AN ONLINE ALGORITHM TO OPTIMIZE FILE LAYOUT IN A DYNAMIC ENVIRONMENT

SHAHRAM GHANDEHARIZADEH, DOUG IERARDI, AND ROGER ZIMMERMANN

ABSTRACT. We describe an algorithm to manage the storage and layout of files cached on mechanical devices, such as magnetic disk drives. The algorithms respond in an online manner to maintain a dynamically changing working set of disk–resident files, while providing a guaranteed degree of contiguity in the layout of each file on the device, with fewer than $\lceil \lg N \rceil$ breaks for each disk–resident file of N blocks.

KEYWORDS: data processing, file systems, file geometry, caching, continuous media

1. INTRODUCTION

A trend in the area of databases has been an increase in the number of repositories whose primary function is to disseminate information. These systems are expected to play a major role in scientific applications; library, health care and general data warehousing; in the entertainment industry; and in the deployment of knowledge-based applications. Such repositories typically provide on-line access to vast amount of data. The large size of their databases has led to the use of hierarchical storage structures consisting of a combination of fast and slow devices: the database resides permanently on the slowest devices, and the system controls the placement of data among the strata of the hierarchy to hide their high latency using its faster devices, such as magnetic disks or disk arrays. Data is cached on the disks, and swapped in and out based upon expected future access patterns, with the objective of minimizing the frequency of access to slower devices.

A hierarchical storage structure may consist of a variety of devices, such as magnetic and optical disks and tape juke boxes. These devices share several properties, at least from the point of view presented by the device driver interface: (1) Data is stored on the medium in a linear manner. (2) The read head of the device may be moved to physical location on the medium; this operation is termed a *seek*. Seeks generally involve mechanical operations, and so the time required for a seek is often substantial. (3) Once the seek is complete, a device can read sequentially from the medium and can generally sustain a relatively high transfer rate. In this sense, seek operations are wasteful and should be minimized in order to maximize the net transfer rate.

This study describes an algorithm to manage the storage and placement of objects or files on such devices. Its goal is to maximize contiguity in the layout of objects stored on the device, and thus to minimize the number of seeks incurred when retrieving an object in a sequential manner, termed *intra-file seeks* in [GVK95]. To simplify the discussion, we shall assume that objects will be cached on magnetic disks.

Date: October 24, 1995.

1.1. **Motivation.** This work was motivated by the design of a continuous media server for isochronous media [Bu94], such as digital audio and video. Our focus in this paper is on the design of the storage manager for the disk drives. All objects in the system are read–only, and a copy of each resides permanently on the slowest device in the hierarchy (*e.g.* a tape juke box). The storage manager and file system components cooperate to place data across devices and to lay out data within devices so that the time required for retrieval is both minimal and predictable. As a consequence, each object in the system will be retrieved in units of a particular size, where that size is determined by its bit-rate, the length of a scheduling period, and the degree of striping. This study focuses on the layout of such units upon a single device.

In multimedia systems of the sort sketched above [GC92], sequential (contiguous) layout and various forms of constrained allocation have been used to minimize or eliminate the occurrence of intrafile seeks, and thus to optimize performance of the disk subsystem. However, these studies have focused largely on the case in which all objects have the same bit-rate, and hence the unit of retrieval is uniform across all objects. The situation becomes more complex in the case of a collection of objects with different bandwidth requirements. This motivates the question: to what extent can one maximize the contiguity of a heterogeneous collection of objects or files, and thus minimize the occurrence of such intrafile seeks? In a dynamic environment, where the disk-resident population changes over time, one would expect a tradeoff between the degree of contiguity that can be achieved and the amount of work that goes into reorganizing the disk's configuration. This is the problem addressed below.

Since we focus only on the space management issue, we assume an external module that determines the set of objects to be cached on the disk, such as that of [GIZ94]. This module issues a (potentially infinite) sequence of requests to materialize and free objects on the device. We further assume that whenever a request to materialize an object of n blocks is issued, there are at least n free blocks on the disk. So the external module selects victims to be deleted to accommodate objects that are to be materialized, and has issued the necessary requests in an appropriate order.

Given such a module, it is the duty of the disk manager to maintain the requested working set on disk. We assume that the disk can be viewed as a linear sequence[1] of C physical blocks. Its performance will be judged by two criteria: (1) the number of *breaks* or discontiguities in the layout of each disk–resident object in the worst case, and (2) the additional work required to maintain this disk organization. The algorithm developed in §3 requires copying at most n additional blocks when materializing an n-block object on disk, and guarantees that it will always be laid out with at most $\lceil \lg n \rceil$ breaks. The two algorithms of §2 and §3 present a tradeoff between the number of additional maintenance operations incurred and the complexity of the disk's bookkeeping structure. Note that within the context that motivates this work, these objects may be either entire multimedia objects, such as a video or audio clips, or just the parts of objects that are retrieved during each scheduling period. In the first case, the bound on the number of breaks in the

[1]This model of a disk drive omits important characteristics of the device. Nevertheless, the model is consistent with the approaches taken in a variety of implementations, such as [And92] in which this linear sequence is given by the logical addresses of disk blocks, or [GC92] which also took into account intrafile seeks that may result from a disk drive's remapping of defective sectors.

layout corresponds to the total number of intrafile seeks incurred during retrieval of the entire object; in the latter case, this number of intrafile seeks may be incurred during each period.

1.2. **The model.** Assume that the storage medium has been partitioned into C *physical blocks* of a fixed size, where a block is the minimum unit of space allocation for the physical device. All blocks are assumed to have the same size. Let $B_0, B_1, \ldots, B_{C-1}$ denote the ordered sequence of physical blocks on the disk. Assume also a fixed collection of objects (or files). Each object o is composed of an ordered sequence $\langle o_1, \ldots, o_n \rangle$ of n *pages*, for some $n > 0$ (its *size*, denoted $|o|$). Each physical block of the disk can hold exactly one page of data. An object is *disk–resident* if all its pages reside on the disk. The assignment of pages to blocks is given by a partial function ℓ, which assigns no two pages to the same physical block. A disk block is *free* if ℓ does not assign a page of any disk–resident object to it. Let R_ℓ be the set of disk–resident objects for ℓ. We assume that every object that is not disk–resident has no pages residing on the disk. So the algorithm cannot take advantage of "partially resident" objects.

Let o be any disk resident object, and o_i a page of o. Then there is a *break at* o_i under ℓ if $i > 0$ and its predecessor o_{i-1} is not assigned to the block immediately preceding $\ell(o_i)$. That is, in a continuous retrieval of the pages of o in sequence, the device is forced to perform an intrafile seek to block $\ell(o_i)$ after reading $\ell(o_{i-1})$. o's *internal fragmentation* under a given layout ℓ is the number of breaks in the file's layout.

1.3. **Basic operations and their costs.** Materialization of an object o on the disk implies that the set of resident objects and their layout both change. To simplify the analysis, assume no cost for writing a page of o to disk during materialization, because any algorithm must incur this cost. Instead we focus on two quantities that the algorithm intends to minimize: (1) the internal fragmentation of an object, and (2) the number of pages that are copied from one physical block to another. It is assumed that the various data–structures that record the state of the system (such as directories and the free list) are maintained in main memory.

2. The Basic Algorithm

To manage the disk's space, we first impose an ordered d-ary tree structure on the sequence of physical blocks, in which leaves correspond to blocks on the disk, and their order corresponds to the blocks' physical sequence. To simplify the description of the algorithm, we take $d = 2$. Generalization to the case $d > 2$ is straightforward.

The tree structure imposed on the physical blocks is built up in the following manner. Let $B[i, j]$ denote the contiguous sequence of blocks $B_i, \ldots, B_j$. For each integer $h = 0, \ldots, \lfloor \lg C \rfloor$, and each $i = 0, \ldots, \lfloor C/2^h \rfloor$, the interval

$$B[i\,2^h, (i+1)\,2^h - 1]$$

is the i^{th} *section of height* h. The sections of height 0, consisting of single blocks, are the leaves. Each section of height $h > 0$ contains exactly two sections of height $h-1$, which are its children. By taking these sections as internal nodes, the blocks of the disk are now organized into an ordered forest of complete binary trees. See Figure 1 for an illustration. Contiguous sections that are siblings in this tree also called "buddies" [Kno65]; so each contiguous section of height h can be decomposed

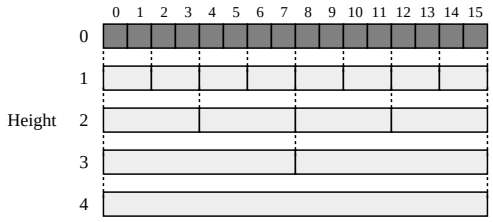

Figure 1: Illustration of proposed organization of sections on a 16-block disk.

into a pair of buddy sections of height $h-1$. Conversely, each pair of buddies of height $h-1$ can be combined to form a single contiguous section of height h. There can be at most one section at each height that has no buddy, which we call *unpaired* sections. Each of these is the root of a complete binary tree in this forest. For example, if a disk has a capacity of $261 = 2^8 + 2^2 + 2^0$ blocks then one could organize the first 256 blocks of the disk into a complete binary tree of height 8. The remaining 5 blocks would be organized into trees of height 2 and 0. There are three non-paired sections — each corresponding to the root of one of these complete binary trees — of heights 8, 2 and 0.

We say that a section is occupied by object o if some subsequence of o's pages are laid out contiguously in the blocks of this section. By "the sections of o", we mean the maximal sections occupied by o (*i.e.* those of maximal height under containment).

2.1. **The free list.** A section is free if every page in it is free. For any layout, the free list is a list of the free sections that are not contained in other free sections. The current free list is recorded in a data structure, maintained as a sequence of lists — one for each possible section height from 0 to $\lfloor \lg C \rfloor$. The address of each maximal section of height h is enqueued in a list that handles sections of height h only.

2.2. **Invariant properties.** For any n ($0 \le n < C$), let n_h denote the hth bit in the binary expansion n, so that

$$n = \sum_{h=0}^{\lfloor \lg C \rfloor} n_h 2^h \ .$$

In its simplest version, the algorithm for managing data on the disk will maintain the following properties.

1. If an object occupies a section, then all of its pages in that section are stored contiguously in the blocks of that section.
2. Let o be any disk–resident object and $n = |o|$. Then o has exactly n_h maximal sections of height $h \le \lfloor \lg C \rfloor$.
3. Suppose there are f free blocks on the disk. Then the free list contains exactly f_h maximal free sections of height h, for each $h \le \lfloor \lg C \rfloor$.

For example, suppose the disk has a capacity of 260 blocks. So the sections range in height from 0 to 8. There are two unpaired sections, one of a complete binary tree of height 8, and another of height 2. If an object o of 13 pages is resident,

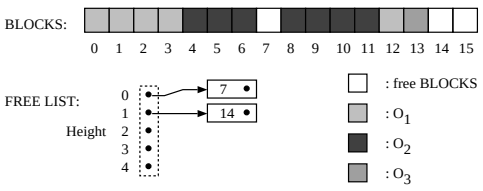

Figure 2: A simple example showing a possible layout of three objects (o_1, o_2, o_3) on a 16-block disk. Two sections (7 and 14) are on the free list.

then the properties above require that the pages of o occupy three sections: one of height 3 (with 8 blocks), one section of height 2 (with 4 blocks) and one of height 0 (with 1 block). There will be at most two breaks in the layout of o. Similarly, if the free list contains 32 free blocks, then these must all occupy a single section of height 4.

To record the state of the system (*e.g.* a directory), it suffices to record the set of disk resident objects, their maximal sections, and the sections on the free list. Since each maximal section can be specified by its starting address and its height, the disposition of a resident object of n blocks can be recorded in $O(\lg n)$ space, and the sections on the free list can be recorded in $O(\lg C)$ space. So a record of the entire system requires space at most

$$\sum_{o \in R} O(\lg |o|) + O(\lg C) \leq O(|R| \lg C) \ ,$$

where R denotes the set of objects that are currently resident. A sample layout, together with its free list, is shown in Figure 2.

2.3. **Materialization.** Materialization of an object is handled by the following algorithm. Assume that a request is made to materialize object o of n pages. Let $n = \sum_{h=0}^{\lfloor \lg C \rfloor} n_h 2^h$. Partition the blocks of o into intervals, with n_h intervals of size 2^h, for each h.

For each height $h = \lfloor \lg C \rfloor, \ldots, 0$, if $n_h > 0$, allocate a section of height h as follows.

1. If there is a section of height h on the free list, then allocate this section.
2. Otherwise, recursively allocate one section of height $h+1$. Partition this into 2 sections of height h. Enqueue one of these on the free list, and return the other.

Once all sections are allocated, each interval of o is copied contiguously into a section of the appropriate height.

Lemma 2.1. *Allocation preserves all the properties of §2.2. It requires at most $O(\lg C)$ processing time and writes exactly n disk blocks.*

Proof. If there are f free blocks and $f = \sum_h f_h 2^h$, then for each h there are f_h maximal free sections of height h on the free list. The algorithm simply mimics the algorithm for computing the difference of f and n as binary numerals. Properties 1 and 2 above follow immediately from this observation. Property 3 follows by induction on the number of allocations. □

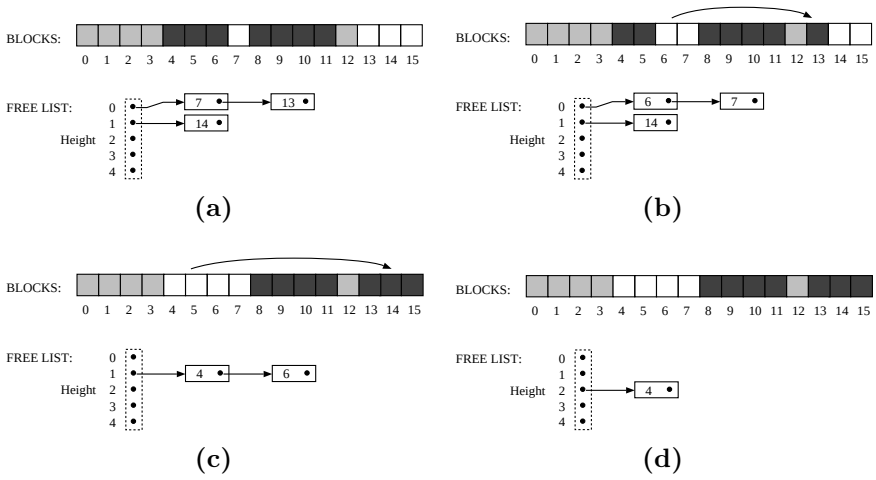

Figure 3: Deletion of an object from the layout of Figure 2.

2.4. **Deletion.** The following steps remove object o from the disk-resident set and reclaim its space. First, all sections of o are enqueued on the appropriate free lists. Then, for each height $h = 0, \ldots, \lfloor \lg C \rfloor$ the space is compacted. First, in memory, a new layout is determined using the following algorithm. While there are more than d sections on the list, these steps are repeated:

1. Choose any two sections f_1 and f_2. From these, choose one that has a buddy. Without loss of generality, assume this is f_1 with buddy b_2.
   (a) Remove f_2 from the free list.
   (b) Record that the page stored at block b_2 is to be copied to f_2.
   (c) Add b_2 to the free list.
2. Now f_1 and its buddy are both on the free list at height h. Remove them from the list, merge them, and place the resulting section of height $h + 1$ on the next free list height.

Once the new layout is determined, it is realized on the disk by copying pages directly to the final locations computed by the algorithm. Of course, if the algorithm determined that a page was to be copied multiple times — first to one block and from there to another — the copying phase merely moves it directly to its final location.

Example . Suppose object o_1 is deleted from the configuration in Figure 2. This leaves the configuration in Figure 3(a) in which there are two sections of height 0 (blocks 7 and 13) on the free list. These should be combined into a single section of height 1. To do this, we arbitrarily choose to free the buddy of block 7. Data is copied from 6 to 13, and and 6 and 7 are combined into a single section (6,7) of height 1 (Figure 3(b)). The resulting configuration now has two sections of height 1, as recorded on the updated free-list shown below. To compact these, data will be copied from (4,5) — the buddy of section (6,7) — to (14,15) (see Figure 3(c)). Sections (4,5) and (6,7) are then combined into a single section of height 2. The final configuration is shown in Figure 3(d).

Lemma 2.2. *The deletion algorithm preserves all the properties of* §*2.2.* □

The proof follows from the observation that freeing the space allocated to an object is formally equivalent to adding n to f as binary numerals and by induction on the number of deletions.

3. A "Lazy" Variant of the Algorithm

The implementation of the basic operations presented above yields a rather high cost for the deletion of objects, even in an amortized sense. In the worst case, the deletion of an object can cause a cascade of compactions involving sections of larger height. For example, in the case where each of the sublists at heights $0, \ldots, k$ has exactly one section, freeing a single block can require copying nearly 2^{k+1} additional blocks. Thus, because the copying costs associated with larger sections is also larger, the cost of a deletion cannot be bounded immediately by the size of the object deleted.

These costs can be bounded by a simple variant of the algorithm above in which the compaction of the deallocation procedure is made "lazy". In other words, once a new layout for the data on the disk has been determined after deallocation of an object, the data is not immediately reorganized. Instead, a record is kept of the changes needed to realized this new organization. As a consequence, a deletion will not initiate any disk activity and, as argued below, the additional overhead incurred by materializing an object will be proportional to its size.

To realize this, each section on the free list will be designated either *dirty* or *clean*: If no physical block of a section contains valid data (*i.e.* a page of a disk–resident object that is not stored in some other block) then the section is clean; otherwise it is dirty. Each dirty section also carries with it a list of the target blocks or sections that are to receive the valid data that its own blocks or subsections contains. The materialization and deletion procedures above are then modified as follows.

3.1. **Deletion.** Whenever sections are merged during the deallocation procedure, their respective lists are catenated and the physical movement of data to new blocks is postponed. Hence invariant properties 2 and 3 of §2.2 may in fact be violated on the disk — since there may be more than one unoccupied section of each height — yet the property is maintained in the memory–resident free list. Since the deletion procedure affects only these data structures, there is no copying of disk–resident data.

3.2. **Materialization.** When an object is materialized, the algorithm of §2 section is used, but with a minor modification. Before the object is written to disk, the valid data occupying the sections allocated to the object are first moved to the target locations recorded previously.

Theorem 3.1. *When modified as above, the storage management algorithm can dynamically maintain a disk-resident set of objects. For each object o of n blocks, the algorithm guarantees: (1) that while o is disk-resident, there are at most $\lceil \lg n \rceil$ breaks in the layout of o; and (2) that materialization of o on disk requires fewer than n additional disk reads and writes more than the algorithm of §2; (3) that deletion of o from the disk requires no additional disk activity.*

Proof. The first guarantee follows from the fact that invariant properties 1 and 2 continue to hold, both in memory and on disk. Hence each object o has at most $\lceil \lg |o| \rceil$ sections. The second point follows from the observation that the number of blocks moved during a write of an object o is strictly less than the number of

blocks contained in o itself. The last guarantee is immediate from the description of the deallocation procedure. □

Note, however, that cost incurred by adopting this lazy behavior is a larger size for the memory–resident free list.

4. Discussion

The algorithms suggested above are related to the buddy system for efficient main memory storage allocation proposed in [Kno65] and discussed in [LD91]. However, these buddy systems are required to allocate a contiguous chunk of storage for each object materialized. In the case of memory allocation, this results in wasted space and fragmentation, and requires either a re-organization process or a garbage collector. [GR93] The overall structure of the algorithms also resembles that of memory algorithms for certain dynamic data structures, like the binomial heap. [CLR91] The proposed disk organization can also be seen as a natural hierarchical extension of the 2-level partitioning of a disk (into blocks and fragments) in the Unix Fast File system of [MJLF84]. The algorithms have been implemented in the Everest storage manager, a component of an experimental continuous media server currently under development. An empirical study of Everest is currently in progress.

Finally, we believe that the lazy algorithm proposed above in §3 provides an optimal organization for bounding the number of breaks in the layout of each object when caching a working set of objects on a linear storage medium, while both minimizing the overhead required to maintain the organization as the working set changes and maximizing the utilization of the medium (the amount of data that is cached). More precisely, we conjecture that under the assumptions of §1, any storage management scheme which guarantees that the layout of every disk–resident object o has fewer than $\lceil \lg |o| \rceil - 1$ breaks, either requires more than $|o| - 1$ additional disk copies for materializing some object o, or cannot fully utilize the space of the device, from some sequence of requests.

Acknowledgements

We would like to thank the members of the Database Systems Laboratory at the University of Southern California for their comments and suggestions on this work at various stages in its development; the anonymous referees for their numerous suggestions for improving the focus and presentation of this paper; and Jeffrey Ullman for suggesting the conjecture described in §4. This research was supported in part by the National Science Foundation under grants IRI-9110522, IRI-9258362 (NYI award), CDA-9216321, CCR-9207422 and CCR-9402819, and by a gift from Hewlett-Packard.

References

[And92] D.P. Anderson, et al. A File System for Continuous Media. *ACM Trans. on Computer Systems* (10) 4. November 1992. pp. 311-337.

[Bu94] J.F.K. Buford. *Multimedia Systems*, Addison–Wesley, 1994, p. 49.

[CLR91] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1991, 400–419.

[GVK95] D. Gemmell, H.M. Vin, D.D. Kandlur, P.V. Rangan, L.A. Rowe. Multimedia Storage Servers: A Tutorial. *IEEE Computer*, 28(5):43–45, 1995.

[GC92] D. Gemmell and C. Christodoulakis. Principles of Delay–Sensitive Multimedia Data Storage and Retrieval. *ACM Trans. on Information Systems*, 10(1), 1992.

[GIZ94] S. Ghandeharizadeh, D. Ierardi and R. Zimmermann. *Management of Space in Hierarchical Storage Systems*, Technical Report USC-CS–94-598, University of Southern California, Department of Computer Science, November 1994.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993, 670–671.

[Kno65] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, 1965.

[LD91] H. R. Lewis and L. Denenberg. *Data Structures & Their Algorithms*. Harper Collins, 1991, 367–372.

[MJLF84] M. Mckusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, August 1984.

Figures appearing in the text

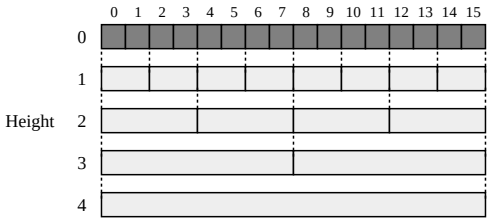

Figure 1

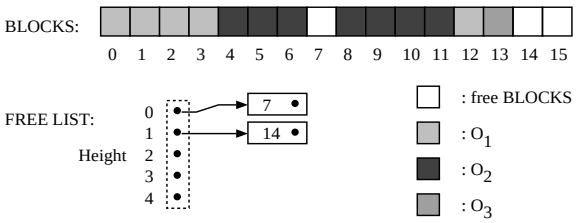
BLOCKS:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
FREE LIST:
Height
0
1
2
3
4
7
14
: free BLOCKS
: O1
: O2
: O3

FIGURE 2

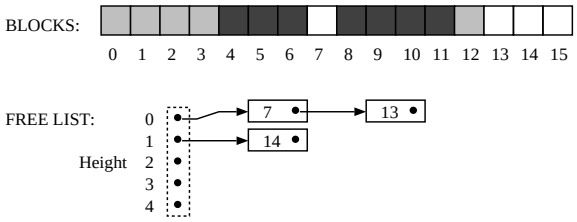
BLOCKS:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
FREE LIST:
Height
0
1
2
3
4
7
13
14

FIGURE 3(A)

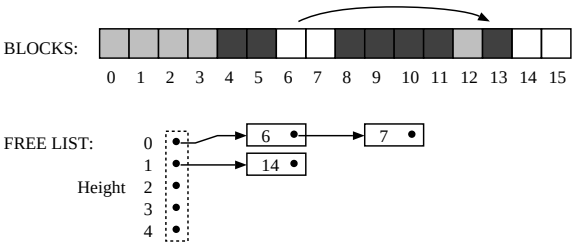
BLOCKS:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
FREE LIST:
Height
0
1
2
3
4
6
7
14

FIGURE 3(B)

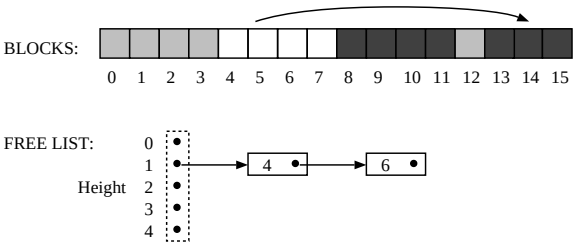

FIGURE 3(C)

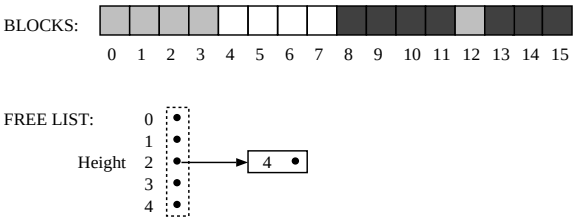

FIGURE 3(D)

Department of Computer Science, University of Southern California, Los Angeles, CA 90089-0781