Chapter 20 B-Trees and External Memory

Columbia Supercomputer at NASA's Advanced Supercomputing Facility at Ames Research Center, 2006. U.S. government image. Credit: Trower, NASA.

Contents

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.

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 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***, ***core memory***, or ***random access 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, CDs, DVDs, 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 20.1.)

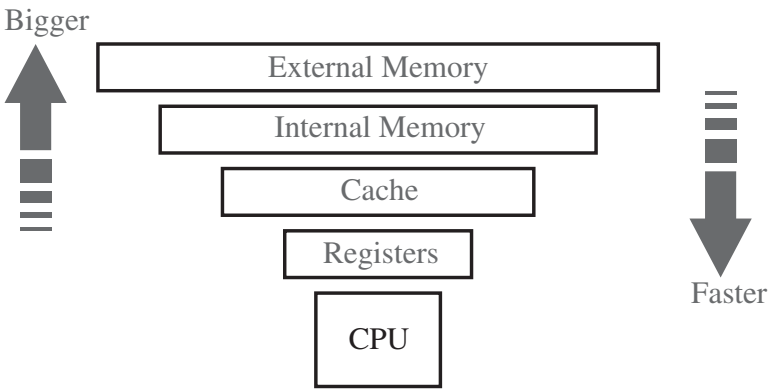

Figure 20.1: The memory hierarchy.

In most applications, however, only two levels really matter—the one that can hold all the data items in our problem 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. In this chapter, we focus on algorithms that accommodate this phenomenon or facilitate it, including B-trees, external-memory sorting, and online caching algorithms.

20.1 External Memory

Which two levels matter most to solving a particular problem depends on the size of that problem. For a problem that can fit entirely in main memory, the important two 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 important two 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.

Hierarchical Memory Management

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 analysis 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 this chapter—but there are two fundamental justifications for why it is actually a reasonable assumption to make.

The first justification 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 Java 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 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.

The second 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 will exhibit both kinds of locality.

Caching and Blocking

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 into the primary-level memory, data located in the secondary 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. For, by bringing data into primary memory, we are hoping that it will be accessed again soon, and we will be able to quickly respond 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 20.2.) This concept is known as ***blocking***, and it is motivated by the expectation that other secondary-level memory locations close to l will soon be accessed. In the interface between cache memory and internal memory, such blocks are often called ***cache lines***, and in the interface between internal memory and external memory, such blocks are often called ***pages***.

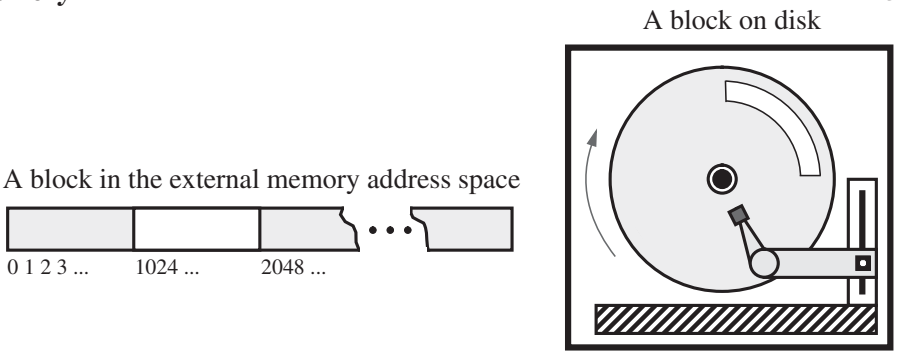

Figure 20.2: Blocks in external memory.

Incidentally, blocking for disk and CD/DVD-ROM drives is also motivated by the properties of these hardware technologies. A reading arm on a disk or CD/DVD-ROM takes a relatively long time to position itself for reading a certain location, but, once the arm is positioned, it can quickly read many contiguous locations, because the medium it is reading is spinning very fast. (See Figure 20.2.) Even without this motivation, however, blocking is fully justified by the spatial locality property that most programs have.

Thus, 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 that we discuss in the remainder of this section.

A Model for External Searching

The first problem we address is that of implementing a dictionary for a large collection of items that do not fit in primary memory. Recall that a dictionary stores key-element pairs (items) subject to insertions, removals, and key-based searches. Since one of the main applications of large dictionaries is in database systems, 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***. Even though we use this terminology, the search techniques we discuss in this section apply also when the primary memory is the CPU cache and the secondary memory is the main (internal) memory. We use the disk-based viewpoint because it is concrete and also because it is more prevalent.

20.2 (2,4) Trees and B-Trees

Some search trees base their efficiency on rules that explicitly bound their depth. In fact, such trees typically define a depth function, or a "pseudo-depth" function closely related to depth, so that every external node is at the same depth or pseudo-depth. In so doing, they maintain every external node to be at depth $O(\log n)$ in a tree storing n elements. These trees are not ideally suited for external memory, however, and in such scenarios another approach is better.

20.2.1 Multi-Way Search Trees

Some bounded-depth search trees are multi-way trees, that is, trees with internal nodes that have two or more children. In this section, we describe how multi-way trees can be used as search trees, including how multi-way trees store items and how we can perform search operations in multi-way search trees. Recall that the ***items*** that we store in a search tree are pairs of the form (k, x), where k is the ***key*** and x is the element associated with the key.

Let v be a node of an ordered tree. We say that v is a d***-node*** if v has d children. We define a ***multi-way search tree*** to be an ordered tree T that has the following properties (which are illustrated in Figure 20.3a):

- Each internal node of T has at least two children. That is, each internal node is a d-node, where $d \geq 2$.
- Each internal node of T stores a collection of items of the form (k, x), where k is a key and x is an element.
- Each d-node v of T, with children $v_1, \ldots, v_d$, stores $d-1$ items $(k_1, x_1), \ldots, (k_{d-1}, x_{d-1})$, where $k_1 \leq \cdots \leq k_{d-1}$.
- Let us define $k_0 = -\infty$ and $k_d = +\infty$. For each item (k, x) stored at a node in the subtree of v rooted at v_i, $i = 1, \ldots, d$, we have $k_{i-1} \leq k \leq k_i$.

That is, if we think of the set of keys stored at v as including the special fictitious keys $k_0 = -\infty$ and $k_d = +\infty$, then a key k stored in the subtree of T rooted at a child node v_i must be "in between" two keys stored at v. This simple viewpoint gives rise to the rule that a node with d children stores $d - 1$ regular keys, and it also forms the basis of the algorithm for searching in a multi-way search tree.

By the above definition, the external nodes of a multi-way search do not store any items and serve only as "placeholders." Thus, we can view a binary search tree (Section 3.1.1) as a special case of a multi-way search tree. At the other extreme, a multi-way search tree may have only a single internal node storing all the items. In addition, while the external nodes could be **null**, we make the simplifying assumption here that they are actual nodes that don't store anything.

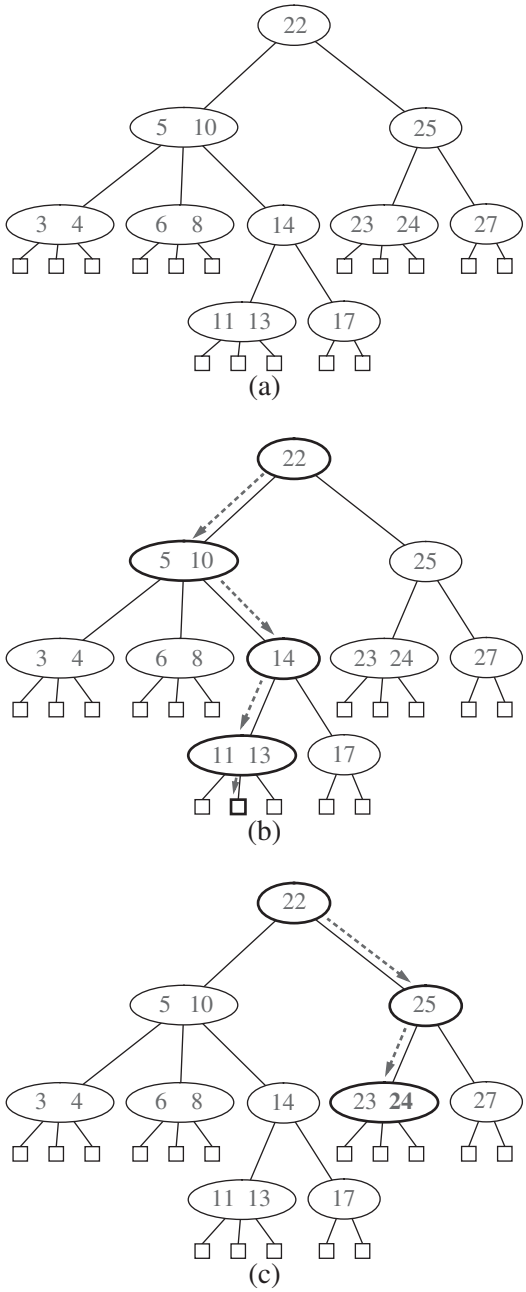

Figure 20.3: (a) A multi-way search tree T; (b) search path in T for key 12 (unsuccessful search); (c) search path in T for key 24 (successful search).

Whether internal nodes of a multi-way tree have two children or many, however, there is an interesting relationship between the number of items and the number of external nodes.

Theorem 20.1: *A multi-way search tree storing n items has $n+1$ external nodes.*

We leave the justification of this theorem as an exercise (C-4.11).

Searching in a Multi-Way Tree

Given a multi-way search tree T, searching for an element with key k is simple. We perform such a search by tracing a path in T starting at the root. (See Figure 20.3b and c.) When we are at a d-node v during this search, we compare the key k with the keys $k_1, \ldots, k_{d-1}$ stored at v. If $k = k_i$ for some i, the search is successfully completed. Otherwise, we continue the search in the child v_i of v such that $k_{i-1} < k < k_i$. (Recall that we consider $k_0 = -\infty$ and $k_d = +\infty$.) If we reach an external node, then we know that there is no item with key k in T, and the search terminates unsuccessfully.

Data Structures for Multi-Way Search Trees

In Section 2.3.4, we discussed different ways of representing general trees. Each of these representations can also be used for multi-way search trees. In fact, in using a general multi-way tree to implement a multi-way search tree, the only additional information that we need to store at each node is the set of items (including keys) associated with that node. That is, we need to store with v a reference to some container or collection object that stores the items for v.

Recall that when we use a binary tree to represent an ordered dictionary D, we simply store a reference to a single item at each internal node. In using a multi-way search tree T to represent D, we must store a reference to the ordered set of items associated with v at each internal node v of T. This reasoning may at first seem like a circular argument, since we need a representation of an ordered dictionary to represent an ordered dictionary. We can avoid any circular arguments, however, by using the ***bootstrapping*** technique, where we use a previous (less-advanced) solution to a problem to create a new (more-advanced) solution. In this case, bootstrapping consists of representing the ordered set associated with each internal node using a dictionary data structure that we have previously constructed (for example, a lookup table based on an ordered vector, as shown in Section 3.1). In particular, assuming we already have a way of implementing ordered dictionaries, we can realize a multi-way search tree by taking a tree T and storing such a dictionary at each d-node v of T.

The dictionary we store at each node v is known as a ***secondary*** or ***auxiliary*** data structure, for we are using it to support the bigger, ***primary*** data structure. We denote the dictionary stored at a node v of T as $D(v)$. The items we

store in $D(v)$ will allow us to find which child node to move to next during a search operation. Specifically, for each node v of T, with children $v_1, \ldots, v_d$ and set of items, $\{(k_1, x_1), \ldots, (k_{d-1}, x_{d-1})\}$, we store in the dictionary $D(v)$ the items (k_1, x_1, v_1), (k_2, x_2, v_2), $\ldots$, $(k_{d-1}, x_{d-1}, v_{d-1})$, $(+\infty, null, v_d)$. That is, an item (k_i, x_i, v_i) of dictionary $D(v)$ has key k_i and element (x_i, v_i). Note that the last item stores the special key $+\infty$.

With the above realization of a multi-way search tree T, processing a d-node v while searching for an element of T with key k can be done by performing a search operation to find the item (k_i, x_i, v_i) in $D(v)$ with smallest key greater than or equal to k, such as in the closestElemAfter(k) operation. We distinguish two cases:

- If $k < k_i$, then we continue the search by processing child v_i. (Note that if the special key $k_d = +\infty$ is returned, then k is greater than all the keys stored at node v, and we continue the search by processing child v_d.)
- Otherwise ($k = k_i$), then the search terminates successfully.

Performance Issues for Multi-Way Search Trees

Consider the space requirement for the above realization of a multi-way search tree T storing n items. By Theorem 20.1, using any of the common realizations of ordered dictionaries (Section 6.1) for the secondary structures of the nodes of T, the overall space requirement for T is $O(n)$.

Consider next the time spent answering a search in T. The time spent at a d-node v of T during a search depends on how we realize the secondary data structure $D(v)$. If $D(v)$ is realized with a vector-based sorted sequence (that is, a lookup table), then we can process v in $O(\log d)$ time. If instead $D(v)$ is realized using an unsorted sequence (that is, a log file), then processing v takes $O(d)$ time. Let $d_{\max}$ denote the maximum number of children of any node of T, and let h denote the height of T. The search time in a multi-way search tree is either $O(hd_{\max})$ or $O(h \log d_{\max})$, depending on the specific implementation of the secondary structures at the nodes of T (the dictionaries $D(v)$). If $d_{\max}$ is a constant, the running time for performing a search is $O(h)$, irrespective of the implementation of the secondary structures.

Thus, the prime efficiency goal for a multi-way search tree is to keep the height as small as possible, that is, we want h to be a logarithmic function of n, the number of total items stored in the dictionary. A search tree with logarithmic height, such as this, is called a ***balanced search tree***. Bounded-depth search trees satisfy this goal by keeping each external node at exactly the same depth level in the tree.

Next, we discuss a bounded-depth search tree that is a multi-way search tree that caps $d_{\max}$ at 4. In Section 20.2.3, we discuss a more general kind of multi-way search tree that has applications where our search tree is too large to completely fit into the internal memory of our computer.

20.2.2 (2,4) Trees

In using a multi-way search tree in practice, we desire that it be balanced, that is, have logarithmic height. The multi-way search tree we study next is fairly easy to keep balanced. It is the $(2, 4)$ tree, which is sometimes also called the 2-4 tree or 2-3-4 tree. In fact, we can maintain balance in a $(2, 4)$ tree by maintaining two simple properties (see Figure 20.4):

***Size Property*:** Every node has at most four children.

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

Enforcing the size property for $(2, 4)$ trees keeps the size of the nodes in the multi-way search tree constant, for it allows us to represent the dictionary $D(v)$ stored at each internal node v using a constant-sized array. The depth property, on the other hand, maintains the balance in a $(2, 4)$ tree, by forcing it to be a bounded-depth structure.

Theorem 20.2: *The height of a $(2, 4)$ tree storing n items is $\Theta(\log n)$.*

Proof: Let h be the height of a $(2, 4)$ tree T storing n items. Note that, by the size property, we can have at most 4 nodes at depth 1, at most 4^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at most 4^h. Likewise, by the depth property and the definition of a $(2, 4)$ tree, we must have at least 2 nodes at depth 1, at least 2^2 nodes at depth 2, and so on. Thus, the number of external nodes in T is at least 2^h. In addition, by Theorem 20.1, the number of external nodes in T is $n + 1$. Therefore, we obtain

$$2^h \leq n + 1 \quad \text{and} \quad n + 1 \leq 4^h.$$

Taking the logarithm in base 2 of each of the above terms, we get that

$$h \leq \log(n + 1) \quad \text{and} \quad \log(n + 1) \leq 2h,$$

which justifies our theorem. ■

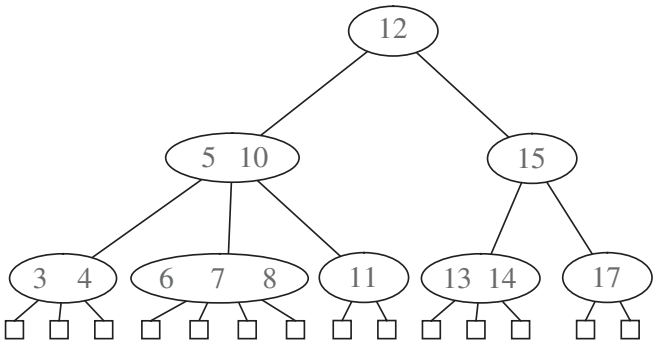

Figure 20.4: A $(2, 4)$ tree.

Insertion in a $(2, 4)$ Tree

Theorem 20.2 states that the size and depth properties are sufficient for keeping a multi-way tree balanced. Maintaining these properties requires some effort after performing insertions and removals in a $(2, 4)$ tree, however. In particular, to insert a new item (k, x), with key k, into a $(2, 4)$ tree T, we first perform a search for k. Assuming that T has no element with key k, this search terminates unsuccessfully at an external node z. Let v be the parent of z. We insert the new item into node v and add a new child w (an external node) to v on the left of z. That is, we add item (k, x, w) to the dictionary $D(v)$.

Our insertion method preserves the depth property, since we add a new external node at the same level as existing external nodes. Nevertheless, it may violate the size property. Indeed, if a node v was previously a 4-node, then it may become a 5-node after the insertion, which causes the tree T to no longer be a $(2, 4)$ tree. This type of violation of the size property is called an ***overflow*** at node v, and it must be resolved in order to restore the properties of a $(2, 4)$ tree. Let $v_1, \ldots, v_5$ be the children of v, and let $k_1, \ldots, k_4$ be the keys stored at v. To remedy the overflow at node v, we perform a ***split*** operation on v as follows (see Figure 20.5):

- Replace v with two nodes v' and v'', where
  - v' is a 3-node with children v_1, v_2, v_3 storing keys k_1 and k_2
  - v'' is a 2-node with children v_4, v_5 storing key k_4.
- If v was the root of T, create a new root node u; else, let u be the parent of v.
- Insert key k_3 into u and make v' and v'' children of u, so that if v was child i of u, then v' and v'' become children i and $i + 1$ of u, respectively.

We show a sequence of insertions in a $(2, 4)$ tree in Figure 20.6.

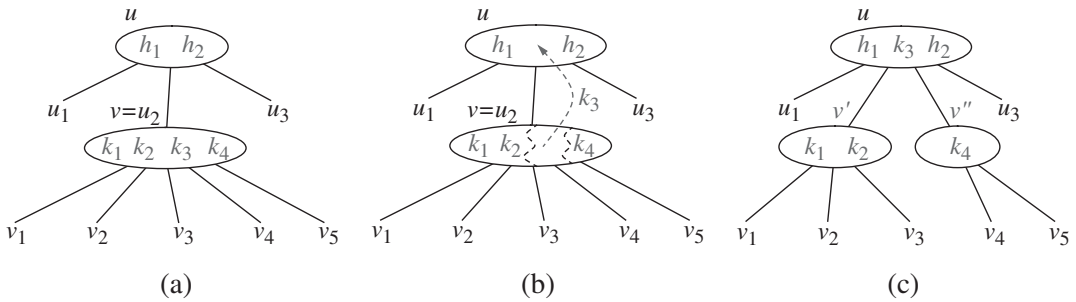

Figure 20.5: A node split: (a) overflow at a 5-node v; (b) the third key of v inserted into the parent u of v; (c) node v replaced with a 3-node v' and a 2-node v''.

A split operation affects a constant number of nodes of the tree and $O(1)$ items stored at such nodes. Thus, it can be implemented to run in $O(1)$ time.

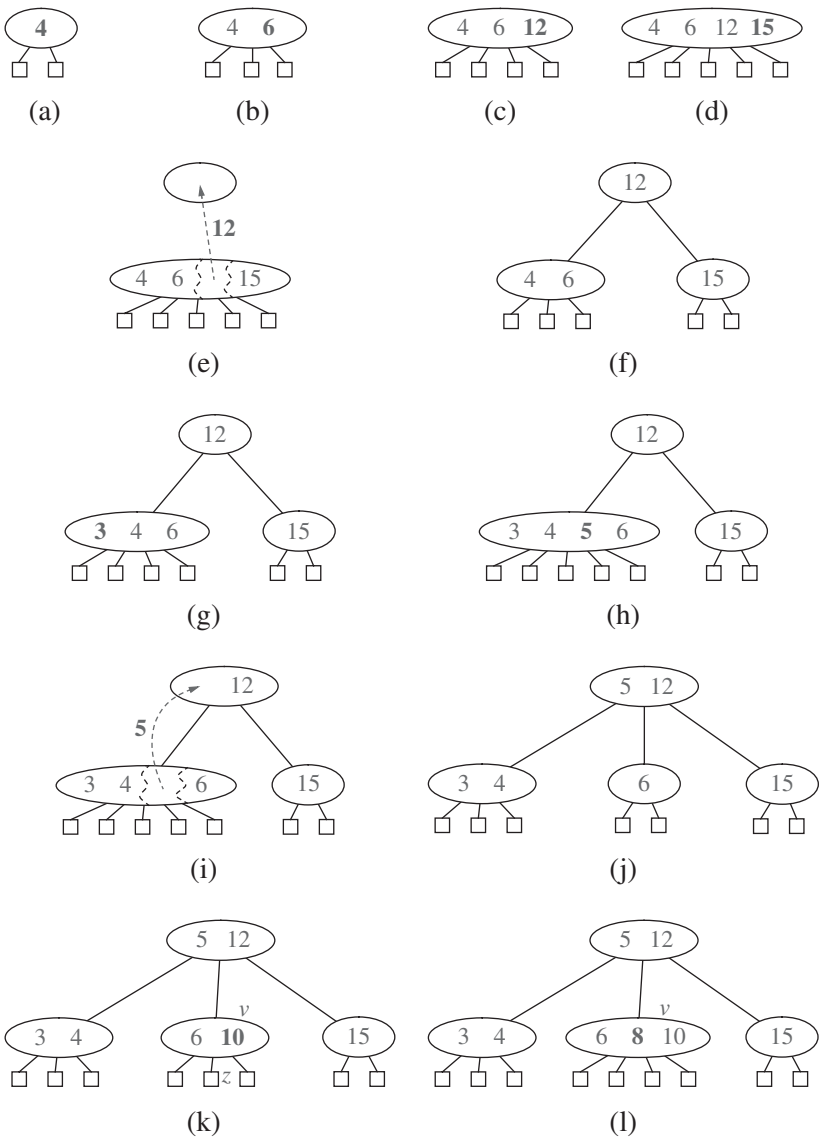

Figure 20.6: A sequence of insertions into a $(2, 4)$ tree: (a) initial tree with one item; (b) insertion of 6; (c) insertion of 12; (d) insertion of 15, which causes an overflow; (e) split, which causes the creation of a new root node; (f) after the split; (g) insertion of 3; (h) insertion of 5, which causes an overflow; (i) split; (j) after the split; (k) insertion of 10; (l) insertion of 8.

Performance of $(2, 4)$ Tree Insertion

As a consequence of a split operation on node v, a new overflow may occur at the parent u of v. If such an overflow occurs, it triggers, in turn, a split at node u. (See Figure 20.7.) A split operation either eliminates the overflow or propagates it into the parent of the current node. Indeed, this propagation can continue all the way up to the root of the search tree. But if it does propagate all the way to the root, it will finally be resolved at that point. We show such a sequence of splitting propagations in Figure 20.7.

Thus, the number of split operations is bounded by the height of the tree, which is $O(\log n)$ by Theorem 20.2. Therefore, the total time to perform an insertion in a $(2, 4)$ tree is $O(\log n)$.

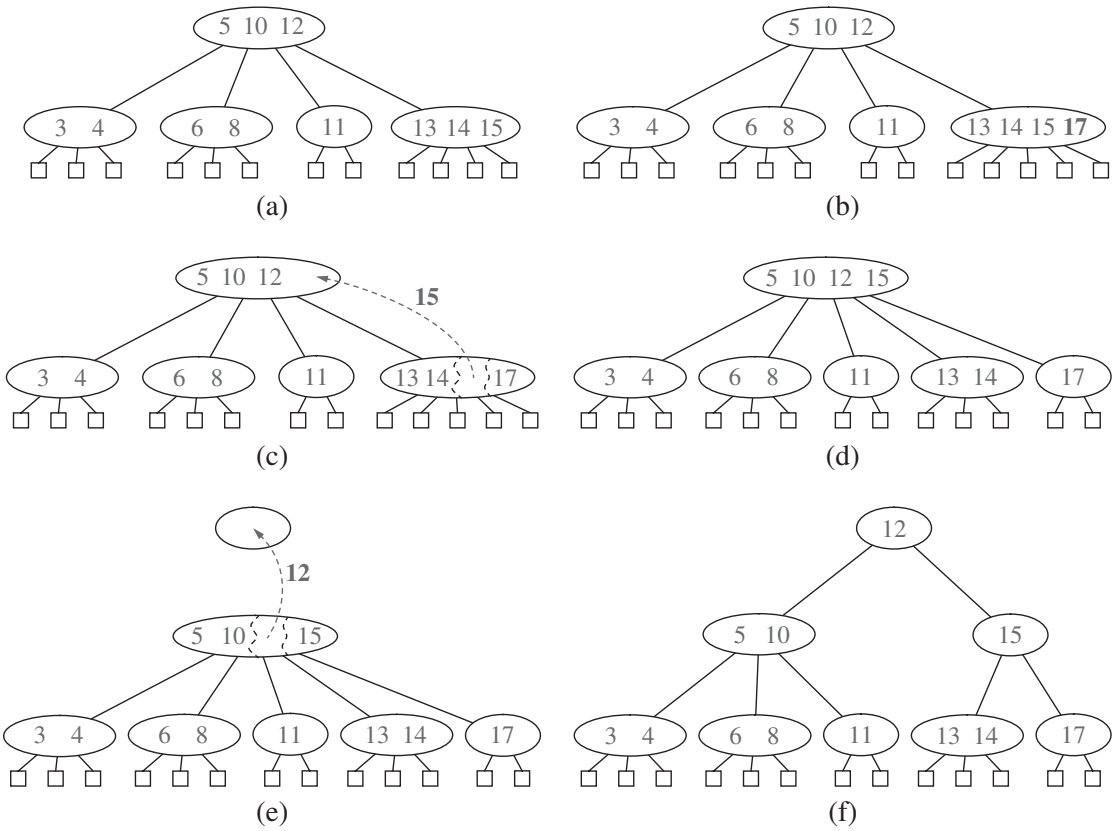

Figure 20.7: An insertion in a $(2, 4)$ tree that causes a cascading split: (a) before the insertion; (b) insertion of 17, causing an overflow; (c) a split; (d) after the split a new overflow occurs; (e) another split, creating a new root node; (f) final tree.

Removal in a $(2, 4)$ Tree

Let us now consider the removal of an item with key k from a $(2, 4)$ tree T. We begin such an operation by performing a search in T for an item with key k. Removing such an item from a $(2, 4)$ tree can always be reduced to the case where the item to be removed is stored at a node v whose children are external nodes. Suppose, for instance, that the item with key k that we wish to remove is stored in the ith item (k_i, x_i) at a node z that has only internal-node children. In this case, we swap the item (k_i, x_i) with an appropriate item that is stored at a node v with external-node children as follows (Figure 20.8d):

1. We find the right-most internal node v in the subtree rooted at the ith child of z, noting that the children of node v are all external nodes.
2. We swap the item (k_i, x_i) at z with the last item of v.

Once we ensure that the item to remove is stored at a node v with only external-node children (because either it was already at v or we swapped it into v), we simply remove the item from v (that is, from the dictionary $D(v)$) and remove the ith external node of v.

Removing an item (and a child) from a node v as described above preserves the depth property, for we always remove an external-node child from a node v with only external-node children. However, in removing such an external node we may violate the size property at v. Indeed, if v was previously a 2-node, then it becomes a 1-node with no items after the removal (Figure 20.8d and e), which is not allowed in a $(2, 4)$ tree. This type of violation of the size property is called an ***underflow*** at node v. To remedy an underflow, we check whether an immediate sibling of v is a 3-node or a 4-node. If we find such a sibling w, then we perform a ***transfer*** operation, in which we move a child of w to be a child of v, a key of w to the parent u of v and w, and a key of u to v. (See Figure 20.8b and c.) If v has only one sibling that is a 2-node, or if both immediate siblings of v are 2-nodes, then we perform a ***fusion*** operation, in which we merge v with a sibling, creating a new node v', and move a key from the parent u of v to v'. (See Figure 20.9e and f.)

A fusion operation at node v may cause a new underflow to occur at the parent u of v, which in turn triggers a transfer or fusion at u. (See Figure 20.9.) Hence, the number of fusion operations is bounded by the height of the tree, which is $O(\log n)$ by Theorem 20.2. If an underflow propagates all the way up to the root, then the root is simply deleted. (See Figure 20.9c and d.) We show a sequence of removals from a $(2, 4)$ tree in Figures 20.8 and 20.9.

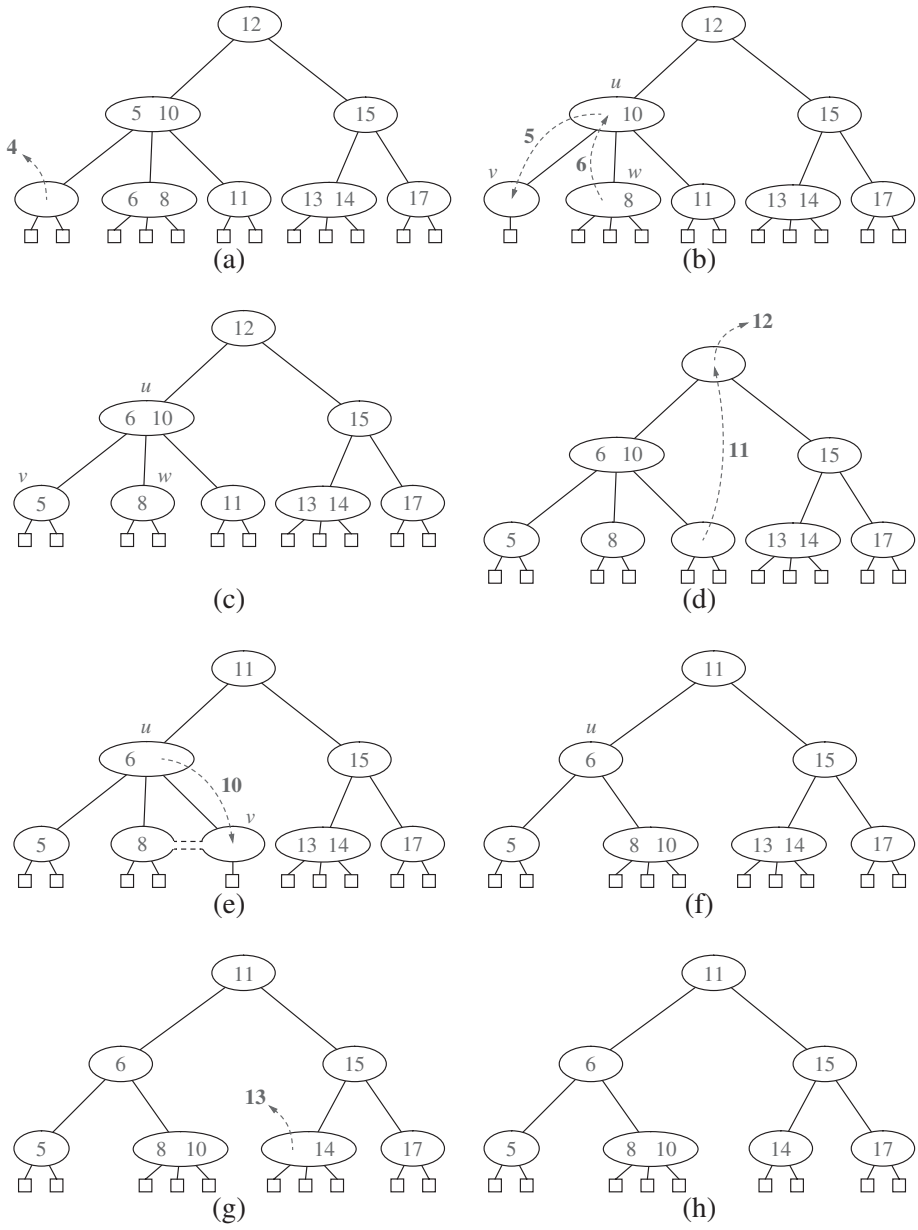

Figure 20.8: A sequence of removals from a $(2, 4)$ tree: (a) removal of 4, causing an underflow; (b) a transfer operation; (c) after the transfer operation; (d) removal of 12, causing an underflow; (e) a fusion operation; (f) after the fusion operation; (g) removal of 13; (h) after removing 13.

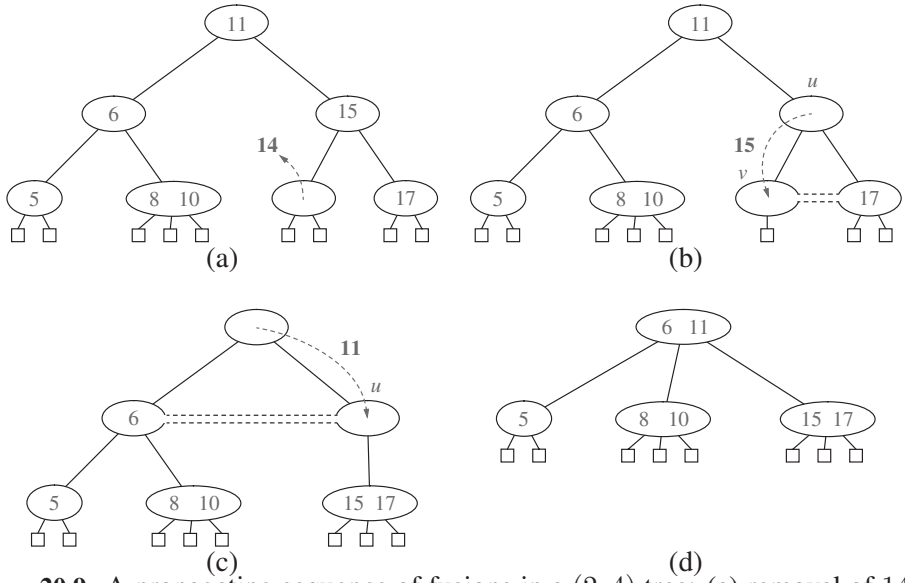

Figure 20.9: A propagating sequence of fusions in a $(2, 4)$ tree: (a) removal of 14, which causes an underflow; (b) fusion, which causes another underflow; (c) second fusion operation, which causes the root to be removed; (d) final tree.

Analysis of $(2, 4)$ Trees

The following theorem summarizes the running times of the main operations of a dictionary realized with a $(2, 4)$ tree.

Theorem 20.3: *A $(2, 4)$ tree for n key-element items uses $O(n)$ space and implements the operations of a dictionary data structure with the following running times. Finding an item, inserting an item, and removing an item each take time $O(\log n)$.*

The time complexity analysis is based on the following:

- The height of a $(2, 4)$ tree storing n items is $O(\log n)$, by Theorem 20.2.
- A split, transfer, or fusion operation takes $O(1)$ time.
- A search, insertion, or removal of an item visits $O(\log n)$ nodes.

Thus, $(2, 4)$ trees provide for fast dictionary search and update operations. $(2, 4)$ trees also have an interesting relationship to the data structure we discuss next, which is better suited for external memory.

20.2.3 (a, b) Trees and B-Trees

Recalling the great time difference that exists between main memory accesses and disk accesses, the main goal of maintaining a dictionary 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 dictionary implementations by counting the number of disk transfers each would require to perform the standard dictionary search and update operations.

Let us first consider some external-memory inefficient dictionary implementations based on sequences. If the sequence representing a dictionary is implemented as an unsorted, doubly linked list, then insert and remove can be performed with $O(1)$ transfers each, assuming we know which block holds an item to be removed. But, in this case, searching requires $\Theta(n)$ 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-20.1), 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, for we may have to access all blocks to move elements up or down. Thus, sequence dictionary implementations are not efficient for external memory.

If sequence implementations are inefficient, then perhaps 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 dictionary items at the nodes of a binary tree or of a graph. In the worst case, 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 is good, but we can do better. In particular, we can perform dictionary queries and updates using only $O(\log_B n) = O(\log n/ \log B)$ transfers.

The main idea for improving the external-memory performance of the dictionary implementations discussed above is that we should be willing to perform up to $O(B)$ internal-memory accesses to avoid a single disk transfer, where B denotes the size of a block. The hardware and software that drives the disk performs this many internal-memory accesses just to bring a block into internal memory, and this is only a small part of the cost of a disk transfer. Thus, $O(B)$ high-speed, internal-memory accesses are a small price to pay to avoid a time-consuming disk transfer.

(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 dictionary using a multi-way search tree, which is a generalization of the $(2, 4)$ tree data structure to a structure known as an (a, b) tree.

Formally, 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 $b - 1$ items. The algorithms for searching, inserting, and removing elements 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 generalized class of trees provides a flexible search structure, where the size of the nodes and the running time of the various dictionary 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 performance for external memory.

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.

Theorem 20.4: *The height of an (a, b) tree storing n items is $\Omega(\log n / \log b)$ and $O(\log n / \log a)$.*

Proof: Let T be an (a, b) tree storing n elements, and let h be the height of T. We justify the theorem 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 Theorem 20.1, $n'' = n + 1$. Thus,

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

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

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

■

We recall that in a multi-way search tree T, each node v of T holds a secondary structure $D(v)$, which is itself a dictionary (Section 20.2.1). If T is an (a, b) tree, then $D(v)$ stores at most b items. Let $f(b)$ denote the time for performing a search in a $D(v)$ dictionary. The search algorithm in an (a, b) tree is exactly like the one for multi-way search trees given in Section 20.2.1. Hence, searching in an (a, b) tree T with n items takes $O(\frac{f(b)}{\log a} \log n)$. Note that if b is a constant (and thus a is

also), then the search time is $O(\log n)$, independent of the specific implementation of the secondary structures.

The main application of (a, b) trees is for dictionaries stored in external memory (for example, on a disk or CD/DVD-ROM). 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.

Insertion and Removal in an (a, b) Tree

The insertion algorithm for an (a, b) tree is similar to that for a $(2, 4)$ tree. An overflow occurs when an item 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 item 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 $D(v')$ and $D(v'')$.

Removing an element from an (a, b) tree is also 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 either 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. Here, we see another reason for requiring $a \leq (b+1)/2$. Note that as a consequence of the fusion, we need to build the secondary structure $D(w)$.

Table 20.10 shows the running time of the main operations of a dictionary realized by means of an (a, b) tree T.

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

Table 20.10: Time complexity of the main methods of a dictionary realized by an (a, b) tree. We let $f(b)$ denote the time to search a b-node and $g(b)$ the time to split or fuse a b-node. We also denote the number of elements in the dictionary with n. The space complexity is $O(n)$.

The bounds in Table 20.10 are based on the following assumptions and facts:

- The (a, b) tree T is represented using the data structure described in Section 20.2.1, and 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 $O(1)$ in the context where we are only counting disk transfers.
- The height of an (a, b) tree storing n elements is at most $O((\log n)/(\log a))$ (Theorem 20.4).
- A search visits $O((\log n)/(\log a))$ nodes on a path between the root and an external node, and spends $f(b)$ time per node.
- A transfer operation takes $f(b)$ time.
- A split or fusion operation takes $g(b)$ time and builds a secondary structure of size $O(b)$ for the new node(s) created.
- An insertion or removal of an element visits $O((\log n)/(\log a))$ nodes on a path between the root and an external node, and spends $g(b)$ time per node.

Thus, we may summarize as follows.

Theorem 20.5: *An (a, b) tree implements an n-item dictionary to support performing insertions and removals in $O((g(b)/\log a)\log n)$ time, and performing find queries in $O((f(b)/\log a)\log n)$ time.*

B-Trees

A specialized version of the (a, b) tree data structure, which is an efficient method for maintaining a dictionary in external memory, is the data structure known as the "B-tree." (See Figure 20.11.) A ***B-tree of order*** d is simply an (a, b) tree with $a = \lceil d/2 \rceil$ and $b = d$. Since we discussed the standard dictionary query and update methods for (a, b) trees above, we restrict our discussion here to the analysis of the input/output (I/O) performance of B-trees.

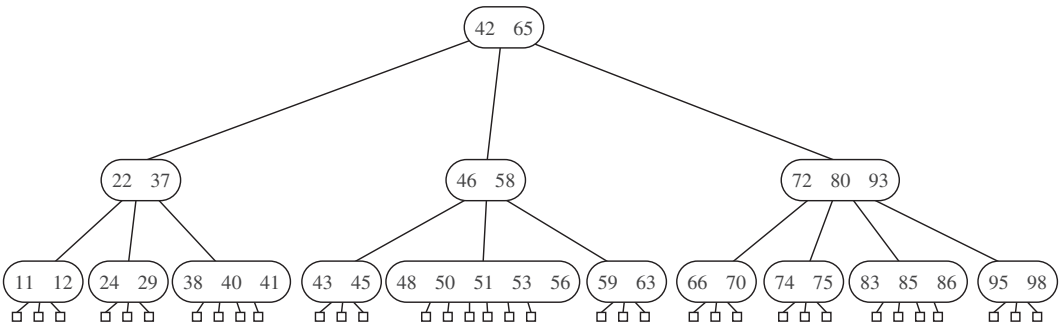

Figure 20.11: A B-tree of order 6.

Parameterizing B-trees for External Memory

The most important observation about 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. That is, we choose d so that

$$d \text{ is } \Theta(B).$$

This choice also implies that we may assume that a and b are $\Theta(B)$ in the analysis of the search and update operations on (a,b) trees. Also, recall that we are interested primarily in the number of disk transfers needed to perform various operations. Thus, the choice for d also implies that

$$f(b) = c,$$

and

$$g(b) = c,$$

for some constant $c \geq 1$, for each time we access a node to perform a search or an update operation, we need only perform a single disk transfer. That is, $f(b)$ and $g(b)$ are both $O(1)$. 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 dictionary search or update operation on a B-tree requires only

$$\begin{aligned} O(\log_{\lceil d/2 \rceil} n) &= O(\log n / \log B) \\ &= O(\log_B n) \end{aligned}$$

disk transfers. For example, an insert operation proceeds down the B-tree to locate the node in which to insert the new item. 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, in a remove operation, we remove an item from a node, and, if this results in a node ***underflow*** (to have $\lceil d/2 \rceil - 1$ children), then we either 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 will continue up the B-tree for at most $O(\log_B n)$ levels. Thus, we have the following:

Theorem 20.6: *A B-tree with n items executes $O(\log_B n)$ disk transfers in a search or update operation, where B is the number of items that can fit in one block.*

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. Analytical and experimental study of the average block usage in a B-tree is that it is closer to 67%, which is quite good.

20.3 External-Memory Sorting

In addition to data structures, such as dictionaries, 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.

A Lower Bound for External-Memory Sorting

As we discussed above, there can be a big difference between an algorithm's performance in internal memory and its performance in external memory. For example, the performance of the radix-sorting algorithm is bad in external memory, yet good in internal memory. Other algorithms, such as the merge-sort algorithm, are reasonably good in both internal memory and external memory, however. The number of block transfers performed by the traditional merge-sorting algorithm is $O((n/B)\log_2 n)$, where B is the size of disk blocks. While this is much better than the $O(n)$ block transfers performed by an external version of radix sort, it is, nevertheless, not the best that is achievable for the sorting problem. In fact, we can show the following lower bound, whose proof is beyond the scope of this book.

Theorem 20.7: *Sorting n elements stored in external memory requires*

$$\Omega\left(\frac{n}{B}\cdot\frac{\log(n/B)}{\log(M/B)}\right)$$

block transfers, where M is the size of the internal memory.

The ratio M/B is the number of external-memory blocks that can fit into internal memory. Thus, this theorem is saying that the best performance we can achieve for the sorting problem is equivalent to the work of scanning through the input set (which takes $\Theta(n/B)$ transfers) at least a logarithmic number of times, where the base of this logarithm is the number of blocks that fit into internal memory. We will not formally justify this theorem, but we will show how to design an external-memory sorting algorithm whose running time comes within a constant factor of this lower bound.

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, ..., 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 will be within a constant factor of the lower bound given in Theorem 20.7. We choose

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

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.

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 then 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 will associate the smallest object from among all the lists with r. This completes the initialization for the d-way merge. (See Figure 20.12.)

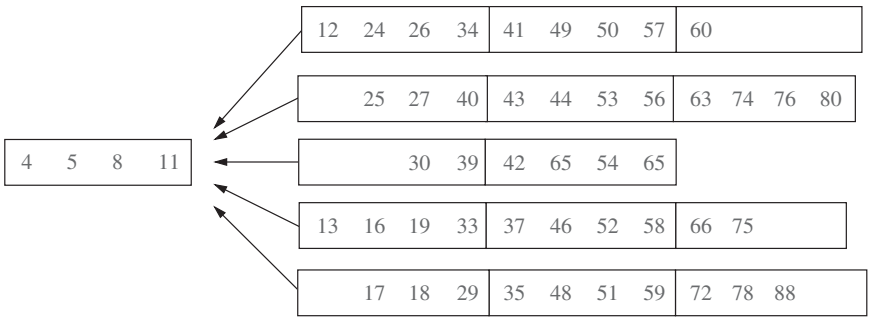

Figure 20.12: A d-way merge. We show a five-way merge with $B = 4$.

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 with the node i 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 the following:

Theorem 20.8: *Given an array, S, of n elements stored in external memory, we can sort S using $O((n/B)\log(n/B)/\log(M/B))$ block transfers (I/Os) 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.*

Achieving "Near" Machine Independence

Using B-trees and external sorting algorithms can produce significant reductions in the number of block transfers. The most important piece of information that made such reductions possible was knowing the value of B, the size of a disk block (or cache line). This information is, of course, machine-dependent, but it is one of the few truly machine-dependent pieces of information that are needed, with one of the others being the ability to store keys continuously in arrays.

From our description of B-trees and external sorting, we might think that we also require low-level access to the external-memory device driver, but this is not strictly needed in order to achieve the claimed results to within a constant factor. In particular, in addition to knowing the block size, the only other thing we need to know is that large arrays of keys are partitioned into blocks of continuous cells. This allows us to implement the "blocks" in B-trees and our external-memory sorting algorithm as separate B-sized arrays, which we call ***pseudo-blocks***. If arrays are allocated to blocks in the natural way, any such pseudo-block will be allocated to at most two real blocks. Thus, even if we are relying on the operating system to perform block replacement (for example, using FIFO, LRU, or the Marker policy discussed later in Section 20.4), we can be sure that accessing any pseudo-block takes at most two, that is, $O(1)$, real block transfers. By using pseudo-blocks, then, instead of real blocks, we can implement a dictionary to achieve search and update operations that use only $O(\log_B n)$ block transfers. We can, therefore, design external-memory data structures and algorithms without taking complete control of the memory hierarchy from the operating system.

20.4 Online Caching Algorithms

An ***online algorithm*** responds to a sequence of ***service requests***, each an associated ***cost***. For example, a web page replacement policy maintains pages in a cache, subject to a sequence of access requests, with the cost of a web page request being zero if the page is in the cache and one if the page is outside the cache. In an ***online*** setting, the algorithm must completely finish responding to a service request before it can receive the next request in the sequence. If an algorithm is given the entire sequence of service requests in advance, it is said to be an ***offline*** algorithm. To analyze an online algorithm, we often employ a competitive analysis, where we compare a particular online algorithm A to an optimal offline algorithm, OPT. Given a particular sequence $P = (p_1, p_2, \ldots, p_n)$ of service requests, let $cost(A, P)$ denote the cost of A on P and let $cost(OPT, P)$ denote the cost of the optimal algorithm on P. The algorithm A is said to be ***c-competitive*** for P if

$$cost(A, P) \leq c \cdot cost(OPT, P) + b,$$

for some constant $b \geq 0$. If A is c-competitive for every sequence P, then we simply say that A is c-competitive, and we call c the ***competitive ratio*** of A. If $b = 0$, then we say that the algorithm A has a ***strict*** competitive ratio of c.

A well-known online problem, explained with a story, is the ***ski rental problem***. Alice has decided to try out skiing, but is uncertain whether she will like it or whether she will be injured and have to stop. Each time Alice goes skiing, it costs her x dollars to "rent" the necessary skiing equipment. Suppose it costs y dollars to buy skis and the equipment that goes with them. Let us say, for the sake of the story, that y is 10 times larger than x, that is, $y = 10x$. The dilemma for Alice is to decide if and when she should buy the skiing equipment instead of renting this equipment each time she goes skiing. For example, if she buys before her first skiing trip and then decides she doesn't like skiing, then she has spent 10 times more than she should. But if she skis many times and never buys the equipment, then she will spend potentially even more than 10 times more than she should. In fact, if she skis n times, then this strategy of always "renting" will cause her to spend $n/10$ times as many dollars as she should. That is, a strategy of buying the first time has a worst-case competitive ratio of 10 and the always-rent strategy has a worst-case competitive ratio of $n/10$. Neither of these choices is good.

Fortunately, Alice has a strategy with a competitive ratio of 2. Namely, she can rent for 9 times and then buy the skiing equipment on the 10th day she skis. The worst-case scenario is that she never uses the skis she just bought. So, in this case, she spends $9x + y = 1.9y$ dollars, when she should have spent $y = 10x$; hence, this strategy has a competitive ratio of 1.9. In fact, no matter how much bigger y is than x, if Alice buys on day $\lceil y/x \rceil$, and then buys, she will have a competitive ratio of at most 2.

20.4.1 Caching Algorithms

There are several applications that must deal with revisiting information presented in pages. For instance, web page revisits have been shown to exhibit localities of reference, both in time and in space. Similarly, the way in which a CPU access disk pages tends to exhibit such localities as well. To exploit these localities of reference, it is often advantageous to store copies of such 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," each of which can contain a web or disk page, depending on the application. We assume that a 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 pages. Each time its requests such a web page l, it must determine (using a quick test) whether l is unchanged and currently contained in the cache. If l is contained in the cache, then the request can be satisfied using the cached copy. If l is not in the cache, however, the page for l is requested and transferred into the cache. If one of the m slots in the cache is available, then the new page, l is assigned to one of the empty slots. But if all the m cells of the cache are occupied, then the computer must determine which previously loaded 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. Some of the better-known page replacement policies include the following (see Figure 20.13):

- **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.

The Random strategy is easy to implement, for 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, 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.

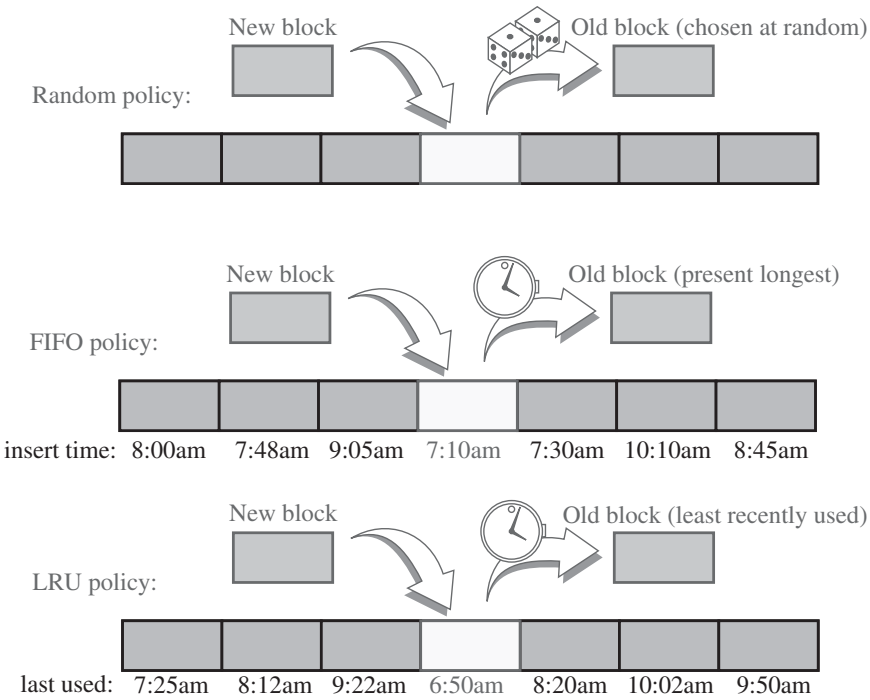

Figure 20.13: The Random, FIFO, and LRU page replacement policies.

The LRU strategy goes a step further than the FIFO strategy, which assumes that the page that has been in the cache the longest among all those present is the least likely to be requested in the near future. The LRU strategy explicitly takes advantage of temporal locality, by always evicting the page that was least recently used. From a policy point of view, this strategy 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 $O(1)$. Whenever 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 above implementation, 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 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. The experiments have focused primarily on the Random, FIFO, and LRU policies. Based on these experimental comparisons, the ordering of policies, from best to worst, is as follows: (1) LRU, (2) FIFO, and (3) Random. In fact, LRU is significantly better than the others on typical request sequences, but it still has poor performance in the worst case, as the following theorem shows.

Theorem 20.9: *The FIFO and LRU page replacement policies for a cache with m pages have competitive ratio at least m.*

Proof: We observed above that there is a sequence $P = (p_1, p_2, \ldots, p_n)$ of page requests causing FIFO and LRU to perform a page replacement with each request—the loop of $m + 1$ requests. We compare this performance with that of the optimal offline algorithm, *OPT*, which, in the case of the page replacement problem, is to evict from the cache the page that is requested the furthest into the future. This strategy can only be implemented, of course, in the offline case, when we are given the entire sequence P in advance, unless the algorithm is "prophetic." When applied to the loop sequence, the *OPT* policy will perform a page replacement once every m requests (for it evicts the most recently referenced page each time, as this one is referenced furthest in the future). Thus, both FIFO and LRU are c-competitive on this sequence P, where

$$c = \frac{n}{n/m} = m.$$

Observe that if any portion $P' = (p_i, p_{i+1}, \ldots, p_j)$ of P makes requests to m different pages (with p_{i-1} and/or p_{j+1} not being one of them), then even the optimal algorithm must evict one page. In addition, the most number of pages the FIFO and LRU policies evict for such a portion P' is m, each time evicting a page that was referenced prior to p_i. Therefore, FIFO and LRU have a competitive ratio of m, and this is the best possible competitive ratio for these strategies in the worst case. ■

The Randomized Marker Algorithm

Even though the deterministic FIFO and LRU policies can have poor worst-case competitive ratios compared to the "prophetic" optimal algorithm, we can show that a randomized policy that attempts to simulate LRU has a good competitive ratio. Specifically, let us study the competitive ratio of a randomized strategy that tries to emulate the LRU policy. From a strategic viewpoint, this policy, which is known as the ***Marker strategy***, emulates the best aspects of the deterministic LRU policy, while using randomization to avoid the worst-case situations that are bad for the LRU strategy. The policy for Marker is as follows:

- **Marker**: Associate, with each page in the cache, a Boolean variable "marked," which is initially set to "false" for every page in the cache. If a browser requests a page that is already in the cache, that page's marked variable is set to "true." Otherwise, if a browser requests a page that is not in the cache, a random page whose marked variable is "false" is evicted and replaced with the new page, whose marked variable is immediately set to "true." If all the pages in the cache have marked variables set to "true," then all of them are reset to "false." (See Figure 20.14.)

Competitive Analysis for a Randomized Online Algorithm

Armed with the above policy definition, we would now like to perform a competitive analysis of the Marker strategy. Before we can do this analysis, however, we must first define what we mean by the competitive ratio of a randomized online algorithm. Since a randomized algorithm A, like the Marker policy, can have many different possible runs, depending upon the random choices it makes, we define such an algorithm to be ***c*-competitive** for a sequence of requests P if

$$E(cost(A, P)) \leq c \cdot cost(\mathbf{OPT}, P) + b,$$

for some constant $b \geq 0$, where $E(cost(A, P))$ denotes the expected cost of algorithm A on the sequence P (with this expectation taken over all possible random choices for the algorithm A). If A is c-competitive for every sequence P, then we simply say that A is c-competitive, and we call c the ***competitive ratio*** for A.

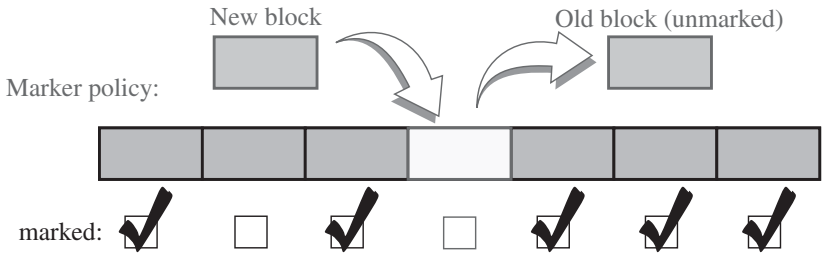

Figure 20.14: The Marker page replacement policy.

Theorem 20.10: *The Marker page policy for a cache with m pages has competitive ratio $2 \log m$.*

Proof: Let $P = (p_1, p_2, \ldots, p_n)$ be a sufficiently long sequence of page requests. The Marker policy implicitly partitions the requests in P into ***rounds***. Each round begins with all the pages in the cache having "false" marked labels, and a round ends when all the pages in the cache have "true" marked labels (with the next request beginning the next round, since the policy then resets each such label to "false"). Consider the ith round in P, and call a page requested in round i ***fresh*** if it is not in the Marker policy's cache at the beginning of round i. Also, we refer to a page in the Marker's cache that has a false marked label ***stale***. Thus, at the beginning of a round i, all the pages in the Marker policy's cache are stale. Let m_i denote the number of fresh pages referenced in the ith round, and let b_i denote the number of pages that are in the cache for the *OPT* algorithm at the beginning of round i and are not in the cache for the Marker policy at this time. Since the Marker policy has to perform a page replacement for each of the m_i requests, algorithm *OPT* must perform at least $m_i - b_i$ page replacements in round i. (See Figure 20.15.) In addition, since each of the pages in the Marker policy's cache at the end of round i are requested in round i, algorithm *OPT* must perform at least b_{i+1} page replacements in round i. Thus, the algorithm *OPT* must perform at least

$$\max\{m_i - b_i, b_{i+1}\} \geq \frac{m_i - b_i + b_{i+1}}{2}$$

page replacements in round i. Summing over all k rounds in P then, we see that algorithm *OPT* must perform at least the following number of page replacements:

$$L = \sum_{i=1}^{k} \frac{m_i - b_i + b_{i+1}}{2} = (b_{k+1} - b_1)/2 + \frac{1}{2} \sum_{i=1}^{k} m_i.$$

Next, let us consider the expected number of page replacements performed by the Marker policy.

We have already observed that the Marker policy has to perform at least m_i page replacements in round i. It may actually perform more than this, however, if it evicts stale pages that are then requested later in the round. Thus, the expected number of page replacements performed by the Marker policy is $m_i + n_i$, where n_i is the expected number of stale pages that are referenced in round i after having been evicted from the cache. The value n_i is equal to the sum, over all stale pages referenced in round i, of the probability that these pages are outside of the cache when referenced. At the point in round i when a stale page v is referenced, the probability that v is out of the cache is at most f/g, where f is the number of fresh pages referenced before page v and g is the number of stale pages that have not yet been referenced. This is because each reference to a fresh page evicts some unmarked stale page at random. The cost to the Marker policy will be highest then, if all m_i requests to fresh pages are made before any requests to stale pages. So, assuming this worst-case viewpoint, the expected number of evicted stale pages

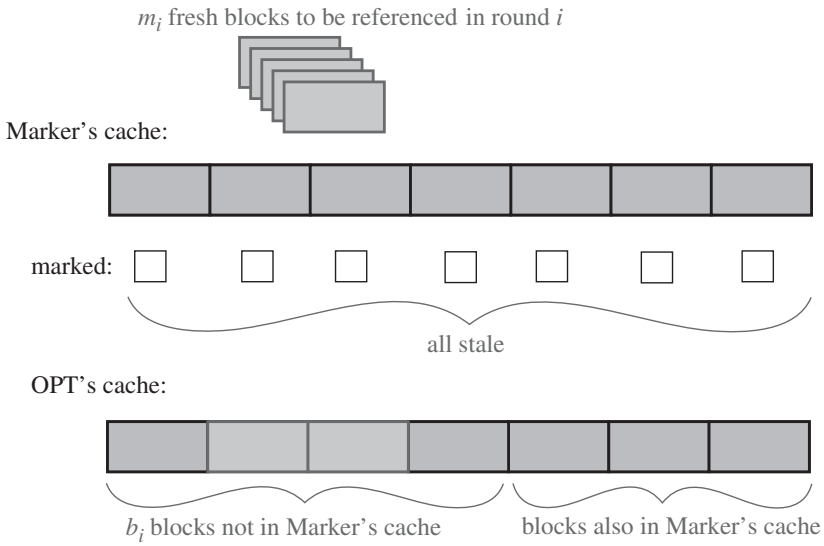

Figure 20.15: The state of Marker's cache and OPT's cache at the beginning of round i.

referenced in round i can be bounded as follows:

$$\begin{aligned} n_i &\leq \frac{m_i}{m} + \frac{m_i}{m-1} + \frac{m_i}{m-2} + \cdots + \frac{m_i}{m_i+1} \\ &\leq m_i \sum_{j=1}^{m} \frac{1}{j}, \end{aligned}$$

since there are $m - m_i$ references to stale pages in round i. Noting that this summation is known as the mth harmonic number, which is denoted H_m, we have

$$n_i \leq m_i H_m.$$

Thus, the expected number of page replacements performed by the Marker policy is at most

$$U = \sum_{i=1}^{k} m_i(H_m + 1) = (H_m + 1) \sum_{i=1}^{k} m_i.$$

Therefore, the competitive ratio for the Marker policy is at most

$$\begin{aligned} \frac{U}{L} &= \frac{(H_m + 1) \sum_{i=1}^{k} m_i}{(1/2) \sum_{i=1}^{k} m_i} \\ &= 2(H_m + 1). \end{aligned}$$

Using an approximation for H_m, namely that $H_m \leq \log m$, the competitive ratio for the Marker policy is at most $2 \log m$. ■

Thus, the competitive analysis shows that the Marker policy is fairly efficient.

20.5 Exercises

Reinforcement

R-20.1 Describe, in detail, the insertion and removal algorithms for an (a, b) tree.

R-20.2 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-20.3 For what values of d is the tree T of the previous exercise an order-d B-tree?

R-20.4 Draw the order-7 B-tree resulting from inserting the following keys (in this order) into an initially empty tree T:

$$(4, 40, 23, 50, 11, 34, 62, 78, 66, 22, 90, 59, 25, 72, 64, 77, 39, 12).$$

R-20.5 Show each level of recursion in performing a four-way, external-memory merge-sort of the sequence given in the previous exercise.

R-20.6 Consider the generalization of the renter's dilemma where Alice can buy or rent her skis separate from her boots. Say that renting skis costs a dollars, whereas buying skis costs b dollars. Likewise, say that renting boots costs c dollars, whereas buying boots costs b dollars. Describe a 2-competitive online algorithm for Alice to try to minimize the costs for going skiing subject to the uncertainty of her not knowing how many times she will continue to go skiing in the future.

R-20.7 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-20.8 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-20.9 Consider an initially empty memory cache consisting of four pages. How many page misses does the marker algorithm incur on the following page-request sequence: $(2, 3, 4, 1, 2, 5, 1, 3, 5, 4, 1, 2, 3)$? Show the random choices your algorithm made.

R-20.10 Consider an initially empty memory cache consisting of four pages. Construct a sequence of memory requests that would cause the marker algorithm to go through four rounds.

Creativity

C-20.1 Show how to implement a dictionary 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-20.2 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-20.3 Another possible external-memory dictionary 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-20.4 Suppose that instead of having the node-search function $f(d) = 1$ in an order-d B-tree T, we instead have $f(d) = \log d$. What does the asymptotic running time of performing a search in T now become?

C-20.5 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-20.6 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-20.7 Show that LRU is m-competitive for any sequence of n page requests, where m is the size of the memory cache.

C-20.8 Show that FIFO is m-competitive for any sequence of n page requests, where m is the size of the memory cache.

C-20.9 What is the expected number of block replacements performed by the Random policy on a cache of size m, for an access sequence of length n, that iteratively accesses $m+1$ blocks in a cyclic fashion (assuming n is much larger than m)?

C-20.10 Show that the Marker algorithm is H_m-competitive when the size of the cache is m and there are $m+1$ possible pages that can be accessed, where H_m denotes the mth Harmonic number.

Applications

A-20.1 Suppose you are processing a large number of operations in a consumer-producer process, such as a buffer for a large media stream. Describe an external-memory data structure to implement a queue so that the total number of disk transfers needed to process a sequence of n enqueue and dequeue operations is $O(n/B)$.

A-20.2 Imagine that you are trying to construct a minimum spanning tree for a large network, such as is defined by a popular social networking website. Based on using Kruskal's algorithm, the bottleneck is the maintenance of a union-find data structure. Describe how to use a B-tree to implement a union-find data structure (from Section 7.1) so that union and find operations each use at most $O(\log n/\log B)$ disk transfers each.

A-20.3 Suppose you are processing an automated course registration program. The data set in this case is a large file of N course numbers, one for each course request made by a student. Show that you can count the number of requests made for each course, using $O((N/B)\log(N/B)/\log(M/B))$ I/Os.

A-20.4 In the MapReduce framework, for performing a parallel computation, a crucial step involves an input that consists of a set of n key-value pairs, (k, v), for which we need to collect each subset of key-value pairs that have the same key, k, into a single file. Describe an efficient external-memory algorithm for constructing all such files. How many disk transfers does your algorithm perform?

A-20.5 Suppose Alice is faced with the ski rental problem, where buying skis is 20 times more expensive than renting. In this case, however, Alice notices that she has a fair coin in her pocket and is willing to consider a randomized strategy. Show that she can use her coin to come up with a strategy with an expected competitive ratio of 1.8 or better.

Chapter Notes

B-trees were invented by Bayer and McCreight [23] and Comer [47] provides a very nice overview of this structure. The books by Mehlhorn [157] and Samet [182] also discuss B-trees and their variants. Aho, Hopcroft, and Ullman [8] discuss $(2, 3)$ trees, which are similar to $(2, 4)$ trees. Arge and Vitter [12] present a weight-balanced B-tree, which has a number of nice properties. Knuth [131] has very nice discussions about external-memory sorting and searching. Aggarwal and Vitter [6] study the I/O complexity of sorting and related problems, establishing upper and lower bounds. Goodrich *et al.* [90] 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 book by Vitter [211].

The reader interested in further study of online algorithms is referred to the book by Borodin and El-Yaniv [34] or the paper by Koutsoupias and Papadimitriou [134]. The marker caching algorithm is discussed in the book by Borodin and El-Yaniv; our discussion is modeled after a similar discussion in by Motwani and Raghavan [162]. Exercises C-20.7 and C-20.8 come from Sleator and Tarjan [196].