```
procedure boundary(s^d, M, M_b)
  if not findData(M, s^d) then
    addData(M, s^d, true)
    for every f in faces(s^d) do
      s'^d = up(f)
      if (s'^d = null) then add(M_b, f)
      else boundary(s'^d, M, M_b)
```

Fig. 2. Pseudocode for computing the $d-1$ boundary M_b of a d simplicial mesh M.

returns the user data ud associated with s^k in M. If there is no associated data then **null** is returned.

The interface as described can be used for most applications that traverse and update a simplicial mesh. Figure 2 gives an example of code that traverses a d-simplicial mesh with boundary and returns the $(d-1)$ boundary mesh. It recursively traverses the mesh in depth-first order storing flags on the d simplices when visited. Whenever a boundary $d-1$ simplex is found (s'^d = **null** in the code), it is added to the output mesh. The code assumes the boundary is a simplicial mesh.

5. Data Structure

Here we describe our 2D and 3D data structures for simplicial meshes (simplicial orientable pseudo manifolds). We first describe uncompressed versions of the data structures and then describe how to compress them. Our data structures are based on storing the link for a set of $(d-2)$-simplices. In 2D this is similar to the half-edge structure,[30] and in 3D it is similar to the Dobkin and Laszlo[34] structure. We note, however, that all references are to vertex labels instead of pointers to other higher-dimensional simplex structures, allowing us to compress based on vertex labels. Our data structures have the property that if the degree of all vertices is bounded all queries take constant time. We first describe a version for manifold complexes.

Our 2D data structure maps each vertex to its link, represented as a cycle of the labels of its neighboring vertices. The cycle is ordered radially around the vertex in the orientation of the complex, *e.g.*, clockwise. A **findUp** query on the ordered edge (v_1, v_2) can be answered by looking up the link for v_1, finding v_2 in the link, and returning the next vertex in the link. A **findUp** on a vertex can be answered by selecting the first two vertices off of its link.

The link can be stored as a list of labels starting at an arbitrary point on the cycle. If the vertex has bounded degree, the lookup takes constant time. To analyze the space note that each edge appears in two cycles, and each appearance requires two pointers, one to the vertex label and one to the next element in the list. The total space is therefore 4 pointers/edge + 1 pointer/vertex. This is identical in space usage to the triangle-based structure, assuming that it also maintains a pointer from