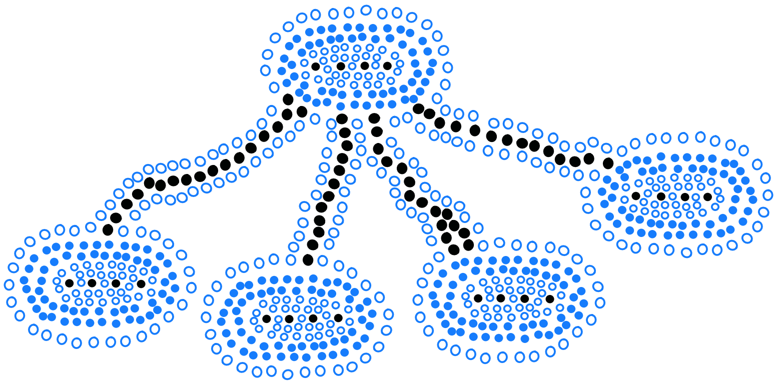
Chapter

15 Memory Management and B-Trees



Contents

15.1 Memory Management

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, as portrayed in Figure 15.1.

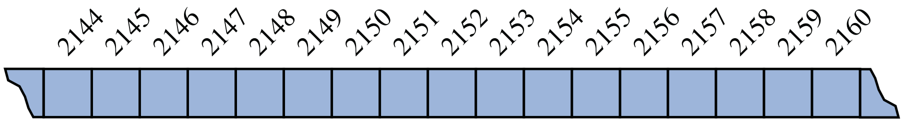


Figure 15.1: Memory addresses.

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 for various purposes in a Java program, and the way in which portions of memory are deallocated and reclaimed, when no longer needed.

15.1.1 Stacks in the Java Virtual Machine

A Java program is typically compiled into a sequence of byte codes that serve as "machine" instructions for a well-defined model—the ***Java Virtual Machine*** (***JVM***). The definition of the JVM is at the heart of the definition of the Java language itself. By compiling Java code into the JVM byte codes, rather than the machine language of a specific CPU, a Java program can be run on any computer that has a program that can emulate the JVM.

Stacks have an important application to the runtime environment of Java programs. A running Java program (more precisely, a running Java thread) has a private stack, called the ***Java method stack*** or just ***Java stack*** for short, which is used to keep track of local variables and other important information on methods as they are invoked during execution. (See Figure 15.2.)

More specifically, during the execution of a Java program, the Java Virtual Machine (JVM) maintains a stack whose elements are descriptors of the currently active (that is, nonterminated) invocations of methods. These descriptors are called ***frames***. A frame for some invocation of method "fool" stores the current values of the local variables and parameters of method fool, as well as information on method "cool" that called fool and on what needs to be returned to method "cool".

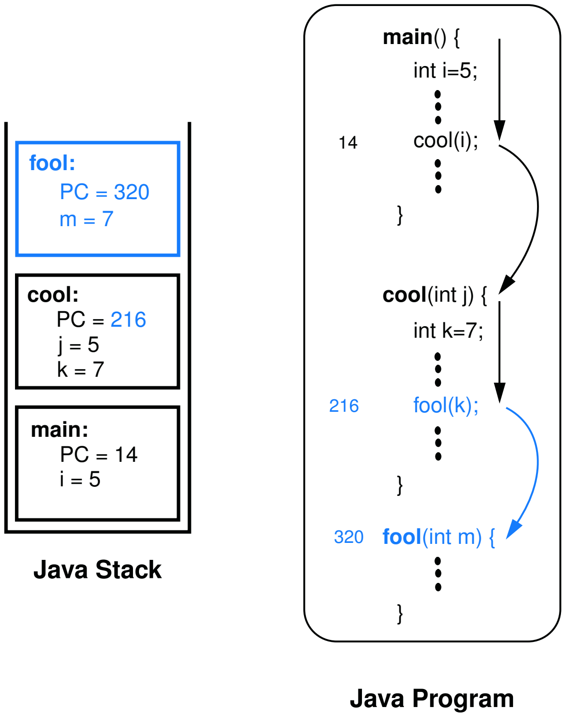


Figure 15.2: An example of a Java method stack: method fool has just been called by method cool, which itself was previously called by method main. Note the values of the program counter, parameters, and local variables stored in the stack frames. When the invocation of method fool terminates, the invocation of method cool will resume its execution at instruction 217, which is obtained by incrementing the value of the program counter stored in the stack frame.

Keeping Track of the Program Counter

The JVM keeps a special variable, called the ***program counter***, to maintain the address of the statement the JVM is currently executing in the program. When a method "cool" invokes another method "fool", the current value of the program counter is recorded in the frame of the current invocation of cool (so the JVM will know where to return to when method fool is done). At the top of the Java stack is the frame of the ***running method***, that is, the method that currently has control of the execution. The remaining elements of the stack are frames of the ***suspended methods***, that is, methods that have invoked another method and are currently waiting for it to return control to them upon its termination. The order of the elements in the stack corresponds to the chain of invocations of the currently active methods. When a new method is invoked, a frame for this method is pushed onto the stack. When it terminates, its frame is popped from the stack and the JVM resumes the processing of the previously suspended method.

Implementing Recursion

One of the benefits of using a stack to implement method invocation is that it allows programs to use ***recursion***. That is, it allows a method to call itself, as discussed in Chapter 5. We implicitly described the concept of the call stack and the use of frames within our portrayal of ***recursion traces*** in that chapter. Interestingly, early programming languages, such as Cobol and Fortran, did not originally use call stacks to implement function and procedure calls. But because of the elegance and efficiency that recursion allows, all modern programming languages, including the modern versions of classic languages like Cobol and Fortran, utilize a runtime stack for method and procedure calls.

Each box of a recursion trace corresponds to a frame of the Java method stack. At any point in time, the contents of the Java method stack corresponds to the chain of boxes from the initial method invocation to the current one.

To better illustrate how a runtime stack allows recursive methods, we refer back to the Java implementation of the classic recursive definition of the factorial function,

$$n! = n(n-1)(n-2)\cdots 1,$$

with the code originally given in Code Fragment 5.1, and the recursion trace in Figure 5.1. The first time we call method factorial(n), its stack frame includes a local variable storing the value n. The method recursively calls itself to compute $(n-1)!$, which pushes a new frame on the Java runtime stack. In turn, this recursive invocation calls itself to compute $(n-2)!$, etc. The chain of recursive invocations, and thus the runtime stack, only grows up to size $n+1$, with the most deeply nested call being factorial(0), which returns 1 without any further recursion. The runtime stack allows several invocations of the factorial method to exist simultaneously. Each has a frame that stores the value of its parameter n as well as the value to be returned. When the first recursive call eventually terminates, it returns $(n-1)!$, which is then multiplied by n to compute $n!$ for the original call of the factorial method.

The Operand Stack

Interestingly, there is actually another place where the JVM uses a stack. Arithmetic expressions, such as $((a+b)*(c+d))/e$, are evaluated by the JVM using an ***operand stack***. A simple binary operation, such as $a+b$, is computed by pushing a on the stack, pushing b on the stack, and then calling an instruction that pops the top two items from the stack, performs the binary operation on them, and pushes the result back onto the stack. Likewise, instructions for writing and reading elements to and from memory involve the use of pop and push methods for the operand stack. Thus, the JVM uses a stack to evaluate arithmetic expressions in Java.

15.1.2 Allocating Space in the Memory Heap

We have already discussed (in Section 15.1.1) how the Java Virtual Machine allocates a method's local variables in that method's frame on the Java runtime stack. The Java stack is not the only kind of memory available for program data in Java, however.

Dynamic Memory Allocation

Memory for an object can also be allocated dynamically during a method's execution, by having that method utilize the special **new** operator built into Java. For example, the following Java statement creates an array of integers whose size is given by the value of variable k:

```
int[ ] items = new int[k];
```

The size of the array above is known only at runtime. Moreover, the array may continue to exist even after the method that created it terminates. Thus, the memory for this array cannot be allocated on the Java stack.

The Memory Heap

Instead of using the Java stack for this object's memory, Java uses memory from another area of storage—the ***memory heap*** (which should not be confused with the "heap" data structure presented in Chapter 9). We illustrate this memory area, together with the other memory areas, in a Java Virtual Machine in Figure 15.3. 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.

To simplify the discussion, let us assume that blocks in the memory heap are of a fixed size, say, 1,024 bytes, and that one block is big enough for any object we might want to create. (Efficiently handling the more general case is actually an interesting research problem.)

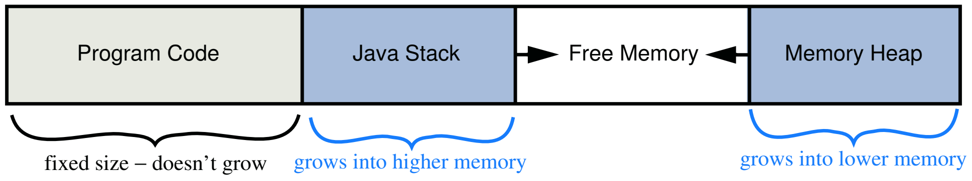


Figure 15.3: A schematic view of the layout of memory addresses in the Java Virtual Machine.

Memory Allocation Algorithms

The Java Virtual Machine definition requires that the memory heap be able to quickly allocate memory for new objects, but it does not specify the algorithm that should be used to do this. 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).

Two kinds of fragmentation 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. A runtime environment can not do much 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 runtime environment has control over where to allocate memory when it is requested (for example, when the **new** keyword is used in Java), the runtime environment should allocate memory in a way to try to reduce external fragmentation.

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 3.3). 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 if this list were maintained as a priority queue (Chapter 9). In each algorithm, the requested amount of memory is subtracted from the chosen memory hole and the leftover part of that hole is returned to the free list.

Although it might sound good at first, the best-fit algorithm tends to produce the worst external fragmentation, since the leftover parts of the chosen holes tend to be small. The first-fit algorithm is fast, but it tends to produce a lot of external fragmentation at the front of the free list, which slows down future searches. The next-fit algorithm spreads fragmentation more evenly throughout the memory heap, thus keeping search times low. This spreading also makes it more difficult to allocate large blocks, however. The worst-fit algorithm attempts to avoid this problem by keeping contiguous sections of free memory as large as possible.

15.1.3 Garbage Collection

In some languages, like C and C++, the memory space for objects must be explicitly deallocated by the programmer, which is a duty often overlooked by beginning programmers and is the source of frustrating programming errors even for experienced programmers. The designers of Java instead placed the burden of memory management entirely on the runtime environment.

As mentioned above, memory for objects is allocated from the memory heap and the space for the instance variables of a running Java program are placed in its method stacks, one for each running thread (for the simple programs discussed in this book there is typically just one running thread). Since instance variables in a method stack can refer to objects in the memory heap, all the variables and objects in the method stacks of running threads are called ***root objects***. All those objects that can be reached by following object references that start from a root object are called ***live objects***. The live objects are the active objects currently being used by the running program; these objects should ***not*** be deallocated. For example, a running Java program may store, in a variable, a reference to a sequence S that is implemented using a doubly linked list. The reference variable to S is a root object, while the object for S is a live object, as are all the node objects that are referenced from this object and all the elements that are referenced from these node objects.

From time to time, the Java virtual machine (JVM) may notice that available space in the memory heap is becoming scarce. At such times, the JVM can elect to reclaim the space that is being used for objects that are no longer live, and return the reclaimed memory to the free list. This reclamation process is known as ***garbage collection***. There are several different algorithms for garbage collection, but one of the most used is the ***mark-sweep algorithm***.

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 Java stacks of the currently running threads and we mark all the root objects in these stacks 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 referencing 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 don't have this much memory to use. So our only alternative is to find a way to perform DFS in-place rather than recursively.

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 now switch the edge we modified to point back to w. Of course, we need to have some way of identifying which edge we need to change back. One possibility is to number the references going out of v as 1, 2, and so on, and store, in addition to the mark bit (which we are using for the "visited" tag in our DFS), a count identifier that tells us which edges we have modified.

Using a count identifier requires an extra word of storage per object. This extra word can be avoided in some implementations, however. For example, many implementations of the Java virtual machine represent an object as a composition of a reference with a type identifier (which indicates if this object is an Integer or some other type) and as a reference to the other objects or data fields for this object. Since the type reference is always supposed to be the first element of the composition in such implementations, we can use this reference to "mark" the edge we changed when leaving an object v and going to some object w. We simply swap the reference at v that refers to the type of v with the reference at v that refers to w. When we return to v, we can quickly identify the edge (v,w) we changed, because it will be the first reference in the composition for v, and the position of the reference to v's type will tell us the place where this edge belongs in v's adjacency list.

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 15.4.) 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.

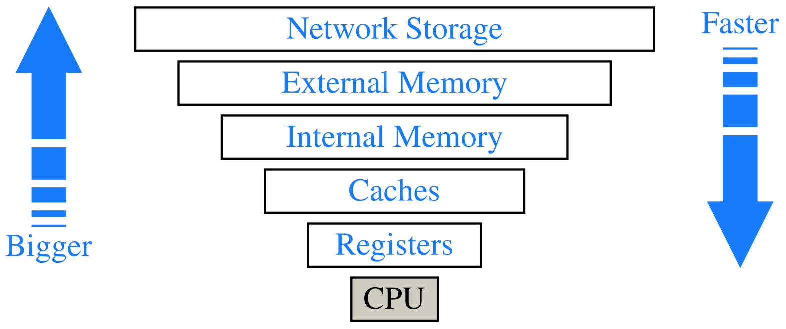


Figure 15.4: The memory hierarchy.

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 $100,000$ to $1,000,000$ 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 thus far described in this book 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 Java 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.

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% of its time in 10% 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 15.5.) 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***.

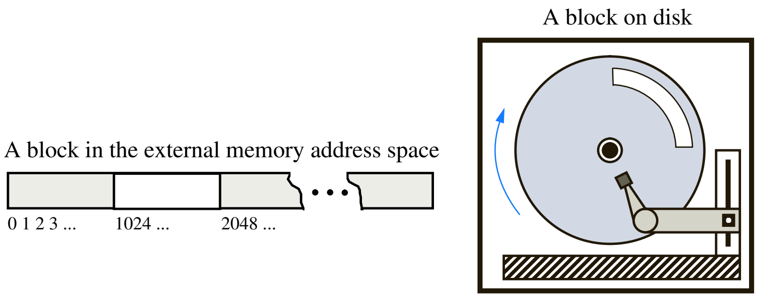


Figure 15.5: Blocks in external memory.

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 will 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.

Page Replacement Algorithms

Some of the better-known page replacement policies include the following (see Figure 15.6):

- **First-in, first-out (FIFO)**: Evict the page that has been in the cache the longest, that is, the page that was transferred to the cache furthest in the past.
- **Least recently used (LRU)**: Evict the page whose last request occurred furthest in the past.

In addition, we can consider a simple and purely random strategy:

- **Random**: Choose a page at random to evict from the cache.

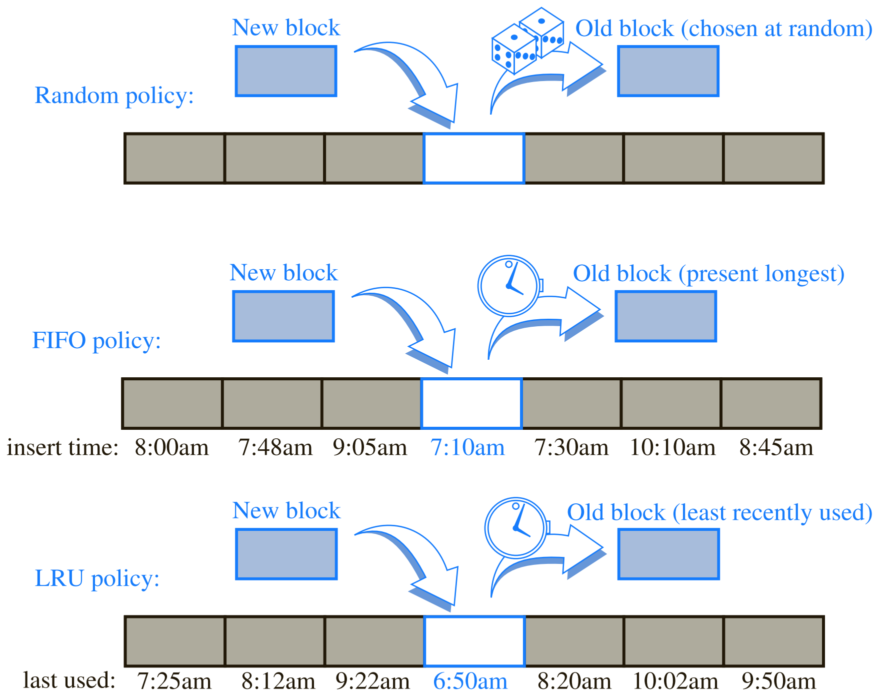

New block
Old block (chosen at random)
Random policy:
New block
Old block (present longest)
FIFO policy:
insert time: 8:00am 7:48am 9:05am 7:10am 7:30am 10:10am 8:45am
New block
Old block (least recently used)
LRU policy:
last used: 7:25am 8:12am 9:22am 6:50am 8:20am 10:02am 9:50am

Figure 15.6: The Random, FIFO, and LRU page replacement policies.

The Random strategy is one of the easiest policies to implement, for it only requires a random or pseudorandom number generator. The overhead involved in implementing this policy is an $O(1)$ additional amount of work per page replacement. Moreover, there is no additional overhead for each page request, other than to determine whether a page request is in the cache or not. Still, this policy makes no attempt to take advantage of any temporal locality exhibited by a user's browsing.

The FIFO strategy is quite simple to implement, as it only requires a queue Q to store references to the pages in the cache. Pages are enqueued in Q when they are referenced by a browser, and then are brought into the cache. When a page needs to be evicted, the computer simply performs a dequeue operation on Q to determine which page to evict. Thus, this policy also requires $O(1)$ additional work per page replacement. Also, the FIFO policy incurs no additional overhead for page requests. Moreover, it tries to take some advantage of temporal locality.

The LRU strategy goes a step further than the FIFO strategy, for the LRU strategy explicitly takes advantage of temporal locality as much as possible, by always evicting the page that was least-recently used. From a policy point of view, this is an excellent approach, but it is costly from an implementation point of view. That is, its way of optimizing temporal and spatial locality is fairly costly. Implementing the LRU strategy requires the use of an adaptable priority queue Q that supports updating the priority of existing pages. If Q is implemented with a sorted sequence based on a linked list, then the overhead for each page request and page replacement is $O(1)$. When we insert a page in Q or update its key, the page is assigned the highest key in Q and is placed at the end of the list, which can also be done in $O(1)$ time. Even though the LRU strategy has constant-time overhead, using the implementation above, the constant factors involved, in terms of the additional time overhead and the extra space for the priority queue Q, make this policy less attractive from a practical point of view.

Since these different page replacement policies have different trade-offs between implementation difficulty and the degree to which they seem to take advantage of localities, it is natural for us to ask for some kind of comparative analysis of these methods to see which one, if any, is the best.

From a worst-case point of view, the FIFO and LRU strategies have fairly unattractive competitive behavior. For example, suppose we have a cache containing m pages, and consider the FIFO and LRU methods for performing page replacement for a program that has a loop that repeatedly requests $m+1$ pages in a cyclic order. Both the FIFO and LRU policies perform badly on such a sequence of page requests, because they perform a page replacement on every page request. Thus, from a worst-case point of view, these policies are almost the worst we can imagine—they require a page replacement on every page request.

This worst-case analysis is a little too pessimistic, however, for it focuses on each protocol's behavior for one bad sequence of page requests. An ideal analysis would be to compare these methods over all possible page-request sequences. Of course, this is impossible to do exhaustively, but there have been a great number of experimental simulations done on page-request sequences derived from real programs. Based on these experimental comparisons, the LRU strategy has been shown to be usually superior to the FIFO strategy, which is usually better than the Random strategy.

15.3 External Searching and B-Trees

Consider the problem of maintaining a large collection of items that does not fit in main memory, such as a typical database. In this context, we refer to the secondary-memory blocks as ***disk blocks***. Likewise, we refer to the transfer of a block between secondary memory and primary memory as a ***disk transfer***. Recalling the great time difference that exists between main memory accesses and disk accesses, the main goal of maintaining such a collection in external memory is to minimize the number of disk transfers needed to perform a query or update. We refer to this count as the ***I/O complexity*** of the algorithm involved.

Some Inefficient External-Memory Representations

A typical operation we would like to support is the search for a key in a map. If we were to store n items unordered in a doubly linked list, searching for a particular key within the list requires n transfers in the worst case, since each link hop we perform on the linked list might access a different block of memory.

We can reduce the number of block transfers by storing the sequence in an array. A sequential search of an array can be performed using only $O(n/B)$ block transfers because of spatial locality of reference, where B denotes the number of elements that fit into a block. This is because the block transfer when accessing the first element of the array actually retrieves the first B elements, and so on with each successive block. It is worth noting that the bound of $O(n/B)$ transfers is only achieved when using an array of primitives in Java. For an array of objects, the array stores the sequence of references; the actual objects that are referenced are not necessarily stored near each other in memory, and so there may be n distinct block transfers in the worst case.

If a sequence is stored in *sorted* order within an array, a binary search performs $O(\log_2 n)$ transfers, which is a nice improvement. But we do not get significant benefit from block transfers because each query during a binary search is likely in a different block of the sequence. As usual, update operations are expensive for a sorted array.

Since these simple implementations are I/O inefficient, we should consider the logarithmic-time internal-memory strategies that use balanced binary trees (for example, AVL trees or red-black trees) or other search structures with logarithmic average-case query and update times (for example, skip lists or splay trees). Typically, each node accessed for a query or update in one of these structures will be in a different block. Thus, these methods all require $O(\log_2 n)$ transfers in the worst case to perform a query or update operation. But we can do better! We can perform map queries and updates using only $O(\log_B n) = O(\log n/\log B)$ transfers.

15.3.1 (a,b) Trees

To reduce the number of external-memory accesses when searching, we can represent our map using a multiway search tree (Section 11.5.1). This approach gives rise to a generalization of the $(2,4)$ tree data structure known as the (a,b) tree.

An (a,b) tree is a multiway search tree such that each node has between a and b children and stores between $a-1$ and $b-1$ entries. The algorithms for searching, inserting, and removing entries in an (a,b) tree are straightforward generalizations of the corresponding ones for $(2,4)$ trees. The advantage of generalizing $(2,4)$ trees to (a,b) trees is that a parameterized class of trees provides a flexible search structure, where the size of the nodes and the running time of the various map operations depends on the parameters a and b. By setting the parameters a and b appropriately with respect to the size of disk blocks, we can derive a data structure that achieves good external-memory performance.

Definition of an (a,b) Tree

An (a,b) ***tree***, where parameters a and b are integers such that $2 \le a \le (b+1)/2$, is a multiway search tree T with the following additional restrictions:

***Size Property*:** Each internal node has at least a children, unless it is the root, and has at most b children.

***Depth Property*:** All the external nodes have the same depth.

Proposition 15.1: *The height of an (a,b) tree storing n entries is $\Omega(\log n/\log b)$ and $O(\log n/\log a)$.*

Justification: Let T be an (a,b) tree storing n entries, and let h be the height of T. We justify the proposition by establishing the following bounds on h:

$$\frac{1}{\log b}\log(n+1) \le h \le \frac{1}{\log a}\log\frac{n+1}{2}+1.$$

By the size and depth properties, the number n'' of external nodes of T is at least $2a^{h-1}$ and at most b^h. By Proposition 11.6, $n'' = n+1$. Thus,

$$2a^{h-1} \le n+1 \le b^h.$$

Taking the logarithm in base 2 of each term, we get

$$(h-1)\log a + 1 \le \log(n+1) \le h\log b.$$

An algebraic manipulation of these inequalities completes the justification. ■

Search and Update Operations

We recall that in a multiway search tree T, each node w of T holds a secondary structure $M(w)$, which is itself a map (Section 11.5.1). If T is an (a,b) tree, then $M(w)$ stores at most b entries. Let $f(b)$ denote the time for performing a search in a map, $M(w)$. The search algorithm in an (a,b) tree is exactly like the one for multiway search trees given in Section 11.5.1. Hence, searching in an (a,b) tree T with n entries takes $O(\frac{f(b)}{\log a}\log n)$ time. Note that if b is considered a constant (and thus a is also), then the search time is $O(\log n)$.

The main application of (a,b) trees is for maps stored in external memory. Namely, to minimize disk accesses, we select the parameters a and b so that each tree node occupies a single disk block (so that $f(b) = 1$ if we wish to simply count block transfers). Providing the right a and b values in this context gives rise to a data structure known as the B-tree, which we will describe shortly. Before we describe this structure, however, let us discuss how insertions and removals are handled in (a,b) trees.

The insertion algorithm for an (a,b) tree is similar to that for a $(2,4)$ tree. An overflow occurs when an entry is inserted into a b-node v, which becomes an illegal $(b+1)$-node. (Recall that a node in a multiway tree is a d-node if it has d children.) To remedy an overflow, we split node w by moving the median entry of w into the parent of w and replacing w with a $\lceil (b+1)/2 \rceil$-node w' and a $\lfloor (b+1)/2 \rfloor$-node w''. We can now see the reason for requiring $a \leq (b+1)/2$ in the definition of an (a,b) tree. Note that as a consequence of the split, we need to build the secondary structures $M(w')$ and $M(w'')$.

Removing an entry from an (a,b) tree is similar to what was done for $(2,4)$ trees. An underflow occurs when a key is removed from an a-node w, distinct from the root, which causes w to become an illegal $(a-1)$-node. To remedy an underflow, we perform a transfer with a sibling of w that is not an a-node or we perform a fusion of w with a sibling that is an a-node. The new node w' resulting from the fusion is a $(2a-1)$-node, which is another reason for requiring $a \leq (b+1)/2$.

Table 15.1 shows the performance of a map realized with an (a,b) tree.

Method	Running Time
get	$O\left(\frac{f(b)}{\log a}\log n\right)$
put	$O\left(\frac{g(b)}{\log a}\log n\right)$
remove	$O\left(\frac{g(b)}{\log a}\log n\right)$

Table 15.1: Time bounds for an n-entry map realized by an (a,b) tree T. We assume the secondary structure of the nodes of T support search in $f(b)$ time, and split and fusion operations in $g(b)$ time, for some functions $f(b)$ and $g(b)$, which can be made to be $O(1)$ when we are only counting disk transfers.

15.3.2 B-Trees

A version of the (a,b) tree data structure, which is the best-known method for maintaining a map in external memory, is called the "B-tree." (See Figure 15.7.) A ***B-tree of order*** d is an (a,b) tree with $a = \lceil d/2 \rceil$ and $b = d$. Since we discussed the standard map query and update methods for (a,b) trees above, we restrict our discussion here to the I/O complexity of B-trees.

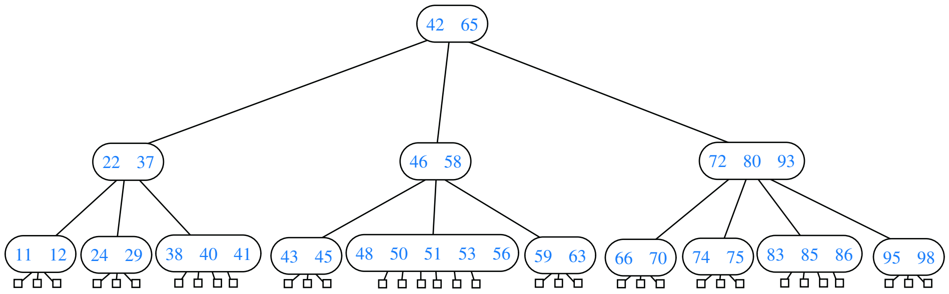


Figure 15.7: A B-tree of order 6.

An important property of B-trees is that we can choose d so that the d children references and the $d-1$ keys stored at a node can fit compactly into a single disk block, implying that d is proportional to B. This choice allows us to assume that a and b are also proportional to B in the analysis of the search and update operations on (a,b) trees. Thus, $f(b)$ and $g(b)$ are both $O(1)$, for each time we access a node to perform a search or an update operation, we need only perform a single disk transfer.

As we have already observed above, each search or update requires that we examine at most $O(1)$ nodes for each level of the tree. Therefore, any map search or update operation on a B-tree requires only $O(\log_{\lceil d/2 \rceil} n)$, that is, $O(\log n/\log B)$, disk transfers. For example, an insert operation proceeds down the B-tree to locate the node in which to insert the new entry. If the node would ***overflow*** (to have $d+1$ children) because of this addition, then this node is ***split*** into two nodes that have $\lfloor (d+1)/2 \rfloor$ and $\lceil (d+1)/2 \rceil$ children, respectively. This process is then repeated at the next level up, and will continue for at most $O(\log_B n)$ levels.

Likewise, if a remove operation results in a node ***underflow*** (to have $\lceil d/2 \rceil - 1$ children), then we move references from a sibling node with at least $\lceil d/2 \rceil + 1$ children or we perform a ***fusion*** operation of this node with its sibling (and repeat this computation at the parent). As with the insert operation, this will continue up the B-tree for at most $O(\log_B n)$ levels. The requirement that each internal node have at least $\lceil d/2 \rceil$ children implies that each disk block used to support a B-tree is at least half full. Thus, we have the following:

Proposition 15.2: *A B-tree with n entries has I/O complexity $O(\log_B n)$ for search or update operation, and uses $O(n/B)$ blocks, where B is the size of a block.*

15.4 External-Memory Sorting

In addition to data structures, such as maps, that need to be implemented in external memory, there are many algorithms that must also operate on input sets that are too large to fit entirely into internal memory. In this case, the objective is to solve the algorithmic problem using as few block transfers as possible. The most classic domain for such external-memory algorithms is the sorting problem.

Multiway Merge-Sort

An efficient way to sort a set S of n objects in external memory amounts to a simple external-memory variation on the familiar merge-sort algorithm. The main idea behind this variation is to merge many recursively sorted lists at a time, thereby reducing the number of levels of recursion. Specifically, a high-level description of this ***multiway merge-sort*** method is to divide S into d subsets $S_1, S_2, \ldots, S_d$ of roughly equal size, recursively sort each subset S_i, and then simultaneously merge all d sorted lists into a sorted representation of S. If we can perform the merge process using only $O(n/B)$ disk transfers, then, for large enough values of n, the total number of transfers performed by this algorithm satisfies the following recurrence equation:

$$t(n) = d \cdot t(n/d) + cn/B,$$

for some constant $c \geq 1$. We can stop the recursion when $n \leq B$, since we can perform a single block transfer at this point, getting all of the objects into internal memory, and then sort the set with an efficient internal-memory algorithm. Thus, the stopping criterion for $t(n)$ is

$$t(n) = 1 \quad \text{if } n/B \leq 1.$$

This implies a closed-form solution that $t(n)$ is $O((n/B)\log_d(n/B))$, which is

$$O((n/B)\log(n/B)/\log d).$$

Thus, if we can choose d to be $\Theta(M/B)$, where M is the size of the internal memory, then the worst-case number of block transfers performed by this multiway merge-sort algorithm will be quite low. For reasons given in the next section, we choose

$$d = (M/B) - 1.$$

The only aspect of this algorithm left to specify, then, is how to perform the d-way merge using only $O(n/B)$ block transfers.

15.4.1 Multiway Merging

In a standard merge-sort (Section 12.1), 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 15.8.)

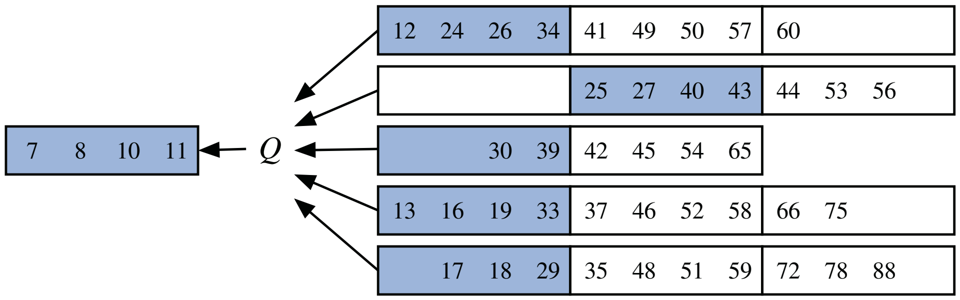


Figure 15.8: A d-way merge with $d = 5$ and $B = 4$. Blocks that currently reside in main memory are shaded.

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 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.*

15.5 Exercises

Reinforcement

R-15.1 Julia just bought a new computer that uses 64-bit integers to address memory cells. Argue why Julia will never in her life be able to upgrade the main memory of her computer so that it is the maximum-size possible, assuming that you have to have distinct atoms to represent different bits.

R-15.2 Consider an initially empty memory cache consisting of four pages. How many page misses does the LRU algorithm incur on the following page request sequence: $(2,3,4,1,2,5,1,3,5,4,1,2,3)$?

R-15.3 Consider an initially empty memory cache consisting of four pages. How many page misses does the FIFO algorithm incur on the following page request sequence: $(2,3,4,1,2,5,1,3,5,4,1,2,3)$?

R-15.4 Consider an initially empty memory cache consisting of four pages. What is the maximum number of page misses that the random algorithm incurs on the following page request sequence: $(2,3,4,1,2,5,1,3,5,4,1,2,3)$? Show all of the random choices the algorithm made in this case.

R-15.5 Describe, in detail, algorithms for adding an item to, or deleting an item from, an (a,b) tree.

R-15.6 Suppose T is a multiway tree in which each internal node has at least five and at most eight children. For what values of a and b is T a valid (a,b) tree?

R-15.7 For what values of d is the tree T of the previous exercise an order-d B-tree?

R-15.8 Draw the result of inserting, into an initially empty order-7 B-tree, entries with keys $(4,40,23,50,11,34,62,78,66,22,90,59,25,72,64,77,39,12)$, in this order.

Creativity

C-15.9 Describe an efficient external-memory algorithm for removing all the duplicate entries in an array list of size n.

C-15.10 Describe an external-memory data structure to implement the stack ADT so that the total number of disk transfers needed to process a sequence of k push and pop operations is $O(k/B)$.

C-15.11 Describe an external-memory data structure to implement the queue ADT so that the total number of disk transfers needed to process a sequence of k enqueue and dequeue operations is $O(k/B)$.

C-15.12 Describe an external-memory version of the PositionalList ADT (Section 7.3), with block size B, such that an iteration of a list of length n is completed using $O(n/B)$ transfers in the worst case, and all other methods of the ADT require only $O(1)$ transfers.

C-15.13 Change the rules that define red-black trees so that each red-black tree T has a corresponding $(4,8)$ tree, and vice versa.

C-15.14 Describe a modified version of the B-tree insertion algorithm so that each time we create an overflow because of a split of a node w, we redistribute keys among all of w's siblings, so that each sibling holds roughly the same number of keys (possibly cascading the split up to the parent of w). What is the minimum fraction of each block that will always be filled using this scheme?

C-15.15 Another possible external-memory map implementation is to use a skip list, but to collect consecutive groups of $O(B)$ nodes, in individual blocks, on any level in the skip list. In particular, we define an ***order-d B-skip list*** to be such a representation of a skip list structure, where each block contains at least $\lceil d/2 \rceil$ list nodes and at most d list nodes. Let us also choose d in this case to be the maximum number of list nodes from a level of a skip list that can fit into one block. Describe how we should modify the skip-list insertion and removal algorithms for a B-skip list so that the expected height of the structure is $O(\log n/\log B)$.

C-15.16 Describe how to use a B-tree to implement the Partition ADT (Section 14.7.3) so that the union and find operations each use at most $O(\log n/\log B)$ disk transfers.

C-15.17 Suppose we are given a sequence S of n elements with integer keys such that some elements in S are colored "blue" and some elements in S are colored "red." In addition, say that a red element e ***pairs*** with a blue element f if they have the same key value. Describe an efficient external-memory algorithm for finding all the red-blue pairs in S. How many disk transfers does your algorithm perform?

C-15.18 Consider the page caching problem where the memory cache can hold m pages, and we are given a sequence P of n requests taken from a pool of $m+1$ possible pages. Describe the optimal strategy for the offline algorithm and show that it causes at most $m+n/m$ page misses in total, starting from an empty cache.

C-15.19 Describe an efficient external-memory algorithm that determines whether an array of n integers contains a value occurring more than $n/2$ times.

C-15.20 Consider the page caching strategy based on the ***least frequently used*** (LFU) rule, where the page in the cache that has been accessed the least often is the one that is evicted when a new page is requested. If there are ties, LFU evicts the least frequently used page that has been in the cache the longest. Show that there is a sequence P of n requests that causes LFU to miss $\Omega(n)$ times for a cache of m pages, whereas the optimal algorithm will miss only $O(m)$ times.

C-15.21 Suppose that instead of having the node-search function $f(d) = 1$ in an order-d B-tree T, we have $f(d) = \log d$. What does the asymptotic running time of performing a search in T now become?

Projects

P-15.22 Write a Java class that simulates the best-fit, worst-fit, first-fit, and next-fit algorithms for memory management. Determine experimentally which method is the best under various sequences of memory requests.

P-15.23 Write a Java class that implements all the methods of the sorted map ADT by means of an (a,b) tree, where a and b are integer constants passed as parameters to a constructor.

P-15.24 Implement the B-tree data structure, assuming a block size of 1024 and integer keys. Test the number of "disk transfers" needed to process a sequence of map operations.

Chapter Notes

The reader interested in the study of the architecture of hierarchical memory systems is referred to the book chapter by Burger et al. [20] or the book by Hennessy and Patterson [44]. The mark-sweep garbage collection method we describe is one of many different algorithms for performing garbage collection. We encourage the reader interested in further study of garbage collection to examine the book by Jones and Lins [52]. Knuth [61] has very nice discussions about external-memory sorting and searching. The handbook by Gonnet and Baeza-Yates [38] compares the performance of a number of different sorting algorithms, many of which are external-memory algorithms. B-trees were invented by Bayer and McCreight [11] and Comer [24] provides a very nice overview of this data structure. The books by Mehlhorn [71] and Samet [81] also have nice discussions about B-trees and their variants. Aggarwal and Vitter [3] study the I/O complexity of sorting and related problems, establishing upper and lower bounds. Goodrich et al. [40] study the I/O complexity of several computational geometry problems. The reader interested in further study of I/O-efficient algorithms is encouraged to examine the survey paper of Vitter [91].