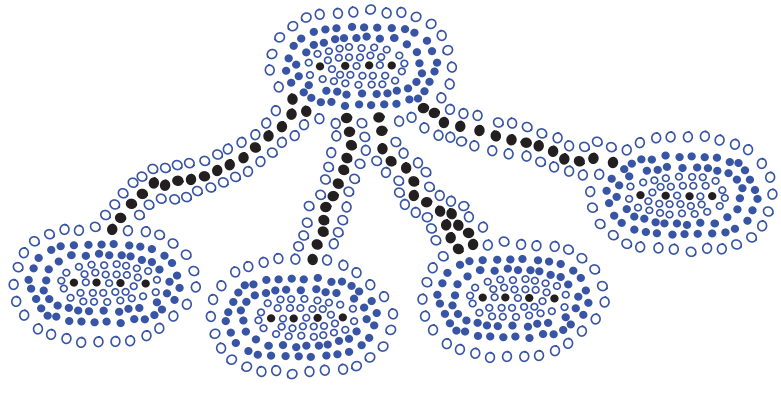
Chapter 14 Memory Management and B-Trees



Contents

14.1 Memory Management

In order to implement any data structure on an actual computer, we need to use computer memory. Computer memory is simply a sequence of memory ***words***, each of which usually 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 ***address***. Thus, the memory in a computer can be viewed as basically one giant array of memory words. Using this memory to construct data structures (and run programs) requires that we ***manage*** the computer's memory to provide the space needed for data—including variables, nodes, pointers, arrays, and character strings—and the programs the computer runs. We discuss the basics of memory management in this section.

The C++ Run-Time Stack

A C++ program is compiled into a binary executable file, which is then executed within the context of the C++ run-time environment. The ***run-time environment*** provides important functions for executing your program, such as managing memory and performing input and output.

Stacks have an important application to the run-time environment of C++ programs. A running program has a private stack, called the ***function call stack*** or just ***call stack*** for short, which is used to keep track of local variables and other important information on functions as they are invoked during execution. (See Figure 14.1.)

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

Keeping Track of the Program Counter

Your computer's run-time system maintains a special variable, called the ***program counter***, which keeps track of which machine instruction is currently being executed. When the function cool() invokes another function fool(), the current value of the program counter is recorded in the frame of the current invocation of cool() (so the system knows where to return to when function fool() is done). At the top of the stack is the frame of the ***running function***, that is, the function that is currently

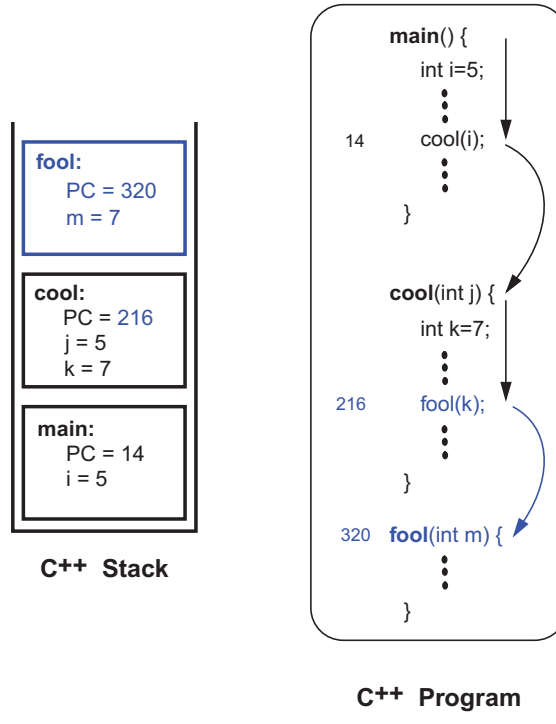


Figure 14.1: An example of the C++ call stack: function fool has just been called by function cool, which itself was previously called by function main. Note the values of the program counter, parameters, and local variables stored in the stack frames. When the invocation of function fool terminates, the invocation of function cool resumes its execution at instruction 217, which is obtained by incrementing the value of the program counter stored in the stack frame.

executing. The remaining elements of the stack are frames of the ***suspended functions***, that is, functions that have invoked another function 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 functions. When a new function is invoked, a frame for this function is pushed onto the stack. When it terminates, its frame is popped from the stack and the system resumes the processing of the previously suspended function.

Understanding Call-by-Value Parameter Passing

The system uses the call stack to perform parameter passing to functions. Unless reference parameters are involved, C++ uses the ***call-by-value*** parameter passing protocol. This means that the current ***value*** of a variable (or expression) is what is passed as an argument to a called function.

If the variable x being passed is not specified as a reference parameter, its value is copied to a local variable in the called function's frame. This applies to primitive types (such as int and float), pointers (such as "int*"), and even to classes (such as "std::vector<int>"). Note that if the called function changes the value of this local variable, it will ***not*** change the value of the variable in the calling function.

On the other hand, if the variable x is passed as a reference parameter, such as "int&," the address of x is passed instead, and this address is assigned to some local variable y in the called function. Thus, y and x refer to the same object. If the called function changes the internal state of the object that y refers to, it will simultaneously be changing the internal state of the object that x refers to (since they refer to the same object).

C++ arrays behave somewhat differently, however. Recall from Section 1.1.3, that a C++ array is represented internally as a pointer to its first element. Thus, passing an array parameter passes a copy of this pointer, not a copy of the array contents. Since the variable x in the calling function and the associated local variable y in the called function share the same copy of this pointer, $x[i]$ and $y[i]$ refer to the same object in memory.

Implementing Recursion

One of the benefits of using a stack to implement function invocation is that it allows programs to use ***recursion***. That is, it allows a function to call itself, as discussed in Section 3.5. Interestingly, early programming languages, such as Cobol and Fortran, did not originally use run-time 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 run-time stack for function and procedure calls.

In the execution of a recursive function, each box of the recursion trace corresponds to a frame of the call stack. Also, the content of the call stack corresponds to the chain of boxes from the initial function invocation to the current one.

To better illustrate how a run-time stack allows for recursive functions, let us consider a C++ implementation of the classic recursive definition of the factorial tion

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

as shown in Code Fragment 14.1.

The first time we call function factorial, its stack frame includes a local variable storing the value n. Function factorial recursively calls itself to compute $(n-1)!$, which pushes a new frame on the call stack. In turn, this recursive invocation calls itself to compute $(n-2)!$, etc. The chain of recursive invocations, and thus the run-time stack, only grows up to size n, because calling factorial(1) returns

```
int recursiveFactorial(int n) {                  // recursive factorial function
  if (n == 0) return 1;                          // basis case
  else return n * recursiveFactorial(n−1);       // recursive case
}
```

Code Fragment 14.1: A recursive implementation of the factorial function.

1 immediately without invoking itself recursively. The run-time stack allows for function factorial to exist simultaneously in several active frames (as many as n at some point). Each frame stores the value of its parameter n as well as the value to be returned. Eventually, when the first recursive call terminates, it returns $(n-1)!$, which is then multiplied by n to compute $n!$ for the original call of the factorial function.

14.1.1 Memory Allocation in C++

We have already discussed (in Section 14.1) how the C++ run-time system allocates a function's local variables in that function's frame on the run-time stack. The stack is not the only kind of memory available for program data in C++, however. Memory can also be allocated dynamically by using the **new** operator, which is built into C++. For example, in Chapter 1, we learned that we can allocate an array of 100 integers as follows:

```
int* items = new int[100];
```

Memory allocated in this manner can be deallocated with "delete [ ] items."

The Memory Heap

Instead of using the run-time stack for this object's memory, C++ uses memory from another area of storage—the ***memory heap*** (which should not be confused with the "heap" data structure discussed in Chapter 8). We illustrate this memory area, together with the other memory areas, in Figure 14.2. 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.)

The memory heap must be able to allocate memory blocks quickly for new objects. Different run-time systems use different approaches. We therefore exercise this freedom and choose to use a queue to manage the unused blocks in the memory heap. When a function uses the **new** operator to request a block of memory for

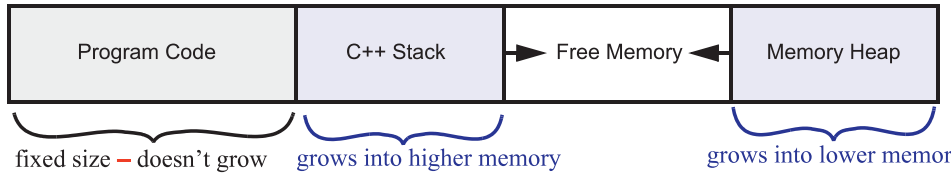


Figure 14.2: A schematic view of the layout of memory in a C++ program.

some new object, the run-time system can perform a dequeue operation on the queue of unused blocks to provide a free block of memory in the memory heap. Likewise, when the user deallocates a block of memory using **delete**, then the run-time system can perform an enqueue operation to return this block to the queue of available blocks.

Memory Allocation Algorithms

It is important that the run-time systems of modern programming languages, such as C++ and Java, are able to quickly allocate memory for new objects. Different systems adopt difference approaches. One popular method is to keep contiguous "holes" of available free memory in a doubly 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***. Of course, 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 not actually used. For example, a program may request an array of size $1,000$ but only use the first 100 cells of this array. There isn't 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 (for example, when the **new** keyword is used in C++), the run-time environment should allocate memory in a way that tries to reduce external fragmentation as much as reasonably possible.

Several heuristics have been suggested for allocating memory from the heap in order 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 3.4.1). 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 8). 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.

14.1.2 Garbage Collection

In C++, the memory space for objects must be explicitly allocated and deallocated by the programmer through the use of the operators **new** and **delete**, respectively. Other programming languages, such as Java, place the burden of memory management entirely on the run-time environment. In this section, we discuss how the run-time systems of languages like Java manage the memory used by objects allocated by the **new** operation.

As mentioned above, memory for objects is allocated from the memory heap and the space for the member variables of a running program are placed in its call stacks, one for each running program. Since member variables in a call stack can refer to objects in the memory heap, all the variables and objects in the call 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 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 run-time environment may notice that available space in the memory heap is becoming scarce. At such times, the system 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***.

In the mark-sweep garbage collection algorithm, we associate a "mark" bit with each object that identifies if that object is live or not. When we determine, at some point, that garbage collection is needed, we suspend all other running threads and clear the mark bits of all the objects currently allocated in the memory heap. We then trace through the call stack of the currently running program and we mark all the (root) objects in this stack 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 use the directed-graph version of the depth-first search traversal (Section 13.3.1). 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 threads. 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 described it in Section 13.3.1, 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. We want a way to perform DFS in-place, 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, corresponds 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.

14.2 External Memory and Caching

There are several computer applications that must deal with a large amount of data. Examples include the analysis of scientific data sets, the processing of financial transactions, and the organization and maintenance of databases (such as telephone directories). In fact, the amount of data that must be dealt with is often too large to fit entirely in the internal memory of a computer.

14.2.1 The Memory Hierarchy

In order to accommodate large data sets, computers have a ***hierarchy*** of different kinds of memories that 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 is the ***cache*** memory. This memory is considerably larger than the register set of a CPU, but accessing it takes longer (and there may even be multiple caches with progressively slower access times). 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. Finally, at the highest 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. Thus, the memory hierarchy for computers can be viewed as consisting of four levels, each of which is larger and slower than the previous level. (See Figure 14.3.)

In most applications, however, only two levels really matter—the one that can hold all data items and the level just below that one. Bringing data items in and out of the higher memory that can hold all items will typically be the computational bottleneck in this case.

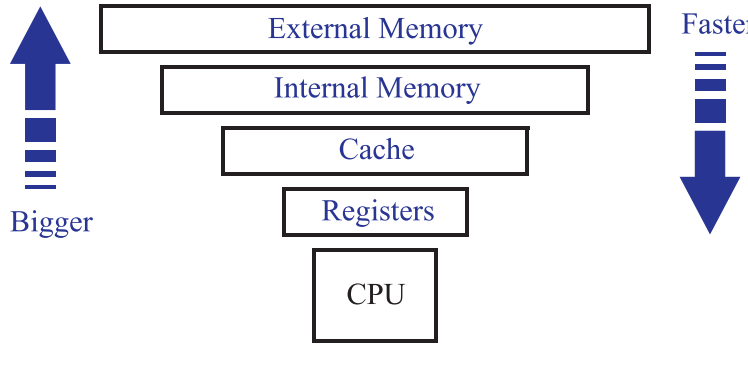


Figure 14.3: The memory hierarchy.

Specifically, the two levels that matter most depend on the size of the problem we are trying to solve. 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 devices, 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 e-mail 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.

14.2.2 Caching Strategies

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 impossible to get. For example, a C++ program that is designed to run on many different computer platforms cannot be defined in terms of a specific computer architecture configuration. We can certainly use architecture-specific information, if we have it (and we show how to exploit such information later in this chapter). But once we have optimized our software for a certain architecture configuration, our software is no longer 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 for 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 it is likely to access this location again in the near future. For example, it is quite 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 it is likely to access other locations that are near this one. For example, a program using an array is 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 for-loop used to scan through an array exhibits both kinds of locality.

Temporal and spatial localities have, in turn, given rise to two fundamental design choices for two-level 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. Because, 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 14.4.) 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***.

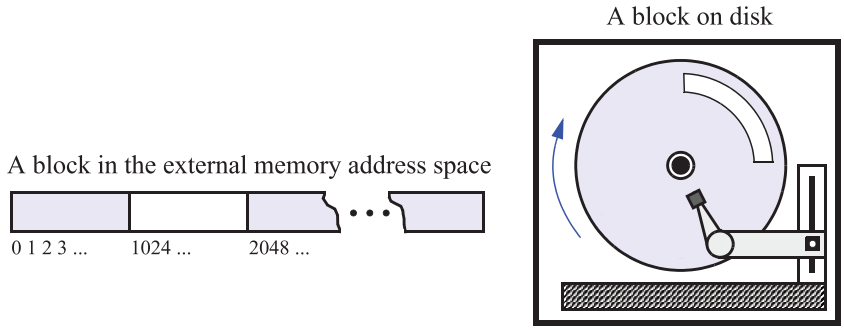


Figure 14.4: Blocks in external memory.

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 how to do this eviction brings up a number of interesting data structure and algorithm design issues.

Caching Algorithms

There are several Web applications that must deal with revisiting information presented in Web pages. These revisits have been shown to exhibit localities of reference, both in time and in space. To exploit these localities 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. 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 l, the browser determines (using a quick test) if l is unchanged and currently contained in the cache. If l is contained in the cache, then the browser satisfies the request using the cached copy. If l is not in the cache, however, the page for l 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 l 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 l 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 14.5):

- **First-in, fist-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.

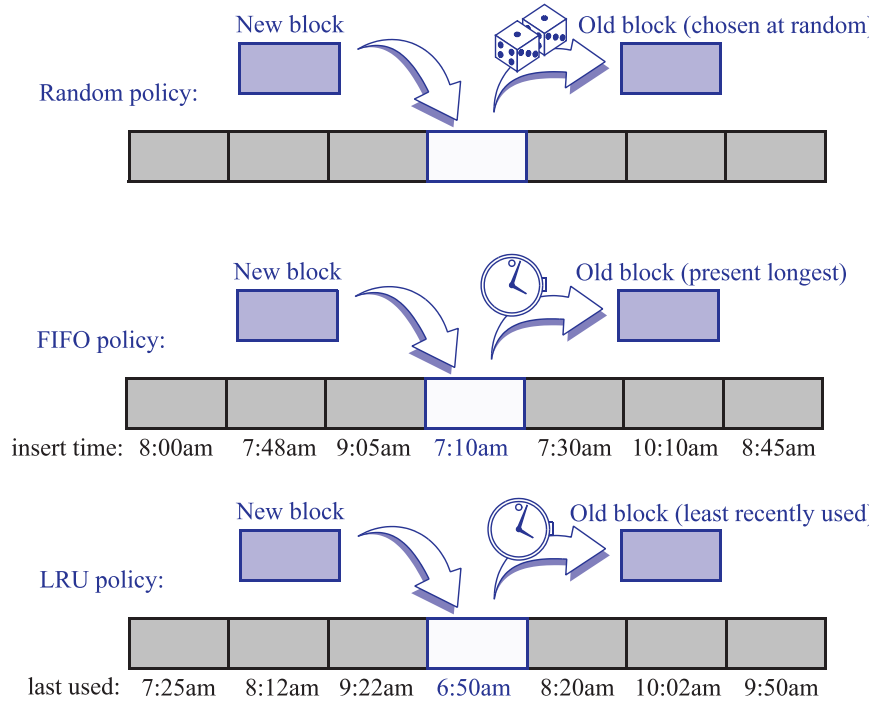


Figure 14.5: The Random, FIFO, and LRU page replacement policies.

The Random strategy is one of the easiest policies to implement, because it only requires a random or pseudo-random 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 or spatial localities that a user's browsing exhibits.

The FIFO strategy is quite simple to implement, because 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, since 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 a priority queue Q that supports searching for existing pages, for example, using special pointers or "locators." 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.

14.3 External Searching and B-Trees

Consider the problem of implementing the map ADT for a large collection of items that do not fit in main memory. Since one of the main uses of a large map is in a database, 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 a map in external memory is to minimize the number of disk transfers needed to perform a query or update. In fact, the difference in speed between disk and internal memory is so great that we should be willing to perform a considerable number of internal-memory accesses if they allow us to avoid a few disk transfers. Let us, therefore, analyze the performance of map implementations by counting the number of disk transfers each would require to perform the standard map search and update operations. We refer to this count as the ***I/O complexity*** of the algorithms involved.

Some Inefficient External-Memory Dictionaries

Let us first consider the simple map implementations that use a list to store n entries. If the list is implemented as an unsorted, doubly linked list, then insert and remove operations can be performed with $O(1)$ transfers each, but removals and searches require n transfers in the worst case, since each link hop we perform could access a different block. This search time can be improved to $O(n/B)$ transfers (see Exercise C-14.2), where B denotes the number of nodes of the list that can fit into a block, but this is still poor performance. We could alternately implement the sequence using a sorted array. In this case, a search performs $O(\log_2 n)$ transfers, via binary search, which is a nice improvement. But this solution requires $\Theta(n/B)$ transfers to implement an insert or remove operation in the worst case, because we may have to access all blocks to move elements up or down. Thus, list-based map implementations are not efficient in external memory.

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). These methods store the map items at the nodes of a binary tree or of a graph. 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. This performance is good, but we can do better. In particular, we can perform map queries and updates using only $O(\log_B n) = O(\log n/\log B)$ transfers.

14.3.1 (a,b) Trees

To reduce the importance of the performance difference between internal-memory accesses and external-memory accesses for searching, we can represent our map using a multi-way search tree (Section 10.4.1). 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 multi-way 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 algorithms for $(2,4)$ trees. The advantage of generalizing $(2,4)$ trees to (a,b) trees is that a generalized 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 a and b are integers, such that $2 \leq a \leq (b+1)/2$, is a multi-way 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 14.1: *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

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

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 10.7, $n'' = n+1$. Thus

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

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

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

■

Search and Update Operations

We recall that in a multi-way search tree T, each node v of T holds a secondary structure $M(v)$, which is itself a map (Section 10.4.1). If T is an (a,b) tree, then $M(v)$ stores at most b entries. Let $f(b)$ denote the time for performing a search in a map, $M(v)$. The search algorithm in an (a,b) tree is exactly like the one for multi-way search trees given in Section 10.4.1. Hence, searching in an (a,b) tree T with n entries takes $O((f(b)/\log a)\log n)$ time. Note that if b is 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 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 multi-way tree is a d-node if it has d children.) To remedy an overflow, we split node v by moving the median entry of v into the parent of v and replacing v with a $\lceil (b+1)/2 \rceil$-node v' and a $\lfloor (b+1)/2 \rfloor$-node v''. 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 $M(v')$ and $M(v'')$.

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 v, distinct from the root, which causes v to become an illegal $(a-1)$-node. To remedy an underflow, we perform a transfer with a sibling of v that is not an a-node or we perform a fusion of v 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 14.1 shows the performance of a map realized with an (a,b) tree.

Operation	*Time*
find	$O\left(\frac{f(b)}{\log a}\log n\right)$
insert	$O\left(\frac{g(b)}{\log a}\log n\right)$
erase	$O\left(\frac{g(b)}{\log a}\log n\right)$

Table 14.1: 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.

14.3.2 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 14.6.) 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.

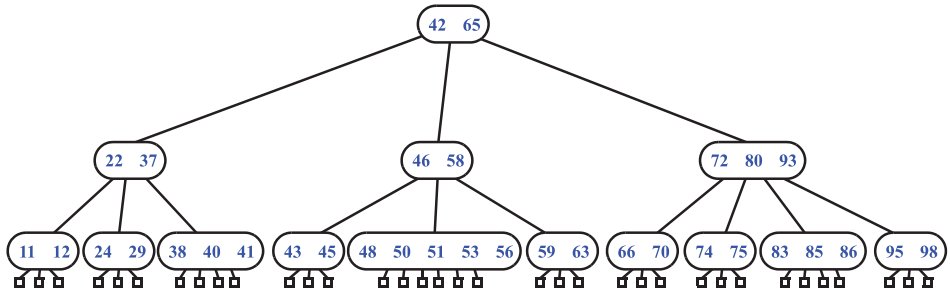


Figure 14.6: A B-tree of order 6.

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 all fit 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 $O(1)$, because 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 ***overflows*** (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 continues 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 need to perform a ***fusion*** operation of this node with its sibling (and repeat this computation at the parent). As with the insert operation, this continues up the B-tree for at most $O(\log_B n)$ levels. The requirement that each internal node has 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 14.2: *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.*

14.4 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.

Multi-Way 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 ***multi-way 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

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

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 $\Theta(M/B)$, then the worst-case number of block transfers performed by this multi-way merge-sort algorithm is quite low. We choose

$$d = (1/2)M/B.$$

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

14.4.1 Multi-Way Merging

We perform the d-way merge by running a "tournament." We let T be a complete binary tree with d external nodes, and we keep T entirely in internal memory. We associate each external node i of T with a different sorted list S_i. We initialize T by reading into each external node i, the first object in S_i. This has the effect of reading into internal memory the first block of each sorted list S_i. For each internal-node parent v of two external nodes, we then compare the objects stored at v's children and we associate the smaller of the two with v. We repeat this comparison test at the next level up in T, and the next, and so on. When we reach the root r of T, we associate the smallest object from among all the lists with r. This completes the initialization for the d-way merge. (See Figure 14.7.)

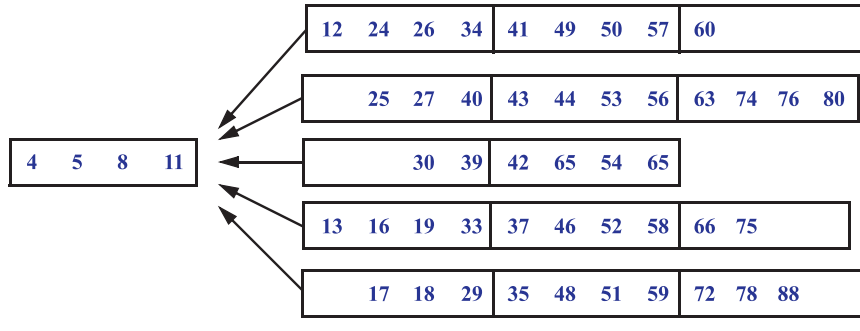


Figure 14.7: A d-way merge. We show a five-way merge with $B = 4$.

In a general step of the d-way merge, we move the object o associated with the root r of T into an array we are building for the merged list S'. We then trace down T, following the path to the external node i that o came from. We then read into i the next object in the list S_i. If o was not the last element in its block, then this next object is already in internal memory. Otherwise, we read in the next block of S_i to access this new object (if S_i is now empty, associate the node i with a pseudo-object with key $+\infty$). We then repeat the minimum computations for each of the internal nodes from i to the root of T. This again gives us the complete tree T. We then repeat this process of moving the object from the root of T to the merged list S', and rebuilding T, until T is empty of objects. Each step in the merge takes $O(\log d)$ time; hence, the internal time for the d-way merge is $O(n \log d)$. The number of transfers performed in a merge is $O(n/B)$, since we scan each list S_i in order once, and we write out the merged list S' once. Thus, we have:

Proposition 14.3: *Given an array-based sequence S of n elements stored in external memory, we can sort S using $O((n/B)\log(n/B)/\log(M/B))$ transfers and $O(n \log n)$ internal CPU time, where M is the size of the internal memory and B is the size of a block.*

14.5 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-14.1 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-14.2 Describe, in detail, add and remove algorithms for an (a,b) tree.

R-14.3 Suppose T is a multi-way 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-14.4 For what values of d is the tree T of the previous exercise an order-d B-tree?

R-14.5 Show each level of recursion in performing a four-way, external-memory merge-sort of the sequence given in the previous exercise.

R-14.6 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-14.7 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-14.8 Consider an initially empty memory cache consisting of four pages. How many page misses can the random algorithm incur 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 your algorithm made in this case.

R-14.9 Draw the result of inserting, into an initially empty order-7 B-tree, the keys $(4, 40, 23, 50, 11, 34, 62, 78, 66, 22, 90, 59, 25, 72, 64, 77, 39, 12)$.

R-14.10 Show each level of recursion in performing a four-way merge-sort of the sequence given in the previous exercise.

Creativity

C-14.1 Describe an efficient external-memory algorithm for removing all the duplicate entries in a vector of size n.

C-14.2 Show how to implement a map in external memory using an unordered sequence so that insertions require only $O(1)$ transfers and searches require $O(n/B)$ transfers in the worst case, where n is the number of elements and B is the number of list nodes that can fit into a disk block.

C-14.3 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-14.4 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 v, we redistribute keys among all of v's siblings, so that each sibling holds roughly the same number of keys (possibly cascading the split up to the parent of v). What is the minimum fraction of each block that will always be filled using this scheme?

C-14.5 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-14.6 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 n enqueue and dequeue operations is $O(n/B)$.

C-14.7 Solve the previous problem for the deque ADT.

C-14.8 Describe how to use a B-tree to implement the partition (union-find) ADT (from Section 11.4.3) so that the union and find operations each use at most $O(\log n/\log B)$ disk transfers.

C-14.9 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-14.10 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-14.11 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-14.12 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?

C-14.13 Describe an efficient external-memory algorithm that determines whether an array of n integers contains a value occurring more than $n/2$ times.

Projects

P-14.1 Write a C++ 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-14.2 Write a C++ class that implements all the functions of the ordered map ADT by means of an (a,b) tree, where a and b are integer constants passed as parameters to a constructor.

P-14.3 Implement the B-tree data structure, assuming a block size of $1,024$ and integer keys. Test the number of "disk transfers" needed to process a sequence of map operations.

P-14.4 Implement an external-memory sorting algorithm and compare it experimentally to any internal-memory sorting algorithm.

Chapter Notes

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 [51].

Knuth [57] has very nice discussions about external-memory sorting and searching, and Ullman [97] discusses external memory structures for database systems. The reader interested in the study of the architecture of hierarchical memory systems is referred to the book chapter by Burger *et al.* [18] or the book by Hennessy and Patterson [44]. The handbook by Gonnet and Baeza-Yates [37] 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 [10] and Comer [23] provides a very nice overview of this data structure. The books by Mehlhorn [73] and Samet [87] also have nice discussions about B-trees and their variants. Aggarwal and Vitter [2] study the I/O

complexity of sorting and related problems, establishing upper and lower bounds, including the lower bound for sorting given in this chapter. 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 [99].