FAST FITS

New methods for dynamic storage allocation

C. J. Stephenson

Thomas J. Watson Research Center
Yorktown Heights, N.Y.

A cartesian tree can be used to manage a pool of storage so that typically the allocation or release of a variable-length area takes time $O(\log M)$, where $M =$ number of discontiguous available blocks. It neither incurs the spacial penalty of 'Buddy' methods nor has the restrictions associated with tags. It can support the same programming interface that is conventionally associated with 'First Fit', without using additional storage, and with much better performance when $M$ is large.

1. Preamble

Consider a storage allocation routine which manages a 'pool' of storage, and has two entry points named (say) allocate and release, which may be called from divergent programs. Allocate is called when a program requires a piece of storage for its own use: the calling program supplies the length needed, and the storage allocation routine finds an available piece in the pool and provides its address. Release is called when a piece that was previously allocated is no longer required and can therefore be made available for re-allocation: in this case the calling program supplies both the address and the length.

Storage allocation routines of this general kind exist in most operating systems and in many other programs. This paper presents a data structure, and some new algorithms, for implementing them.
2. Background

The classical methods of dynamic storage allocation are described by Knuth [1968], and can be classified under the following two headings:

(A) *First Fit and Best Fit*

The available blocks of storage are linked together in address order. Each available block contains a two-word header, consisting of a pointer to the next available block and a length field. In the case of First Fit, storage is allocated from the first available block of sufficient length; and with Best Fit it is allocated from a block having the minimum excess length. Using either method, storage can be allocated in multiples of two words (the length of a header); and an area that is allocated as a unit can be partially released if desired (the remainder being retained for the time being). When a piece of storage is released, it is inserted in the list at the right place, or coalesced with one or both of the neighbouring blocks in the list, if it touches them. In the long run, after numerous pieces of storage of more-or-less random lengths have been allocated and released at more-or-less random times, a number of uselessly small blocks tend to develop, particularly near the beginning of the list. Although these fragments comprise a surprisingly small proportion of the storage (typically around 10 per cent), a lot of time can be wasted chaining through them, visiting blocks that have no relevance to the current allocation or release operation. In machines with a cache, the effect can be ruinous, since the number of blocks visited (during a single allocate or release operation) can exceed the total number of 'lines' in the cache.

(B) *Buddy Methods*

Here the task of managing the pool of storage is reduced in size by structuring the way in which the pool can be divided up, e.g. into blocks of size $2^k$ words ($k = 2, 3, ...$), and limiting the number of different lengths that can be allocated. Both the allocated and available blocks are tagged in storage, and the latter are kept in
doubly-linked lists of equal-sized blocks which are anchored from an array in which the index can be computed from the length. Consequently an available block, of some required size, can usually be found with very little work. (If the most appropriate list is empty, a bigger block is split up.) Similarly, when a piece of storage is released, it is possible to compute the whereabouts of its primary neighbour (or 'buddy'), and to coalesce the two blocks if they are both now available. (This coalescence is repeated until the buddy of the coalesced block is not available.) These tactics eliminate the problem of chaining through long lists of uselessly small blocks, but there are the following disadvantages. (a) It is impossible to support partial release in any useful way. (b) Space is wasted in rounding up the length requested to an allowable size, and typically 30 to 40 per cent more storage is required to satisfy the same allocations than when using First Fit or Best Fit.

There are a number of algorithmic variations. For example, the performance of First Fit and Best Fit can be improved by maintaining 'sub-pools' for blocks of popular sizes, when these are known in advance (see Margolin et al. [1971], Bozeman et al. [1982]). This is effective when there is a single application to be supported. For open-ended situations, Knuth [1968] suggests keeping the available blocks in a doubly-linked list, chained in a more-or-less random order, with tags at the beginning and end of the allocated pieces (as well as the available blocks) which are inspected to discover whether a piece being released has neighbours with which it can be coalesced. Unfortunately this precludes partial release; and also (as it turns out) it tends to use a lot of storage, partly because the tags themselves occupy space, but mostly because small allocated pieces become widely distributed (rather than being packed near the beginning of the pool) and fragment the bigger available blocks (see Shore [1975], Page [1982]).

There have been some ingenious attempts at improving the storage utilization of Buddy Methods, by allowing more variation in the allowable sizes (not just powers of 2). Unfortunately they have been strangely self-defeating, since the saving in 'internal' fragmentation (due to rounding up the length requested) turns out to be almost exactly offset by a corresponding increase in 'external' fragmentation (due to unused available blocks): see Peterson and Norman [1977].
Apart from First Fit and Best Fit, most existing methods preclude partial release of an area that has been allocated as one unit. Other things being equal, the ability to perform partial release is a desirable feature. Here are two examples of its use:

(a) Imagine a program that can compute an upper bound for the size of a table to be constructed but does not know the actual size until the table has been constructed. Given partial release, the program can obtain a piece of storage that is more than adequate and subsequently release the unused part.

(b) In a system that uses paging, a program that is conscious of its working set can acquire several logically separable pieces of storage with a single call to allocate. This ensures their physical contiguity initially, without preventing the program from re-allocating one of the pieces later, should this be necessary.

3. Motivation

Most performance studies of storage allocation have been concerned with the 'steady state' in which there is a continual and more-or-less random succession of calls to allocate and release. As it happens, the ideas presented here were motivated by examining a simpler scenario.

Consider a pool of storage that is managed by a straightforward First Fit or Best Fit routine, and initially consists of a single available block. Suppose that N pieces are allocated from it, one by one (by calling allocate); and that subsequently they are individually released (by calling release). In this situation allocation is cheap, since there is no opportunity for the pool to become fragmented, and there is only one
(shrinking) available block. Now examine what happens during the 'release' phase. If the order of release is the same as that of allocation (or is the opposite), then each piece that is released (except the first) is immediately coalesced with the previously released piece, and the number of available blocks never exceeds 1 or 2. If however the pieces are released in random order (or otherwise scrambled), the storage pool soon starts to resemble a mosaic of available and unavailable chunks, and on average it will be necessary to chain through about half the available blocks in order to insert a piece being released in the right place, or to coalesce it with its neighbours: see Fig. 1 (a). Patterns of this kind can arise in practice when releasing pieces of storage that have been used for sorting, or chained from hash tables.

(Knuth [1968] points out that the need to chain through the available blocks, in order to insert a piece of storage that is being released, can be avoided by maintaining a doubly-linked list, and keeping tags at the beginning and end of each allocated piece. This has several disadvantages, however (see previous section), and the problem as described is therefore of practical interest.)

The average and worst-case results for this scenario are derived in Appendix A. When $N$ is large, and the pieces are released in random order, the average number of available blocks during the 'release' phase has an expected value of $N/6 + O(1)$, and a worst-case maximum of $N/4$. The total number of block-visits to release all $N$ pieces has an expected value of $N^2/12 + O(N)$, and a maximum of $N^2/4$.

The fact that the work required to release $N$ pieces in random order increases with $N^2$ can lead to some bizarre performance problems. Imagine the writer of a virtual storage text editor who implements a 'sort' request, carefully using an $N \log N$ algorithm, and who then discovers that editor termination takes time $O(N^2)$; or a programmer who sets up a hashed symbol table that takes longer to destroy than to create!

Given this problem, it is worth considering a data structure for the available blocks which permits random-order insertions to be made without chaining through a linear list.
Fig. 1 (a). Available blocks of storage chained together in linear list (as for First Fit or Best Fit). Storage address increases to the right.

Fig. 1 (b). The same available blocks of storage chained as binary search tree, ordered by address.

Fig. 1 (c). The same available blocks of storage chained as cartesian tree in which x-coordinate represents address and y-coordinate represents length of block.
One possibility is to link together the available blocks in a binary search tree, ordered by address: see Fig. 1 (b). As with First Fit and Best Fit, the pointers and length fields can be stored in the available blocks themselves, so that no additional storage is required (other than an anchor). Now, when a piece of storage is released, it can be inserted into the tree, or coalesced with neighbours (if any), using standard 'search', 'insert' and 'delete' algorithms (see for example Knuth [1973], chapter 6, algorithms T and D). If the tree maintains good balance, the number of nodes that must be visited is approximately \( \log_2 M \), where \( M \) = number of available blocks. Using the results of Appendix A (summarized above), the average value of \( M \) when releasing \( N \) pieces of storage in random order is expected to be about \( N/6 \) (for \( N \gg 1 \)), and the expected number of nodes that have to be visited to release all \( N \) pieces in random order is therefore somewhat less than \( N \log_2 N \).

(Note that the block headers now require three words, two tree-pointers and a length field. If the two-word granularity of First Fit and Best Fit is to be preserved, the low-order bit of (say) the right-pointer may be used as a flag to indicate that this is a two-word node and the length field is therefore missing: this is always possible, even in a word-addressed machine, since the two-word granularity already constrains blocks to an even address (or possibly an odd one, depending where the pool starts). A more elegant alternative will be presented later, in section 9. For the time being I shall disregard this restriction, and use three-word headers:)

This structure provides a solution to the performance problem outlined above, since the expected time required for random-order release of \( N \) pieces is reduced from \( O(N^2) \) to \( O(N \log N) \). However it is not much help in the general case, where calls to allocate and release are intermixed. Traversing the tree, to find a block which is big enough to satisfy an allocation request (or which offers the best possible fit) is even more costly than for a simple linear list: furthermore it involves using a stack, or temporarily modifying many of the nodes in the tree, which can be particularly expensive in paging systems.

Given these difficulties, it is tempting to see if the tree can be refined so that it also assists in allocation.
4. The cartesian tree

In a binary search tree, the *vertical* relationship between nodes is not significant, and is usually determined incidentally by the order in which the nodes are inserted, and by the side-effects of other operations on the tree. It is however possible to construct a tree in which (a) the horizontal relationships adhere to the rules of a binary search tree, and also (b) the vertical relationships are significant.

This structure is described by Vuillemin [1980], who calls it a 'cartesian' tree, since the horizontal and vertical relationships can be used to represent independent quantities, say $x$ and $y$.

Insertion of a node into such a tree involves a 'passive' binary search, from the root, until the required level in the tree is reached, followed by insertion at the root of the remaining subtree. (For a description of root insertion, see Stephenson [1980].) Deletion of a node involves merging the inner edges of the pendant subtree.

In the case of storage allocation, it turns out that we get nice properties if we use $x$ for address of node and $y$ for length of node, and construct a tree in which for any node $S$:

(i) address of descendants on left (if any) $< \text{ address of } S < \text{ address of descendants on right (if any)}$;

(ii) length of descendants on left (if any) $\leq \text{ length of } S \geq \text{ length of descendants on right (if any)}$.

An example is shown in Fig. 1 (c). See how uselessly small blocks are pushed to the lower levels of the tree.
The essential algorithmic tools for maintaining this tree are *insert* and *delete*, which are described formally in section 6. The *promote* and *demote* algorithms are also useful, though they are both functionally equivalent to *delete* followed by *insert*. These algorithms all involve $O(D)$ or fewer node-visits ($D =$ maximum depth of tree).

Provided the tree does not become grossly unbalanced, this structure preserves the ‘$N \log N$’ execution time when releasing $N$ pieces of storage in random order (as described in the previous section for an ordinary binary search tree). It also lends itself to speedy allocation policies, since a node of adequate length can be found without inspecting any nodes of inadequate length.

It should be noted, however, that a cartesian tree does not necessarily maintain good balance, and in general it cannot be balanced explicitly, since (unless there are duplicate values) the position of each node, relative to the other nodes, is determined uniquely by its $x$ and $y$ coordinates — in this application the address and length of the nodes. Whether the structure will in practice remain well balanced therefore depends upon the pattern of calls to *allocate* and *release*, together with the allocation policy that is used.

(In section 6.2.3, Knuth [1973] gives an exercise, attributed to E. McCreight, in which a storage allocator maintains the available blocks in a balanced tree with an extra field in each node containing the maximum node length in the left-hand subtree. As far as I know, this method has not been used in practice, and I have not studied it in detail. Compared with the methods presented here, it would exhibit better worst-case performance, since the tree is balanced; but it requires longer nodes and therefore forces a larger granularity for the units of allocation.)

5. **Programming notation**

I shall use the following programming notations.

1. The static pointer variable ‘anchor’ contains the address of the tree root, or ‘null’ if the tree is null.
2. Each node in the tree contains a header, with three fields:
   - `left`, `right` pointers to sons (null if son does not exist)
   - `len` length of node

3. If 'a' is an address, then 'O(a)' refers to the contents stored at this address. If 'p' is the address of a node, then 'name(p)' refers to the contents of the 'name' field in the node.

4. The function 'addr of' yields the storage address of its argument.

5. The function 'weight of x', where x is a pointer variable which either contains the address of a node or has the value 'null', yields an integer value defined thus:

   \[ \text{if } x = \text{null} \text{ then 0 else len(x)} \]

6. Addresses and lengths are treated as unsigned integers with values \( \geq 0 \). They are assumed to occupy one machine word each.

7. Arguments are passed by value.

8. Information of a declarative nature is given in English.

6. Algorithmic tools

   Here are the basic algorithmic tools for maintaining a cartesian tree for storage allocation.

   **Insert**

   This routine inserts a new node in a cartesian tree or subtree, placing it in the correct position, horizontally and vertically, with respect to the existing nodes.
Call is:

\[ \text{insert } (a,b,p) \]

where:

- \( a, b \) contain address and length of new node (\( a \neq \text{null}, b > 0 \))
- \( p \) contains address of pointer to root of cartesian tree or subtree (\( O(p) \) may be null)

The algorithm may be described informally as follows. Starting from the root, a passive binary search is made for the new node. If this search was allowed to continue, it would eventually fail (since there cannot already be a node at the given address); but in fact it stops when it reaches a node in the tree which has a length less than that of the new node (or when it reaches a null tree pointer). The new node is then inserted at the root of the subtree for which the shorter node forms the old root (or in place of the null pointer).

Here is the algorithm. The pointer variables 'x', 'left_hook' and 'right_hook' are used as scratch variables.

\[
x := 0(p); \quad \text{\{addr of root\}}
\]

\[
\text{while weight of } x \geq b \text{ do} \quad \text{\{* see note below *\}}
\]

\[
\begin{align*}
\text{begin} \\
\quad \text{if } a < x \text{ then} \\
\quad \quad p := \text{addr of left}(x) \quad \text{\{descend to the left...\}} \\
\quad \text{else} \\
\quad \quad p := \text{addr of right}(x); \quad \text{...or to the right} \\
\quad x := 0(p) \\
\text{end;}
\end{align*}
\]

\[
0(p) := a; \quad \text{\{attach new node at 0(p)\}}
\]

\[
\text{left_hook} := \text{addr of left}(a); \quad \text{\{initialize the hooks\}}
\]
right_hook := addr of right(a);
len(a) := b; \hspace{1cm} \text{\{set length field\}}

\textbf{while} x \neq \text{null} \textbf{do} \hspace{1cm} \text{\{perform root-insertion\}}
  \hspace{1cm} \textbf{if} x < a \textbf{then}
    \hspace{1cm} \begin{align*}
      &\text{O(left_hook)} := x; \\
      &\text{left_hook} := \text{addr of right(x)}; \\
      &x := \text{right(x)}
    \end{align*}
  \hspace{1cm} \textbf{end}
  \hspace{1cm} \textbf{else}
    \hspace{1cm} \begin{align*}
      &\text{O(right_hook)} := x; \\
      &\text{right_hook} := \text{addr of left(x)}; \\
      &x := \text{left(x)}
    \end{align*}
  \hspace{1cm} \textbf{end;}

\text{O(left_hook)} := \text{null}; \hspace{1cm} \text{\{clear remaining hooks\}}
\text{O(right_hook)} := \text{null}

Notes:

1. In the algorithm as given, the new node is inserted as low as possible in the tree. (This entails the least disturbance to the tree.) To insert it as high as possible, which would be equally valid, the operator in the first 'while' loop condition may be changed from '≥' to '>'. The results will differ only if the node that is inserted has the same length as one or more of the nodes already in the branch of the tree that is followed.

2. In the root insertion sequence as given, an assignment is made to the field addressed by 'left_hook' or 'right_hook' each time around the loop, and to both fields on exit from the loop, irrespective of whether they already contain the required values. On average, about half these assignments are unnecessary, and can be avoided, if desired, by making a small change to the program.

3. The nodes visited are the same ones that would be visited if the same given node was attached as a leaf to the same binary search tree, without regard to vertical ordering, though in general more of the nodes are modified.
Delete

This routine deletes a node from a cartesian tree.

Call is:

    delete (p)

where:

    p contains address of pointer to node which is to be deleted (0(p) ≠ null)

The algorithm may be described informally as follows. The left and right sons of the node to be deleted define two subtrees which are to be merged and attached in place of the deleted node. Each node on the inside edges of these two subtrees is examined and the longer nodes are placed above the shorter ones.

Here is the algorithm. The pointer variables 'x', 'left_branch' and 'right_branch' are used as scratch variables.

\[
\begin{align*}
    x & := 0(p); & \quad \text{[addr of node to be deleted]} \\
    \text{left\_branch} & := \text{left}(x); & \quad \text{[initialize the branch ptrs]} \\
    \text{right\_branch} & := \text{right}(x); \\

    \text{while } \text{left\_branch} \neq \text{right\_branch} \text{ do } \{\text{until both are null}\} & \\
    & \quad \text{if weight of left\_branch} \geq \text{weight of right\_branch} \text{ then} & \\
    & & \quad \quad \text{begin} \\
    & & \quad \quad \quad 0(p) := \text{left\_branch}; & \quad \text{[promote the left branch]} \\
    & & \quad \quad \quad \text{p} := \text{addr of right(left\_branch)}; & \\
    & & \quad \quad \quad \text{left\_branch} := \text{right(left\_branch)} & \\
    & & \quad \quad \text{end} & \\
    & \quad \text{else} & \\
    & & \quad \quad \text{begin} & \\
    & & \quad \quad \quad 0(p) := \text{right\_branch}; & \quad \text{[promote the right branch]} \\
    & & \quad \quad \quad \text{p} := \text{addr of left(right\_branch)}; & \\
    & & \quad \quad \quad \text{right\_branch} := \text{left(right\_branch)} & \\
    & & \quad \quad \text{end}; & \\
    & \quad \text{end;} & \\
    & \quad 0(p) := \text{null} & \\
\end{align*}
\]
Notes:

1. In the algorithm as given, the nodes in the left subtree are arbitrarily placed above those in the right subtree when they are of equal length. It would be possible to do the opposite by making a small change to the program.

2. The nodes visited are those on the inside edges of the two subtrees defined by the deleted node, together with the deleted node itself and the node originally containing the pointer to it.

**Promote**

This routine promotes a node in a cartesian tree, if necessary, to establish the required vertical ordering. All the nodes in the tree, including the one to be promoted, must be correctly ordered horizontally; and all the nodes except the one to be promoted must also be correctly positioned vertically. The node to be promoted may already be correctly positioned vertically, or it may have a length exceeding that of its father.

The need for promotion occurs when (for example) the length of a node is increased above that its father as a result of being coalesced with a piece of storage that is released. Promotion is functionally equivalent to deletion followed by insertion, but generally involves visiting fewer nodes.

Call is:

promote (a,p)

where:

- a contains address of node to be promoted (a ≠ null)
- p contains address of pointer to root of tree or subtree containing the node (0(p) ≠ null)

The algorithm may be described informally as follows. Starting from the root, a passive binary search is made for the node which is to be promoted. If this search was allowed to continue, it would eventually find
the given node; but in fact it stops when it reaches a node in the tree which has a length less than that of the node to be promoted. The given node is then promoted to the root of the subtree for which the shorter node forms the old root.

Here is the algorithm. The pointer variables 'x', 'left_branch', 'right_branch', 'left_hook' and 'right_hook' are used as scratch variables.

\[
x := 0(p); \quad \{\text{addr of root}\}
\]

\[\text{while weight of } x \geq \text{ weight of } a \text{ do}\]
\[\text{begin}\]
\[\quad \text{if } a < x \text{ then}\]
\[\quad \quad p := \text{addr of left}(x); \quad \{\text{descend to the left...}\}
\]
\[\quad \text{else}\]
\[\quad \quad p := \text{addr of right}(x); \quad \{\ldots \text{ or to the right}\}
\]
\[\quad x := 0(p)
\]
\[\text{end;}
\]
\[0(p) := a; \quad \{\text{attach given node at } 0(p)\}\]
\[\text{left_branch} := \text{left}(a); \quad \{\text{remember its old sons}\}
\]
\[\text{right_branch} := \text{right}(a);
\]
\[\text{left_hook} := \text{addr of left}(a); \quad \{\text{initialize the hooks}\}
\]
\[\text{right_hook} := \text{addr of right}(a);
\]
\[\text{while } x \neq a \text{ do}\]
\[\quad \text{if } x < a \text{ then}\]
\[\quad \quad 0(\text{left_hook}) := x;
\quad \quad \text{left_hook} := \text{addr of right}(x);
\quad \quad x := \text{right}(x)
\]
\[\quad \text{end}
\]
\[\quad \text{else}\]
\[\quad \quad 0(\text{right_hook}) := x;
\quad \quad \text{right_hook} := \text{addr of left}(x);
\quad \quad x := \text{left}(x)
\]
\[\quad \text{end;}
\]
\[0(\text{left_hook}) := \text{left_branch}; \quad \{\text{set remaining hooks}\}
\]
\[0(\text{right_hook}) := \text{right_branch}
\]
Notes:

1. In the algorithm as given, the new node is promoted as little as possible in the tree (cf. note 1 under \textit{insert}).

2. In the second 'while' loop as given, an assignment is made to the field addressed by 'left\_hook' or 'right\_hook' each time around the loop (cf. note 2 under \textit{insert}).

3. The nodes visited are those that lie on the path from the given root to the given node (including both of these).

\textbf{Demote}

This routine demotes a node in a cartesian tree, if necessary, to establish the required vertical ordering. All the nodes in the tree, including the one to be demoted, must be correctly ordered horizontally; and all the nodes except the one to be demoted must also be correctly positioned vertically. The node to be demoted may already be correctly positioned vertically, or it may have a length which is less than that of one or both of its sons.

The need for demotion occurs when (for example) the length of a node is decreased below that of one or both of its sons as a result of allocating a piece of storage from it. Demotion is functionally equivalent to deletion followed by insertion, but generally involves visiting fewer nodes.

Call is:

\begin{verbatim}
demote (p)
\end{verbatim}

where:

\begin{verbatim}
p contains address of pointer to node which is to be demoted (O(p) \neq null)
\end{verbatim}

The algorithm may be described informally as follows. The left and right sons of the node to be demoted define two subtrees which are to be partially merged and attached in place of the demoted node. The nodes on the inside edges of these two subtrees are examined and the
longer nodes are placed above the shorter ones, until a node is reached which has a length no greater than that of the node being demoted (or until a null pointer is reached). The node being demoted is then attached at this point, and the remaining subtrees (if any) become its sons.

Here is the algorithm. The pointer variables 'x', 'left_branch' and 'right_branch' are used as scratch variables.

```
x := 0(p);                         \{addr of node to be demoted\}
left_branch := left(x);            \{initialize the branch ptrs\}
right_branch := right(x);

while (weight of left_branch > weight of x) or
       (weight of right_branch > weight of x) do
  if weight of left_branch ≥ weight of right_branch then
    begin
      0(p) := left_branch; \{promote the left branch\}
p := addr of right(left_branch);
left_branch := 0(p)
    end
  else
    begin
      0(p) := right_branch; \{promote the right branch\}
p := addr of left(right_branch);
right_branch := 0(p)
    end;

0(p) := x; \{attach demoted node here\}
left(x) := left_branch;
right(x) := right_branch
```

Notes:

1. In the algorithm as given, the node is demoted as little as possible in the tree (cf. note 1 under insert). Also, the nodes on the left branch are arbitrarily promoted above those on the right branch when they are of equal length (cf. note 1 under delete).

2. The nodes visited are those that are placed between the old and new positions of the demoted node, together with the demoted node itself and the node previously containing the pointer to it.
General comment on the algorithms

These algorithms are not special to storage allocation, and can easily be generalized to situations in which the x and y coordinates are used for properties other than address and length.

7. Procedures for allocation and release

I will postpone discussing the allocation policy (i.e. the choice of a node from which to allocate a piece of storage). In the meantime, here are procedures which, given a node from which allocation is to be made, or given a piece of storage to be released, make the necessary changes in the tree.

Allocation

Suppose the pointer variable 'p' contains the address of the pointer to a node from which a piece of storage, of length 'b', is to be allocated (0 < b ≤ length of node). If the selected node is the root, p contains the address of 'anchor'. The following sequence sets the pointer variable 'a' to the address of the allocated piece, and removes the piece (a,b) from the tree. It is assumed that the length of the node, and the length to be allocated, are both multiples of k, where k ≥ length of node header. The pointer variable 'x' is used as a scratch variable.

```plaintext
a := 0(p);          {* see note below *}
if len(a) = b then
  delete (p)        {delete node if nothing left}
else
  begin
    x := a + b;      {create new node header here}
    left(x) := left(a);
    right(x) := right(a);
    len(x) := len(a) - b;
    0(p) := x;       {chain new node to father}
    demote (p)       {demote if necessary}
  end
```
Note:

When the length of the node exceeds the length to be allocated, this procedure allocates from the beginning of the block and creates a new node header following the piece allocated. It would be equally valid (and in fact somewhat simpler) to allocate from the end of the block, but see section 8 below.

Release

Suppose the pointer variable ‘a’ contains the address of a piece of storage, of length ‘b’, which is to be released. The following sequence searches for neighbours (existing available blocks that can be coalesced with the piece being released) which it deletes from the tree and combines with the given piece. It then inserts the piece, as a new node, at the appropriate place in the tree. For simplicity it is assumed that the given values are valid: for example, the given address and length must be aligned appropriately (a = u + nk, and b = nk, where u = address of storage pool and k ≥ length of node-header), and the piece being released must not overlap any existing nodes. The pointer variables ‘p’ and ‘x’ are used as scratch variables.

```
p := addr of anchor;    {addr of ptr to root}
while 0(p) ≠ null do    {search for neighbours}
  begin
    x := 0(p);         {step down a level}
    if x + len(x) = a then
      begin
        a := x;         {absorb left neighbour}
        b := len(x) + b;
        delete (p)      {delete left neighbour}
      end
    else
      if a + b = x then
        begin
          b := b + len(x); {absorb right neighbour}
          delete (p)      {delete right neighbour}
        end
  end
```
else
    if a < x then  \{keep searching\}
        p := addr of left(x)
    else
        p := addr of right(x)
end;

    p := addr of anchor;  \{addr of ptr to root\}
    insert (a,b,p)  \{insert new node (a,b)\}

Note:

The procedure as given is straightforward but involves two complete descents through the tree (to search for neighbours, and to insert the new node), and zero, one or two partial descents (to delete neighbours, if any). It is possible, with some extra program complexity, to reduce the amount of work done by (a) combining the search for neighbours with a provisional insertion of the new node (which is attached at the highest legal point during this initial descent), and then (b) promoting the new node only if its final length exceeds that of its father. It is observed that in practice the second step is not usually required.

8. Allocation policies

It can sometimes happen that the caller of allocate requires the longest piece of storage available, or requires a piece that is longer than any of the available nodes. (There are in fact situations where this is quite common.) This is one of the nastier situations for First Fit and Best Fit, since the entire list must be searched. (It helps only slightly to remember the whereabouts of the longest available node, since this information must be invalidated, or updated, whenever the longest node is selected for allocation.) The cartesian tree handles this situation nicely, since the longest node is always at the root, and its length is therefore available immediately.

Now consider the case in which the length required does not exceed that of the root. How should a node be selected for allocation? Clearly, it would always be possible to select the root itself (and demote what remains, if necessary), but in general this would needlessly fragment the
larger chunks of storage. There are two policies which are more attrac-
tive than this, and which I have dubbed Leftmost Fit and Better Fit, as
follows.

Leftmost Fit

This method selects the leftmost node of sufficient length. For a
given sequence of calls to allocate and release, the pieces allocated are
identical to those that would be allocated if the same pool of storage
was managed by First Fit, i.e. they have the same addresses. (For this
to be true, it is necessary to allocate from the beginning of the selected
block: see note under Allocation in section 7 above.) Consequently the
fragmentation properties are also identical, and are known to be quite
good (see for example Shore [1977]). Since, however, allocation is
always made from the left-hand side of the tree (or in the limiting case
from the root), the tree will tend to become unbalanced, and it is not
obvious whether the performance will be any better than that of First
Fit.

Leftmost Fit is simple to define. Let ‘b’ contain the required
length, and assume 0 < b ≤ weight of root. The following sequence sets
the pointer variable ‘a’ to the address of the leftmost node of sufficient
length, and the pointer variable ‘p’ to the address of the pointer to this
node. (If ‘a’ points to the root, ‘p’ points to ‘anchor’. ) The pointer
variable ‘left_son’ is used as a scratch variable.

p := addr of anchor;
a := 0(p);
left_son := left(a);

while weight of left_son ≥ b do
begin
    p := a;
a := left_son;
    left_son := left(a)
end
**Better Fit**

This method selects a node by descending the tree, from the root, so that at each decision point the better fitting son is chosen (i.e. the shorter one, provided it is long enough, or the longer one, otherwise). The descent stops when both sons are too short. This policy has some attractive properties:

(a) The larger nodes remain intact until they are needed, irrespective of their address.

(b) There is less tendency (than with Leftmost Fit) for the tree to become unbalanced.

(c) The policy is 'stable', in the following sense. Allocation of a piece of storage from some node, say \( S \), will result in \( S \) becoming shorter, which (in the absence of other changes) will result in its being selected again for the next allocation of the same length, assuming it is still big enough. If it ceases to be big enough, allocation will switch to a node which is near \( S \) in the tree, and is therefore likely to be near it in storage. This suggests that consecutively allocated pieces will often have reasonably good proximity, which in turn suggests desirable overall working set characteristics.

Here is the formal description. As before, let \( 'b' \) contain the required length, and assume \( 0 < b \leq \text{weight of root} \). The following sequence sets the pointer variable \( 'a' \) to the address of the selected node, and the pointer variable \( 'p' \) to the address of the pointer to this node. The pointer variables \( 'left\_son' \) and \( 'right\_son' \) are used as scratch variables.

\[
p := \text{addr of anchor};
\]

\[
a := 0(p);
\]

\[
\text{left\_son} := \text{left}(a);
\]

\[
\text{right\_son} := \text{right}(a);
\]

\[
\text{while (weight of left\_son} \geq b) \text{ or (weight of right\_son} \geq b) \text{ do begin}
\]

\[
p := a;
\]

\[
\text{if weight of left\_son} \leq \text{weight of right\_son then}
\]
if weight of left_son ≥ b then
  a := left_son
else
  a := right_son
else
  if weight of right_son ≥ b then
    a := right_son
  else
    a := left_son;
    left_son := left(a);
    right_son := right(a)
end

Note:
In the sequence as given, the left son is arbitrarily selected in preference to the right son when they are of equal length.

9. A structural refinement

In the algorithms and procedures shown above, I have used three fields in each node, the left pointer, the right pointer and the length of the node; and I have disregarded the problem of granularity by assuming that the length to be allocated (or released) is always a multiple of k, where k ≥ length of node header. In section 3 above, I mentioned that a granularity of two words can be achieved by using one flag bit per node. There is however a more elegant and efficient technique, in which the length of a node is stored in the same place as its address (i.e. in the father), and which takes advantage of the fact that in the cartesian tree any descendants of a short node are also short. It uses a node header laid out thus:

```
left pointer  right pointer  left weight  right weight
```

The first two words contain the address of the left and right son (or are null), exactly as before. The second two words, if present, contain the corresponding 'weight'. They are present only if the node containing
them exceeds two words in length; otherwise the weight in words of (for example) the left son is:

\[
\text{if } \text{left\_ptr} = \text{null} \text{ then 0 else 2}
\]

The length of the root must now be stored separately (e.g. in the same place as its address).

This node layout has the incidental advantage that it enables some procedures to be executed while visiting fewer nodes. For example, when descending the tree to select a node according to Better Fit, it is unnecessary to visit a node's sons in order to select the better fitting one, and therefore the inadequate and less well-fitting nodes are not visited at all.

On the other hand, it makes the programming somewhat more complicated. For example, when allocating a piece of length \( b \) words from a node of length \( b+2 \) words, the remaining node, though short, may temporarily have long sons (until it has been demoted). The process of splitting a node, and demoting it, must therefore be combined in the program. The details are not given here.

10. Performance measurements

Comparing methods of storage allocation is not completely straightforward. Their behaviour depends upon the sequence of calls to \texttt{allocate and release}; and for a given sequence it depends to some extent on the details of implementation.

Instrumented routines, which keep track of the nodes visited, the nodes changed, and the maximum length of occupied storage, were written for the following methods:

- First Fit (as summarized in section 2)
- Best Fit (as summarized in section 2)
- Leftmost Fit (as summarized in sections 6, 7, 8)
- Better Fit (as summarized in sections 6, 7, 8)
- Buddy (as summarized in section 2)
The first four methods are strictly comparable, in the sense that each supports exactly the same operations (including partial release with a granularity of two words). The last one (Buddy) does not support partial release.

In each case, the implementation provides a flexible boundary at the high-address end of the pool. When necessary to satisfy an allocation, the boundary is moved forward into the 'wilderness' (unused area following the pool); or it retreats when the piece adjoining the wilderness is released. This provides a clear-cut way of defining how much storage is occupied at any time. It is also realistic, since it permits there to be two allocation routines (e.g. one for control blocks and the other for paging) which compete for storage by taking from opposite ends of the wilderness, without predefining their maximum shares.

The following quantities were measured:

- **Nodes visited**: Average number of distinct node headers the contents of which are referred to (or changed) during each call to *allocate* and *release*. This is a rough measure of the working set size.

- **Nodes changed**: Average number of distinct node headers the contents of which are changed during each call to *allocate* and *release*. This is a rough measure of the change set size.

- **Maximum storage occupied**: Maximum length of storage used (i.e. maximum encroachment into the wilderness). This quantity, divided by the maximum sum of the instantaneously allocated lengths, is a measure of the the amount of storage that is actually required (to satisfy the allocations) compared with the amount that would be required if the allocated pieces could at any time be packed down so that they were contiguous.

Estimates of working set size and change set size that are derived from the average number of nodes visited and nodes changed are more useful for evaluating the performance of a cache, where a slot is typically
comparable in length to a small node in a list or tree, than for a paging system, where there may be several or many nodes in a page.

11. Simulations

Pieces of storage, of various lengths, were continually allocated and released until a reasonably steady state was achieved (see below). The lengths were randomly selected, in units of words, from a skewed distribution shown in Fig. 2 (see below). A piece was allocated after each unit of time, and assigned a 'lifetime' that was independent of its length and randomly selected from a truncated exponential distribution. The average length and lifetime varied from run to run.

At the beginning of a run there were no pieces allocated, and the wilderness occupied all the available storage. As pieces started to be allocated, the occupied space grew in size and the wilderness retreated. At the end of the run, the remaining pieces were released and a check was made to ensure that the wilderness had regained all its original ground.

Each run consisted of 500,000 allocations. The results are shown in Table I. The reported average number of nodes visited and nodes changed are based on measurements made during the second half of the run. The reported maximum storage occupied is the maximum for the entire run.

For most of the trials, a run length of 500,000 allocations appears to be more than sufficient (a) for the data structure to reach a limiting size and shape, and (b) for the results to be little affected by short term random fluctuations. This opinion is based upon the observation that if only the first half of the runs are used, the results are generally within about 1 per cent of those shown in Table I. The runs for Better Fit are a possible exception: this method seems to be less stable than the others, and in my experiments it displayed apparently random long term fluctuations of up to 6 per cent. It is not entirely clear whether Better Fit achieves a well-defined steady state.

The skewed length distribution was used in preference to a simpler uniform or exponential distribution for reasons of verisimilitude, since in
### Table I. Steady-state results for allocating and releasing pieces of storage with randomly chosen lengths and lifetimes. The number of pieces allocated is the theoretical average number (i.e. the ratio of average lifetime to inter-arrival time). Under 'Nodes visited' and 'Nodes changed', the two columns contain the approximate average number of distinct nodes visited or changed in allocate (A) and release (R). Under 'Maximum storage occupied' is given the maximum length of storage occupied during the run divided by the maximum sum of the lengths that were instantaneously allocated.

**Average allocated length = 10 words**

<table>
<thead>
<tr>
<th>Method</th>
<th>100 pieces allocated</th>
<th>1000 pieces allocated</th>
<th>10,000 pieces allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes visited (A)</td>
<td>Nodes visited (R)</td>
<td>Maximum storage occupied</td>
</tr>
<tr>
<td>First Fit</td>
<td>13 13 1.4 1.7 1.18</td>
<td>71 71 1.3 1.8 1.14</td>
<td>461 462 1.2 1.8 1.10</td>
</tr>
<tr>
<td>Best Fit</td>
<td>13 9 1.3 1.7 1.14</td>
<td>36 27 1.1 1.8 1.08</td>
<td>90 72 1.0 1.9 1.07</td>
</tr>
<tr>
<td>Leftmost Fit</td>
<td>6 8 2.5 3.7 1.18</td>
<td>13 16 3.1 4.9 1.14</td>
<td>25 29 3.5 6.0 1.10</td>
</tr>
<tr>
<td>Better Fit</td>
<td>5 6 2.0 3.1 1.16</td>
<td>8 10 2.5 4.0 1.11</td>
<td>11 15 2.8 5.0 1.12</td>
</tr>
<tr>
<td>Buddy</td>
<td>2 3 1.9 2.0 1.61</td>
<td>2 3 2.0 2.0 1.55</td>
<td>2 3 2.0 2.0 1.54</td>
</tr>
</tbody>
</table>

**Average allocated length = 100 words**

<table>
<thead>
<tr>
<th>Method</th>
<th>100 pieces allocated</th>
<th>1000 pieces allocated</th>
<th>10,000 pieces allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes visited (A)</td>
<td>Nodes visited (R)</td>
<td>Maximum storage occupied</td>
</tr>
<tr>
<td>First Fit</td>
<td>24 24 1.9 1.7 1.18</td>
<td>209 209 1.9 1.7 1.12</td>
<td>1855 1854 1.8 1.7 1.08</td>
</tr>
<tr>
<td>Best Fit</td>
<td>37 21 1.8 1.7 1.11</td>
<td>221 131 1.6 1.8 1.07</td>
<td>1114 712 1.4 1.8 1.03</td>
</tr>
<tr>
<td>Leftmost Fit</td>
<td>8 9 3.6 4.4 1.18</td>
<td>16 17 5.2 6.2 1.12</td>
<td>26 28 6.6 7.8 1.08</td>
</tr>
<tr>
<td>Better Fit</td>
<td>6 8 3.1 3.8 1.14</td>
<td>10 12 4.3 5.3 1.10</td>
<td>13 17 5.2 6.3 1.09</td>
</tr>
<tr>
<td>Buddy</td>
<td>2 3 1.9 2.0 1.56</td>
<td>2 3 2.0 2.0 1.47</td>
<td>2 3 2.0 2.0 1.47</td>
</tr>
</tbody>
</table>
Fig. 2. Skewed distribution used for randomly selected lengths (Appendix B).

<table>
<thead>
<tr>
<th>Method</th>
<th>Approx average number of machine instructions executed per node visited (A)</th>
<th>(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Fit</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Best Fit</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Leftmost Fit</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Better Fit</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>Buddy</td>
<td>18</td>
<td>18</td>
</tr>
</tbody>
</table>

Table II. Approximate average number of machine instructions executed per node visited in allocate (A) and release (R).
practice it is rare for very small or large allocations to dominate. The lengths were generated by the method described in Appendix B.

In addition, for each method, an estimate was made of the average number of machine instructions executed per node visited. The results are shown in Table II. For this purpose, I assume a register machine which supports instructions such as LOAD, STORE, ADD, SUBTRACT, AND, OR, EXCLUSIVE OR, COMPARE and CONDITIONAL BRANCH. The instruction counts, in conjunction with the counts of nodes visited, can be used to make a rough comparison of raw execution times. It should be made clear, however, that the actual instructions, and their speed, depend upon the machinery. Also the number of instructions executed depends upon the cleverness of the programming, or the compiler, and no claim is made that the counts reported are the minimum that can be achieved. Finally, the counts given for First Fit and Best Fit can be approached only when the number of nodes visited is fairly large: when only a few nodes are visited, the initial and final bookkeeping dominates, and inflates the count.

12. Summary of simulation results

In all cases, Best Fit required the least storage (from 3 to 14 percent above the maximum allocated amount), and generally ran slightly faster than First Fit. Both these methods, however, had intolerable execution time when the number of pieces was large.

As expected from section 8, First Fit and Leftmost Fit required exactly the same amount of storage (which in these tests was from 8 to 18 percent above the maximum allocated amount), but Leftmost Fit ran much faster when the number of pieces was large. The only disadvantageous statistic of Leftmost Fit (compared with First Fit) is that a few more nodes were changed.

In these tests, Better Fit required about the same amount of storage as Leftmost Fit, and ran up to twice as fast (because the tree was more nearly balanced). There are however known to be cases in which Better Fit uses more storage (see next section).
The Buddy Method required the most storage (from 47 to 61 percent above the maximum allocated amount), but exhibited the smallest execution time.

There are the following observations on the overall performance of Leftmost Fit (or Better Fit) compared with the Buddy Method. For the situations studied here, the Buddy Method requires about 1.4 times the storage, but has an execution time that is between 1 and 6 times smaller. If the proportion of total execution time that is spent in the storage allocation routine is small (say a few percent), and the amount of storage that is managed by it is fairly large (say several megabytes), then the better storage utilization of Leftmost Fit (or Better Fit) may outweigh its greater execution time.

13. Additional observations

Uniform length distribution

Another set of simulations was run, with the same average lengths and the same lifetime distribution as those used for Table I, but with the individual lengths randomly selected from a uniform distribution instead of from a skewed distribution. The uniform distribution is thought to be less realistic, but has often been used in the literature for reasons of simplicity. In fact, the two sets of results were very similar, and the second set is not presented here.

Length, arrival and lifetime distributions taken from a real system

Bozman et al. [1982] measured the average inter-arrival times and lifetimes of storage acquisitions in CP/370 (component of the IBM virtual machine system) as a function of their length. The performance of Leftmost Fit and Better Fit has been studied by them, using these data, with inter-arrival times and lifetimes randomly selected from truncated exponential distributions having the appropriate averages. Leftmost Fit performed well, and is recommended by them as the preferred method when sub-pools are not applicable (or are full). There was, however, a surprising result for Better Fit, which required almost 1.5 times as much storage as Leftmost Fit (for the same allocations). There is at present no satisfactory explanation for this unexpected
behaviour, but it is thought to have something to do with cycles of short and long lifetimes as a function of length (a feature which does not normally appear in synthetic simulations). For this reason Better Fit cannot unequivocally be recommended over Leftmost Fit.

Pathological cases

Since the performance of Leftmost Fit and Better Fit depend upon the shape of the cartesian tree, it is possible to find sequences of calls to allocate and release which result in gross lack of balance, and consequently give rise to large execution times. Imagine, for example, starting with a clean pool, allocating pieces of increasing (or decreasing) length, and then releasing every alternate piece. The tree will now comprise a single edge. If the subsequent activity consists of alternately allocating and releasing a small piece, the entire edge will be examined on every call. Fortunately such patterns have not been observed in practice.

14. Final remarks

For various reasons, First Fit is a widely used method of dynamic storage allocation. It utilizes storage reasonably efficiently, it permits partial release to be supported, and it is simple to program. When the number of pieces allocated becomes large, however, the execution time becomes unacceptable.

Leftmost Fit can be viewed as an alternative implementation of First Fit which runs much faster when the number of pieces is large. Probably its only practical disadvantage (compared with First Fit) is increased program complexity.

In cases where good storage packing is an important factor, Leftmost Fit may also result in better overall performance than Buddy Methods.

Better Fit seems to run about twice as fast as Leftmost Fit, with storage packing which is usually about the same, or slightly better. There are however known to be situations in which it requires more storage, so it should be used with caution.
Acknowledgements

I wish to thank the University of Hong Kong for providing computer facilities which enabled me to perform some initial experiments with these ideas in 1981, while I was a visitor there; my colleagues at IBM Research, and the referees, for some good suggestions when shown a draft of this paper; and Cathy May, in particular, for also improving its appearance.
Suppose that $N$ pieces of contiguous storage are inserted, in random order, into an ordered linear linked list, and coalesced when possible. We will assume the list is initially empty; then when all the pieces have been inserted, it will contain a single block. However, when some but not all of the pieces have been inserted, there will in general be several or many blocks in the list, and on average it will be necessary to chain through about half of them in order to insert a piece in the right place, or to coalesce it with existing blocks if it touches them.

This is the situation that prevails when pieces of storage that have been allocated using the simple First Fit or Best Fit method are released, one by one, in random order.

Let us consider how many block headers, on average, must be visited during an insertion, i.e. how many distinct headers are referred to (or modified). Here are the answers for some small values of $N$:

If $N = 1$, there are no existing blocks to be visited, but a new header is stored in the piece being inserted. The number of headers visited is therefore $1$.

If $N = 2$, there are two possible sequences of insertion, viz. $1\ 2$ (for which the number of headers visited is $1 + 1$) and $2\ 1$ (for which it is $1 + 2$). Each of these sequences is equally likely, and the average number of headers visited per insertion is therefore $5/4$.

If $N = 3$, there are six possible sequences of insertion, each of which is equally likely, and the number of headers visited is as follows:
<table>
<thead>
<tr>
<th>Order of insertion</th>
<th>Number of headers visited</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>1 + 1 + 1 = 3</td>
</tr>
<tr>
<td>1 3 2</td>
<td>1 + 2 + 2 = 5</td>
</tr>
<tr>
<td>2 1 3</td>
<td>1 + 2 + 1 = 4</td>
</tr>
<tr>
<td>2 3 1</td>
<td>1 + 1 + 2 = 4</td>
</tr>
<tr>
<td>3 1 2</td>
<td>1 + 1 + 2 = 4</td>
</tr>
<tr>
<td>3 2 1</td>
<td>1 + 2 + 2 = 5</td>
</tr>
</tbody>
</table>

In this case, therefore, the average number of headers visited per insertion is $25/18$.

I am assuming that headers are stored at the left-hand end of a block; consequently a piece of storage with a left neighbour in the list does not itself need to be visited during its insertion.

In general, let $V_N$ be the average number of headers visited during the insertion of a single piece of storage when there are $N$ pieces altogether. It turns out that we can derive $V_N$ in terms of $V_{N-1}$ for $N \geq 2$. To do this, we decompose the problem with $N$ pieces into the problem with $N-1$ pieces together with an extra piece of storage at the beginning. Consider (a) how many headers are visited to insert the extra piece, (b) how many additional headers are visited to insert what is now the second piece (over and above the number when the first piece does not exist), and (c) how many additional headers are visited to insert what are now the 3rd, 4th, ..., piece (over the numbers when the first piece does not exist).

(a) If the first piece is inserted before the second, inserting the first piece involves visiting one header (in the first piece itself). If the second piece is inserted first, inserting the first piece involves visiting two headers (in the first piece and the second piece). Each of these is equally likely, and therefore the average number of headers visited to insert the extra piece at the beginning is $3/2$.

(b) If the first piece is inserted before the second, inserting the second piece does not involve visiting any more headers than if the first
piece did not exist (the first piece is visited, but the second is not). Similarly, if the second piece is inserted first, the number of headers visited (when inserting it) is not affected by the existence of the first piece. Therefore the number of additional headers visited to insert the second piece is 0.

(c) Here is the tricky part of the argument. Represent the uninserted state by 0 (piece is not in list), and the inserted state by 1 (piece is in list). Before any pieces have been inserted into the list, the first two pieces are in the state 0 0, and finally they are in the state 1 1. They pass from the initial to the final state by one of the following two paths:

\[
\begin{align*}
0 0 & \quad \rightarrow \quad 0 1 \quad \rightarrow \quad 1 1 \\
0 0 & \quad \rightarrow \quad 1 0 \quad \rightarrow \quad 1 1
\end{align*}
\]

Each path is equally likely, and within each path each state is equally likely; however only the state 0 1 results in an increase (by 1) in the number of headers visited when inserting the 3rd, 4th, ..., piece (compared with inserting the same piece in the absence of the first piece). This state exists (on average) 1/6 of the time. Therefore the average number of additional headers visited to insert the kth piece (k > 2) is 1/6.

Summing (a), (b) and (c), we get

\[
V_N = \frac{1}{N} \left\{ \frac{3}{2} N^2 + \frac{N-2}{6} + (N-1)V_{N-1} \right\}, \text{ for } N \geq 2. \tag{1}
\]

With the known solutions for small values of N, we obtain the solution:

\[
V_N = \frac{1}{12N} \left( N^2 + 15N - 4 \right). \tag{2}
\]

Summarizing, for N \gg 1, V_N = N/12 + O(1). Furthermore, since on average roughly half the blocks in the list are visited when inserting a new piece of storage, we can conclude that the average number of blocks in the list is N/6 + O(1).
The worst case occurs when the order of insertion is 1 3 5 ... N ... 6 4 2. Then the average number of blocks in the list, and the average number of headers visited when inserting a piece of storage, are both \( N/4 + O(1) \).
APPENDIX B

GENERATION OF RANDOM LENGTHS FROM SKEWED DISTRIBUTION

The randomly chosen lengths that were used for the simulations described in sections 10, 11 and 12 were selected from the skewed distribution shown in Fig. 2. This is thought to be a more realistic model for typical storage acquisitions than a uniform or exponential distribution, which have sometimes been used for reasons of simplicity. For lengths much smaller than the mean, the density of the distribution increases approximately linearly with the length; and for lengths larger than the mean it decreases roughly exponentially, until the distribution is finally truncated. To be precise, a length \( L \) was generated from a pseudo-random number \( X \) by the following function:

\[
L = \left[ K \left(1 - X^2\right)^{\frac{1}{2}} (1 - \ln X) \right],
\]

where \( X \) was chosen with equal probabilities from the sequence \( 2^{-31}, 2 \times 2^{-31}, 3 \times 2^{-31}, \ldots, 1 - 2^{-31} \), and \( K \) was a constant chosen to yield the desired mean.

The truncation of the distribution is a convenient side-effect of the method of generation. When the mean length is 100 words, for example, the maximum possible length is 1298 words, which occurs when \( X = 2^{-31} \). If a distribution with a very long tail is used, some of the results (especially the maximum storage occupied) can be dominated by one or two exceptionally large allocations: this may be realistic, but it tends to conceal the differences between the methods rather than exposing them.
References

Bozman, G., Buco, W., Daly, T. and Tetzlaff, W., Analysis of free storage algorithms — revisited, private communication (1982), to be published in IBM Sys. J.


Peterson, J. L. and Norman, T. A., Buddy systems, CACM 20, 6, 421 (1977).

Shore, J. E., On the external storage fragmentation produced by first fit and best fit allocation policies, CACM 18, 8, 433 (1975).
