

3. Describe a method for storing a file of text on an external memory medium, such as a floppy disk.
4. Discuss how text can be represented in main memory for use by a word processor. Discuss some possible roles for "invisible text codes" in this text representation.

7.5 EXERCISES

1. Suppose *S* is a variable containing a Java String object. Implement the Java String method *S.toLowerCase()*.
2. Suppose *S* and *T* are variables containing Java String objects. Implement the Java boolean String method *S.equalsIgnoreCase(T)*.
3. Verify that the behavior of the *StrngBuffer* class given in Program 7.14 is equivalent to the Java *StringBuffer* class in the *java.lang* package, first by systematically substituting "StringBuffer" for "StrngBuffer" in Program 7.13, second by removing the *StrngBuffer* class definition from Program 7.13, and finally by executing Program 7.13. Do you get completely identical results?

7.6 Dynamic Memory Allocation

LEARNING OBJECTIVES

1. To understand the difference between static and dynamic memory allocation.
2. To learn about available space lists and garbage collection.
3. To learn how heaps can be used to support dynamic memory allocation.
4. To learn about fragmentation and coalescing.
5. To learn about handles and heap compaction.
6. To learn how reference counts can be used to support incremental storage reclamation.

We begin by reviewing the difference between *static* and *dynamic* memory allocation. Suppose we have a Java applet, *MyApplet*, that (1) constructs generalized list nodes using the node types defined in Program 7.9, and (2) declares two variables: (a) a static integer *year*, and (b) a variable *L* that is of type *GenList*, (see Program 7.17). Now suppose that later in a method *M* inside *MyApplet*, we encounter the statements in Program 7.18.

As you can see, when the statements in Program 7.18 are executed, they cause a new general list node to be allocated, and they cause a new first list item to be inserted whose item field contains an *Integer* object having the year 1998 as its value.

```

| public class MyApplet extends Applet {
|
|     static int year;
|     GenList L;
|

```

Program 7.17 Two Variables in Java Applet Class Definition

Data Structures in Java
 Thomas A. Standish
 Addison Wesley

static versus
dynamic allocation

static

dynamic

```

|   year = 1998;           // let 1998 be the value of the class variable, year
|                           // let L be a new GenList, and insert
20 |   L = new GenList( );   // a new Integer item on the list L
|   L.insertItem(new Integer(year)); // whose value is the year 1998

```

Program 7.18 Some Statements in a Method M in MyApplet

Let's now compare the times at which storage is allocated for the static int year class variable and at which storage is allocated for the node referenced by the pointer in L. The space for the class variable year can be allocated before the execution of program MyApplet begins. That is, the compiler that compiles the code for MyApplet can analyze the information in the declaration of year, and it can compile code that, when executed, will access and update a static int storage location. The locations of such static variables can be fixed relative to the base address of the run-time storage allocated for executing MyApplet. That is, the space for them is reserved in advance of the execution of MyApplet. In this case, we say that the class variable year has been *statically* allocated. The word *static* means "not in motion." MyApplet is not in motion at the time this *static* storage allocation takes place, because the applet has not been executed yet.

By contrast, let's examine the time when storage is allocated for the general list node referenced by the pointer in the variable, L. The space for this node is allocated at the time the statement L = new GenList(); is executed (on line 20 of Program 7.18). We say that storage for this node is *dynamically* allocated, because it gets allocated only when the program is executed. The word *dynamic* can be taken to mean "in motion," and it describes an action that is taken during program execution.

Static memory allocation can take place for all static class variables declared at the beginning of a Java class definition. This is because Java's rules compel us to provide enough information, when declaring static class variables, that their exact size and memory requirements can be calculated from analyzing the text of the Java program at compile time.

On the other hand, when we execute a Java program that invokes methods, the number of method calls cannot be predicted in advance of the running of the program. Rather, as each method call is encountered during program execution, we can allocate new space on a run-time stack associated with a given Java execution thread to accommodate the call's storage requirements. Such stack-based allocation is one form of dynamic memory allocation.

Another kind of dynamic memory allocation occurs when we execute a statement, such as L = new GenList(); in a Java program. The space for the node referenced by L, that gets allocated when we execute this statement, cannot be placed on a run-time stack, since this space may need to remain in existence regardless of whether we call or return from various methods. If we reserved space for the node referenced by L's pointer on a thread's run-time stack, then the space would have to disappear the moment we returned from the currently executing method. To guarantee the persistence of the storage for the node L independently of method calls and returns, we can use a separate region of memory, and we can allocate blocks of memory for object instances of various sizes from this region whenever the need arises. Dynamic memory

static memory allocation

dynamic memory allocation

dynamic allocation in heaps

TION

such as

processor.

on.

String

Java

equiva-

atically

moving

cuting

ocation.

t nodes

s: (a) a

1 7.17).

ements

cause a

insert-

..

allocation can then take place using blocks allocated from this region anytime during the running of a program. We refer to such a region of memory as a *heap*.

In addition to using stacks and heaps to support dynamic memory allocation, it is also possible to base dynamic memory allocation on the use of a list of available space—a form of dynamic allocation useful for lists formed by linking together nodes of identical size.

In the subsections that follow, we will discuss the kind of dynamic memory allocation that occurs when we organize a region of memory into an available space list and the techniques for dynamic memory allocation in heaps.

Available Space Lists and Garbage Collection

In Fig. 7.12 we saw how to divide memory into two zones for use in list processing applications. One zone of memory held nodes containing pairs of pointers, and the second zone held representations of atomic values tagged with their types. In such a scheme, before execution of the main program begins, the zone of memory holding nodes for pairs of pointers is initialized by organizing it into a single one-way linked list called the *available space list*. The available space list is a pool of free memory containing nodes that can be allocated on demand whenever the need arises. Generalized lists can be constructed from building blocks removed from the front of this available space list. Thus the available space list turns memory into a flexible commodity that can be allocated on demand during the running of a program.

Now we consider two policies for recycling used list nodes. These are called *storage reclamation policies* since they reclaim space for nodes that are no longer in use. Policy 1 places deallocation of list nodes under direct programmer control. Policy 2 automatically collects deallocated nodes when they are needed.

- *Policy 1—Explicit Deallocation by the Programmer*

In some list processing systems, when a list node is no longer needed, it is the direct responsibility of the programmer to return it to the available space list (by linking it back onto the available space list, either at the beginning or the end of the available space list). The early list processing language IPL IV used this policy for recycling unused list nodes for further use.

- *Policy 2—Automatic Storage Reclamation*

In other list processing systems, while list nodes are removed from the front of the available space list when needed, the programmer is not responsible for returning unused nodes back to the available space list. When the available space list runs out of nodes to allocate (i.e., when it becomes empty), an automatic storage reclamation process, called *garbage collection*, is invoked. For example, the LISP programming language uses automatic storage reclamation policies.

Let's look for a moment at how the garbage collection process works. First of all, it is assumed that each list node has a special bit allocated in it, called the *mark bit*. This mark bit can be set to one of two values, *free* or *reserved* (where, for example, *free* is represented by the bit value "0" and *reserved* is represented by the bit value "1").

the available space list

explicit deallocation

automatic reclamation

how garbage collection works

using
mark al

The process proceeds in three phases. In the first phase, called the *initialization phase*, a pass is made through the list memory region setting the mark bits on each node to *free*. In the second phase, called the *marking phase*, all list nodes currently in use are identified and are marked by having their mark bits set to *reserved*. The idea is to mark all the nodes in current use to prevent them from being collected during the final phase. In the third phase, called the *gathering phase*, another sweep is made through the list space, and all nodes marked *free* are linked together into a new available space list. The available space list now consists of all the inaccessible nodes that were not previously incorporated into structures in use. These inaccessible nodes are sometimes called *garbage*, and the process of identifying them and linking them together into a new available space list is called *garbage collection*.

Program Strategy 7.19 outlines the main steps in the "mark and gather" strategy for garbage collection. On line 8 of this program strategy, there is an item field in the ListNode class that contains an Object pointer. However, we assume that the item pointer in a list node containing a pointer pair can point either to another list node in the *ordered pair zone* or to an atomic value node in the *atomic value zone*. (See Fig. 7.12 for an illustration of these two zones in the case of a LISP-like list processing system.) An *address range discrimination* can be used at the assembly language level to distinguish whether a pointer points to an address inside the ordered pair zone or the atomic value zone. An assembly language test can thus distinguish which kind of operations to apply to a node, depending on whether the node resides in the ordered pair zone or the atomic value zone. (This is one way to implement the instanceof operator in a Java boolean expression such as (L.item instanceof ListNode) that is used in line 9 of Program Strategy 7.20 on p. 229.)

An unfinished method to implement in Program Strategy 7.19 is the marking method markListNodesInUse, which is called on line 31. Let's investigate how to organize a marking process. Program Strategy 7.20 provides a sketch. We assume that we start with a single ListNode pointer, as the value of the variable L, and our task is to trace out and mark all ListNodes accessible from L via any path of pointers starting with either the item field or the link field of L's referent.

You can see that Program Strategy 7.20 makes two recursive calls (on lines 10 and 14). These recursive calls apply the node marking process to the nodes referenced by the item and link pointers of the node referenced by the pointer in L. When the function markListNodesInUse looks at the node referenced by the pointer in L, it immediately tests to see if the node is already marked as RESERVED. If a node is already marked as RESERVED, it has already been visited by the marking process along some other path of pointers, and it does not need to be considered further. However, if the node is not yet marked as RESERVED, its markBit field is immediately marked as RESERVED, and the marking process proceeds further to examine the nodes referenced by the item and link pointers. If the pointer in the item field points to a ListNode, then the marking process is applied to the node it references. In any event, the marking process is applied to the node referenced by the pointer in the link field. The process eventually terminates when every ListNode accessible via a path of pointers from the initial pointer in L is marked as RESERVED.

using recursive calls to
mark all accessible nodes

```

static final byte FREE = 0;
static final byte RESERVED = 1;

/* each ListNode has a markBit which is either FREE or RESERVED */
5
class ListNode {
    byte    markBit;           // FREE or RESERVED
    Object  item;
    ListNode link;
10
}

/* assume further that all ListNodes are allocated inside a region of */
/* memory as an array of nodes called the listNodeArray, as follows: */

15
    ListNode listNodeArray[listNodeArraySize];
    ListNode avail;           // avail will point to the available space list

void garbageCollection() {
20
    int i;                   // i is a local variable that indexes the listNodeArray

    /* phase 1—Initialization Phase—mark all ListNodes FREE */

    for ( i = 0; i < listNodeArraySize; i++ ) {
25
        listNodeArray[i].markBit = FREE;
    }

    /* phase 2—Marking Phase—mark all ListNodes in use RESERVED */
30
    // use the method markListNodesInUse of Program Strategy 7.20
    // to mark all list nodes in use

    /* phase 3—Gathering Phase—link all FREE ListNodes together */
35
    avail = null;
    for ( i = 0; i < listNodeArraySize; i++ ) {
        if ( listNodeArray[i].markBit == FREE ) {
40
            listNodeArray[i].link = avail;
            avail = listNodeArray[i];
        }
    } // at the conclusion, avail is the new available space list
}
}

```

Program Strategy 7.19 Garbage Collection by Marking and Gathering

Heaps and Dynamic Memory Allocation

To support dynamic memory allocation for Java objects of different sizes we cannot use the available space list technique discussed in the previous subsection. This is because the available space list contains linked blocks of *identical* sizes, whereas what we need to provide is a means for allocating blocks of memory of *differing* sizes.

allocating blocks of
different sizes

```

void markListNodesInUse(ListNode L) {
    if ( ( L != null ) && ( L.markBit != RESERVED ) ) {
        L.markBit = RESERVED;

        if (L.item instanceof ListNode) {
            markListNodesInUse(L.item);
        }

        markListNodesInUse(L.link);
    }
}
    
```

Program Strategy 7.20 Marking List Nodes in Use

The problem we face is this. Given a request size, n , for a block of memory of n bytes, how can we allocate a block of size n ? More generally, how can we organize a region of memory to allow us to allocate and free blocks of memory of various different sizes, given a sequence of requests to do so?

Figure 7.21 shows a region of memory called a *heap*. It contains blocks of memory of various sizes that have been reserved in response to allocation requests. The shaded region represents unallocated memory.

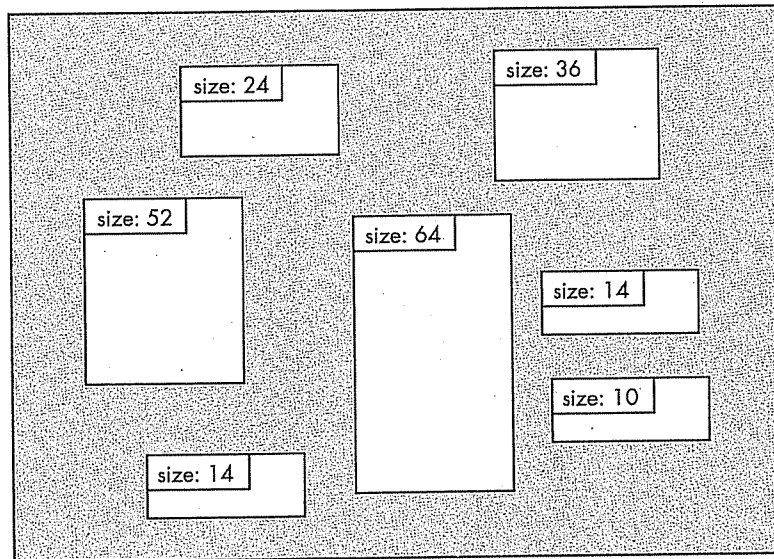


Figure 7.21 A Heap Containing Reserved Blocks of Various Sizes

3

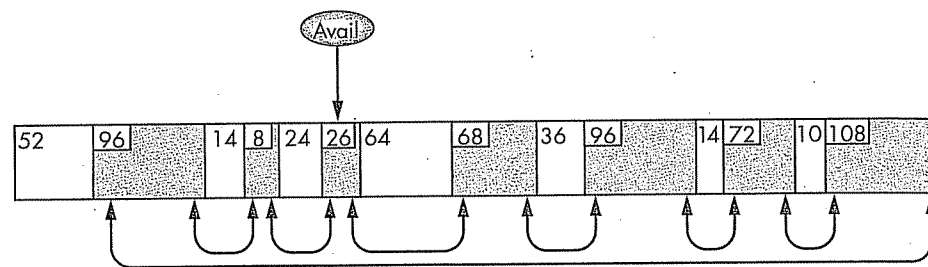


Figure 7.22 The Two-Way Linked List of Free Blocks in a Heap

The question now arises: How can we organize the unallocated space sitting between the reserved blocks so that future allocation requests can be satisfied? Two of the allocation policies we shall examine use a circular, two-way linked list of unallocated blocks. Figure 7.22 gives us another picture of the heap shown in Fig. 7.21. It shows the heap as being arranged in one linear zone of memory (with memory addresses increasing, say, from left to right), with shaded blocks of free memory containing their sizes and arranged into a two-way linked list using double-headed arrows to symbolize the two-way links. Since this two-way list is circular, we will call it a *ring*, in what follows. (Note: Even though Fig. 7.22 shows the two-way linked list as linking the free blocks together in increasing order of their addresses, in general, this will not be the case in actual applications. Instead, the free blocks can be linked into a ring in any order.)

Suppose we are given a pointer, *Avail*, to one of the free blocks on the ring of free blocks. Starting from any block on the ring, we can move either left or right to neighboring blocks. We can also search the ring in either direction to find a block, *B*, of suitable size to satisfy a memory allocation request. We can also remove block *B* from the ring by linking *B*'s left and right neighbors to one another, and we can mark *B* as being reserved, to put it in service to satisfy a request.

Let's now take a look at two different policies for allocating a new block of memory, given a request for a block of size n .

First-Fit

In the *first-fit* policy, we travel around the ring of free blocks starting at the block referenced by the pointer, *Avail*, until we find the first block *B* such that $\text{Size}(B) \geq n$, and we use *B* to satisfy the request. If $\text{Size}(B) > n$, then we break *B* into two blocks, B_1 of size n and B_2 of size $(\text{Size}(B) - n)$. We use B_1 to satisfy the request, and we link B_2 back into the ring of free blocks. If $\text{Size}(B) == n$, then we have what is called an *exact fit*, and we detach *B* from the ring of free blocks and use it directly to satisfy the request. In this second *exact fit* case, there exists no remaining unused block B_2 to link back into the ring of free blocks.

For example, in Fig. 7.22, suppose the request size is 72. We start at the block whose size is 26, which is referenced by the pointer *Avail*, and we compare the request size 72 with the block size 26. Since the block of size 26 is not big enough to satisfy

using circular two-way lists
of unallocated blocks

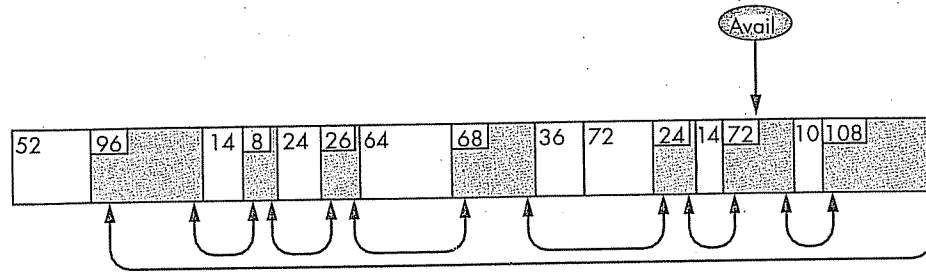


Figure 7.23 First-Fit Allocation of a Block of Size 72

the request, we advance around the ring (moving, let us say, in the rightward direction), and we consider next the block of size 68. This block is not big enough either, so we again move to the right to the block of size 96. This time, the block of size 96 is big enough. So we divide it into a block of size 72 and a block of size 24. We use the block of size 72 to satisfy the request, and we link the remaining block of size 24 back onto the ring of free blocks. The result of doing this is shown in Fig. 7.23. Note that the *Avail* pointer has been reset to point to the block of size 72 to the right of the original block of size 96 that we split in order to satisfy the request.

Best-Fit

Under the *best-fit policy*, we travel around the ring attempting to find a block that is the "best fit" for the request size n , as follows. Starting at the block, B , referenced by the pointer *Avail*, we compare $Size(B)$ to the request size n . If $Size(B) == n$, we have an *exact fit*, so we use block B to satisfy the request immediately by detaching it from the ring of free blocks and marking it as reserved. But if $Size(B) \neq n$, then we continue to search around the ring. We stop whenever we find an exact fit. But if we travel all the way around the ring without having found an exact fit, we use the closest fit we discovered on our trip around the ring. The closest fit is the block B on the ring having the property that $(Size(B) - n)$ is as small as possible. Again, we split the block B into a block B_1 of size n and a block B_2 of size $(Size(B) - n)$. We mark B_1 as reserved, and we link B_2 back into the ring of free blocks.

Figure 7.24 (on page 232) shows the result of using the best-fit policy to allocate a block of size 72, starting with the situation given in Fig. 7.22. Starting with the situation in Fig. 7.24, if we use the best-fit policy to satisfy a request to allocate a block of size 64, the block of size 68 would be selected (since it is the tightest fit, having the least excess storage of any of the blocks on the ring of sizes 108, 96, 68, and 96 that are large enough to satisfy a request of size 64).

Fragmentation and Coalescing

fragmentation If we operate a heap storage allocation system for a while, using either the first-fit policy or the best-fit policy, the ring of free blocks will tend to contain smaller and small-

sitting
Two of
unallo-
7.21. It
memory
ry con-
l arrows
t a ring,
linking
will not
ring in

g of free
, neigh-
k, B, of
B from
ark B as

of mem-

ock ref-
≥ n, and
s, B₁ of
link B₂
an exact
isfy the
to link

e block
request
satisfy

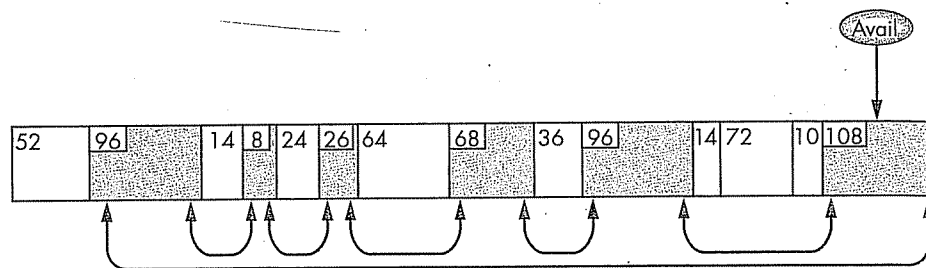


Figure 7.24 Best-Fit Allocation of a Block of Size 72

er blocks. This happens because we keep splitting larger blocks to satisfy requests for more storage. This tendency is called *fragmentation*.

allocation failure

The problem with fragmentation is that if the free blocks on the ring keep getting divided up into smaller and smaller size blocks, there may come a time when we encounter a request to allocate a block of size n that cannot be satisfied because all the free blocks are too small. This is called *allocation failure*.

coalescing

To reduce the chances of encountering allocation failure, we can perform what is called *coalescing*. When we *coalesce* two blocks that are sitting next to one another in memory, we join them into a single larger block. A good moment to try coalescing is when we free the storage for a block, B , and attempt to return it to the ring of free blocks. Instead of just linking B into the ring of free blocks, we can look at B 's immediate left and right neighbor blocks (in address order, not ring order) and if either of these neighbors is free, we can join it to B to make a larger coalesced block before putting this larger block back on the ring of free blocks.

For example, if we attempted to free the storage for the block of size 36 in Fig. 7.24, we could coalesce it with its two neighbors of sizes 68 and 96, getting a new large coalesced block of size $200 == (68 + 36 + 96)$ to return to the ring of free blocks. (To make this coalescing policy work efficiently, we need to store each free block's size and a mark bit designating it as *free* not only at the top of each block but also at its bottom boundary.)

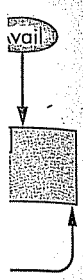
equilibrium

We say that a storage allocation policy is operating under *equilibrium* when, over time, the average amount of space in new blocks being reserved is equal to the average amount of space in blocks being freed, and when, in addition, the distribution of block sizes in the system remains the same. We need to use a coalescing policy to combat the tendency toward storage fragmentation, since, in most cases, without using a coalescing policy, the system will not achieve equilibrium over time (because the distribution of block sizes will keep getting smaller and smaller and the risk of allocation failure will increase).

using roving pointers

One more detail is worth mentioning in connection with the use of the first-fit and best-fit policies. This concerns the importance of using a *roving pointer*, *Avail*, to point to blocks on the ring of free blocks. In the roving pointer technique, we let *Avail* point to the successive blocks on the ring of free blocks when we search for a block B to satisfy a request, and after B has been located, we make *Avail* point to the block immediately beyond B on the ring. (By contrast, in a nonroving pointer tech-

movin



nique, we always start our search for block B at a fixed initial block referenced by *Avail*.) The result of using the roving pointer technique is that *Avail* keeps circulating around and around the ring of free blocks, and immediately ahead of it, the blocks have had the longest time among any others to coalesce and form bigger new blocks due to the return of freed blocks. If we keep scanning along the ring of free blocks from a fixed initial starting point (as we do in the nonroving pointer technique), then small fragments will tend to concentrate in the initial part of the search path, increasing the time required to satisfy larger requests. This inefficiency is quite noticeable and severe in some cases.

Compacting to Deal with Allocation Failures

When we make a request to allocate a block of size *n*, and the system finds that no block on the ring of free blocks is large enough to satisfy the request, it may still be the case that the sum of the block sizes on the ring of free blocks is larger than *n*. In this case, allocation failure could be said to have occurred because of the fragmentation of free blocks on the ring of free blocks. In particular, if we had a way of coalescing all the free blocks together into a single large block, we could proceed to satisfy the original request.

The process of moving all the reserved blocks into one end of the heap, while moving all the free blocks into the opposite end and coalescing them all into one large free block is called *compacting the heap* (or *heap compaction*). But to accomplish this, we need to make it possible not only for the reserved blocks to shift position but also for all external pointers to these blocks to be updated to point to the new addresses to which the blocks have been moved.

One technique for making this process easy is to use what are called *handles*. Handles are *double pointers*—or pointers to pointers. To create handles, we set aside a region of the heap to contain a zone of *master pointers*. These master pointers point to reserved blocks. Then we use a pointer to a block B's master pointer as the *handle* to B. We use handles both to provide external access to blocks from outside the heap, and also to allow blocks to contain pointers to point to one another inside the heap. Figure 7.25 shows a heap divided into a normal block allocation heap zone and a master pointer zone.

If we agree always to access reserved blocks in the normal heap zone using their handles, then the stage is set to make it easy to move the blocks around in the heap. All we have to do when we move a block, B, is to change B's master pointer to point to the new location to which B has been moved. After B has been moved to a new location, external access to B through its handle still works. To make it easy to update B's master pointer when we move B, we can agree to store a copy of B's handle inside B (along with B's size and its *mark bit* telling whether B is *free* or *reserved*).

Compacting the heap then consists of scanning the heap zone from bottom to top, moving each reserved block down as far as possible and updating the block's master pointer to point to its new location. Finally, the free space at the top is joined into one large free block. The heap is then said to be *compacted*. Figure 7.26 shows the result of compacting the heap illustrated in Fig. 7.25.

ests for
getting
then we
use all

what is
ther in
scing is
of free
imme-
ther of
before

in Fig.
a new
blocks.
block's
also at

n, over
e aver-
tion of
licy to
without
because
risk of

first-fit
vail, to
we let
h for a
to the
r tech-

compacting the heap

handles

moving blocks in the heap
and updating their
master pointers

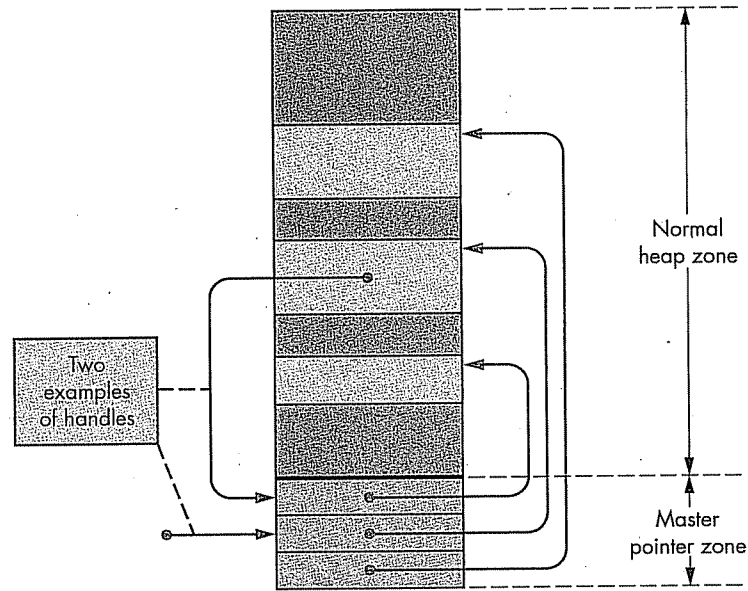


Figure 7.25 A Heap with a Master Pointer Zone and Handles

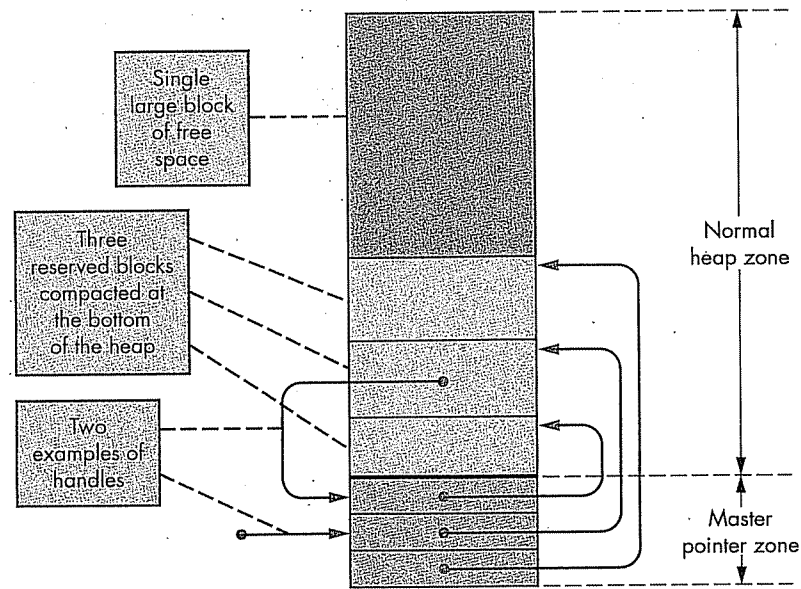


Figure 7.26 A Compacted Heap

double dereferencing

If the heap in an implementation of the Java Virtual Machine (JVM) were to be represented using blocks of memory for objects referenced by handles, then accessing objects from their reference values would have to be implemented by *double pointer dereferencing*. For example, to access the item field of a ListNode, L, using the expression L.item, the reference value in L would be a handle that accesses the actual block of storage for a ListNode, and implicit double dereferencing would be required to access the actual block of memory containing the item field.

handles versus single pointers

Comparing Uses of Heaps in Applications

Comparing the use of single pointers with use of handles, we observe that it costs more both in *space* (to store two pointers instead of just one) and in *time* (to perform double dereferencing instead of single dereferencing) when we use handles instead of pointers. What we gain, however, is convenience in arranging for the underlying storage allocation scheme to be able to accommodate a range of allocation request sizes and to be able to compact memory easily so we can coalesce all fragmented blocks of free space into one large free space block.

applications of heaps

It is perhaps not surprising then that the use of heaps containing blocks referenced by handles is a popular way to organize memory in some contemporary computing systems. One vendor's operating system divides a computer memory into a *system heap* (to contain operating system data structures) and various *application heaps*, each supporting given software applications. Software vendors can write software (such as spreadsheets, word processors, painting and drawing programs, electronic mail programs, and so forth) relying on the availability of such heaps to support their applications. Application software typically uses blocks in an application heap to contain representations of data supporting menus, windows, pictures, text, file control blocks, device control blocks, buttons, scroll bars, and all manner of application support data structures.

pausing for storage reclamation

Some modern object-oriented programming languages allocate data structures representing *objects* in heaps and automatically assume that these objects are referenced by handles, so the programmer never needs to use double-dereferencing notation in a program to refer to objects.

Reference Counts

There is one more concept worth mentioning before we close this section on dynamic memory allocation techniques. This is the idea of *reference counts*.

One of the troubles with the storage reclamation techniques we have discussed so far is that they cause a program to pause while they reorganize memory. The garbage collection technique uses a three pass algorithm to sweep memory setting mark bits to free, then to mark all list nodes in use, and finally to link all unused list nodes into a reconstituted available space list. The heap compaction algorithm also requires making a sweep of the heap zone during which reserved blocks get moved and their master pointers get updated. Each of these processes requires time $O(N)$, where N is the size of the memory being reorganized.

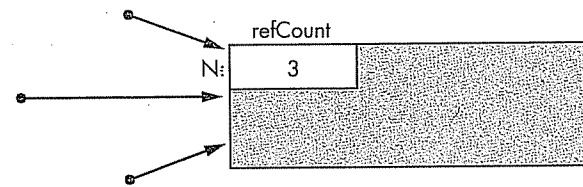


Figure 7.27 A node, N, with a Reference Count of Three

responding in real time

If we have a *real-time application*, requiring a system to respond quickly—within a fraction of a second, let's say, to incoming stimulus events (as is the case in systems such as robots used in manufacturing assembly lines or in collision avoidance systems in airline aircraft)—then we have to guarantee that the system will not be taken offline for a period of time longer than the required real-time response interval. Garbage collection and heap compaction algorithms both have two unfortunate features: (a) it is unpredictable when the moment to execute them will arise, and (b) they take a considerable amount of time to run to completion (often more than a few fractions of a second when a system has a huge list or heap memory, especially on slower computers or computers with huge random access memories). Consequently, it is important to know about alternative *storage reclamation techniques* that work *incrementally* instead of all at once. (An *incremental* technique is one that performs its work in tiny increments, doing its job bit by bit every now and then.)

The use of *reference counts* can provide an incremental storage reclamation technique under suitable circumstances. A *reference count field* in a node, N, is a data field containing a number that counts the number of pointers that reference N. Figure 7.27 illustrates a node N having three pointers that point to it and having a reference count of three in its refCount field.

incremental storage reclamation

Each time we create and use a new pointer that references N, we increase N's refCount by one, and each time we destroy a pointer to N, we decrease N's refCount by one. Whenever N's refCount reaches zero, we know that no more pointers in the system refer to N, so we can return the block of storage used to hold N to the available space list (or to the ring of free blocks in the heap, whichever is appropriate for the storage allocation technique we are using). This policy accomplishes something important. Namely, it automatically frees the storage for N the instant N's storage is no longer in use. Free storage in the system is therefore reclaimed incrementally (instead of all at once, as is the case for mark-and-gather garbage collection or heap compaction). If a real-time system needs to use dynamic memory allocation techniques, the reference count technique may provide a solution that helps the system meet its real-time response constraints.

7.6 REVIEW QUESTIONS

1. What is the difference between static and dynamic memory allocation?
2. Explain what is done in each of the three phases of garbage collection using the marking and gathering technique.

3. What is a heap? How is it used to support dynamic memory allocation?
4. What is the difference between the first-fit and best-fit policies for dynamic memory allocation?
5. What do fragmentation and coalescing refer to in the case of heaps?
6. What are handles? How are they affected by heap compaction?
7. What are reference counts? How do they provide for incremental storage reclamation?

7.6 EXERCISES

1. Why is it the case that, to make the free block coalescing policy work efficiently, we need to store each free block's size and a mark bit designating it as free not only at the top of each block but also at its bottom boundary?
2. Can you draw a picture of several nodes with nonzero reference counts that are linked to one another but are not referenced from the outside? What defect does this imply in the reference count method for incremental storage reclamation?

Pitfalls

- *Improper initialization of your data representations*

Did you remember to initialize all data structures properly? When creating data representations for lists, strings, or dynamic memory allocation support packages, it is easy to overlook proper initialization. Does each one-way, linked list terminate with a null link? Is each string buffer's capacity field initialized properly? Did you properly initialize all the links, sizes, tags, master pointers, and boundary markers in your heap?

The use of formatted debugging aids is highly useful in detecting deficiencies in your implementations. If you can see a printed representation of each of your data structures, your errors, especially those errors related to improper initialization, will tend to leap out at you. In fact, a good policy is to implement the formatted debugging aids before implementing the algorithms for the operations. That way, you will be ready to check out the operation of each newly implemented algorithm thoroughly.

- *Unannounced overflows and allocation failures*

When using sequential representations that can overflow some implementations do not give any overflow warnings or do not throw any Java exceptions that can be handled. In some dynamic memory allocation systems using heaps, allocation failure can happen unannounced. One policy used to signal an allocation failure in a heap is to return a null handle as the reply to an allocation request to allocate a block of size n . In a properly implemented Java system, the JVM will signal an `OutOfMemoryError` when you exhaust the free memory at run time.