12

Lists and Arrays Revisited

Simple lists and arrays are the right tools for the many applications. Other situations require support for operations that cannot be implemented efficiently by the standard list representations of Chapter 4. This chapter presents a range of topics, whose unifying thread is that the data structures included are all list- or array-like. These structures overcome some of the problems of simple linked list and contiguous array representations. This chapter also seeks to reinforce the concept of logical representation versus physical implementation, as some of the "list" implementations have quite different organizations internally.

Section 12.1 describes a series of representations for multilists, which are lists that may contain sublists. Section 12.2 discusses representations for implementing sparse matrices, large matrices where most of the elements have zero values. Section 12.3 discusses memory management techniques, which are essentially a way of allocating variable-length sections from a large array.

12.1 Multilists

Recall from Chapter 4 that a list is a finite, ordered sequence of items of the form $\langle x_0, x_1, ..., x_{n-1} \rangle$ where $n \geq 0$. We can represent the empty list by **`null`** or $\langle\rangle$. In Chapter 4 we assumed that all list elements had the same data type. In this section, we extend the definition of lists to allow elements to be arbitrary in nature. In general, list elements are one of two types.

1. An **atom**, which is a data record of some type such as a number, symbol, or string.
2. Another list, which is called a **sublist**.

A list containing sublists will be written as

$$\langle \mathit{x1}, \langle \mathit{y1}, \langle \mathit{a1}, \mathit{a2} \rangle, \mathit{y3} \rangle, \langle \mathit{z1}, \mathit{z2} \rangle, \mathit{x4} \rangle.$$

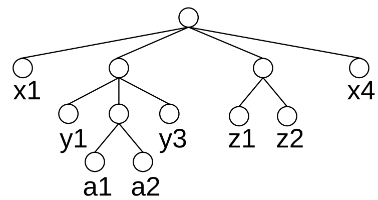

Figure 12.1 Example of a multilist represented by a tree.

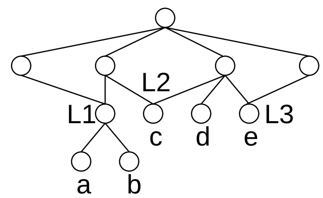

Figure 12.2 Example of a reentrant multilist. The shape of the structure is a DAG (all edges point downward).

In this example, the list has four elements. The second element is the sublist $\langle y1, \langle a1, a2\rangle, y3\rangle$ and the third is the sublist $\langle z1, z2\rangle$. The sublist $\langle y1, \langle a1, a2\rangle, y3\rangle$ itself contains a sublist. If a list **L** has one or more sublists, we call **L** a **multilist**. Lists with no sublists are often referred to as **linear lists** or **chains**. Note that this definition for multilist fits well with our definition of sets from Definition 2.1, where a set's members can be either primitive elements or sets.

We can restrict the sublists of a multilist in various ways, depending on whether the multilist should have the form of a tree, a DAG, or a generic graph. A **pure list** is a list structure whose graph corresponds to a tree, such as in Figure 12.1. In other words, there is exactly one path from the root to any node, which is equivalent to saying that no object may appear more than once in the list. In the pure list, each pair of angle brackets corresponds to an internal node of the tree. The members of the list correspond to the children for the node. Atoms on the list correspond to leaf nodes.

A **reentrant list** is a list structure whose graph corresponds to a DAG. Nodes might be accessible from the root by more than one path, which is equivalent to saying that objects (including sublists) may appear multiple times in the list as long as no cycles are formed. All edges point downward, from the node representing a list or sublist to its elements. Figure 12.2 illustrates a reentrant list. To write out this list in bracket notation, we can duplicate nodes as necessary. Thus, the bracket notation for the list of Figure 12.2 could be written

$$\langle\langle\langle a, b\rangle\rangle, \langle\langle a, b\rangle, c\rangle, \langle c, d, e\rangle, \langle e\rangle\rangle.$$

For convenience, we will adopt a convention of allowing sublists and atoms to be labeled, such as "*L1*:". Whenever a label is repeated, the element corresponding to

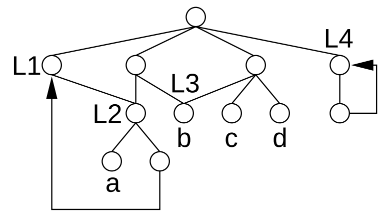

Figure 12.3 Example of a cyclic list. The shape of the structure is a directed graph.

that label will be substituted when we write out the list. Thus, the bracket notation for the list of Figure 12.2 could be written

$$\langle\langle L1 : \langle a, b\rangle\rangle, \langle L1, L2 : c\rangle, \langle L2, d, L3 : e\rangle, \langle L3\rangle\rangle.$$

A **cyclic list** is a list structure whose graph corresponds to any directed graph, possibly containing cycles. Figure 12.3 illustrates such a list. Labels are required to write this in bracket notation. Here is the bracket notation for the list of Figure 12.3.

$$\langle L1 : \langle L2 : \langle a, L1\rangle\rangle, \langle L2, L3 : b\rangle, \langle L3, c, d\rangle, L4 : \langle L4\rangle\rangle.$$

Multilists can be implemented in a number of ways. Most of these should be familiar from implementations suggested earlier in the book for list, tree, and graph data structures.

One simple approach is to use a simple array to represent the list. This works well for chains with fixed-length elements, equivalent to the simple array-based list of Chapter 4. We can view nested sublists as variable-length elements. To use this approach, we require some indication of the beginning and end of each sublist. In essence, we are using a sequential tree implementation as discussed in Section 6.5. This should be no surprise, because the pure list is equivalent to a general tree structure. Unfortunately, as with any sequential representation, access to the nth sublist must be done sequentially from the beginning of the list.

Because pure lists are equivalent to trees, we can also use linked allocation methods to support direct access to the list of children. Simple linear lists are represented by linked lists. Pure lists can be represented as linked lists with an additional tag field to indicate whether the node is an atom or a sublist. If it is a sublist, the data field points to the first element on the sublist. This is illustrated by Figure 12.4.

Another approach is to represent all list elements with link nodes storing two pointer fields, except for atoms. Atoms just contain data. This is the system used by the programming language LISP. Figure 12.5 illustrates this representation. Either the pointer contains a tag bit to identify what it points to, or the object being pointed to stores a tag bit to identify itself. Tags distinguish atoms from list nodes. This

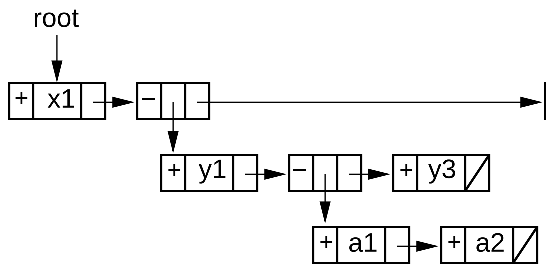

Figure 12.4 Linked representation for the pure list of Figure 12.1. The first field in each link node stores a tag bit. If the tag bit stores "+," then the data field stores an atom. If the tag bit stores "−," then the data field stores a pointer to a sublist.

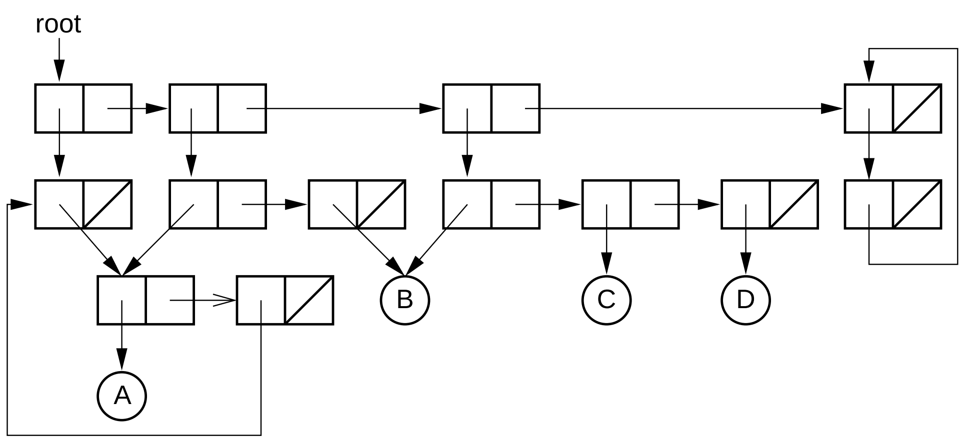

Figure 12.5 LISP-like linked representation for the cyclic multilist of Figure 12.3. Each link node stores two pointers. A pointer either points to an atom, or to another link node. Link nodes are represented by two boxes, and atoms by circles.

implementation can easily support reentrant and cyclic lists, because non-atoms can point to any other node.

12.2 Matrix Representations

Sometimes we need to represent a large, two-dimensional matrix where many of the elements have a value of zero. One example is the lower triangular matrix that results from solving systems of simultaneous equations. A lower triangular matrix stores zero values at all positions $[r, c]$ such that $r < c$, as shown in Figure 12.6(a). Thus, the upper-right triangle of the matrix is always zero. Another example is representing undirected graphs in an adjacency matrix (see Project 11.2). Because all edges between Vertices i and j go in both directions, there is no need to store both. Instead we can just store one edge going from the higher-indexed vertex to

a_{00}	0	0	0
a_{10}	a_{11}	0	0
a_{20}	a_{21}	a_{22}	0
a_{30}	a_{31}	a_{32}	a_{33}

(a)

a_{00}	a_{01}	a_{02}	a_{03}
0	a_{11}	a_{12}	a_{13}
0	0	a_{22}	a_{23}
0	0	0	a_{33}

(b)

Figure 12.6 Triangular matrices. (a) A lower triangular matrix. (b) An upper triangular matrix.

the lower-indexed vertex. In this case, only the lower triangle of the matrix can have non-zero values.

We can take advantage of this fact to save space. Instead of storing $n(n+1)/2$ pieces of information in an $n \times n$ array, it would save space to use a list of length $n(n+1)/2$. This is only practical if some means can be found to locate within the list the element that would correspond to position [r, c] in the original matrix.

We will derive an equation to convert position [r, c] to a position in a one-dimensional list to store the lower triangular matrix. Note that row 0 of the matrix has one non-zero value, row 1 has two non-zero values, and so on. Thus, row r is preceded by r rows with a total of $\sum_{k=1}^{r} k = (r^2 + r)/2$ non-zero elements. Adding c to reach the cth position in the rth row yields the following equation to convert position [r, c] in the original matrix to the correct position in the list.

$$\text{matrix}[r, c] = \text{list}[(r^2 + r)/2 + c].$$

A similar equation can be used to convert coordinates in an upper triangular matrix, that is, a matrix with zero values at positions [r, c] such that $r > c$, as shown in Figure 12.6(b). For an $n \times n$ upper triangular matrix, the equation to convert from matrix coordinates to list positions would be

$$\text{matrix}[r, c] = \text{list}[rn - (r^2 + r)/2 + c].$$

A more difficult situation arises when the vast majority of values stored in an $n \times m$ matrix are zero, but there is no restriction on which positions are zero and which are non-zero. This is known as a **sparse matrix**.

One approach to representing a sparse matrix is to concatenate (or otherwise combine) the row and column coordinates into a single value and use this as a key in a hash table. Thus, if we want to know the value of a particular position in the matrix, we search the hash table for the appropriate key. If a value for this position is not found, it is assumed to be zero. This is an ideal approach when all queries to the matrix are in terms of access by specified position. However, if we wish to find the first non-zero element in a given row, or the next non-zero element below the current one in a given column, then the hash table requires us to check sequentially through all possible positions in some row or column.

Another approach is to implement the matrix as an **orthogonal list**. Consider the following sparse matrix:

$$\begin{array}{rrrrrrr}
10 & 23 & 0 & 0 & 0 & 0 & 19 \\
45 & 5 & 0 & 93 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
40 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 32 & 0 & 12 & 0 & 0 & 7
\end{array}$$

The corresponding orthogonal array is shown in Figure 12.7. Here we have a list of row headers, each of which contains a pointer to a list of matrix records. A second list of column headers also contains pointers to matrix records. Each non-zero matrix element stores pointers to its non-zero neighbors in the row, both following and preceding it. Each non-zero element also stores pointers to its non-zero neighbors following and preceding it in the column. Thus, each non-zero element stores its own value, its position within the matrix, and four pointers. Non-zero elements are found by traversing a row or column list. Note that the first non-zero element in a given row could be in any column; likewise, the neighboring non-zero element in any row or column list could be at any (higher) row or column in the array. Thus, each non-zero element must also store its row and column position explicitly.

To find if a particular position in the matrix contains a non-zero element, we traverse the appropriate row or column list. For example, when looking for the element at Row 7 and Column 1, we can traverse the list either for Row 7 or for Column 1. When traversing a row or column list, if we come to an element with the correct position, then its value is non-zero. If we encounter an element with a higher position, then the element we are looking for is not in the sparse matrix. In this case, the element's value is zero. For example, when traversing the list for Row 7 in the matrix of Figure 12.7, we first reach the element at Row 7 and Column 1. If this is what we are looking for, then the search can stop. If we are looking for the element at Row 7 and Column 2, then the search proceeds along the Row 7 list to next reach the element at Column 3. At this point we know that no element at Row 7 and Column 2 is stored in the sparse matrix.

Insertion and deletion can be performed by working in a similar way to insert or delete elements within the appropriate row and column lists.

Each non-zero element stored in the sparse matrix representation takes much more space than an element stored in a simple $n \times n$ matrix. When is the sparse matrix more space efficient than the standard representation? To calculate this, we need to determine how much space the standard matrix requires, and how much

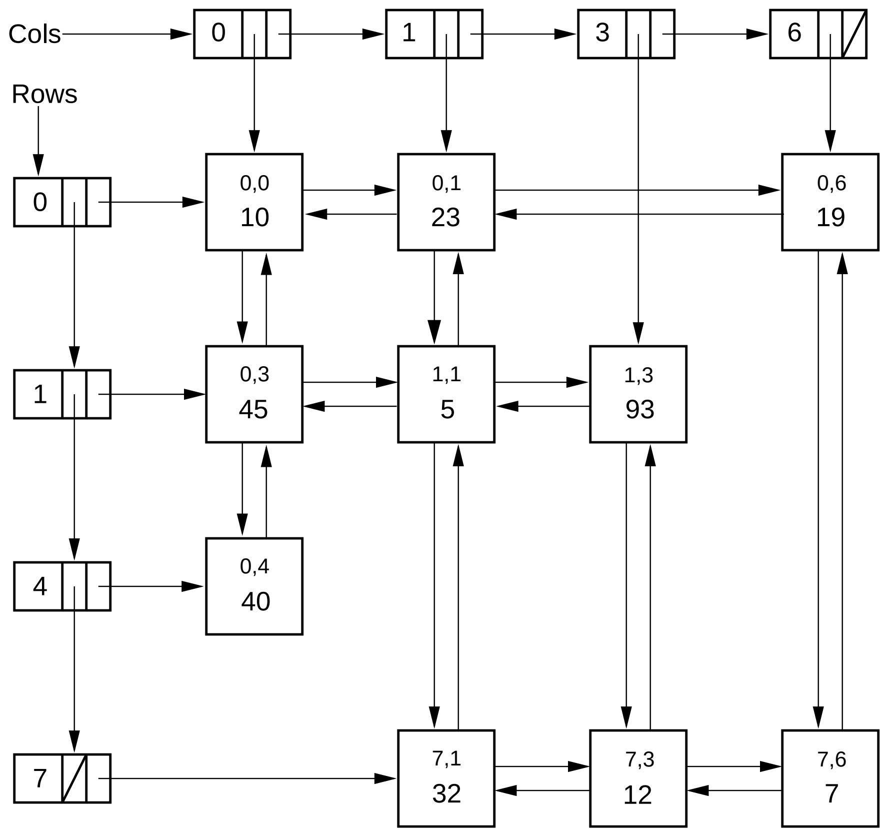

Figure 12.7 The orthogonal list sparse matrix representation.

the sparse matrix requires. The size of the sparse matrix depends on the number of non-zero elements (we will refer to this value as *NNZ*), while the size of the standard matrix representation does not vary. We need to know the (relative) sizes of a pointer and a data value. For simplicity, our calculation will ignore the space taken up by the row and column header (which is not much affected by the number of elements in the sparse array).

As an example, assume that a data value, a row or column index, and a pointer each require four bytes. An $n \times m$ matrix requires $4nm$ bytes. The sparse matrix requires 28 bytes per non-zero element (four pointers, two array indices, and one data value). If we set X to be the percentage of non-zero elements, we can solve for the value of X below which the sparse matrix representation is more space efficient. Using the equation

$$28X = 4mn$$

and solving for X, we find that the sparse matrix using this implementation is more space efficient when $X < 1/7$, that is, when less than about 14% of the elements are non-zero. Different values for the relative sizes of data values, pointers, or matrix indices can lead to a different break-even point for the two implementations.

The time required to process a sparse matrix should ideally depend on NNZ. When searching for an element, the cost is the number of elements preceding the desired element on its row or column list. The cost for operations such as adding two matrices should be $\Theta(n + m)$ in the worst case when the one matrix stores n non-zero elements and the other stores m non-zero elements.

Another representation for sparse matrices is sometimes called the Yale representation. Matlab uses a similar representation, with a primary difference being that the Matlab representation uses column-major order.[1] The Matlab representation stores the sparse matrix using three lists. The first is simply all of the non-zero element values, in column-major order. The second list stores the start position within the first list for each column. The third list stores the row positions for each of the corresponding non-zero values. In the Yale representation, the matrix of Figure 12.7 would appear as:

Values:	10 45 40 23 5 32 93 12 19 7
Column starts:	0 3 5 5 7 7 7 7
Row positions:	0 1 4 0 1 7 1 7 0 7

If the matrix has c columns, then the total space required will be proportional to $c + 2NNZ$. This is good in terms of space. It allows fairly quick access to any column, and allows for easy processing of the non-zero values along a column. However, it does not do a good job of providing access to the values along a row, and is terrible when values need to be added or removed from the representation. Fortunately, when doing computations such as adding or multiplying two sparse matrices, the processing of the input matrices and construction of the output matrix can be done reasonably efficiently.

12.3 Memory Management

Most data structures are designed to store and access objects of uniform size. A typical example would be an integer stored in a list or a queue. Some applications require the ability to store variable-length records, such as a string of arbitrary length. One solution is to store in the list or queue fixed-length pointers to the variable-length strings. This is fine for data structures stored in main memory. But if the collection of strings is meant to be stored on disk, then we might need to worry about where exactly these strings are stored. And even when stored in main memory, something has to figure out where there are available bytes to hold the string. We could easily store variable-size records in a queue or stack, where

[1]Scientific packages tend to prefer column-oriented representations for matrices since this the dominant access need for the operations to be performed.

```
/** Memory Manager interface */
interface MemManADT {

  /** Store a record and return a handle to it */
  public MemHandle insert(byte[] info);

  /** Get back a copy of a stored record */
  public byte[] get(MemHandle h);

  /** Release the space associated with a record */
  public void release(MemHandle h);
}
```

Figure 12.8 A simple ADT for a memory manager.

the restricted order of insertions and deletions makes this easy to deal with. But in a language like **C**++ or Java, programmers can allocate and deallocate space in complex ways through use of **new**. Where does this space come from? This section discusses memory management techniques for the general problem of handling space requests of variable size.

The basic model for memory management is that we have a (large) block of contiguous memory locations, which we will call the **memory pool**. Periodically, memory requests are issued for some amount of space in the pool. The **memory manager** has the job of finding a contiguous block of locations of at least the requested size from somewhere within the memory pool. Honoring such a request is called a **memory allocation**. The memory manager will typically return some piece of information that the requester can hold on to so that later it can recover the record that was just stored by the memory manager. This piece of information is called a **handle**. At some point, space that has been requested might no longer be needed, and this space can be returned to the memory manager so that it can be reused. This is called a **memory deallocation**. The memory manager should then be able to reuse this space to satisfy later memory requests. We can define an ADT for the memory manager as shown in Figure 12.8.

The user of the **MemManager** ADT provides a pointer (in parameter **info**) to space that holds some record or message to be stored or retrieved. This is similar to the Java basic file read/write methods presented in Section 8.4. The fundamental idea is that the client gives messages to the memory manager for safe keeping. The memory manager returns a "receipt" for the message in the form of a **MemHandle** object. Of course to be practical, a **MemHandle** must be much smaller than the typical message to be stored. The client holds the **MemHandle** object until it wishes to get the message back.

Method **insert** lets the client tell the memory manager the length and contents of the message to be stored. This ADT assumes that the memory manager will remember the length of the message associated with a given handle (perhaps in the

Figure 12.9 Dynamic storage allocation model. Memory is made up of a series of variable-size blocks, some allocated and some free. In this example, shaded areas represent memory currently allocated and unshaded areas represent unused memory available for future allocation.

handle itself), thus method **`get`** does not include a length parameter but instead returns the length of the message actually stored. Method **`release`** allows the client to tell the memory manager to release the space that stores a given message.

When all inserts and releases follow a simple pattern, such as last requested, first released (stack order), or first requested, first released (queue order), memory management is fairly easy. We are concerned here with the general case where blocks of any size might be requested and released in any order. This is known as **dynamic storage allocation**. One example of dynamic storage allocation is managing free store for a compiler's runtime environment, such as the system-level **`new`** operation in Java. Another example is managing main memory in a multitasking operating system. Here, a program might require a certain amount of space, and the memory manager must keep track of which programs are using which parts of the main memory. Yet another example is the file manager for a disk drive. When a disk file is created, expanded, or deleted, the file manager must allocate or deallocate disk space.

A block of memory or disk space managed in this way is sometimes referred to as a **heap**. The term "heap" is being used here in a different way than the heap data structure discussed in Section 5.5. Here "heap" refers to the memory controlled by a dynamic memory management scheme.

In the rest of this section, we first study techniques for dynamic memory management. We then tackle the issue of what to do when no single block of memory in the memory pool is large enough to honor a given request.

12.3.1 Dynamic Storage Allocation

For the purpose of dynamic storage allocation, we view memory as a single array which, after a series of memory requests and releases tends to become broken into a series of variable-size blocks, where some of the blocks are **free** and some are **reserved** or already allocated to store messages. The memory manager typically uses a linked list to keep track of the free blocks, called the **freelist**, which is used for servicing future memory requests. Figure 12.9 illustrates the situation that can arise after a series of memory allocations and deallocations.

When a memory request is received by the memory manager, some block on the freelist must be found that is large enough to service the request. If no such

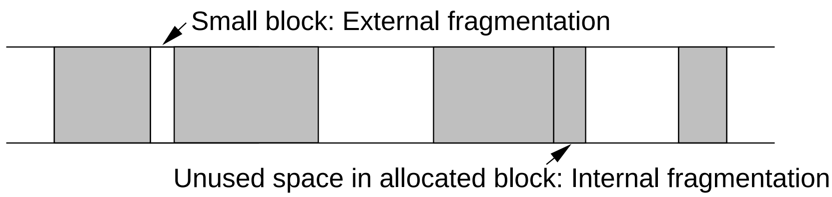

Figure 12.10 An illustration of internal and external fragmentation. The small white block labeled "External fragmentation" is too small to satisfy typical memory requests. The small grey block labeled "Internal fragmentation" was allocated as part of the grey block to its left, but it does not actually store information.

block is found, then the memory manager must resort to a **failure policy** such as discussed in Section 12.3.2.

If there is a request for m words, and no block exists of exactly size m, then a larger block must be used instead. One possibility in this case is that the entire block is given away to the memory allocation request. This might be desirable when the size of the block is only slightly larger than the request. This is because saving a tiny block that is too small to be useful for a future memory request might not be worthwhile. Alternatively, for a free block of size k, with $k > m$, up to $k - m$ space may be retained by the memory manager to form a new free block, while the rest is used to service the request.

Memory managers can suffer from two types of **fragmentation**, which refers to unused space that is too small to be useful. **External fragmentation** occurs when a series of memory requests and releases results in small free blocks. **Internal fragmentation** occurs when more than m words are allocated to a request for m words, wasting free storage. This is equivalent to the internal fragmentation that occurs when files are allocated in multiples of the cluster size. The difference between internal and external fragmentation is illustrated by Figure 12.10.

Some memory management schemes sacrifice space to internal fragmentation to make memory management easier (and perhaps reduce external fragmentation). For example, external fragmentation does not happen in file management systems that allocate file space in clusters. Another example of sacrificing space to internal fragmentation so as to simplify memory management is the **buddy method** described later in this section.

The process of searching the memory pool for a block large enough to service the request, possibly reserving the remaining space as a free block, is referred to as a **sequential fit** method.

Sequential Fit Methods

Sequential-fit methods attempt to find a "good" block to service a storage request. The three sequential-fit methods described here assume that the free blocks are organized into a doubly linked list, as illustrated by Figure 12.11.

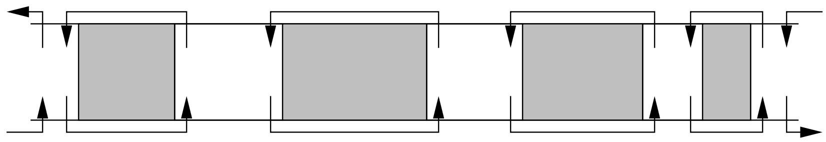

Figure 12.11 A doubly linked list of free blocks as seen by the memory manager. Shaded areas represent allocated memory. Unshaded areas are part of the freelist.

There are two basic approaches to implementing the freelist. The simpler approach is to store the freelist separately from the memory pool. In other words, a simple linked-list implementation such as described in Chapter 4 can be used, where each node of the linked list contains a pointer to a single free block in the memory pool. This is fine if there is space available for the linked list itself, separate from the memory pool.

The second approach to storing the freelist is more complicated but saves space. Because the free space is free, it can be used by the memory manager to help it do its job. That is, the memory manager can temporarily "borrow" space within the free blocks to maintain its doubly linked list. To do so, each unallocated block must be large enough to hold these pointers. In addition, it is usually worthwhile to let the memory manager add a few bytes of space to each reserved block for its own purposes. In other words, a request for m bytes of space might result in slightly more than m bytes being allocated by the memory manager, with the extra bytes used by the memory manager itself rather than the requester. We will assume that all memory blocks are organized as shown in Figure 12.12, with space for tags and linked list pointers. Here, free and reserved blocks are distinguished by a tag bit at both the beginning and the end of the block, for reasons that will be explained. In addition, both free and reserved blocks have a size indicator immediately after the tag bit at the beginning of the block to indicate how large the block is. Free blocks have a second size indicator immediately preceding the tag bit at the end of the block. Finally, free blocks have left and right pointers to their neighbors in the free block list.

The information fields associated with each block permit the memory manager to allocate and deallocate blocks as needed. When a request comes in for m words of storage, the memory manager searches the linked list of free blocks until it finds a "suitable" block for allocation. How it determines which block is suitable will be discussed below. If the block contains exactly m words (plus space for the tag and size fields), then it is removed from the freelist. If the block (of size k) is large enough, then the remaining $k - m$ words are reserved as a block on the freelist, in the current location.

When a block F is freed, it must be merged into the freelist. If we do not care about merging adjacent free blocks, then this is a simple insertion into the doubly linked list of free blocks. However, we would like to merge adjacent blocks,

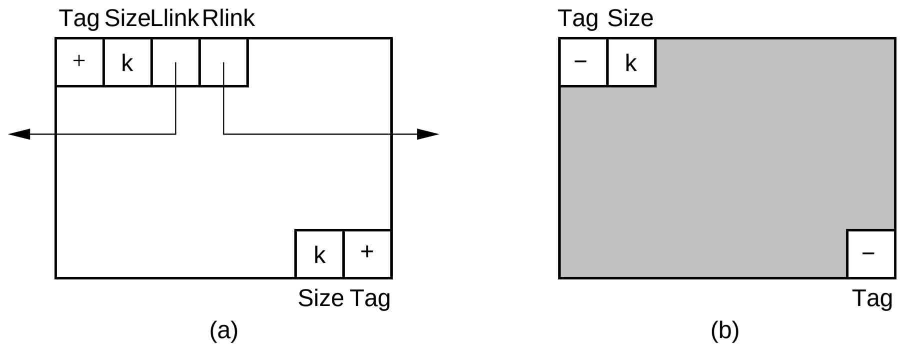

Figure 12.12 Blocks as seen by the memory manager. Each block includes additional information such as freelist link pointers, start and end tags, and a size field. (a) The layout for a free block. The beginning of the block contains the tag bit field, the block size field, and two pointers for the freelist. The end of the block contains a second tag field and a second block size field. (b) A reserved block of k bytes. The memory manager adds to these k bytes an additional tag bit field and block size field at the beginning of the block, and a second tag field at the end of the block.

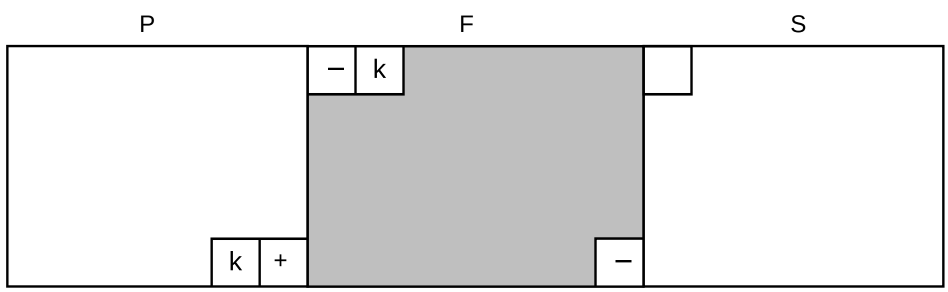

Figure 12.13 Adding block *F* to the freelist. The word immediately preceding the start of *F* in the memory pool stores the tag bit of the preceding block *P*. If *P* is free, merge *F* into *P*. We find the end of *F* by using *F*'s size field. The word following the end of *F* is the tag field for block *S*. If *S* is free, merge it into *F*.

because this allows the memory manager to serve requests of the largest possible size. Merging is easily done due to the tag and size fields stored at the ends of each block, as illustrated by Figure 12.13. Here, the memory manager first checks the unit of memory immediately preceding block *F* to see if the preceding block (call it *P*) is also free. If it is, then the memory unit before *P*'s tag bit stores the size of *P*, thus indicating the position for the beginning of the block in memory. *P* can then simply have its size extended to include block *F*. If block *P* is not free, then we just add block *F* to the freelist. Finally, we also check the bit following the end of block *F*. If this bit indicates that the following block (call it *S*) is free, then *S* is removed from the freelist and the size of *F* is extended appropriately.

We now consider how a "suitable" free block is selected to service a memory request. To illustrate the process, assume that we have a memory pool with 200 units of storage. After some series of allocation requests and releases, we have reached a point where there are four free blocks on the freelist of sizes 25, 35, 32, and 45 (in that order). Assume that a request is made for 30 units of storage. For our examples, we ignore the overhead imposed for the tag, link, and size fields discussed above.

The simplest method for selecting a block would be to move down the free block list until a block of size at least 30 is found. Any remaining space in this block is left on the freelist. If we begin at the beginning of the list and work down to the first free block at least as large as 30, we select the block of size 35. 30 units of storage will be allocated, leaving a free block with 5 units of space. Because this approach selects the first block with enough space, it is called **first fit**. A simple variation that will improve performance is, instead of always beginning at the head of the freelist, remember the last position reached in the previous search and start from there. When the end of the freelist is reached, search begins again at the head of the freelist. This modification reduces the number of unnecessary searches through small blocks that were passed over by previous requests.

There is a potential disadvantage to first fit: It might "waste" larger blocks by breaking them up, and so they will not be available for large requests later. A strategy that avoids using large blocks unnecessarily is called **best fit**. Best fit looks at the entire list and picks the smallest block that is at least as large as the request (i.e., the "best" or closest fit to the request). Continuing with the preceding example, the best fit for a request of 30 units is the block of size 32, leaving a remainder of size 2. Best fit has the disadvantage that it requires that the entire list be searched. Another problem is that the remaining portion of the best-fit block is likely to be small, and thus useless for future requests. In other words, best fit tends to maximize problems of external fragmentation while it minimizes the chance of not being able to service an occasional large request.

A strategy contrary to best fit might make sense because it tends to minimize the effects of external fragmentation. This is called **worst fit**, which always allocates the largest block on the list hoping that the remainder of the block will be useful for servicing a future request. In our example, the worst fit is the block of size 45, leaving a remainder of size 15. If there are a few unusually large requests, this approach will have less chance of servicing them. If requests generally tend to be of the same size, then this might be an effective strategy. Like best fit, worst fit requires searching the entire freelist at each memory request to find the largest block. Alternatively, the freelist can be ordered from largest to smallest free block, possibly by using a priority queue implementation.

Which strategy is best? It depends on the expected types of memory requests. If the requests are of widely ranging size, best fit might work well. If the requests tend to be of similar size, with rare large and small requests, first or worst fit might work well. Unfortunately, there are always request patterns that one of the three sequential fit methods will service, but which the other two will not be able to service. For example, if the series of requests 600, 650, 900, 500, 100 is made to a freelist containing blocks 500, 700, 650, 900 (in that order), the requests can all be serviced by first fit, but not by best fit. Alternatively, the series of requests 600, 500, 700, 900 can be serviced by best fit but not by first fit on this same freelist.

Buddy Methods

Sequential-fit methods rely on a linked list of free blocks, which must be searched for a suitable block at each memory request. Thus, the time to find a suitable free block would be $\Theta(n)$ in the worst case for a freelist containing n blocks. Merging adjacent free blocks is somewhat complicated. Finally, we must either use additional space for the linked list, or use space within the memory pool to support the memory manager operations. In the second option, both free and reserved blocks require tag and size fields. Fields in free blocks do not cost any space (because they are stored in memory that is not otherwise being used), but fields in reserved blocks create additional overhead.

The buddy system solves most of these problems. Searching for a block of the proper size is efficient, merging adjacent free blocks is simple, and no tag or other information fields need be stored within reserved blocks. The buddy system assumes that memory is of size 2^N for some integer N. Both free and reserved blocks will always be of size 2^k for $k \leq N$. At any given time, there might be both free and reserved blocks of various sizes. The buddy system keeps a separate list for free blocks of each size. There can be at most N such lists, because there can only be N distinct block sizes.

When a request comes in for m words, we first determine the smallest value of k such that $2^k \geq m$. A block of size 2^k is selected from the free list for that block size if one exists. The buddy system does not worry about internal fragmentation: The entire block of size 2^k is allocated.

If no block of size 2^k exists, the next larger block is located. This block is split in half (repeatedly if necessary) until the desired block of size 2^k is created. Any other blocks generated as a by-product of this splitting process are placed on the appropriate freelists.

The disadvantage of the buddy system is that it allows internal fragmentation. For example, a request for 257 words will require a block of size 512. The primary advantages of the buddy system are (1) there is less external fragmentation; (2) search for a block of the right size is cheaper than, say, best fit because we need only find the first available block on the block list for blocks of size 2^k; and (3) merging adjacent free blocks is easy.

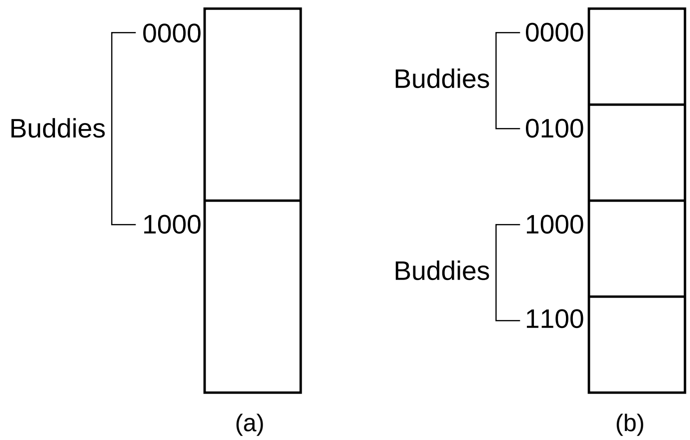

Figure 12.14 Example of the buddy system. (a) Blocks of size 8. (b) Blocks of size 4.

The reason why this method is called the buddy system is because of the way that merging takes place. The **buddy** for any block of size 2^k is another block of the same size, and with the same address (i.e., the byte position in memory, read as a binary value) except that the kth bit is reversed. For example, the block of size 8 with beginning address 0000 in Figure 12.14(a) has buddy with address 1000. Likewise, in Figure 12.14(b), the block of size 4 with address 0000 has buddy 0100. If free blocks are sorted by address value, the buddy can be found by searching the correct block-size list. Merging simply requires that the address for the combined buddies be moved to the freelist for the next larger block size.

Other Memory Allocation Methods

In addition to sequential-fit and buddy methods, there are many ad hoc approaches to memory management. If the application is sufficiently complex, it might be desirable to break available memory into several memory **zones**, each with a different memory management scheme. For example, some zones might have a simple memory access pattern of first-in, first-out. This zone can therefore be managed efficiently by using a simple queue. Another zone might allocate only records of fixed size, and so can be managed with a simple freelist as described in Section 4.1.2. Other zones might need one of the general-purpose memory allocation methods discussed in this section. The advantage of zones is that some portions of memory can be managed more efficiently. The disadvantage is that one zone might fill up while other zones have excess free memory if the zone sizes are chosen poorly.

Another approach to memory management is to impose a standard size on all memory requests. We have seen an example of this concept already in disk file management, where all files are allocated in multiples of the cluster size. This approach leads to internal fragmentation, but managing files composed of clusters is easier than managing arbitrarily sized files. The cluster scheme also allows us to relax the restriction that the memory request be serviced by a contiguous block of memory. Most disk file managers and operating system main memory managers work on a cluster or page system. Block management is usually done with a buffer pool to allocate available blocks in main memory efficiently.

12.3.2 Failure Policies and Garbage Collection

At some point when processing a series of requests, a memory manager could encounter a request for memory that it cannot satisfy. In some situations, there might be nothing that can be done: There simply might not be enough free memory to service the request, and the application may require that the request be serviced immediately. In this case, the memory manager has no option but to return an error, which could in turn lead to a failure of the application program. However, in many cases there are alternatives to simply returning an error. The possible options are referred to collectively as **failure policies**.

In some cases, there might be sufficient free memory to satisfy the request, but it is scattered among small blocks. This can happen when using a sequential-fit memory allocation method, where external fragmentation has led to a series of small blocks that collectively could service the request. In this case, it might be possible to **compact** memory by moving the reserved blocks around so that the free space is collected into a single block. A problem with this approach is that the application must somehow be able to deal with the fact that its data have now been moved to different locations. If the application program relies on the absolute positions of the data in any way, this would be disastrous. One approach for dealing with this problem involves the handles returned by the memory manager. A handle works as a second level of indirection to a memory location. The memory allocation routine does not return a pointer to the block of storage, but rather a pointer to a the handle that in turn gives access to the storage. The handle never moves its position, but the position of the block might be moved and the value of the handle updated. Of course, this requires that the memory manager keep track of the handles and how they associate with the stored messages. Figure 12.15 illustrates the concept.

Another failure policy that might work in some applications is to defer the memory request until sufficient memory becomes available. For example, a multi-tasking operating system could adopt the strategy of not allowing a process to run until there is sufficient memory available. While such a delay might be annoying to the user, it is better than halting the entire system. The assumption here is that other processes will eventually terminate, freeing memory.

Another option might be to allocate more memory to the memory manager. In a zoned memory allocation system where the memory manager is part of a larger system, this might be a viable option. In a Java program that implements its own

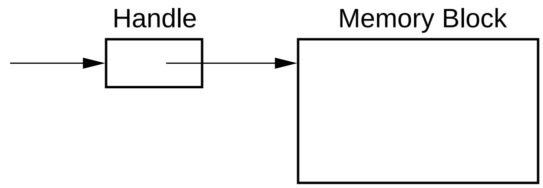

Figure 12.15 Using handles for dynamic memory management. The memory manager returns the address of the handle in response to a memory request. The handle stores the address of the actual memory block. In this way, the memory block might be moved (with its address updated in the handle) without disrupting the application program.

memory manager, it might be possible to get more memory from the system-level **new** operator, such as is done by the freelist of Section 4.1.2.

The last failure policy that we will consider is **garbage collection**. Consider the following series of statements.

```
int[] p = new int[5];
int[] q = new int[10];
p = q;
```

While in Java this would be no problem (due to automatic garbage collection), in languages such as **C**++, this would be considered bad form because the original space allocated to **p** is lost as a result of the third assignment. This space cannot be used again by the program. Such lost memory is referred to as **garbage**, also known as a **memory leak**. When no program variable points to a block of space, no future access to that space is possible. Of course, if another variable had first been assigned to point to **p**'s space, then reassigning **p** would not create garbage.

Some programming languages take a different view towards garbage. In particular, the LISP programming language uses the multilist representation of Figure 12.5, and all storage is in the form either of internal nodes with two pointers or atoms. Figure 12.16 shows a typical collection of LISP structures, headed by variables named *A*, *B*, and *C*, along with a freelist.

In LISP, list objects are constantly being put together in various ways as temporary variables, and then all reference to them is lost when the object is no longer needed. Thus, garbage is normal in LISP, and in fact cannot be avoided during routine program behavior. When LISP runs out of memory, it resorts to a garbage collection process to recover the space tied up in garbage. Garbage collection consists of examining the managed memory pool to determine which parts are still being used and which parts are garbage. In particular, a list is kept of all program variables, and any memory locations not reachable from one of these variables are considered to be garbage. When the garbage collector executes, all unused memory locations are placed in free store for future access. This approach has the advantage that it allows for easy collection of garbage. It has the disadvantage, from a user's

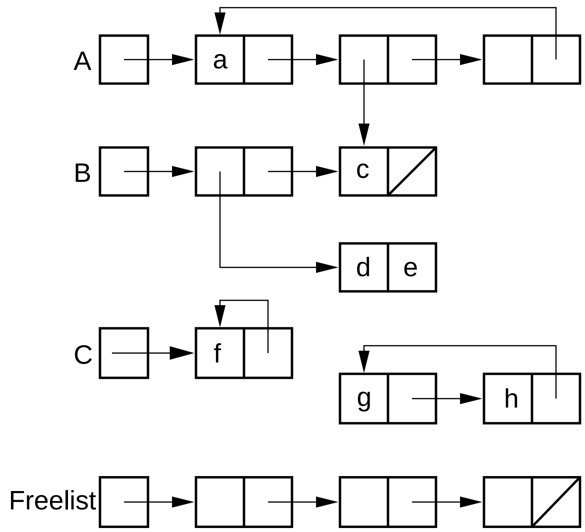

Figure 12.16 Example of LISP list variables, including the system freelist.

point of view, that every so often the system must halt while it performs garbage collection. For example, garbage collection is noticeable in the Emacs text editor, which is normally implemented in LISP. Occasionally the user must wait for a moment while the memory management system performs garbage collection.

The Java programming language also makes use of garbage collection. As in LISP, it is common practice in Java to allocate dynamic memory as needed, and to later drop all references to that memory. The garbage collector is responsible for reclaiming such unused space as necessary. This might require extra time when running the program, but it makes life considerably easier for the programmer. In contrast, many large applications written in **C**++ (even commonly used commercial software) contain memory leaks that will in time cause the program to fail.

Several algorithms have been used for garbage collection. One is the **reference count** algorithm. Here, every dynamically allocated memory block includes space for a count field. Whenever a pointer is directed to a memory block, the reference count is increased. Whenever a pointer is directed away from a memory block, the reference count is decreased. If the count ever becomes zero, then the memory block is considered garbage and is immediately placed in free store. This approach has the advantage that it does not require an explicit garbage collection phase, because information is put in free store immediately when it becomes garbage.

Reference counts are used by the UNIX file system. Files can have multiple names, called links. The file system keeps a count of the number of links to each file. Whenever a file is "deleted," in actuality its link field is simply reduced by one. If there is another link to the file, then no space is recovered by the file system. When the number of links goes to zero, the file's space becomes available for reuse.

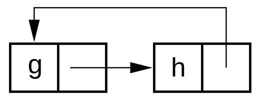

Figure 12.17 Garbage cycle example. All memory elements in the cycle have non-zero reference counts because each element has one pointer to it, even though the entire cycle is garbage (i.e., no static variable in the program points to it).

Reference counts have several major disadvantages. First, a reference count must be maintained for each memory object. This works well when the objects are large, such as a file. However, it will not work well in a system such as LISP where the memory objects typically consist of two pointers or a value (an atom). Another major problem occurs when garbage contains cycles. Consider Figure 12.17. Here each memory object is pointed to once, but the collection of objects is still garbage because no pointer points to the collection. Thus, reference counts only work when the memory objects are linked together without cycles, such as the UNIX file system where files can only be organized as a DAG.

Another approach to garbage collection is the **mark/sweep** strategy. Here, each memory object needs only a single mark bit rather than a reference counter field. When free store is exhausted, a separate garbage collection phase takes place as follows.

1. Clear all mark bits.
2. Perform depth-first search (DFS) following pointers beginning with each variable on the system's list of static variables. Each memory element encountered during the DFS has its mark bit turned on.
3. A "sweep" is made through the memory pool, visiting all elements. Unmarked elements are considered garbage and placed in free store.

The advantages of the mark/sweep approach are that it needs less space than is necessary for reference counts, and it works for cycles. However, there is a major disadvantage. This is a "hidden" space requirement needed to do the processing. DFS is a recursive algorithm: Either it must be implemented recursively, in which case the compiler's runtime system maintains a stack, or else the memory manager can maintain its own stack. What happens if all memory is contained in a single linked list? Then the depth of the recursion (or the size of the stack) is the number of memory cells! Unfortunately, the space for the DFS stack must be available at the worst conceivable time, that is, when free memory has been exhausted.

Fortunately, a clever technique allows DFS to be performed without requiring additional space for a stack. Instead, the structure being traversed is used to hold the stack. At each step deeper into the traversal, instead of storing a pointer on the stack, we "borrow" the pointer being followed. This pointer is set to point back to the node we just came from in the previous step, as illustrated by Figure 12.18. Each borrowed pointer stores an additional bit to tell us whether we came down

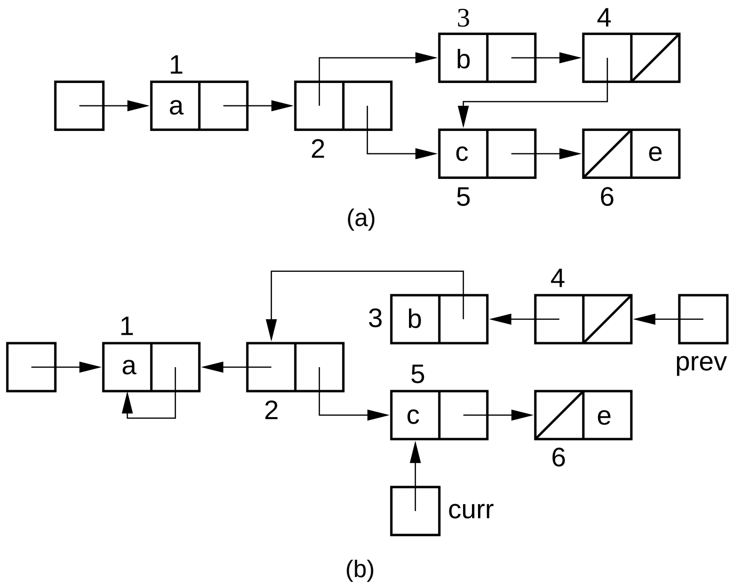

Figure 12.18 Example of the Deutsch-Schorr-Waite garbage collection algorithm. (a) The initial multilist structure. (b) The multilist structure of (a) at the instant when link node 5 is being processed by the garbage collection algorithm. A chain of pointers stretching from variable `prev` to the head node of the structure has been (temporarily) created by the garbage collection algorithm.

the left branch or the right branch of the link node being pointed to. At any given instant we have passed down only one path from the root, and we can follow the trail of pointers back up. As we return (equivalent to popping the recursion stack), we set the pointer back to its original position so as to return the structure to its original condition. This is known as the Deutsch-Schorr-Waite garbage collection algorithm.

12.4 Further Reading

For information on LISP, see *The Little LISPer* by Friedman and Felleisen [FF89]. Another good LISP reference is *Common LISP: The Language* by Guy L. Steele [Ste90]. For information on Emacs, which is both an excellent text editor and a programming environment, see the *GNU Emacs Manual* by Richard Stallman [Sta11b]. You can get more information about Java's garbage collection system from *The Java Programming Language* by Ken Arnold and James Gosling [AG06].

For more details on sparse matrix representations, the Yale representation is described by Eisenstat, Schultz and Sherman [ESS81]. The MATLAB sparse matrix representation is described by Gilbert, Moler, and Schreiber [GMS91].

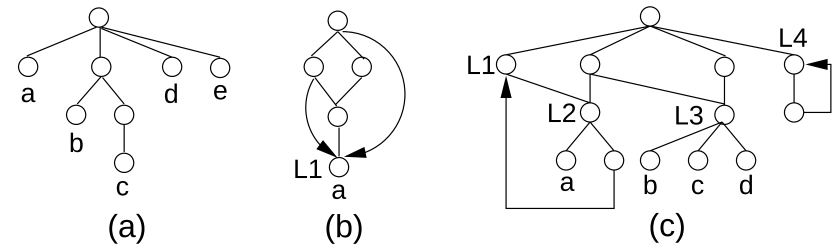

Figure 12.19 Some example multilists.

An introductory text on operating systems covers many topics relating to memory management issues, including layout of files on disk and caching of information in main memory. All of the topics covered here on memory management, buffer pools, and paging are relevant to operating system implementation. For example, see *Operating Systems* by William Stallings[Sta11a].

12.5 Exercises

12.1 For each of the following bracket notation descriptions, draw the equivalent multilist in graphical form such as shown in Figure 12.2.

(a) $\langle a, b, \langle c, d, e\rangle, \langle f, \langle g\rangle, h\rangle\rangle$

(b) $\langle a, b, \langle c, d, L1\!:\!e\rangle, L1\rangle$

(c) $\langle L1\!:\!a, L1, \langle L2\!:\!b\rangle, L2, \langle L1\rangle\rangle$

12.2 **(a)** Show the bracket notation for the list of Figure 12.19(a).

(b) Show the bracket notation for the list of Figure 12.19(b).

(c) Show the bracket notation for the list of Figure 12.19(c).

12.3 Given the linked representation of a pure list such as

$$\langle x_1, \langle y_1, y_2, \langle z_1, z_2\rangle, y_4\rangle, \langle w_1, w_2\rangle, x_4\rangle,$$

write an in-place reversal algorithm to reverse the sublists at all levels including the topmost level. For this example, the result would be a linked representation corresponding to

$$\langle x_4, \langle w_2, w_1\rangle, \langle y_4, \langle z_2, z_1\rangle, y_2, y_1\rangle, x_1\rangle.$$

12.4 What fraction of the values in a matrix must be zero for the sparse matrix representation of Section 12.2 to be more space efficient than the standard two-dimensional matrix representation when data values require eight bytes, array indices require two bytes, and pointers require four bytes?

12.5 Write a function to add an element at a given position to the sparse matrix representation of Section 12.2.

12.6 Write a function to delete an element from a given position in the sparse matrix representation of Section 12.2.

12.7 Write a function to transpose a sparse matrix as represented in Section 12.2.

12.8 Write a function to add two sparse matrices as represented in Section 12.2.

12.9 Write memory manager allocation and deallocation routines for the situation where all requests and releases follow a last-requested, first-released (stack) order.

12.10 Write memory manager allocation and deallocation routines for the situation where all requests and releases follow a last-requested, last-released (queue) order.

12.11 Show the result of allocating the following blocks from a memory pool of size 1000 using first fit for each series of block requests. State if a given request cannot be satisfied.

(a) Take 300 (call this block A), take 500, release A, take 200, take 300.
(b) Take 200 (call this block A), take 500, release A, take 200, take 300.
(c) Take 500 (call this block A), take 300, release A, take 300, take 200.

12.12 Show the result of allocating the following blocks from a memory pool of size 1000 using best fit for each series of block requests. State if a given request cannot be satisfied.

(a) Take 300 (call this block A), take 500, release A, take 200, take 300.
(b) Take 200 (call this block A), take 500, release A, take 200, take 300.
(c) Take 500 (call this block A), take 300, release A, take 300, take 200.

12.13 Show the result of allocating the following blocks from a memory pool of size 1000 using worst fit for each series of block requests. State if a given request cannot be satisfied.

(a) Take 300 (call this block A), take 500, release A, take 200, take 300.
(b) Take 200 (call this block A), take 500, release A, take 200, take 300.
(c) Take 500 (call this block A), take 300, release A, take 300, take 200.

12.14 Assume that the memory pool contains three blocks of free storage. Their sizes are 1300, 2000, and 1000. Give examples of storage requests for which

(a) first-fit allocation will work, but not best fit or worst fit.
(b) best-fit allocation will work, but not first fit or worst fit.
(c) worst-fit allocation will work, but not first fit or best fit.

12.6 Projects

12.1 Implement the orthogonal list sparse matrix representation of Section 12.2. Your implementation should support the following operations on the matrix:

- insert an element at a given position,
- delete an element from a given position,
- return the value of the element at a given position,
- take the transpose of a matrix,

- add two matrices, and
- multiply two matrices.

12.2 Implement the Yale model for sparse matrices described at the end of Section 12.2. Your implementation should support the following operations on the matrix:

- insert an element at a given position,
- delete an element from a given position,
- return the value of the element at a given position,
- take the transpose of a matrix,
- add two matrices, and
- multiply two matrices.

12.3 Implement the **`MemManager`** ADT shown at the beginning of Section 12.3. Use a separate linked list to implement the freelist. Your implementation should work for any of the three sequential-fit methods: first fit, best fit, and worst fit. Test your system empirically to determine under what conditions each method performs well.

12.4 Implement the **`MemManager`** ADT shown at the beginning of Section 12.3. Do not use separate memory for the free list, but instead embed the free list into the memory pool as shown in Figure 12.12. Your implementation should work for any of the three sequential-fit methods: first fit, best fit, and worst fit. Test your system empirically to determine under what conditions each method performs well.

12.5 Implement the **`MemManager`** ADT shown at the beginning of Section 12.3 using the buddy method of Section 12.3.1. Your system should support requests for blocks of a specified size and release of previously requested blocks.

12.6 Implement the Deutsch-Schorr-Waite garbage collection algorithm that is illustrated by Figure 12.18.