**C5.** [Add to AVAIL list.] Set SIZE(P − 1) ← SIZE(P0), LINK(P0) ← AVAIL, LINK(P0 + 1) ← LOC(AVAIL), LINK(AVAIL + 1) ← P0, AVAIL ← P0, TAG(P0) ← TAG(P − 1) ← "−". ▮

The steps of Algorithm C are straightforward consequences of the storage layout (7); a slightly longer algorithm that is a little faster appears in exercise 15. In step C5, AVAIL is an abbreviation for LINK(LOC(AVAIL)), as shown in (9).

**C. The "buddy system."** We will now study another approach to dynamic storage allocation, suitable for use with binary computers. This method uses one bit of overhead in each block, and it requires all blocks to be of length 1, 2, 4, 8, or 16, etc. If a block is not $2^k$ words long for some integer $k$, the next higher power of 2 is chosen and extra unused space is allocated accordingly.

The idea of this method is to keep separate lists of available blocks of each size $2^k$, $0 \le k \le m$. The entire pool of memory space under allocation consists of $2^m$ words, which can be assumed to have the addresses 0 through $2^m - 1$. Originally, the entire block of $2^m$ words is available. Later, when a block of $2^k$ words is desired, and if nothing of this size is available, a larger available block is *split* into two equal parts; ultimately, a block of the right size $2^k$ will appear. When one block splits into two (each of which is half as large as the original), these two blocks are called *buddies*. Later when both buddies are available again, they coalesce back into a single block; thus the process can be maintained indefinitely, unless we run out of space at some point.

The key fact underlying the practical usefulness of this method is that if we know the address of a block (the memory location of its first word), and if we also know the size of that block, we know the address of its buddy. For example, the buddy of the block of size 16 beginning in binary location 101110010110000 is a block starting in binary location 101110010100000. To see why this must be true, we first observe that as the algorithm proceeds, *the address of a block of size $2^k$ is a multiple of $2^k$*. In other words, the address in binary notation has at least $k$ zeros at the right. This observation is easily justified by induction: If it is true for all blocks of size $2^{k+1}$, it is certainly true when such a block is halved.

Therefore a block of size, say, 32 has an address of the form $xx\ldots x00000$ (where the $x$'s represent either 0 or 1); if it is split, the newly formed buddy blocks have the addresses $xx\ldots x00000$ and $xx\ldots x10000$. In general, let $\text{buddy}_k(x) =$ address of the buddy of the block of size $2^k$ whose address is $x$; we find that

$$\text{buddy}_k(x) = \begin{cases} x + 2^k, & \text{if } x \bmod 2^{k+1} = 0; \\ x - 2^k, & \text{if } x \bmod 2^{k+1} = 2^k. \end{cases} \tag{10}$$

This function is readily computed with the "exclusive or" instruction (sometimes called "selective complement" or "add without carry") usually found on binary computers; see exercise 28.

The buddy system makes use of a one-bit TAG field in each block:

$$\begin{aligned} &\texttt{TAG(P)} = 0, && \text{if the block with address P is reserved;} \\ &\texttt{TAG(P)} = 1, && \text{if the block with address P is available.} \end{aligned} \tag{11}$$

Besides this TAG field, which is present in all blocks, *available* blocks also have two link fields, LINKF and LINKB, which are the usual forward and backward links of a doubly linked list; and they also have a KVAL field to specify $k$ when their size is $2^k$. The algorithms below make use of the table locations AVAIL[0], AVAIL[1], ..., AVAIL[$m$], which serve respectively as the heads of the lists of available storage of sizes 1, 2, 4, ..., $2^m$. These lists are doubly linked, so as usual the list heads contain two pointers (see Section 2.2.5):

$$\begin{aligned} \texttt{AVAILF}[k] &= \texttt{LINKF}(\texttt{LOC}(\texttt{AVAIL}[k])) = \text{link to rear of } \texttt{AVAIL}[k] \text{ list;} \\ \texttt{AVAILB}[k] &= \texttt{LINKB}(\texttt{LOC}(\texttt{AVAIL}[k])) = \text{link to front of } \texttt{AVAIL}[k] \text{ list.} \end{aligned} \tag{12}$$

Initially, before any storage has been allocated, we have

$$\begin{aligned} \texttt{AVAILF}[m] &= \texttt{AVAILB}[m] = 0, \\ \texttt{LINKF}(0) &= \texttt{LINKB}(0) = \texttt{LOC}(\texttt{AVAIL}[m]), \\ \texttt{TAG}(0) &= 1, \qquad \texttt{KVAL}(0) = m \end{aligned} \tag{13}$$

(indicating a single available block of length $2^m$, beginning in location 0), and

$$\texttt{AVAILF}[k] = \texttt{AVAILB}[k] = \texttt{LOC}(\texttt{AVAIL}[k]), \qquad \text{for } 0 \le k < m \tag{14}$$

(indicating empty lists for available blocks of lengths $2^k$ for all $k < m$).

From this description of the buddy system, the reader may find it enjoyable to design the necessary algorithms for reserving and freeing storage areas before looking at the algorithms given below. Notice the comparative ease with which blocks can be halved in the reservation algorithm.

**Algorithm R** (*Buddy system reservation*). This algorithm finds and reserves a block of $2^k$ locations, or reports failure, using the organization of the buddy system as explained above.

**R1.** [Find block.] Let $j$ be the smallest integer in the range $k \le j \le m$ for which AVAILF[$j$] $\neq$ LOC(AVAIL[$j$]), that is, for which the list of available blocks of size $2^j$ is not empty. If no such $j$ exists, the algorithm terminates unsuccessfully, since there are no known available blocks of sufficient size to meet the request.

**R2.** [Remove from list.] Set L $\leftarrow$ AVAILF[$j$], P $\leftarrow$ LINKF(L), AVAILF[$j$] $\leftarrow$ P, LINKB(P) $\leftarrow$ LOC(AVAIL[$j$]), and TAG(L) $\leftarrow$ 0.

**R3.** [Split required?] If $j = k$, the algorithm terminates (we have found and reserved an available block starting at address L).

**R4.** [Split.] Decrease $j$ by 1. Then set P $\leftarrow$ L + $2^j$, TAG(P) $\leftarrow$ 1, KVAL(P) $\leftarrow$ $j$, LINKF(P) $\leftarrow$ LINKB(P) $\leftarrow$ LOC(AVAIL[$j$]), AVAILF[$j$] $\leftarrow$ AVAILB[$j$] $\leftarrow$ P. (This splits a large block and enters the unused half in the AVAIL[$j$] list, which was empty.) Go back to step R3. ▮

**Algorithm S** (*Buddy system liberation*). This algorithm returns a block of $2^k$ locations, starting in address L, to free storage, using the organization of the buddy system as explained above.

**S1.** [Is buddy available?] Set P $\leftarrow$ buddy$_k$(L). (See Eq. (10).) If $k = m$ or if TAG(P) $= 0$, or if TAG(P) $= 1$ and KVAL(P) $\neq k$, go to S3.

**S2.** [Combine with buddy.] Set

$$\texttt{LINKF(LINKB(P))} \leftarrow \texttt{LINKF(P)}, \qquad \texttt{LINKB(LINKF(P))} \leftarrow \texttt{LINKB(P)}.$$

(This removes block P from the AVAIL[$k$] list.) Then set $k \leftarrow k + 1$, and if P $<$ L set L $\leftarrow$ P. Return to S1.

**S3.** [Put on list.] Set TAG(L) $\leftarrow 1$, P $\leftarrow$ AVAILF[$k$], LINKF(L) $\leftarrow$ P, LINKB(P) $\leftarrow$ L, KVAL(L) $\leftarrow k$, LINKB(L) $\leftarrow$ LOC(AVAIL[$k$]), AVAILF[$k$] $\leftarrow$ L. (This puts block L on the AVAIL[$k$] list.) ▮

**D. Comparison of the methods.** The mathematical analysis of these dynamic storage-allocation algorithms has proved to be quite difficult, but there is one interesting phenomenon that is fairly easy to analyze, namely the "fifty-percent rule":

> *If Algorithms A and B are used continually in such a way that the system tends to an equilibrium condition, where there are $N$ reserved blocks in the system, on the average, each equally likely to be the next one deleted, and where the quantity K in Algorithm A takes on nonzero values (or, more generally, values $\geq c$ as in step A4′) with probability $p$, then the average number of available blocks tends to approximately $\frac{1}{2}pN$.*

This rule tells us approximately how long the AVAIL list will be. When the quantity $p$ is near 1 — this will happen if $c$ is very small and if the block sizes are infrequently equal to each other — we have about half as many available blocks as unavailable ones; hence the name "fifty-percent rule."

It is not hard to derive this rule. Consider the following memory map:


This shows the reserved blocks divided into three categories:

$A$: when freed, the number of available blocks will decrease by one;
$B$: when freed, the number of available blocks will not change;
$C$: when freed, the number of available blocks will increase by one.

Now let $N$ be the number of reserved blocks, and let $M$ be the number of available ones; let $A$, $B$, and $C$ be the number of blocks of the types identified above. We have

$$\begin{aligned} N &= A + B + C \\ M &= \tfrac{1}{2}(2A + B + \epsilon) \end{aligned} \tag{15}$$

where $\epsilon = 0$, 1, or 2 depending on conditions at the lower and upper boundaries.

Let us assume that $N$ is essentially constant, but that $A$, $B$, $C$, and $\epsilon$ are random quantities that reach a stationary distribution after a block is freed and a (slightly different) stationary distribution after a block is allocated. The average change in $M$ when a block is freed is the average value of $(C - A)/N$; the average change in $M$ when a block is allocated is $1 - p$. So the equilibrium assumption