would be to consider programs in nonincreasing order of a_i. If there is enough space left on the tape for P_i, then it is included in Q. Assume the programs are ordered so that $a_1 \geq a_2 \geq \cdots \geq a_n$. Write a function incorporating this strategy. What is its time and space complexity?

(e) Show that the strategy of part (d) doesn't necessarily yield a subset that maximizes $(\sum_{P_i \in Q} a_i)/l$. How small can this ratio get? Prove your bound.

4. Assume n programs of lengths $l_1, l_2, \ldots, l_n$ are to be stored on a tape. Program i is to be retrieved with frequency f_i. If the programs are stored in the order $i_1, i_2, \ldots, i_n$, the *expected retrieval time* (ERT) is

$$\left[\sum_j (f_{i_j} \sum_{k=1}^{j} l_{i_k})\right] / \sum f_i$$

(a) Show that storing the programs in nondecreasing order of l_i does not necessarily minimize the ERT.

(b) Show that storing the programs in nonincreasing order of f_i does not necessarily minimize the ERT.

(c) Show that the ERT is minimized when the programs are stored in nonincreasing order of f_i/l_i.

5. Consider the tape storage problem of this section. Assume that two tapes $T1$ and $T2$, are available and we wish to distribute n given programs of lengths $l_1, l_2, \ldots, l_n$ onto these two tapes in such a manner that the maximum retrieval time is minimized. That is, if A and B are the sets of programs on the tapes $T1$ and $T2$ respectively, then we wish to choose A and B such that max $\{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$ is minimized. A possible greedy approach to obtaining A and B would be to start with A and B initially empty. Then consider the programs one at a time. The program currently being considered is assigned to set A if $\sum_{i \in A} l_i$ = min $\{ \sum_{i \in A} l_i, \sum_{i \in B} l_i \}$; otherwise it is assigned to B. Show that this does not guarantee optimal solutions even if $l_1 \leq l_2 \leq \cdots \leq l_n$. Show that the same is true if we require $l_1 \geq l_2 \geq \cdots \geq l_n$.

4.7 OPTIMAL MERGE PATTERNS

In Section 3.4 we saw that two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time $O(n+m)$. When more than two sorted files are to be merged together, the merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if files x_1, x_2, x_3, and x_4 are to be merged, we could first merge x_1 and x_2 to get a file y_1. Then we could merge y_1 and x_3 to get y_2. Finally, we could merge y_2 and x_4 to get the desired sorted file. Alternatively, we could first merge x_1 and x_2 getting y_1, then merge x_3 and x_4 and get y_2, and finally merge y_1 and y_2 and get the desired sorted file. Given n sorted files, there are many ways in which to pairwise merge them into a single sorted file. Different pairings require differing amounts of computing time. The problem we address ourselves to now is that of determining an optimal way (one requiring the fewest comparisons) to pairwise merge n sorted files. Since this problem calls for an ordering among the pairs to be merged, it fits the ordering paradigm.

Example 4.9 The files x_1, x_2, and x_3 are three sorted files of length $30, 20$, and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If, instead, we first merge x_2 and x_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first. □

A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n-record file and an m-record file requires possibly $n + m$ record moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together. Thus, if we have five files $(x_1, \ldots, x_5)$ with sizes $(20, 30, 10, 5, 30)$, our greedy rule would generate the following merge pattern: merge x_4 and x_3 to get z_1 ($|z_1| = 15$), merge z_1 and x_1 to get z_2 ($|z_2| = 35$), merge x_2 and x_5 to get z_3 ($|z_3| = 60$), and merge z_2 and z_3 to get the answer z_4. The total number of record moves is 205. One can verify that this is an optimal merge pattern for the given problem instance.

The merge pattern such as the one just described will be referred to as a *two-way merge pattern* (each merge step involves the merging of two files). The two-way merge patterns can be represented by binary merge trees. Figure 4.11 shows a binary merge tree representing the optimal merge pattern obtained for the above five files. The leaf nodes are drawn as squares and represent the given five files. These nodes are called *external nodes.* The remaining nodes are drawn as circles and are called *internal nodes.* Each internal node has exactly two children, and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e., the number of records) of the file represented by that node.

The external node x_4 is at a distance of 3 from the root node z_4 (a node at level i is at a distance of $i - 1$ from the root). Hence, the records of file x_4 are moved three times, once to get z_1, once again to get z_2, and finally one more time to get z_4. If d_i is the distance from the root to the external

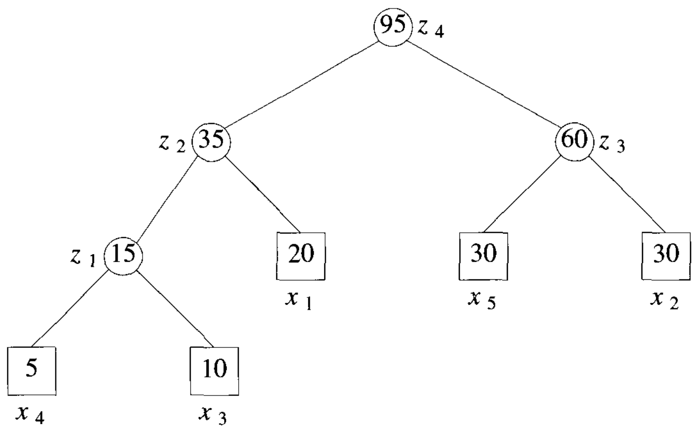

Figure 4.11 Binary merge tree representing a merge pattern

node for file x_i and q_i, the length of x_i is then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^{n} d_i q_i$$

This sum is called the *weighted external path length* of the tree.

An optimal two-way merge pattern corresponds to a binary merge tree with minimum weighted external path length. The function Tree of Algorithm 4.13 uses the greedy rule stated earlier to obtain a two-way merge tree for n files. The algorithm has as input a list *list* of n trees. Each node in a tree has three fields, *lchild*, *rchild*, and *weight*. Initially, each tree in *list* has exactly one node. This node is an external node and has *lchild* and *rchild* fields zero whereas *weight* is the length of one of the n files to be merged. During the course of the algorithm, for any tree in *list* with root node t, $t \rightarrow$ *weight* is the length of the merged file it represents ($t \rightarrow$ *weight* equals the sum of the lengths of the external nodes in tree t). Function Tree uses two functions, Least(*list*) and Insert(*list*, t). Least(*list*) finds a tree in *list* whose root has least *weight* and returns a pointer to this tree. This tree is removed from *list*. Insert(*list*, t) inserts the tree with root t into *list*. Theorem 4.10 shows that Tree (Algorithm 4.13) generates an optimal two-way merge tree.

```
    treenode = record {
        treenode * lchild; treenode * rchild;
        integer weight;
    };

1   Algorithm Tree($n$)
2   // list is a global list of $n$ single node
3   // binary trees as described above.
4   {
5       for $i := 1$ to $n - 1$ do
6       {
7           $pt$ := new treenode; // Get a new tree node.
8           $(pt \to lchild)$ := Least($list$); // Merge two trees with
9           $(pt \to rchild)$ := Least($list$); // smallest lengths.
10          $(pt \to weight) := ((pt \to lchild) \to weight)$
11                  $+((pt \to rchild) \to weight)$;
12          Insert($list, pt$);
13      }
14      return Least($list$); // Tree left in list is the merge tree.
15  }
```

Algorithm 4.13 Algorithm to generate a two-way merge tree

Example 4.10 Let us see how algorithm Tree works when *list* initially represents six files with lengths $(2, 3, 5, 7, 9, 13)$. Figure 4.12 shows *list* at the end of each iteration of the **for** loop. The binary merge tree that results at the end of the algorithm can be used to determine which files are merged. Merging is performed on those files which are lowest (have the greatest depth) in the tree. □

The main **for** loop in Algorithm 4.13 is executed $n - 1$ times. If *list* is kept in nondecreasing order according to the *weight* value in the roots, then Least($list$) requires only $O(1)$ time and Insert($list, t$) can be done in $O(n)$ time. Hence the total time taken is $O(n^2)$. In case *list* is represented as a minheap in which the root value is less than or equal to the values of its children (Section 2.4), then Least($list$) and Insert($list, t$) can be done in $O(\log n)$ time. In this case the computing time for Tree is $O(n \log n)$. Some speedup may be obtained by combining the Insert of line 12 with the Least of line 9.