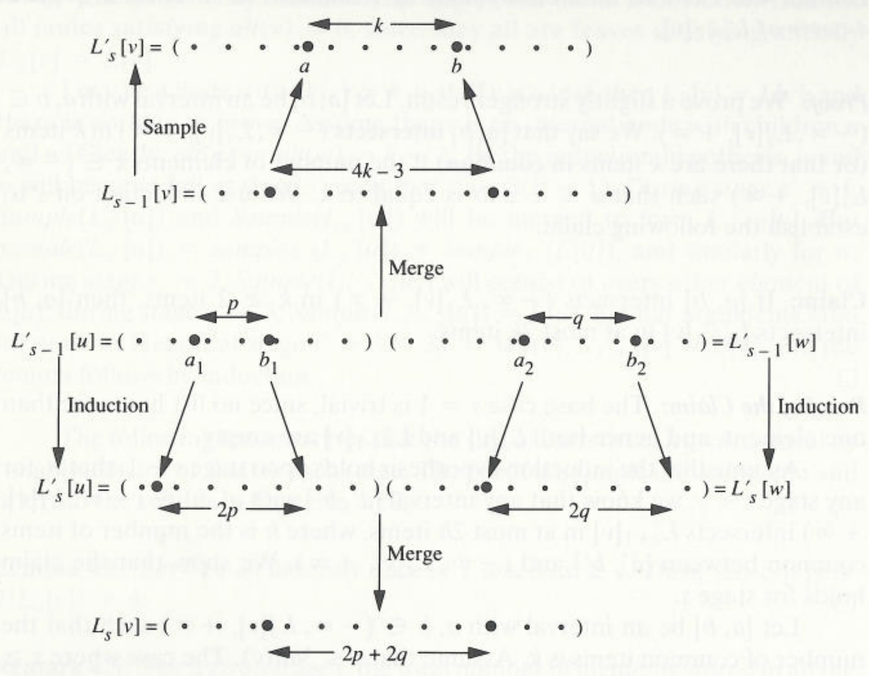
Proof of the Claim: The base case $s = 1$ is trivial, since no list has more than one element, and hence both $L'_s[v]$ and $L'_{s+1}[v]$ are empty.

Assume that the induction hypothesis holds up to stage $s - 1$; that is, for any stage $t < s$, we know that any interval $[a', b']$ with $a', b' \in (-\infty, L'_t[v], +\infty)$ intersects $L'_{t+1}[v]$ in at most $2h$ items, where h is the number of items common between $[a', b']$ and $(-\infty, L'_t[v], +\infty)$. We show that the claim holds for stage s.

Let $[a, b]$ be an interval with $a, b \in (-\infty, L'_s[v], +\infty)$ such that the number of common items is k. Assume that $s \leq 3alt(v)$. The case where $s \geq 3alt(v) + 1$ is straightforward, since $L_{s-1}[v] = L_s[v] = L[v]$. Since $s \leq 3alt(v)$, $L'_s[v] = sample_4\ (L_{s-1}[v])$, and hence $[a, b]$ intersects $(-\infty, L_{s-1}[v], +\infty)$ in $4k - 3$ items.

Recall that we obtained $L_{s-1}[v]$ by merging $L'_{s-1}[u]$ and $L'_{s-1}[w]$, where u and w are the children of v. Hence, the items belonging to $[a, b]$ and $L_{s-1}[v]$ must have come from the set $L'_{s-1}[u] \cup L'_{s-1}[w]$, where each list is viewed as a set of elements. Let $[a_1, b_1]$ be the smallest interval containing $[a, b]$ such that $a_1, b_1 \in (-\infty, L'_{s-1}[u], +\infty)$. Similarly define $[a_2, b_2]$ such that $a_2, b_2 \in (-\infty, L'_{s-1}[w], +\infty)$. Let p be the number of elements in common between $[a_1, b_1]$ and $(-\infty, L'_{s-1}[u], +\infty)$. Similarly, define q to be the number of items in common between $[a_2, b_2]$ and $(-\infty, L'_{s-1}[w], +\infty)$. Since we are assuming that all the elements are distinct, we obtain that $p + q \leq 4k - 1\ (= (4k - 3) + 2$, since two additional elements from $\{a_1, b_1, a_2, b_2\}$ are included).

By the induction hypothesis, $[a_1, b_1]$ intersects $L'_s[u]$ in at most $2p$ elements, and $[a_2, b_2]$ intersects $L'_s[w]$ in at most $2q$ elements (see Fig. 4.8 for a pictorial representation of the relationships among the different lists involved). Now $L_s[v]$ is just the list obtained after merging of $L'_s[u]$ and $L'_s[w]$. Hence, $[a, b]$ intersects $L_s[v]$ in at most $2p + 2q \leq 8k - 2$ elements. Since $L'_{s+1}[v] = sample_4\ (L_s[v])$, we obtain that $[a, b]$ intersects $L'_{s+1}[v]$ in at most $2k$ items, and the claim follows.



$L'_s[v] = (\cdot\ \cdot\ \cdot\ \bullet\ \cdot\ \cdot\ \cdot\ \cdot\ \bullet\ \cdot\ \cdot\ \cdot\ \cdot)$
k
a
b
Sample
$4k-3$
$L_{s-1}[v] = (\ \cdot\ \bullet\ \cdot\ \cdot\ \bullet\ \cdot\ \cdot\ )$
Merge
p
q
$L'_{s-1}[u] = (\ \cdot\ \bullet\ \cdot\ \bullet\ \cdot\ )\ (\ \cdot\ \bullet\ \cdot\ \bullet\ \cdot\ ) = L'_{s-1}[w]$
a_1
b_1
a_2
b_2
Induction
Induction
$L'_s[u] = (\ \cdot\ \bullet\ \cdot\ \cdot\ \cdot\ \bullet\ \cdot\ )\ (\ \cdot\ \bullet\ \cdot\ \cdot\ \cdot\ \cdot\ \bullet\ \cdot\ ) = L'_s[w]$
$2p$
$2q$
Merge
$L_s[v] = (\ \cdot\ \bullet\ \cdot\ \cdot\ \cdot\ \bullet\ \cdot\ )$
$2p + 2q$

FIGURE 4.8
Illustration of the inductive proof of the fact that, if $[a, b]$ intersects $L'_s[v]$ in k elements, then $[a, b]$ intersects $L_s[v]$ in at most $2p + 2q \leq 8k - 2$ elements. This result implies that $[a, b]$ intersects $L'_{s+1}[v]$ in at most $2k$ items.

We now complete the proof of the lemma. Let $a, b \in (-\infty, L'_s[v], +\infty)$, such that a and b are adjacent. Hence, $[a, b]$ intersects $(-\infty, L'_s[v], +\infty)$ in exactly two items. As shown in the claim, $[a, b]$ intersects $L'_{s+1}[v]$ in at most four elements; hence, $L'_s[v]$ is a 4-cover of $L'_{s+1}[v]$. □

Corollary 4.3: *For each internal node v of T, and for each stage $s \geq alt(v)$, $L_s[v]$ is a 4-cover of $L'_{s+1}[u]$ and $L'_{s+1}[w]$, where u and w are the children of v.*

Proof: Let a and b be two adjacent elements of $(-\infty, L_s[v], +\infty)$. Recall that we obtain $L_s[v]$ by merging $L'_s[u]$ and $L'_s[w]$. Let $[a', b']$ be the smallest interval containing a and b such that $a', b' \in (-\infty, L'_s[u], +\infty)$. Clearly, a' and b' are adjacent in $(-\infty, L'_s[u], +\infty)$. Hence, by Lemma 4.6, there are at most four elements in $L'_{s+1}[u]$ between a' and b'. This result implies that there are at most four elements in $L'_{s+1}[u]$ between a and b, which proves that $L_s[v]$ is a 4-cover for $L'_{s+1}[u]$.

We can establish in a similar fashion that $L_s[v]$ is a 4-cover of $L'_{s+1}[w]$. □

Efficient Merging of the Samples. The next lemma shows how to maintain efficiently certain rank information, which will be used to perform the merging operations fast.

Lemma 4.7: *Let $s \geq 2$ be a given stage number of Algorithm 4.4. Suppose that, for every internal node v of T and its two children u and w, we are given the following information:*

1. *$rank(L'_s[v] : L'_{s+1}[v])$*
2. *$rank(L'_s[u] : L'_s[w])$*
3. *$rank(L'_s[w] : L'_s[u])$*

Then, using $O(|L'_{s+1}[u]| + |L'_{s+1}[w]|)$ operations, we can compute the following information in $O(1)$ time:

1. *$rank(L'_{s+1}[v] : L'_{s+2}[v])$*
2. *$rank(L'_{s+1}[u] : L'_{s+1}[w])$*
3. *$rank(L'_{s+1}[w] : L'_{s+1}[u])$*

Proof: We first show how to obtain $rank(L'_{s+1}[u] : L'_{s+1}[w])$ within the stated bounds. Actually, the proof follows from Remark 4.3; for clarity, however, we provide the proof here.

Consider the array $rank(L'_s[u] : L'_{s+1}[u])$, which is given as a part of the input whenever u is an internal node. Since $L'_s[u]$ is a 4-cover of $L'_{s+1}[u]$, by Lemma 4.6, the number of elements in $L'_{s+1}[u]$ between any two consecutive elements of $(-\infty, L'_s[u], +\infty)$ is at most 4. Let S_i be the segment of $L'_{s+1}[u]$ consisting of all the elements between the ith and the $(i+1)$st elements—say, p and q—of $L'_s[u]$. Clearly, $|S_i| \leq 4$. We also know the ranks of p and q in $L'_s[w]$, since the array $rank(L'_s[u] : L'_s[w])$ is given as a part of the input. Let us denote $rank(p : L'_s[w])$ by s_1, and $rank(q : L'_s[w])$ by s_2. Therefore, the elements of S_i are known to lie in the segment S'_i of $L'_s[w]$ determined by the indices s_1 and s_2 (see Fig. 4.9 for an illustration).

The problem of generating $rank(L'_{s+1}[u] : L'_s[w])$ reduces to determining the relative ranks of each element of S_i in the corresponding block S'_i. Since $|S_i| \leq 4$, however, we can use the parallel-search procedure (Algorithm 4.1) to determine the ranks of all the elements of S_i in S'_i. Such computation takes $O(1)$ time, using $O(|S'_i|)$ operations. Since, for $i \neq j$, S'_i and S'_j do not overlap, the array $rank(L'_{s+1}[u] : L'_s[w])$ can be obtained in $O(1)$ time, using $O(|L'_{s+1}[u]| + |L'_s[w]|)$ operations.

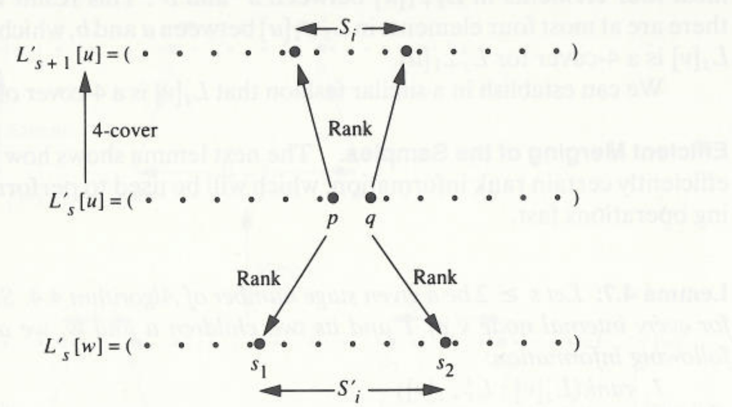


FIGURE 4.9
Illustration of how to compute $rank(L'_{s+1}[u] : L'_s[w])$. Note that $L'_s[u]$ is a 4-cover of $L'_{s+1}[u]$, and that the array $rank(L'_s[u] : L'_s[w])$ is given as part of the input. The elements p and q are the ith and the $(i + 1)$st elements of $L'_s[u]$.

We can easily determine the array $rank(L'_{s+1}[u] : L'_{s+1}[w])$ by using $rank(L'_s[w] : L'_{s+1}[w])$ and the fact that $L'_s[w]$ is a 4-cover of $L'_{s+1}[w]$. We can obtain the array $rank(L'_{s+1}[w] : L'_{s+1}[u])$ in a similar fashion.

We now consider the problem of determining $rank(L'_{s+1}[v] : L'_{s+2}[v])$. Recall that $L'_{s+1}[v]$ is just a sample of $L_s[v]$, and that $L_s[v]$ is obtained by merging of $L'_s[u]$ and $L'_s[w]$.

Let p be any element of $L'_s[u]$. We know the rank of p in $L'_s[u]$, of course. We also know the rank of p in $L'_s[w]$, since we are given the array $rank(L'_s[u] : L'_s[w])$. Since $rank(p : L_s[v]) = rank(p : L'_s[u]) + rank(p : L'_s[w])$, we know, for each $p \in L'_s[u]$, the latter's location in the array $L_s[v]$. Moreover, since the array $rank(L'_s[u] : L'_{s+1}[u])$ is also given, we know $rank(p : L'_{s+1}[u])$.

We can also obtain $rank(p : L'_{s+1}[w])$ as follows. Suppose that $rank(p : L'_s[w]) = r_1$. Then, p is between elements e and f at positions r_1 and $r_1 + 1$, respectively, of $L'_s[w]$. Since $rank(L'_s[w] : L'_{s+1}[w])$ is known, we can determine in $O(1)$ sequential time the boundaries of the segment S_p of $L'_{s+1}[w]$ of all elements between e and f (see Fig. 4.10 for an illustration). Using the fact that $L'_s[w]$ is a 4-cover of $L'_{s+1}[w]$, we conclude that $|S_p| \leq 4$. Therefore, we can determine in $O(1)$ sequential time $rank(p : L'_{s+1}[w])$.

Since $rank(p : L_{s+1}[v]) = rank(p : L'_{s+1}[u]) + rank(p : L'_{s+1}[w])$, the rank of p in $L_{s+1}[v]$ can be determined in $O(1)$ sequential time. However, since $L'_{s+2}[v]$ is a sample of $L_{s+1}[v]$, we can obtain the rank of p in $L'_{s+2}[v]$ in $O(1)$ sequential time.

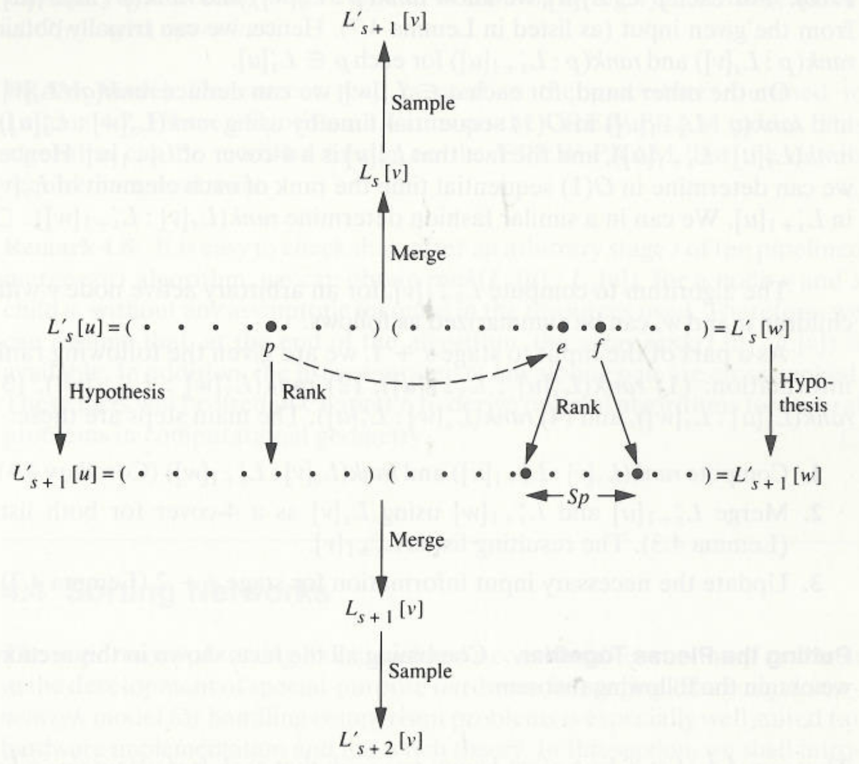


FIGURE 4.10
Determination of $rank(p : L'_{s+2}[v])$ in three steps: (1) determine $rank(p : L'_{s+1}[w])$ using $rank(p : L'_s[w])$, which identifies e and f, followed by ranking p in S_p; (2) determine $rank(p : L_{s+1}[v])$; and (3) use the fact that $L'_{s+2}[v]$ is a sample of $L_{s+1}[v]$ to deduce $rank(p : L'_{s+2}[v])$.

We can perform the same procedure for all the elements in $L'_s[w]$. Therefore, we can determine $rank(L_s[v] : L'_{s+2}[v])$ in $O(1)$ time, using a linear number of operations. Once this value is determined, generating the array $rank(L'_{s+1}[v] : L'_{s+2}[v])$ can be done easily in $O(1)$ time, using $O(|L'_{s+1}[v]|)$ operations. □

Corollary 4.4: *Under the hypothesis of Lemma 4.7, we can determine $rank(L_s[v] : L'_{s+1}[u])$ and $rank(L_s[v] : L'_{s+1}[w])$ in $O(1)$ time using a linear number of operations.*

Proof: For each $p \in L'_s[u]$, we know $rank(p : L'_s[w])$ and $rank(p : L'_{s+1}[u])$ from the given input (as listed in Lemma 4.7). Hence, we can trivially obtain $rank(p : L_s[v])$ and $rank(p : L'_{s+1}[u])$ for each $p \in L'_s[u]$.

On the other hand, for each $q \in L'_s[w]$, we can deduce $rank(q : L_s[v])$ and $rank(q : L'_{s+1}[u])$ in $O(1)$ sequential time by using $rank(L'_s[w] : L'_s[u])$, $rank(L'_s[u] : L'_{s+1}[u])$, and the fact that $L'_s[u]$ is a 4-cover of $L'_{s+1}[u]$. Hence, we can determine in $O(1)$ sequential time the rank of each element of $L_s[v]$ in $L'_{s+1}[u]$. We can in a similar fashion determine $rank(L_s[v] : L'_{s+1}[w])$. □

The algorithm to compute $L_{s+1}[v]$, for an arbitrary active node v with children u and w, can be summarized as follows.

As a part of the input to stage $s + 1$, we are given the following rank information: (1) $rank(L'_s[u] : L'_{s+1}[u])$, (2) $rank(L'_s[w] : L'_{s+1}[w])$, (3) $rank(L'_s[u] : L'_s[w])$, and (4) $rank(L'_s[w] : L'_s[u])$. The main steps are these:

1. Compute $rank(L_s[v] : L'_{s+1}[u])$ and $rank(L_s[v] : L'_{s+1}[w])$ (Corollary 4.4).
2. Merge $L'_{s+1}[u]$ and $L'_{s+1}[w]$ using $L_s[v]$ as a 4-cover for both lists (Lemma 4.3). The resulting list is $L_{s+1}[v]$.
3. Update the necessary input information for stage $s + 2$ (Lemma 4.7).

Putting the Pieces Together. Combining all the facts shown in this section, we obtain the following theorem.

Theorem 4.4: *Let T be a given binary tree such that each leaf v contains a list A(v). Let h(T) be the height of T, and let* $m = \max_v |A(v)|$. *Then, the pipelined merge-sort algorithm (Algorithm 4.4) generates, for each node v of T, a sorted list L[v] containing all the items stored in the subtree rooted at v. The overall algorithm runs in* $O(h(T) + \log m)$ *time, using a total of* $O((n_1 + n_2)(h(T) + \log m))$ *operations, where* n_1 *is the number of nodes in T and* n_2 *is the total number of items in T.*

Proof: The tree T' that we obtain from T by replacing each list $A(v)$ with a balanced binary tree with $|A(v)|$ leaves is of height less than or equal to $h(T) + \log m$ and has no more than $n_1 + n_2$ leaves.

The pipelined merge-sort algorithm applied to T' consists of at most $3(h(T) + \log m)$ stages; each stage can be performed in $O(1)$ time, using $O(n_1 + n_2)$ operations (recall Remark 4.5, which states that the number of elements in all the active nodes of any stage is asymptotically the same as the number of leaves). Therefore, the running time is $O(h(T) + \log m)$, and the number of operations is $O((n_1 + n_2)(h(T) + \log m))$. □

Corollary 4.5: *Sorting n elements can be done in* $O(\log n)$ *time, using a total of* $O(n \log n)$ *operations.* □

PRAM Model: Concurrent read is used in the procedures outlined in Lemma 4.7. Hence, Algorithm 4.4 runs on the CREW PRAM model. This algorithm can be modified to run on the EREW PRAM, but the details required are nontrivial. □

Remark 4.6: It is easy to check that, after an arbitrary stage s of the pipelined merge-sort algorithm, we can obtain $rank(L_s[u] : L_s[v])$, for a node v and a child u, without any asymptotic increase in the resources used. Therefore, we can assume that, at the end of the algorithm, the array $rank(L[u] : L[v])$ is available. In addition, the lists generated at any sibling pair are cross-ranked. These facts will be used in Chapter 6 to derive optimal algorithms for several problems in computational geometry. □

4.4 Sorting Networks

The importance of sorting has stimulated a considerable amount of research in the development of special-purpose hardware for sorting. The *comparator-network* model for handling comparison problems is especially well suited for hardware implementation and has a rich theory. In this section, we shall introduce the classical **bitonic sorting network**, and outline several of its properties.

A **comparator network** is made up of **comparators**, where a comparator is a module whose two inputs are x and y and whose two ordered outputs are $\min\{x, y\}$ and $\max\{x, y\}$. A possible representation of a comparator is shown in Fig. 4.11(a). Note that the direction of the arrow used in this representation is significant.

An example of a comparator network that sorts is shown in Fig. 4.11(b). In this example, the input items are x_1, x_2, x_3, x_4, and the output items y_1, y_2, y_3, y_4 correspond to the sorted sequence.

The **size** of a comparator network is the number of comparators used in the network; the **depth** is the length of the longest path from an input to an output. The sorting network shown in Fig. 4.11(b) is of size 5 and depth 3.

Our main goal in this section is to design comparator networks that sort in small depth and that have a small size. Given a comparator network that sorts, we can easily derive a parallel algorithm whose running time is asymptotically the same as the depth of the network, and whose total number of operations is asymptotically the same as the size of the network.