it is necessary to search down the list of free blocks finding the first block of size $\geq N$ and allocating N words out of this block. Such an allocation strategy is called *first fit*. The algorithm below makes storage allocations using the first fit strategy. An alternate strategy, *best fit*, calls for finding a free block whose size is as close to N as possible, but not less than N. This strategy is examined in the exercises.

```
procedure FF(n, p)
  //AV points to the available space list which is searched for a node
    of size at least n. p is set to the address of a block of size n
    that may be allocated. If there is no block of that size then p = 0.
    It is assumed that the free list has a head node with SIZE field = 0//
  p ← LINK(AV); q ← AV
  while p ≠ 0 do
    if SIZE(p) ≥ n then [SIZE(p) ← SIZE(p) − n
                         if SIZE(p) = 0 then LINK(q) ← LINK(p)
                                        else p ← p + SIZE(p)
                         return]
    q ← p; p ← LINK(p)
  end
  //no block is large enough//
  end FF
```

This algorithm is simple enough to understand. In case only a portion of a free block is to be allocated, the allocation is made from the bottom of the block. This avoids changing any links in the free list unless an entire block is allocated. There are, however, two major problems with *FF*. First, experiments have shown that after some processing time many small nodes are left in the available space list, these nodes being smaller than any requests that would be made. Thus, a request for 9900 words allocated from a block of size 10,000 would leave behind a block of size 100, which may be smaller than any requests that will be made to the system. Retaining these small nodes on the available space list tends to slow down the allocation process as the time needed to make an allocation is proportional to the number of nodes on the available space list. To get around this, we choose some suitable constant ϵ such that if the allocation of a portion of a node leaves behind a node of size $<\epsilon$, then the entire node is allocated. I.e., we allocate more storage than requested in this case. The second problem arises from the fact that the search for a large enough node always begins at the front of the list. As a result of this, all the small nodes tend

to collect at the front so that it is necessary to examine several nodes before an allocation for larger blocks can be made. In order to distribute small nodes evenly along the list, one can begin searching for a new node from a different point in the list each time a request is made. To implement this, the available space list is maintained as a circular list with a head node of size zero. AV now points to the last node from which an allocation was made. We shall see what the new allocation algorithm looks like after we discuss what has to be done to free a block of storage.

The second operation is the freeing of blocks or returning nodes to AV. Not only must we return the node but we also want to recognize if its neighbors are also free so that they can be coalesced into a single block. Looking back at figure 4.15, we see that if $P3$ is the next program to terminate, then rather than just adding this node onto the free list to get the free list of figure 4.17, it would be better to combine the adjacent free blocks corresponding to $P2$ and $P4$, obtaining the free list of figure 4.18. This combining of adjacent free blocks to get bigger free blocks is necessary. The block allocation algorithm splits big blocks

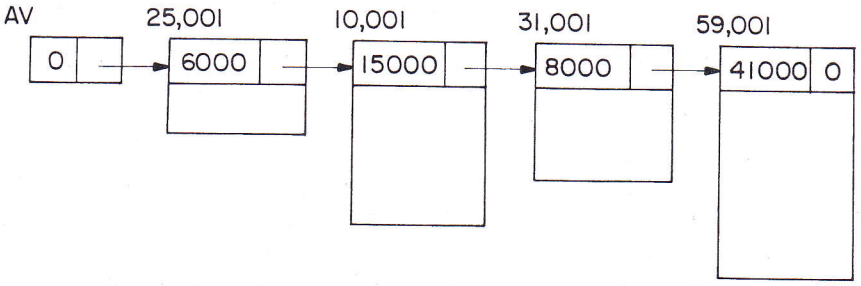


Figure 4.17 Available Space List When Adjacent Free Blocks Are Not Coalesced.

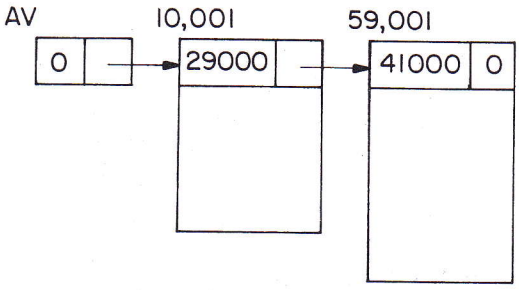


Figure 4.18 Available Space List When Adjacent Free Blocks Are Coalesced.

while making allocations. As a result, available block sizes get smaller and smaller. Unless recombination takes place at some point, we will no longer be able to meet large requests for memory.

With the structure we have for the available space list, it is not easy to determine whether blocks adjacent to the block (n, p) (n = size of block and p = starting location) being returned are free. The only way to do this, at present, is to examine all the nodes in AV to determine whether:

(i) the left adjacent block is free, i.e., the block ending at $p - 1$;

(ii) the right adjacent block is free, i.e., the block beginning at $p + n$.

In order to determine (i) and (ii) above without searching the available space list, we adopt the node structure of figure 4.19 for allocated and free nodes:

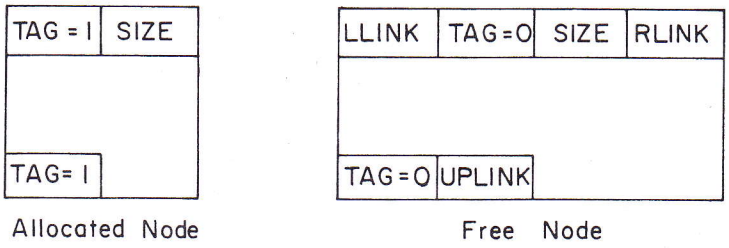
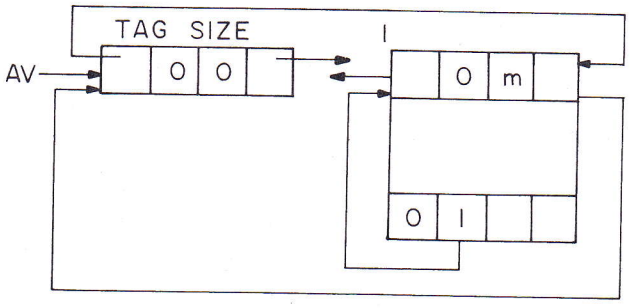


Figure 4.19

The first and last words of each block are reserved for allocation information. The first word of each free block has four fields: LLINK, RLINK, TAG and SIZE. Only the TAG and SIZE field are important for a block in use. The last word in each free block has two fields: TAG and UPLINK. Only the TAG field is important for a block in use. Now by just examining the tag at $p - 1$ and $p + n$ one can determine whether the adjacent blocks are free. The UPLINK field of a free block points to the start of the block. The available space list will now be a doubly linked circular list, linked through the fields LLINK and RLINK. It will have a head node with SIZE = 0. A doubly linked list is needed, as the return block algorithm will delete nodes at random from AV. The need for UPLINK will become clear when we study the freeing algorithm. Since the first and last nodes of each block have TAG fields, this system of allocation and freeing is called the *Boundary Tag method.* It should be noted that the TAG fields in allocated and free blocks occupy the same bit position in the first and last words respectively. This is not obvious from figure 4.19 where the LLINK

field precedes the TAG field in a free node. The labeling of fields in this figure has been done so as to obtain clean diagrams for the available space list. The algorithms we shall obtain for the boundary tag method will assume that memory is numbered 1 to m and that TAG(0) = TAG(m + 1) = 1. This last requirement will enable us to free the block beginning at 1 and the one ending at m without having to test for these blocks as special cases. Such a test would otherwise have been necessary as the first of these blocks has no left adjacent block while the second has no right adjacent block. While the TAG information is all that is needed in an allocated block, it is customary to also retain the size in the block. Hence, figure 4.19 also includes a SIZE field in an allocated block.

Before presenting the allocate and free algorithms let us study the initial condition of the system when all of memory is free. Assuming memory begins at location 1 and ends at m, the AV list initially looks like:



Programs on next two pages

While these algorithms may appear complex, they are a direct consequence of the doubly linked list structure of the available space list and also of the node structure in use. Notice that the use of a head node eliminates the test for an empty list in both algorithms and hence simplifies them. The use of circular linking makes it easy to start the search for a large enough node at any point in the available space list. The UPLINK field in a free block is needed only when returning a block whose left adjacent block is free (see lines 18 and 24 of algorithm FREE). The readability of algorithm FREE has been greatly enhanced by the use of the **case** statement. In lines 20 and 28 AV is changed so that it always points to the start of a free block rather than into

```
     procedure ALLOCATE (n,p)
       //Use first fit to allocate a block of memory of size at least
         n, n > 0. The available space list is maintained as described
         above and it is assumed that no blocks of size < ϵ are to
         be retained. p is set to be the address of the first word in
         the block allocated. AV points to a node on the available list.//
1      p ← RLINK(AV)          //begin search at p//
2      repeat
3        if SIZE (p) ≥ n then          //block is big enough//
4          [diff ← SIZE(p) − n
5          if diff < ϵ then          //allocate whole block//
6            [RLINK (LLINK(p)) ← RLINK(p)          //delete node
                                                     from AV//
7             LLINK (RLINK(p)) ← LLINK(p)
8             TAG(p) ← TAG(p + SIZE(p) − 1) ← 1          //set
                                                          tags//
9             AV ← LLINK(p)          //set starting point of next
                                       search//
10            return]
11                 else          //allocate lower n words//
12           [SIZE(p) ← diff
13            UPLINK(p + diff − 1) ← p
14            TAG(p + diff − 1) ← 0          //set upper portion as
                                             unused//
15            AV ← p          //position for next search//
16            p ← p + diff          //set p to point to start of allocated
                                     block//
17            SIZE(p) ← n
18            TAG(p) ← TAG(p + n − 1) ← 1
                                     //set tags for allocated block//
19            return]]
20         p ← RLINK(p)          //examine next node on list//
21     until p =
22     //no block large enough//
23     p ← 0;
24   end ALLOCATE
```

```
    procedure FREE(p)
      //return a block beginning at p and of size SIZE(p)//
1     n ← SIZE(p)
2     case
3       :TAG(p − 1) = 1 and TAG(p + n) = 1:
                                        //both adjacent blocks in use//
4         TAG(p) ← TAG(p + n − 1) ← 0        //set up a free
                                               block//
5         UPLINK (p + n − 1) ← p
6         LLINK(p) ← AV; RLINK(p) ← RLINK(AV)
                                        //insert at right of AV//
7         LLINK(RLINK(p)) ← p; RLINK(AV) ← p
8       :TAG(p + n) = 1 and TAG(p − 1) = 0:        //only left block
                                                     free//
9         q ← UPLINK(p − 1)          //start of left block//
10        SIZE(q) ← SIZE(q) + n
11        UPLINK(p + n − 1) ← q; TAG(p + n − 1) ← 0
12      :TAG(p + n) = 0 and TAG(p − 1) = 1:
                                  //only right adjacent block free//
13        RLINK(LLINK(p + n)) ← p          //replace block
                                             beginning//
14        LLINK(RLINK(p + n)) ← p          //at p + n by one//
15        LLINK(p) ← LLINK(p + n)          //beginning at p//
16        RLINK(p) ← RLINK(p + n)
17        SIZE(p) ← n + SIZE(p + n)
18        UPLINK(p + SIZE(p) − 1) ← p
19        TAG(p) ← 0
20        if AV = p + n then AV ← p
21        :else: //both adjacent blocks free//
22               //delete right free block from AV list//
                 RLINK(LLINK(p + n)) ←
23               LLINK(RLINK(p + n)) ← LLINK(p + n)
24               q ← UPLINK(p − 1)          //start of left free
                                              block//
25               SIZE(q) ←
26               UPLINK(q + SIZE(q) − 1) ← q
27               if AV = p + n then AV ← LLINK(p + n)
28    end
29  end FREE
```