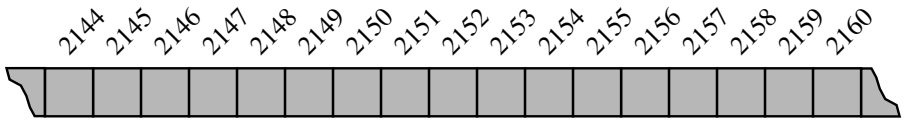
Our study of data structures thus far has focused primarily upon the efficiency of computations, as measured by the number of primitive operations that are executed on a central processing unit (CPU). In practice, the performance of a computer system is also greatly impacted by the management of the computer's memory systems. In our analysis of data structures, we have provided asymptotic bounds for the overall amount of memory used by a data structure. In this chapter, we consider more subtle issues involving the use of a computer's memory system.

We first discuss ways in which memory is allocated and deallocated during the execution of a computer program, and the impact that this has on the program's performance. Second, we discuss the complexity of multilevel memory hierarchies in today's computer systems. Although we often abstract a computer's memory as consisting of a single pool of interchangeable locations, in practice, the data used by an executing program is stored and transferred between a combination of physical memories (e.g., CPU registers, caches, internal memory, and external memory). We consider the use of classic data structures in the algorithms used to manage memory, and how the use of memory hierarchies impacts the choice of data structures and algorithms for classic problems such as searching and sorting.

15.1 Memory Management

In order to implement any data structure on an actual computer, we need to use computer memory. Computer memory is organized into a sequence of ***words***, each of which typically consists of 4, 8, or 16 bytes (depending on the computer). These memory words are numbered from 0 to $N-1$, where N is the number of memory words available to the computer. The number associated with each memory word is known as its ***memory address***. Thus, the memory in a computer can be viewed as basically one giant array of memory words. For example, in Figure 5.1 of Section 5.2, we portrayed a section of the computer's memory as follows:



In order to run programs and store information, the computer's memory must be ***managed*** so as to determine what data is stored in what memory cells. In this section, we discuss the basics of memory management, most notably describing the way in which memory is allocated to store new objects, the way in which portions of memory are deallocated and reclaimed, when no longer needed, and the way in which the Python interpreter uses memory in completing its tasks.

15.2.2 Caching Strategies

The significance of the memory hierarchy on the performance of a program depends greatly upon the size of the problem we are trying to solve and the physical characteristics of the computer system. Often, the bottleneck occurs between two levels of the memory hierarchy—the one that can hold all data items and the level just below that one. For a problem that can fit entirely in main memory, the two most important levels are the cache memory and the internal memory. Access times for internal memory can be as much as 10 to 100 times longer than those for cache memory. It is desirable, therefore, to be able to perform most memory accesses in cache memory. For a problem that does not fit entirely in main memory, on the other hand, the two most important levels are the internal memory and the external memory. Here the differences are even more dramatic, for access times for disks, the usual general-purpose external-memory device, are typically as much as 100000 to 1000000 times longer than those for internal memory.

To put this latter figure into perspective, imagine there is a student in Baltimore who wants to send a request-for-money message to his parents in Chicago. If the student sends his parents an email message, it can arrive at their home computer in about five seconds. Think of this mode of communication as corresponding to an internal-memory access by a CPU. A mode of communication corresponding to an external-memory access that is $500,000$ times slower would be for the student to walk to Chicago and deliver his message in person, which would take about a month if he can average 20 miles per day. Thus, we should make as few accesses to external memory as possible.

Most algorithms are not designed with the memory hierarchy in mind, in spite of the great variance between access times for the different levels. Indeed, all of the algorithm analyses described in this book so far have assumed that all memory accesses are equal. This assumption might seem, at first, to be a great oversight—and one we are only addressing now in the final chapter—but there are good reasons why it is actually a reasonable assumption to make.

One justification for this assumption is that it is often necessary to assume that all memory accesses take the same amount of time, since specific device-dependent information about memory sizes is often hard to come by. In fact, information about memory size may be difficult to get. For example, a Python program that is designed to run on many different computer platforms cannot easily be defined in terms of a specific computer architecture configuration. We can certainly use architecture-specific information, if we have it (and we will show how to exploit such information later in this chapter). But once we have optimized our software for a certain architecture configuration, our software will no longer be device-independent. Fortunately, such optimizations are not always necessary, primarily because of the second justification for the equal-time memory-access assumption.

15.1.1 Memory Allocation

With Python, all objects are stored in a pool of memory, known as the ***memory heap*** or ***Python heap*** (which should not be confused with the "heap" data structure presented in Chapter 9). When a command such as

```
w = Widget( )
```

is executed, assuming Widget is the name of a class, a new instance of the class is created and stored somewhere within the memory heap. The Python interpreter is responsible for negotiating the use of space with the operating system and for managing the use of the memory heap when executing a Python program.

The storage available in the memory heap is divided into ***blocks***, which are contiguous array-like "chunks" of memory that may be of variable or fixed sizes. The system must be implemented so that it can quickly allocate memory for new objects. One popular method is to keep contiguous "holes" of available free memory in a linked list, called the ***free list***. The links joining these holes are stored inside the holes themselves, since their memory is not being used. As memory is allocated and deallocated, the collection of holes in the free lists changes, with the unused memory being separated into disjoint holes divided by blocks of used memory. This separation of unused memory into separate holes is known as ***fragmentation***. The problem is that it becomes more difficult to find large continuous chunks of memory, when needed, even though an equivalent amount of memory may be unused (yet fragmented). Therefore, we would like to minimize fragmentation as much as possible.

There are two kinds of fragmentation that can occur. ***Internal fragmentation*** occurs when a portion of an allocated memory block is unused. For example, a program may request an array of size 1000, but only use the first 100 cells of this array. There is not much that a run-time environment can do to reduce internal fragmentation. ***External fragmentation***, on the other hand, occurs when there is a significant amount of unused memory between several contiguous blocks of allocated memory. Since the run-time environment has control over where to allocate memory when it is requested, the run-time environment should allocate memory in a way to try to reduce external fragmentation as much as reasonably possible.

Several heuristics have been suggested for allocating memory from the heap so as to minimize external fragmentation. The ***best-fit algorithm*** searches the entire free list to find the hole whose size is closest to the amount of memory being requested. The ***first-fit algorithm*** searches from the beginning of the free list for the first hole that is large enough. The ***next-fit algorithm*** is similar, in that it also searches the free list for the first hole that is large enough, but it begins its search from where it left off previously, viewing the free list as a circularly linked list (Section 7.2). The ***worst-fit algorithm*** searches the free list to find the largest hole of available memory, which might be done faster than a search of the entire free list

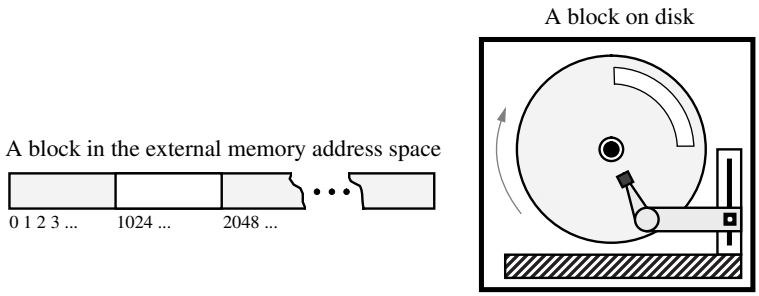


Figure 15.2: Blocks in external memory.

When implemented with caching and blocking, virtual memory often allows us to perceive secondary-level memory as being faster than it really is. There is still a problem, however. Primary-level memory is much smaller than secondary-level memory. Moreover, because memory systems use blocking, any program of substance will likely reach a point where it requests data from secondary-level memory, but the primary memory is already full of blocks. In order to fulfill the request and maintain our use of caching and blocking, we must remove some block from primary memory to make room for a new block from secondary memory in this case. Deciding which block to evict brings up a number of interesting data structure and algorithm design issues.

Caching in Web Browsers

For motivation, we consider a related problem that arises when revisiting information presented in Web pages. To exploit temporal locality of reference, it is often advantageous to store copies of Web pages in a ***cache*** memory, so these pages can be quickly retrieved when requested again. This effectively creates a two-level memory hierarchy, with the cache serving as the smaller, quicker internal memory, and the network being the external memory. In particular, suppose we have a cache memory that has m "slots" that can contain Web pages. We assume that a Web page can be placed in any slot of the cache. This is known as a ***fully associative*** cache.

As a browser executes, it requests different Web pages. Each time the browser requests such a Web page p, the browser determines (using a quick test) if p is unchanged and currently contained in the cache. If p is contained in the cache, then the browser satisfies the request using the cached copy. If p is not in the cache, however, the page for p is requested over the Internet and transferred into the cache. If one of the m slots in the cache is available, then the browser assigns p to one of the empty slots. But if all the m cells of the cache are occupied, then the computer must determine which previously viewed Web page to evict before bringing in p to take its place. There are, of course, many different policies that can be used to determine the page to evict.

Caching and Blocking

Another justification for the memory-access equality assumption is that operating system designers have developed general mechanisms that allow most memory accesses to be fast. These mechanisms are based on two important ***locality-of-reference*** properties that most software possesses:

- **Temporal locality**: If a program accesses a certain memory location, then there is increased likelihood that it accesses that same location again in the near future. For example, it is common to use the value of a counter variable in several different expressions, including one to increment the counter's value. In fact, a common adage among computer architects is that a program spends 90 percent of its time in 10 percent of its code.
- **Spatial locality**: If a program accesses a certain memory location, then there is increased likelihood that it soon accesses other locations that are near this one. For example, a program using an array may be likely to access the locations of this array in a sequential or near-sequential manner.

Computer scientists and engineers have performed extensive software profiling experiments to justify the claim that most software possesses both of these kinds of locality of reference. For example, a nested for loop used to repeatedly scan through an array will exhibit both kinds of locality.

Temporal and spatial localities have, in turn, given rise to two fundamental design choices for multilevel computer memory systems (which are present in the interface between cache memory and internal memory, and also in the interface between internal memory and external memory).

The first design choice is called ***virtual memory***. This concept consists of providing an address space as large as the capacity of the secondary-level memory, and of transferring data located in the secondary level into the primary level, when they are addressed. Virtual memory does not limit the programmer to the constraint of the internal memory size. The concept of bringing data into primary memory is called ***caching***, and it is motivated by temporal locality. By bringing data into primary memory, we are hoping that it will be accessed again soon, and we will be able to respond quickly to all the requests for this data that come in the near future.

The second design choice is motivated by spatial locality. Specifically, if data stored at a secondary-level memory location ℓ is accessed, then we bring into primary-level memory a large block of contiguous locations that include the location ℓ. (See Figure 15.2.) This concept is known as ***blocking***, and it is motivated by the expectation that other secondary-level memory locations close to ℓ will soon be accessed. In the interface between cache memory and internal memory, such blocks are often called ***cache lines***, and in the interface between internal memory and external memory, such blocks are often called ***pages***.

15.4.1 Multiway Merging

In a standard merge-sort (Section 12.2), the merge process combines two sorted sequences into one by repeatedly taking the smaller of the items at the front of the two respective lists. In a d-way merge, we repeatedly find the smallest among the items at the front of the d sequences and place it as the next element of the merged sequence. We continue until all elements are included.

In the context of an external-memory sorting algorithm, if main memory has size M and each block has size B, we can store up to M/B blocks within main memory at any given time. We specifically choose $d = (M/B) - 1$ so that we can afford to keep one block from each input sequence in main memory at any given time, and to have one additional block to use as a buffer for the merged sequence. (See Figure 15.5.)

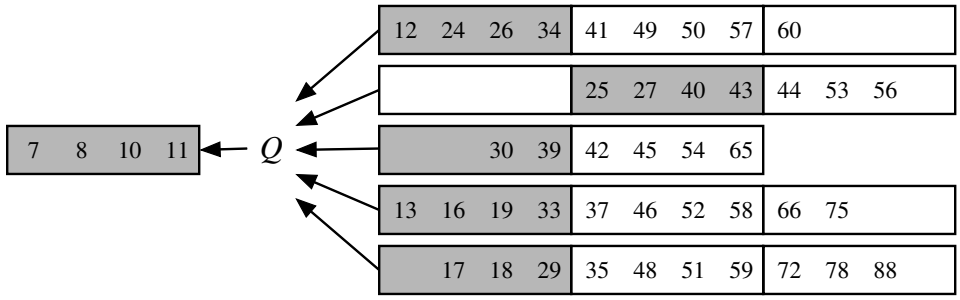


Figure 15.5: A d-way merge with $d = 5$ and $B = 4$. Blocks that currently reside in main memory are shaded.

We maintain the smallest unprocessed element from each input sequence in main memory, requesting the next block from a sequence when the preceding block has been exhausted. Similarly, we use one block of internal memory to buffer the merged sequence, flushing that block to external memory when full. In this way, the total number of transfers performed during a single d-way merge is $O(n/B)$, since we scan each block of list S_i once, and we write out each block of the merged list S' once. In terms of computation time, choosing the smallest of d values can trivially be performed using $O(d)$ operations. If we are willing to devote $O(d)$ internal memory, we can maintain a priority queue identifying the smallest element from each sequence, thereby performing each step of the merge in $O(\log d)$ time by removing the minimum element and replacing it with the next element from the same sequence. Hence, the internal time for the d-way merge is $O(n \log d)$.

Proposition 15.3: *Given an array-based sequence S of n elements stored compactly in external memory, we can sort S with $O((n/B)\log(n/B)/\log(M/B))$ block transfers and $O(n \log n)$ internal computations, where M is the size of the internal memory and B is the size of a block.*

5.2 Low-Level Arrays

To accurately describe the way in which Python represents the sequence types, we must first discuss aspects of the low-level computer architecture. The primary memory of a computer is composed of bits of information, and those bits are typically grouped into larger units that depend upon the precise system architecture. Such a typical unit is a ***byte***, which is equivalent to 8 bits.

A computer system will have a huge number of bytes of memory, and to keep track of what information is stored in what byte, the computer uses an abstraction known as a ***memory address***. In effect, each byte of memory is associated with a unique number that serves as its address (more formally, the binary representation of the number serves as the address). In this way, the computer system can refer to the data in "byte #2150" versus the data in "byte #2157," for example. Memory addresses are typically coordinated with the physical layout of the memory system, and so we often portray the numbers in sequential fashion. Figure 5.1 provides such a diagram, with the designated memory address for each byte.

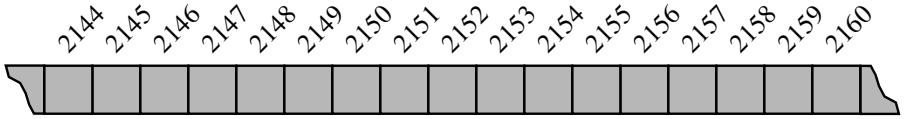


Figure 5.1: A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses.

Despite the sequential nature of the numbering system, computer hardware is designed, in theory, so that any byte of the main memory can be efficiently accessed based upon its memory address. In this sense, we say that a computer's main memory performs as ***random access memory (RAM)***. That is, it is just as easy to retrieve byte #8675309 as it is to retrieve byte #309. (In practice, there are complicating factors including the use of caches and external memory; we address some of those issues in Chapter 15.) Using the notation for asymptotic analysis, we say that any individual byte of memory can be stored or retrieved in $O(1)$ time.

In general, a programming language keeps track of the association between an identifier and the memory address in which the associated value is stored. For example, identifier `x` might be associated with one value stored in memory, while `y` is associated with another value stored in memory. A common programming task is to keep track of a sequence of related objects. For example, we may want a video game to keep track of the top ten scores for that game. Rather than use ten different variables for this task, we would prefer to use a single name for the group and use index numbers to refer to the high scores in that group.

15.2 Memory Hierarchies and Caching

With the increased use of computing in society, software applications must manage extremely large data sets. Such applications include the processing of online financial transactions, the organization and maintenance of databases, and analyses of customers' purchasing histories and preferences. The amount of data can be so large that the overall performance of algorithms and data structures sometimes depends more on the time to access the data than on the speed of the CPU.

15.2.1 Memory Systems

In order to accommodate large data sets, computers have a ***hierarchy*** of different kinds of memories, which vary in terms of their size and distance from the CPU. Closest to the CPU are the internal registers that the CPU itself uses. Access to such locations is very fast, but there are relatively few such locations. At the second level in the hierarchy are one or more memory ***caches***. This memory is considerably larger than the register set of a CPU, but accessing it takes longer. At the third level in the hierarchy is the ***internal memory***, which is also known as ***main memory*** or ***core memory***. The internal memory is considerably larger than the cache memory, but also requires more time to access. Another level in the hierarchy is the ***external memory***, which usually consists of disks, CD drives, DVD drives, and/or tapes. This memory is very large, but it is also very slow. Data stored through an external network can be viewed as yet another level in this hierarchy, with even greater storage capacity, but even slower access. Thus, the memory hierarchy for computers can be viewed as consisting of five or more levels, each of which is larger and slower than the previous level. (See Figure 15.1.) During the execution of a program, data is routinely copied from one level of the hierarchy to a neighboring level, and these transfers can become a computational bottleneck.

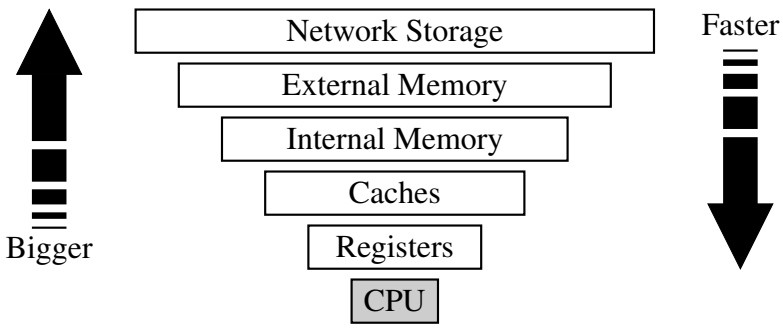


Figure 15.1: The memory hierarchy.

The Mark-Sweep Algorithm

In the mark-sweep garbage collection algorithm, we associate a "mark" bit with each object that identifies whether that object is live. When we determine at some point that garbage collection is needed, we suspend all other activity and clear the mark bits of all the objects currently allocated in the memory heap. We then trace through the active namespaces and we mark all the root objects as "live." We must then determine all the other live objects—the ones that are reachable from the root objects. To do this efficiently, we can perform a depth-first search (see Section 14.3.1) on the directed graph that is defined by objects reference other objects. In this case, each object in the memory heap is viewed as a vertex in a directed graph, and the reference from one object to another is viewed as a directed edge. By performing a directed DFS from each root object, we can correctly identify and mark each live object. This process is known as the "mark" phase. Once this process has completed, we then scan through the memory heap and reclaim any space that is being used for an object that has not been marked. At this time, we can also optionally coalesce all the allocated space in the memory heap into a single block, thereby eliminating external fragmentation for the time being. This scanning and reclamation process is known as the "sweep" phase, and when it completes, we resume running the suspended program. Thus, the mark-sweep garbage collection algorithm will reclaim unused space in time proportional to the number of live objects and their references plus the size of the memory heap.

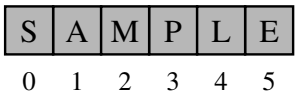
Performing DFS In-Place

The mark-sweep algorithm correctly reclaims unused space in the memory heap, but there is an important issue we must face during the mark phase. Since we are reclaiming memory space at a time when available memory is scarce, we must take care not to use extra space during the garbage collection itself. The trouble is that the DFS algorithm, in the recursive way we have described it in Section 14.3.1, can use space proportional to the number of vertices in the graph. In the case of garbage collection, the vertices in our graph are the objects in the memory heap; hence, we probably do not have this much memory to use. So our only alternative is to find a way to perform DFS in-place rather than recursively, that is, we must perform DFS using only a constant amount of additional storage.

The main idea for performing DFS in-place is to simulate the recursion stack using the edges of the graph (which in the case of garbage collection correspond to object references). When we traverse an edge from a visited vertex v to a new vertex w, we change the edge (v,w) stored in v's adjacency list to point back to v's parent in the DFS tree. When we return back to v (simulating the return from the "recursive" call at w), we can then switch the edge we modified to point back to w, assuming we have some way to identify which edge we need to change back.

5.2.2 Compact Arrays in Python

In the introduction to this section, we emphasized that strings are represented using an array of characters (not an array of references). We will refer to this more direct representation as a ***compact array*** because the array is storing the bits that represent the primary data (characters, in the case of strings).



Compact arrays have several advantages over referential structures in terms of computing performance. Most significantly, the overall memory usage will be much lower for a compact structure because there is no overhead devoted to the explicit storage of the sequence of memory references (in addition to the primary data). That is, a referential structure will typically use 64-bits for the memory address stored in the array, on top of whatever number of bits are used to represent the object that is considered the element. Also, each Unicode character stored in a compact array within a string typically requires 2 bytes. If each character were stored independently as a one-character string, there would be significantly more bytes used.

As another case study, suppose we wish to store a sequence of one million, 64-bit integers. In theory, we might hope to use only 64 million bits. However, we estimate that a Python list will use *four to five times as much memory*. Each element of the list will result in a 64-bit memory address being stored in the primary array, and an **int** instance being stored elsewhere in memory. Python allows you to query the actual number of bytes being used for the primary storage of any object. This is done using the getsizeof function of the sys module. On our system, the size of a typical int object requires 14 bytes of memory (well beyond the 4 bytes needed for representing the actual 64-bit number). In all, the list will be using 18 bytes per entry, rather than the 4 bytes that a compact list of integers would require.

Another important advantage to a compact structure for high-performance computing is that the primary data are stored consecutively in memory. Note well that this is not the case for a referential structure. That is, even though a list maintains careful ordering of the sequence of memory addresses, where those elements reside in memory is not determined by the list. Because of the workings of the cache and memory hierarchies of computers, it is often advantageous to have data stored in memory near other data that might be used in the same computations.

Despite the apparent inefficiencies of referential structures, we will generally be content with the convenience of Python's lists and tuples in this book. The only place in which we consider alternatives will be in Chapter 15, which focuses on the impact of memory usage on data structures and algorithms. Python provides several means for creating compact arrays of various types.