the pairs of adjacent vertices in the polygon vertex lists. Edges may instead be represented explicitly as pairs of vertices, with each face now defined as a list of indices into the list of edges. These representations were discussed in more detail in Section 11.1.1.

12.5.2 The Winged-Edge Representation

Simple representations make certain computations quite expensive. For example, discovering the two faces shared by an edge (e.g., to help prove that a representation encodes a valid solid) requires searching the edge lists of all the faces. More complex b-reps have been designed to decrease the cost of these computations. One of the most popular is the winged-edge data structure developed by Baumgart [BAUM72; BAUM75]. As shown in Fig. 12.16, each edge in the winged-edge data structure is represented by pointers to its two vertices, to the two faces sharing the edge, and to four of the additional edges emanating from its vertices. Each vertex has a backward pointer to one of the edges emanating from it, whereas each face points to one of its edges. Note that we traverse the vertices of an edge in opposite directions when following the vertices of each of its two faces in clockwise order. Labeling the edge’s vertices \(n\) and \(p\), we refer to the face to its right when traversing the edge from \(n\) to \(p\) as its \(p\) face, and the face to its right when traversing the edge from \(p\) to \(n\) as its \(n\) face. For edge \(E_1\) in Fig. 12.16, if \(n\) is \(V_1\) and \(p\) is \(V_2\), then \(F_1\) is \(E_1\)’s \(p\) face, and \(F_2\) is its \(n\) face. The four edges to which each edge points can be classified as follows. The two edges that share the edge’s \(n\) vertex are the next (clockwise) edge of the \(n\) face, and the previous (counterclockwise) edge of the \(p\) face, \(E_3\) and \(E_2\), respectively. The two edges that share the edge’s \(p\) vertex are the next (clockwise) edge of the \(p\) face, and the previous (counterclockwise) edge of the \(n\) face, \(E_4\) and \(E_5\), respectively. These four edges are the “wings” from which the winged-edge data structure gets its name.

Note that the data structure described here handles only faces that have no holes. This limitation can be removed by representing each face as a set of edge loops—a clockwise outer loop and zero or more counterclockwise inner loops for its holes—as described in...
Section 19.1. Alternatively, a special auxiliary edge can be used to join each hole's boundary to the outer boundary. Each auxiliary edge is traversed twice, once in each direction, when a circuit of its face's edges is completed. Since an auxiliary edge has the same face on both of its sides, it can be easily identified because its two face pointers point to the same face.

A b-rep allows us to query which faces, edges, or vertices are adjacent to each face, edge, or vertex. These queries correspond to nine kinds of adjacency relationships. The winged-edge data structure makes it possible to determine in constant time which vertices or faces are associated with an edge. It takes longer to compute other adjacency relationships. One attractive property of the winged edge is that the data structures for the edges, faces, and vertices are each of a small, constant size. Only the number of instances of each data structure varies among objects. Weiler [WEIL85] and Woo [WOO85] discuss the space-time efficiency of the winged edge and a variety of alternative b-rep data structures.

### 12.5.3 Boolean Set Operations

B-reps may be combined, using the regularized Boolean set operators, to create new b-reps [REQU85]. Sarraga [SARR83] and Miller [MILL87] discuss algorithms that determine the intersections between quadric surfaces. Algorithms for combining polyhedral objects are presented in [TURN84, REQU85, PUTN86, LAID86], and Thibault and Naylor [THIB87] describe a method based on the binary space-partitioning tree representation of solids discussed in Section 12.6.4.

One approach [LAID86] is to inspect the polygons of both objects, splitting them if necessary to ensure that the intersection of a vertex, edge, or face of one object with any vertex, edge, or face of another, is a vertex, edge, or face of both. The polygons of each object are then classified relative to the other object to determine whether they lie inside, outside, or on its boundary. Referring back to Table 12.1, we note that since this is a b-rep, we are concerned with only the last six rows, each of which represents some part of one or both of the original object boundaries, A_b and B_b. After splitting, each polygon of one object is either wholly inside the other object (A_b \cap B_b) or wholly outside the other object (A_b - B_b or B_b - A_b), or part of the shared boundary (A_b \cap B_b same or A_b \cap B_b diff).

A polygon may be classified by the ray-casting technique discussed in Section 15.10.1. Here, we construct a vector in the direction of the polygon’s surface normal from a point on the polygon’s interior, and then find the closest polygon that intersects the vector in the other object. If no polygon is intersected, the original polygon is outside the other object. If the closest intersecting polygon is coplanar with the original polygon, then this is a boundary–boundary intersection, and comparing polygon normals indicates what kind of intersection it is (A_b \cap B_b same or A_b \cap B_b diff). Otherwise, the dot product of the two polygons’ normals is inspected. A positive dot product indicates that the original polygon is inside the other object, whereas a negative dot product indicates that it is outside. A zero dot product occurs if the vector is in the plane of the intersected polygon; in this case, the vector is perturbed slightly and is intersected again with the other object’s polygons.

Vertex-adjacency information can be used to avoid the overhead of classifying each polygon in this way. If a polygon is adjacent to (i.e., shares vertices with) a classified...
what the answer to the following simple HW question might look like:

Q: Write pseudocode that takes a ptr to an edge & outputs all adjacent faces for the Requicha b-rep.

A: Function FindFacesFromEdge (input: "InputEdge Ptr" as specified in problem,
and assume also given as input "Object Boundary Ptr" which
points to the first face) You can assume this for the Requicha
problem in the HW too.

Face_Ptr = Object Boundary_Ptr;
do {
    assignment operator (use your favorite programming language syntax)
    if Face_Ptr -> face -> 1st edge == Input Edge_Ptr
        or follow ptr. with this name comparing if equal to (use your favorite programming language syntax)
    else if Face_Ptr -> face -> 2nd edge == Input Edge_Ptr
        or
    else if Face_Ptr -> face -> 3rd edge == Input Edge_Ptr
        add Face_Ptr -> face to output;
    Face_Ptr = Face_Ptr -> next
} while Face_Ptr ≠ NULL;

output the output.