RC 20542 23 Sep 1996
Computer Science 154 pages

Research Report

Stacks and Trees and Strings and Bits and Pieces

C. J. Stephenson
IBM Research Division
T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, N.Y. 10598

LIMITED DISTRIBUTION NOTICE
Portions of this report may be submitted for outside publication and if accepted will probably be copyrighted by the publisher. Distribution of this report outside IBM should therefore be limited to peer communication and specific requests.

Research Division - Almaden, Austin,
Tokyo, Yorktown Heights and Zurich

STACKS AND TREES AND STRINGS AND BITS AND PIECES

C. J. Stephenson

This report comprises a collection of short papers on divers programming techniques which I have stumbled on over the past twenty years while working on the experimental operating system EM-YMS.

The word "paper" is perhaps pretentious, and I use it only for want of a better word. By no means all of the items are suitable for publication in scientific journals. One reason for assembling them here is to place the ideas in the "public domain". This is intended to dissuade anyone who might independently discover the same techniques from attempting to obtain patents for them.

Several of the papers explicitly note that the work was done jointly with one or two other people. All of the work reported here has however benefitted from long-term collaboration, both formal and informal, with some very talented colleagues at IBM Research -- particularly Walter Daniels, Michel Hack, Paul Kosinski (now at Digital Equipment Corporation), and Gerald Spivak.

Although one motivation for producing this report is to protect the ideas in it, I hope it may also make enjoyable reading. Regard it as a collection of short stories.

By the way, I use first and second person pronouns in the following way. The first person singular refers to me the writer; the second person (plural) refers to you the reader; and the first person plural refers to the fleeting collaboration that exists between us.

CJS, Yorktown Heights, September 1996. cjs@vnet.ibm.com

GLOSSARY

CISC	Complex Instruction Set Computer. This simply means "non-RISC". Before the mid-1980s nearly all computer architectures were CISC -- though they did not know it at the time, since the term was not yet established. CISC programs tend to occupy less space than RISC programs, but the hardware is more complicated. IBM S/370 and Intel "x86" are examples of CISC architectures.
EM	Extended machine. This is the "lower" layer of EM-YMS. It contains a file system, a paging manager, IO programs, interruption handlers, debugging facilities and so on.
EM-YMS	An experimental two-layer operating system at IBM Research. The system runs on IBM mainframes and S/370 workstations.
Ending address	Beginning address + length.
lg X	$\log_2 X$
ln X	$\log_e X$
RISC	Reduced Instruction Set Computer. This has been a popular trend in machine architectures since the mid-1980s. Compared with CISC, RISC is characterized by a reduced repertoire of instructions referring to memory, and in addition all the instructions usually have the same length. Advantages include simpler hardware and faster clock cycles. The IBM RS/6000, the "Sparc" station (by Sun Microsystems), and the "PowerPC" (by IBM, Motorola and Apple) are examples of RISC.
S/370	System/370. This was the name of IBM's mainframe architecture from 1970 to 1987. The name was adjusted several times during the 1980s, and at the time of writing it has been renamed the "Enterprise Systems Architecture/390". In this report I mostly use the older name, since it is widely known, and the examples given here do not depend on features that postdate the name change.
YMS	Yorktown Monitor System. This is the "higher" layer of EM-YMS. It supports a runtime environment for application programs.

CONTENTS

Deferred interruption handling	1
The self-adjusting stack	15
Calendric programming	35
Five coding techniques	45
Practical methods for handling self-adjusting binary search trees	59
On traversing and dismantling binary trees	93
The skewed file tree	101
Bulk hashing of file directories	111
Warped hashing	119
Double-ended memory allocator	123
On the latency of Boyer-Moore search	129

DEFERRED INTERRUPTION HANDLING

1. BACKGROUND

Systems usually contain a few sensitive structures whose update requires special care. Consider for example a linked list or tree which records the disposition of memory and is maintained by the memory allocator. Suppose an application program has called the memory allocator to request an area of memory. The allocator has identified a suitable piece and is in the process of updating the list or tree to record the new state of affairs. Temporarily (until the update is complete) the structure in memory will present an inconsistent picture if it is viewed by an observer that does not know what is going on. Now suppose there is an external event, such as a keystroke, which interrupts the memory allocator. The interruption handler must add an element to the keystroke queue -- to be consumed later -- and to do this it needs a few words of memory. The obvious way to obtain these is to call the memory allocator. Normally this would work fine; but it cannot be done safely in the present situation since at the moment of the interruption the structure describing memory is partially updated and presents an inconsistent picture.

The conventional way to circumvent this problem is to run the memory allocator disabled for interruptions. This postpones the reporting of external events. After the memory structure has been updated, and possesses a new consistent state, the allocator reenables for interruptions. Any pending event is then presented to the interruption handler -- which can happily call the memory allocator if it needs to. Finally (of course) the interruption handler resumes execution (enabled) from the point of interruption.

(In systems with several processors and shared memory, additional mechanisms are required to prevent the memory structure from being updated by two or more processors simultaneously. This is outside the scope, and the subject matter, of the present note.)

Fig. 1 shows the relationship between an application program, a service routine that runs disabled, and an interruption handler. S/370 assembler notation is used (see [1]).

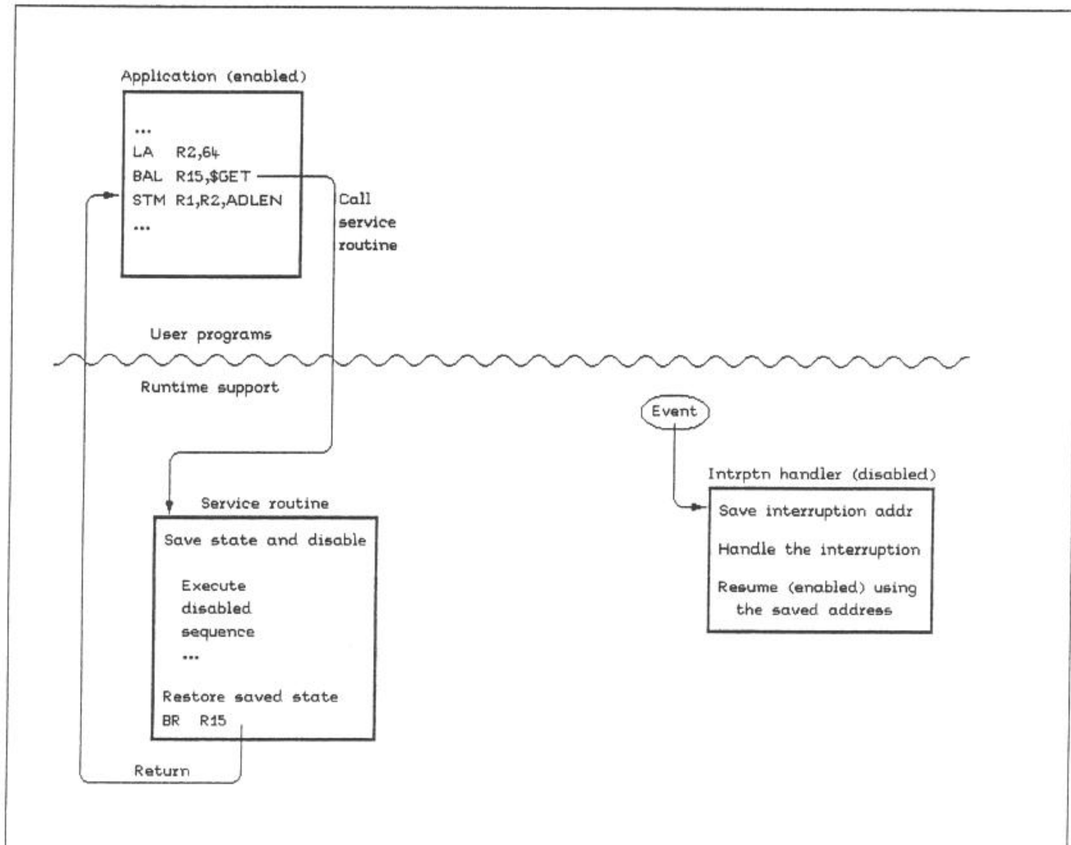


Fig. 1. Conventional interruption handling.

2. PERFORMANCE PROBLEM

The trouble with constantly disabling and reenabling is that it can be expensive.

If the memory allocator runs in supervisor state, it can disable and reenable the processor by executing special-purpose machine instructions. These instructions are often quite slow. On some machines they take about 10 times longer than ordinary instructions such as Load and Branch.

Things get worse.

In modern systems the application programs, along with service routines such as the memory allocator, run in problem state. These programs are not allowed to disable the entire processor. Instead they tell the supervisor (or kernel) that they must not be disturbed for a while; the supervisor then queues their interruptions internally, for presentation later. This has the desired effect; but the cost of asking the supervisor to hold the interruptions (or of cancelling the request) may be 30 or 100 times greater than the cost of an ordinary machine instruction. And since interruptions at this level in the system are fairly rare, the application program may call the memory allocator hundreds or thousands of times per interruption. Therefore these expensive operations nearly all turn out to be unnecessary.

(On some machines the act of disabling, and the act of restoring the prior state, can be combined with the calling mechanism and the return mechanism. In S/370 the SVC instruction (Supervisor Call) is intended for this purpose. If the machine is prepared appropriately, this instruction can transfer control, disable, and change privilege mode, all at the same time. The difference between this method of changing state and the explicit method is not however germane to the present discussion, and for simplicity I will consider only the explicit method.)

I have been citing the memory allocator as an interruption-sensitive routine. It is a good example, but in reality there are usually several such system services. Another one is the routine that delivers a queued keystroke, which must prevent the interruption handler from appending a new keystroke to the queue at the same instant that it is removing the last one already there.

3. OPTIMISTIC INTERRUPTION HANDLING

If a soldier is patrolling a combat zone, he should not wait until he has been shot before taking steps to protect

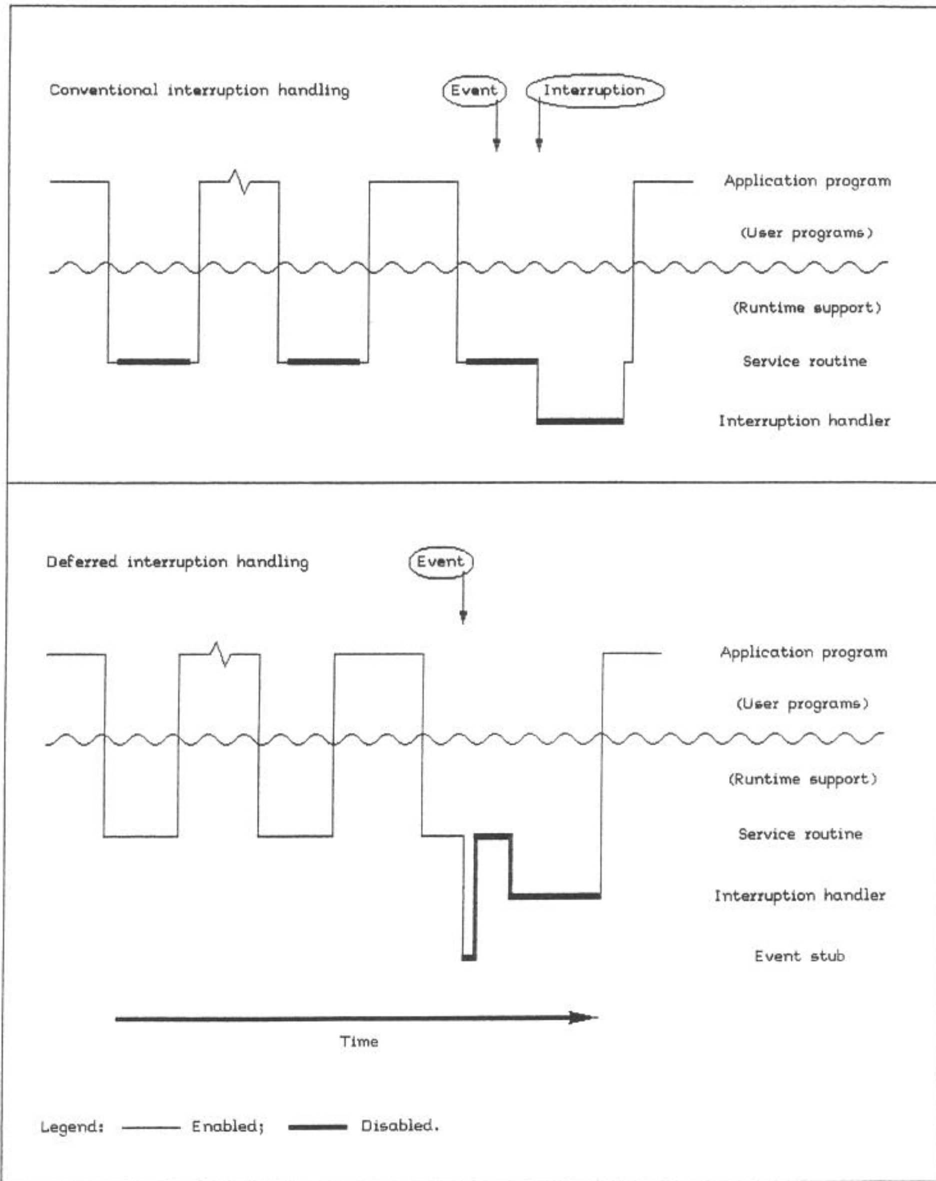


Fig. 2. State transitions during conventional interruption handling and deferred interruption handling.

himself from bullets. But when designing computer systems we can sometimes finagle matters so that it is safe to live dangerously. It is in fact possible for an interruption-sensitive service routine to allow itself to be shot, and obtain protection only if it turns out to have been necessary.

The basic idea is this. An interruption-sensitive service routine (such as the memory allocator) begins in the same state as its caller, i.e. enabled or disabled. If it is called from an application program, it begins enabled. Usually it will run to completion without being interrupted, and without incurring any extraordinary costs (see below). Occasionally however an event will occur while the service routine is running. When this happens, the supervisor transfers control (disabled) to a small "event stub". (The mechanism for doing this does not concern us here; it is the same mechanism that would conventionally be used to transfer control to the interruption handler.) The event stub observes that an interruption-sensitive service routine is running, and immediately resumes execution (disabled) at the point of interruption, inside the service routine. The service routine is therefore not affected by the interruption, except that after the event it runs disabled. (Remember that this is what it always wanted to do, but was inhibited by cost.) The actual handling of the interruption is deferred until the service routine has run to completion.

From the point of view of the interruption handler, an event that occurs while a service routine is running appears to have occurred just as the service routine was about to return to its caller.

The net effect is that the service routine runs as if it was disabled, but without incurring the cost of disabling. Physically it runs enabled most of the time.

If a second event occurs before the first one has been handled, it will be held by the supervisor, since the program continues to run disabled until the first event has been handled. (Remember that this is the behaviour we originally wanted for all events ... but we were inhibited by cost.)

Fig. 2 compares the transitions that occur during conventional interruption handling with those that occur when interruption handling is deferred.

4. IMPLEMENTATION AND OTHER DETAILS

It turns out that this scheme can be implemented elegantly, at low cost, and without using any fancy synchronization features, provided the machinery supports the following two problem-state instructions:

OI loc,imm	Set the byte at the given location to the "or" of its old value with the 8-bit immediate field in the instruction
NI loc,imm	Set the byte at the given location to the "and" of its old value with the 8-bit immediate field in the instruction, and set the condition code to 0 if the resulting value is 0, or to 1 otherwise

We will use these instructions to maintain a flag byte which is reachable from all the programs that comprise the runtime support, i.e. the service routines, the event stub and the interruption handlers. (In S/370 the flag byte can be placed in page zero of the applicable address space.) The bits in this byte will be used as described below to record (a) whether an interruption-sensitive service routine is running, and (b) whether there is a deferred interruption. Let me be specific, and define the flag bits thus:

FLAG:								
Name of bit:	S0	S1	S2	S3	S4	S5	S6	KG
Usual value:	0	0	0	0	0	0	0	1

Here S0, S1, ..., stand for "Service flag 0", "Service flag 1", etc., and "KG" stands for "Keep Going".

There is one more thing to be done before describing how these flags are actually used. The system designer must assign each interruption-sensitive service routine to one of 7 groups, numbered 0, 1, ..., 6, such that the groups reflect the calling hierarchy. Service routines in group 3 (for example) may call service routines in groups 4, 5 and 6, but no others.

Now this is how deferred interruption handling can be implemented.

1. Before a service routine in group "j" enters its critical sequence, it sets bit Sj (using the OI instruction). This publicizes the fact that an interruption-sensitive routine is running.

   (If the service routines have been properly assigned, bit Sj will always be 0 on entry to a service routine in group j.)

2. If an event occurs, the supervisor transfers control to the event stub. On arrival here the program is disabled. The event stub clears the "KG" bit (using the NI

instruction), in order to publicize the fact that there is an interruption to be handled, and it examines the resulting condition code.

If the condition code is 0, then S0,S1,...,S6 must all be 0, which means there are no interruption-sensitive service routines in execution. In this case the event stub immediately passes control to the interruption handler (remaining disabled). The interruption handler restores the "KG" bit to 1 (using the OI instruction) and handles the interruption. While doing so it may call interruption-sensitive service routines if the need arises. Finally it resumes execution (enabled) at the point of interruption. This is the simple (and typical) case.

If the condition code is 1, then one or more of S0, S1,...,S6 are evidently set, which indicates that at least one interruption-sensitive service routine is running (and possibly several). In this case the event stub resumes execution from the point of interruption, inside the service routine. From this point on the service routine runs disabled, so any subsequent interruptions will be held at bay by the supervisor.

3. When a service routine in group "j" is ready to return control to its caller, it clears bit Sj (using the NI instruction) and examines the resulting condition code.

   If the condition code is 1, then one or both of the following statements must be true:

   a. The "KG" bit is set. This indicates that neither this routine, nor any routines it may have called, has been interrupted by an event, and therefore no deferred interruption exists. In this case the program should simply "keep going", i.e. the service routine should return to its caller in the ordinary way.

   b. One or more of Si are set (i<j), meaning that this service routine was called from some other interruption-sensitive service routine. In this case this service routine must return to its caller, irrespective of whether there has been an event; for the system will not be in a fit state to handle a deferred interruption until the last interruption-sensitive routine has run to completion.

   So if the condition code is 1, the service routine returns directly to its caller, without needing to know

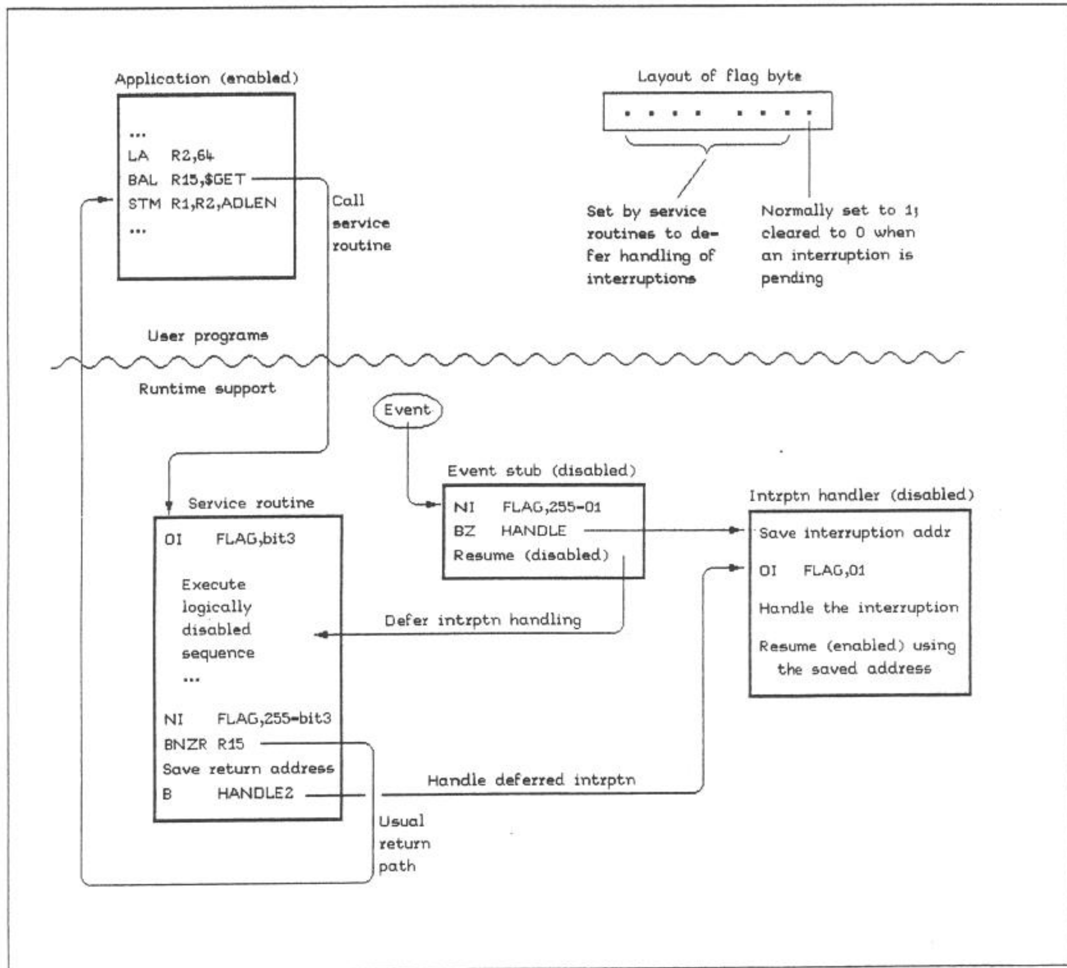


Fig. 3. Deferred interruption handling.

the precise reason why this is the right thing to do.

On the other hand, if the condition code is 0, both of the following statements must be true:

a. The "KG" bit is 0. This means that this service routine, or some other service routine called by it, was momentarily interrupted by an event, and a deferred interruption exists.

b. None of S0,S1,...,S6 is set. This means that there are no interruption-sensitive routines remaining in execution.

In this case the service routine that thought it was about to return to its caller must instead proceed as follows:

> it must save its own return address for use by the interruption handler (see below);
>
> it must then pass control to the interruption handler.

The interruption handler restores the "KG" bit to 1 (using the OI instruction), and handles the interruption. While doing this it may call interruption-sensitive service routines if the need arises. Finally it resumes execution (enabled) at the return address of the service routine that passed control to the interruption handler.

Fig. 3 depicts the technique graphically.

5. LIMITATIONS AND FINE POINTS

1. The technique as it is described above does not allow an interruption-sensitive service routine to pass information to its caller in the condition code -- which is used for other purposes. This restriction can easily be lifted. Doing so requires a few more instructions at the end of the service routines. It also requires that the interruption handler be capable of resuming execution with an explicitly saved condition code (as well as with an explicitly saved address).

2. This technique will not handle interruption-sensitive routines that are recursive. This is because each routine must be assigned ahead of time to a specific group, and is not allowed to call interruption-sensitive routines in the same group or in groups with a lower number.

3. The number of service routine groups could if necessary be increased beyond 7. In the case of S/370, instructions exist which can atomically set a bit (or clear a bit) in a string of up to 2048 bits and at the same time set the condition code (OC, NC). These instructions are functionally akin to the OI and NI instructions shown above, but the immediate field is replaced by an operand in memory. Using these instructions, the number of groups could be increased to 2047 without any impact on program complexity. (In practice this would usually be overkill.)

4. This technique works best when there is a single class of asynchronous interruptions -- or when (if there is more than one class) the classes are all enabled and disabled together as if they formed a single class. I do not know whether the technique could be extended to handle situations in which some classes of interruptions may be enabled while others are disabled.

5. The advantages of this technique diminish if the service routines require different privileges from those enjoyed by the application programs. Suppose for example that so-called storage "keys" are used, and the service routines have access to regions of memory that are out of bounds to the application programs. Then an interaction with the supervisor is unavoidable (to change the privilege), and conventional disabling (or reenabling) can be handled at the same time, at little extra cost.

When the limitations are acceptable, the technique is powerful. It allows a set of service routines to handle interruptions in a fairly general way, while exhibiting performance that is about the same as that of compiler runtime libraries.

6. ARCHITECTURAL IMPLICATIONS

Deferred interruption handling is easy to implement on the S/370, which supports OI and NI instructions similar to those described above. As far as I know, the technique has never been implemented on a "RISC", where usually the only way to gain access to memory is via registers. This presents a symmetric pair of problems, at the beginning and end of each interruption-sensitive service routine:

a. On a "CISC", service routines in the same group can share a static register save area. This can be protected simply by setting the appropriate service flag (Sj) before the caller's registers are stored -- effectively treating the save area as a part of the sensitive

structure. On a RISC, however, the act of setting Sj requires a scratch register; so it is impossible to set the Sj bit (and also save all the registers) unless some more complicated scheme is used for supplying dynamic save areas.

b. On a CISC, a service routine can get ready to return to its caller, including reloading all the registers, and then modify (and examine) the flag byte at the last moment. On a RISC, the act of modifying (and examining) the flags requires a scratch register, so it is impossible to restore all the caller's registers -- unless (as before) a dynamic save area is supplied.

On a RISC, therefore, it seems necessary to adopt a compromise, such as one of the following:

Compromise A — The rules for calling interruption-sensitive service routines must permit at least one register to have undefined contents on return.

Compromise B — Each caller of an interruption-sensitive service routine must supply at least one word of scratch space, thus driving the hard part of the problem back to the application programs (and the other parts of the system that call the services).

If one of these compromises is acceptable, then the technique can be implemented on a RISC, though the details are not as clean as on a CISC. In order to be specific, let me adopt Compromise A. This is consistent with the AIX linkage conventions, which leave several registers (including GPR 0) undefined after a call: see [2]. (Here I will let the undefined register be named "scratch".) Then this is how we might proceed:

1. We must place the "KG" bit in a different byte from S0,S1,... . This allows the event stub to update the "KG" bit reliably -- without tripping over a service routine that is in the process of setting or clearing one of the "service" bits, which on a RISC is perforce a non-atomic operation. It is convenient to place the two flag bytes in contiguous locations, in the same halfword, e.g. thus:

FLAG:								
Name of bit:	S0	S1	S2	S3	S4	S5	S6	--
Usual value:	0	0	0	0	0	0	0	0

```
        FLAG+1:
    Name of bit:   -- -- -- --   -- -- -- KG
    Usual value:    0  0  0  0    0  0  0  1
```

2. At the beginning of an interruption-sensitive service routine, replace the OI instruction that sets bit Sj (e.g. j = 3) by the sequence:

```
lbz   scratch,FLAG          # Fetch first flag byte
ori   scratch,scratch,bit3  # Set my service bit
stb   scratch,FLAG
```

Here and in the following notes I use 32-bit PowerPC assembly language notation with a "big endian" model of memory (see [2,3]).

Remember we are not considering multiple processors with shared memory. Also the event stub and the interruption handler do not modify S0,S1,... . These facts explain why it is not necessary to use fancy synchronizing instructions when setting bit Sj.

3. In the event stub, replace the NI instruction and the conditional branch by the sequence:

```
stw   any,private         # Save a GPR in static

lhz   any,FLAG            # Fetch both flag bytes
andi. any,any,65535-01    #  and clear the KG bit
stb   any,FLAG+1

lwz   any,private         # Quietly reload the GPR
beq   HANDLE              # Br if FLAG = FLAG+1 = 0
```

Here the symbol "any" stands for any general-purpose register.

4. At the end of an interruption-sensitive service routine, replace the NI instruction and the conditional branch by the sequence:

```
lbz   scratch,FLAG        # Fetch first flag byte
andi. scratch,scratch,65535-bit3  # Clear my bit
stb   scratch,FLAG

lhz   scratch,FLAG        # Fetch both flag bytes
or.   scratch,scratch,scratch  # Examine all bits
bnelr                     # Ret unless deferred int
```

Note that there is no point is inspecting the "KG" bit in FLAG+1 until after bit Sj has been cleared in FLAG. Were the old value of "KG" used, the pro-

gram might reach the wrong conclusion if the event stub happened to gain control during execution of the first three instructions above.

5. In the interruption handler, replace the OI instruction by the sequence:

```
lbz   any,FLAG+1        # Fetch 2nd flag byte
ori   any,any,01        #  and set the KG bit
stb   any,FLAG+1
```

   These instructions must of course be executed after the "any" register has been saved. (The registers will usually need to be saved anyway, in order to handle the interruption.)

My colleague Michel Hack observes that, given the need for a scratch register, and given that the "KG" bit must be stored separately, the "service" bits could be replaced by a full-word counter. This would allow the scheme to be used even when recursion was involved.

Historical note

The technique described in this paper has been used since 1976 to handle interruptions in YMS.

REFERENCES

[1] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

[2] AIX Version 3.2 Assembler Language Reference, SC23-2197-02, IBM (1993)

[3] May, C. et al. (editors), The PowerPC Architecture, 2nd edition, IBM and Morgan Kaufmann (1994)

CJS, 1995-01-23.

Empty

THE SELF-ADJUSTING STACK

1. BACKGROUND

Program call stacks are used to support subroutine calls and function invocations in compiled code, runtime libraries, interpreters, editors, operating systems, etc. To call a subroutine, for example, the linkage code typically proceeds thus:

a. obtain a new stack frame of sufficient size,

b. save the necessary parts of the old state in the new stack frame, including the address of the previous stack frame,

c. prepare the new state for the subroutine, and commence execution;

and to return:

d. restore the prior state from the current stack frame (including the address of the prior stack frame), and vacate the current frame, thereby rendering the memory reusable,

e. resume execution following the point of call.

A "frame" is an area of memory within the stack. It is a physical entity. Consecutively assigned frames may or may not be contiguous in memory.

On the other hand, a "stack" is a notional entity. There are many different ways to implement a stack. But whatever the details, all stacks have one very handy property, which allows certain short cuts to be taken in their implementation:

> Frames are vacated in the opposite order from that of their acquisition.

Because a stack is involved on every subroutine call and every return, the performance of stack operations can be very important.

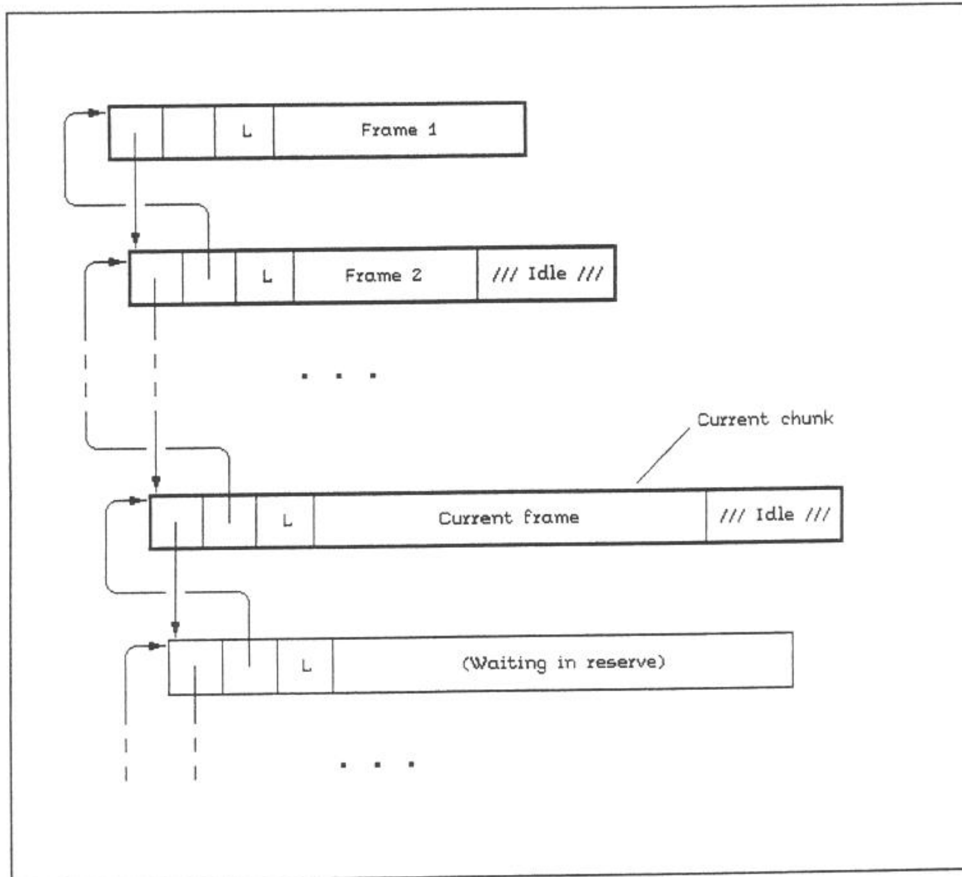


Fig. 1. Program call stack with one frame per chunk. "L" stands for "length of chunk".

2. CONVENTIONAL METHODS OF STACK MANAGEMENT

I will briefly review several commonly used methods of stack management.

Perhaps the simplest approach is to preallocate a single area of memory, which becomes dedicated to the stack. Frames are then suballocated sequentially from within this area, and the program is terminated if the stack ever overflows. This can certainly be made to run fast. It may be adequate in certain constrained situations, e.g. when the frame size and the maximum stack depth are known in advance, or when there is a large private segment of virtual memory dedicated to the task.

In the remainder of this paper, however, I will consider unbounded stacks. These possess no predefined limits, either on the frame size or the stack depth. Such stacks are still capable of failing, e.g. if the address space fills up; but they do not impose a separate constraint of their own. They will fail only if there is an impending crisis of a more general nature.

When implementing unbounded stacks, the following techniques are widely used.

Conventional Method 1 -- One frame per chunk

The memory required for each frame is acquired separately, from the memory allocator, and deallocated when the frame is vacated. This is sometimes done when calling programs in operating systems.

In its simplest form this method can be expensive, since the time taken by the memory allocator may exceed the time required by the subroutine that needs the frame.

A more sophisticated variant avoids deallocating the memory when a frame is vacated, in the hope that the same piece will be usable again, for another stack frame. Each frame must now be preceded by a short prefix which is used to chain the "chunks" together and to record their lengths. When a new frame is required, the first idle chunk (if any) is examined, and reused if it is big enough. This greatly reduces the number of trips through the memory allocator.

The technique is illustrated in Fig. 1.

This variant is not completely straightforward, since different frame sizes may be required on different occasions, and the chunk left over from an old frame may be insufficient for the next one. There are various ways around this. The

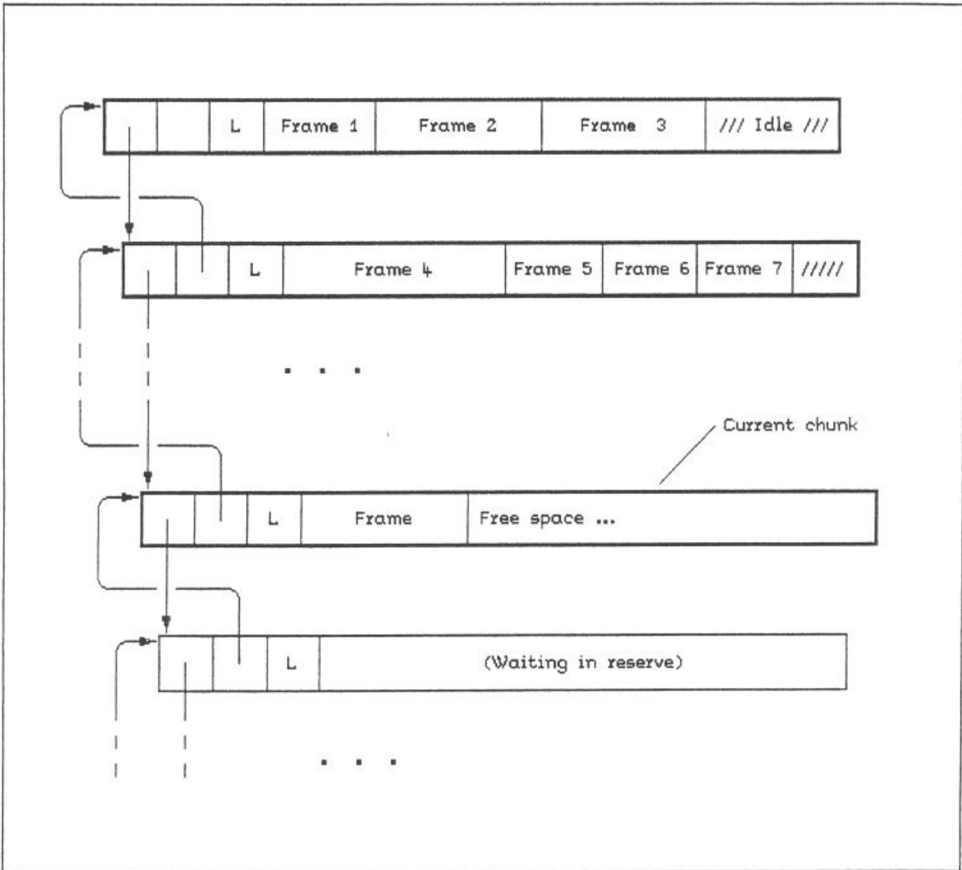


Fig. 2. Program call stack with several frames per chunk. "L" stands for "length of chunk".

strategy that I like best is this:

> If the first idle chunk is too small for a new frame, the chunk is simply deallocated and replaced by one that is sufficient.

During typical program loops, each chunk in the chain will soon reach the maximum size required of it; and thereafter frames will be assigned (and vacated) without any need to call the memory allocator. This method is used in the S/370 interpreters for Exec 2 (see [1]).

In practice this usually works quite well, but there are some disadvantages:

a. Since the chunks are allocated separately, they may end up scattered through memory, with undesirable effects on fragmentation and working set.

b. It is possible to devise pathological scenarios in which each chunk grows to a great size, even though the typical and average requirements may be small.

c. Every time a frame is assigned or vacated, several instructions are required to chain to the adjacent chunk. Also, when chaining down, it is necessary to check whether the adjacent chunk exists at all (and to allocate it if it does not).

The last consideration weighs against the use of this method in compiled code, where there is usually a desire to cut the cost of the calling sequence to its bare bones.

Conventional Method 2 -- Several frames per chunk

This method is often used in compiled code.

A moderately sized chunk of memory is obtained from the memory allocator, and stack frames are suballocated from within it. Usually a frame can be assigned by incrementing an "available" pointer by the frame length, and vacated by moving the pointer back to the address of the vacated frame. (This takes advantage of the fact that frames are vacated in the opposite order from their assignment.)

When a frame is required whose length exceeds that of the remaining free space in the current chunk, a new chunk is obtained from the memory allocator and the next few frames are suballocated from this ... until it too becomes full, and so on.

We can use the same trick as before, and keep empty chunks

for possible reuse. This reduces the number of trips through the memory allocator.

The technique is illustrated in Fig. 2.

This method works well most of the time. Only three or four machine instructions are required to determine whether a frame can be accomodated in the current chunk (and usually it can be), and two or three additional instructions are sufficient to assign the frame and adjust the "available" pointer. But on those occasions when the frame cannot be accomodated in the current chunk, there is noticeably more work to do:

> Is there another chunk of sufficient size already in hand? (If not, obtain a new chunk and append it to the chain.)
>
> Acquire the address of the free space in the next chunk, and its length, and store these in a convenient place for subsequent size checking.
>
> (Assign the stack frame, adjust the "available" pointer, etc., as in the typical case.)

The shortest path through these checks (represented by the parts not in parentheses) involves 7 to 10 extra instructions on most machines.

Similarly, when a frame is vacated, only three or four instructions are required in the common case; but on those occasions when the vacated frame is the last remaining one in the chunk, three or four additional instructions are required to step back to the previous chunk, and to store the "available" pointer and the chunk length in a convenient place for subsequent checking.

So typically a frame may be assigned in about 7 instructions (and vacated in about 3); but occasionally the assignment may take about 15 instructions (and the vacation about 6).

A problem therefore arises if a small subroutine is called in a tight loop, and through bad luck there is insufficient room for its stack frame in the same chunk as that of its caller. Now the extra work for crossing chunk boundaries is incurred on every call and every return, and the performance is insidiously degraded. The explanation for the degradation is not usually apparent to the user, and it may appear capricious or random. It can be pernicious when making performance measurements -- especially as the effect is not reduced, proportionally, by increasing the number of iterations.

3. THE SELF-ADJUSTING STACK

I will now describe a method of stack management which exhibits nearly the same best-case performance as Conventional Method 2, but automatically adjusts the shape of the stack so that the transition from one chunk to another never occurs repeatedly, in a loop.

The basic idea is as follows.

We start off in roughly the same way as Conventional Method 2, with a single chunk that is expected to be sufficient for several stack frames. The "available" pointer (and several other pieces of information) are maintained in a static anchor.

When a stack frame is required, the frame is placed at the beginning of the available space (and the "available" pointer is advanced appropriately), provided there is sufficient free space. When there is insufficient free space in the chunk, we proceed as follows:

1. Return the unused tail of the chunk to the memory allocator.

2. Obtain a new bigger chunk from the memory allocator, having (say) twice the original size of the previous (now full) chunk.

3. Reinitialize the "available" pointer and the other information in the anchor, ready to use the new chunk.

4. Try all over again.

When a stack frame is vacated, what happens depends upon whether the frame lies within the current chunk (i.e. the one most recently acquired). If the frame lies within this chunk, the space it occupies is potentially reusable (for subsequent frames), and the "available" pointer is simply set to the address of the vacated frame. Otherwise the space is deemed not reusable, and is immediately returned to the memory allocator, as an isolated piece; in this case the "available" pointer is not changed.

Therefore all the frames that are not in the current chunk are deallocated one at a time. At first sight this may seem an extravagant proposition -- and in a sense it is. But the saving grace is that it cannot occur in a loop.

Consider for example the sequence:

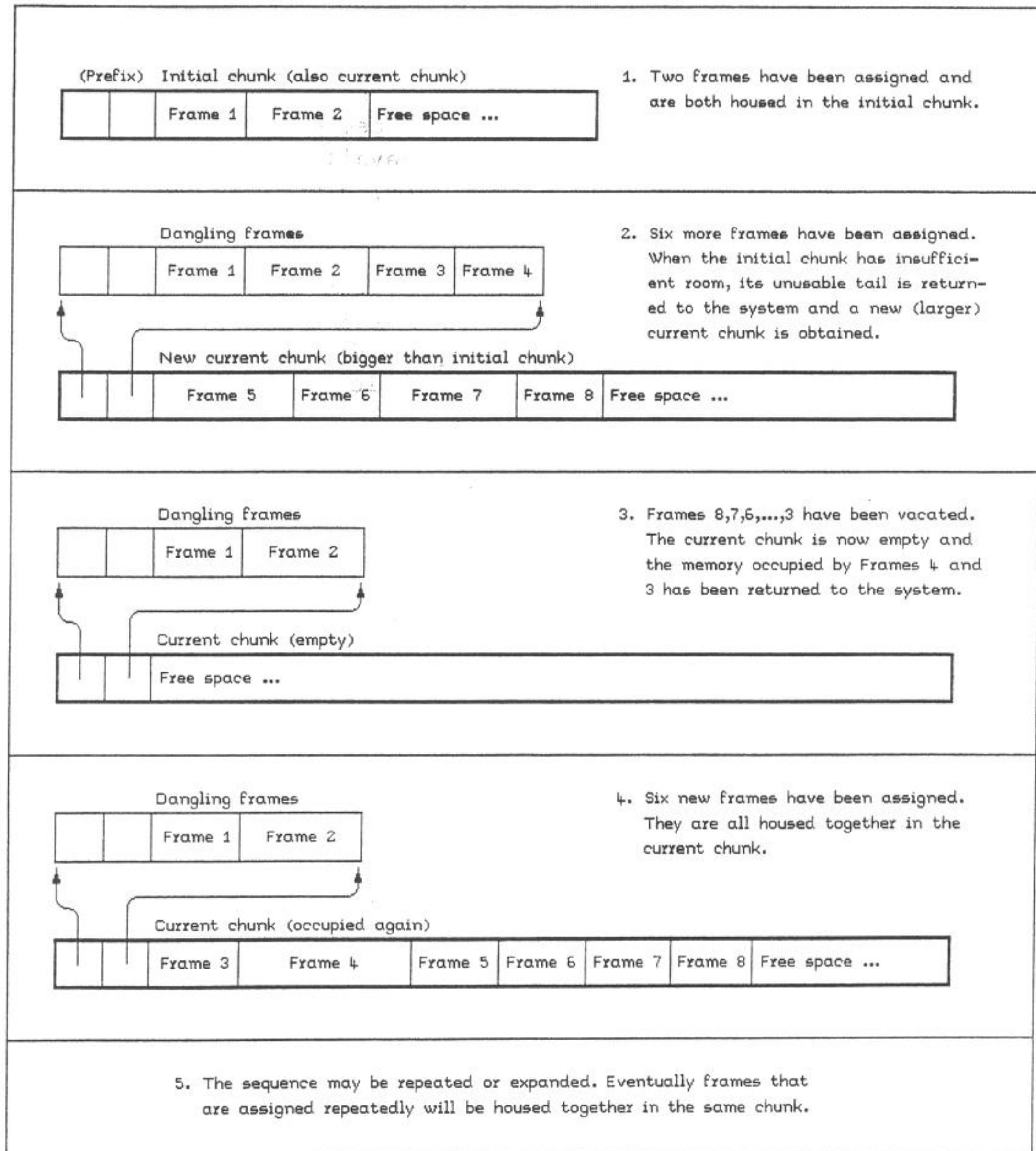


Fig. 3. An example of the self-adjusting stack.

```
A calls B;
B calls C;
C returns;
B returns.
```

Suppose that when A first calls B, B's stack frame nearly reaches the end of the original chunk. When B calls C, a new chunk is therefore required, and C's frame is placed at the beginning of it. (Assigning this particular frame is expensive.) When C returns to B, its frame is vacated (which is cheap) and the current chunk becomes empty (but is retained). When B returns to A, the vacated frame is not in the current chunk, so it is returned to the memory allocator. (This is expensive.)

Now imagine doing it all again. This time, B's frame is placed at the beginning of the new chunk; and C's frame is placed immediately after this. Both these operations are cheap. Vacating these frames is also cheap. So a loop that contains this sequence runs at full speed after the first iteration, without involving any further chunk crossing.

The reusable space is all in the current chunk. The current chunk is replaced when necessary, becoming bigger each time, so that it eventually contains sufficient space for the deepest sequence of repeated calls that occurs in the program.

The technique is illustrated in Fig. 3.

4. IMPLEMENTATION

Here is an implementation of the self-adjusting stack.

The prefix areas of the chunks are laid out as shown in Fig. 3. There is also a 4-word anchor, which resides at a fixed location, as shown in Fig. 4. The last word of the anchor contains the total length (in bytes) of the current chunk, including its two-word prefix.

I will describe two subroutines: "advance" supplies the address of a new frame, and "retreat" vacates the current frame. I will specify the important parts in detail, using S/370 assembler notation, so that we can count the instructions (see [2]). The rest will be described informally, in English.

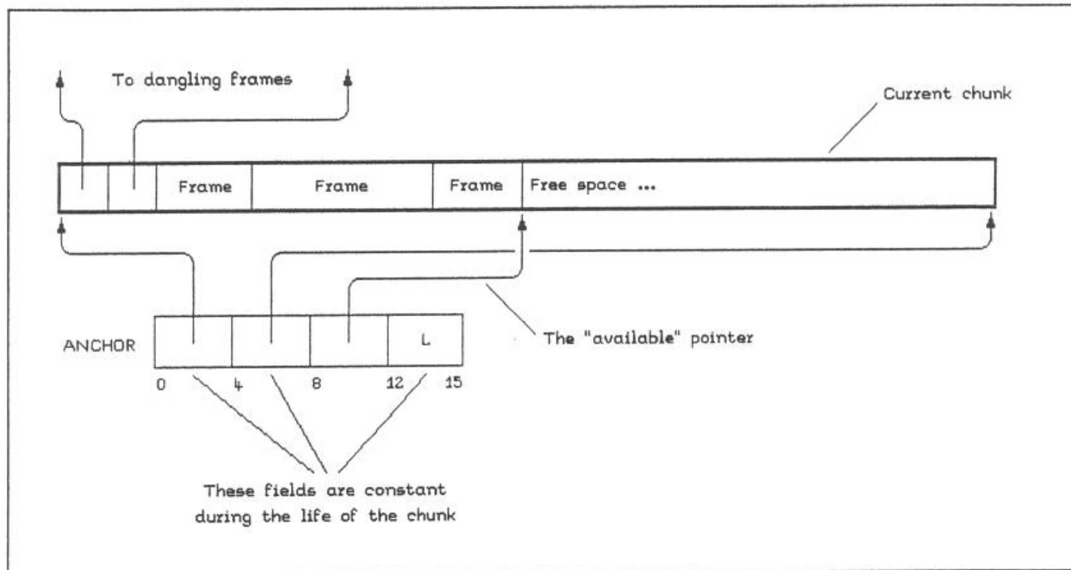


Fig. 4. Layout of anchor for self-adjusting stack. "L" stands for "length of chunk".

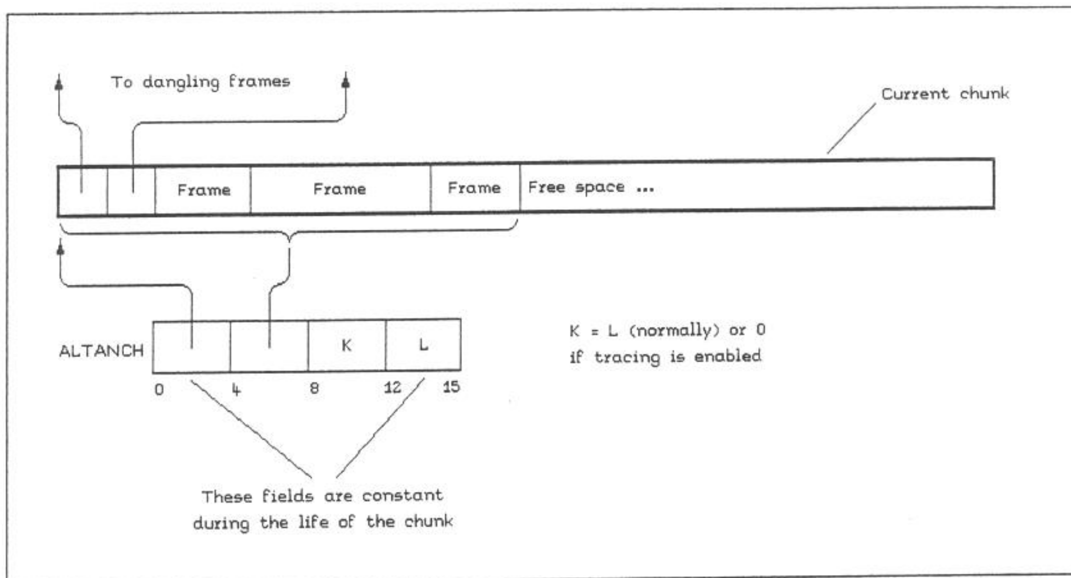


Fig. 5. Anchor for alternative implementation (see section 6 of paper). As before, "L" stands for "length of chunk".

```
*
*   ADVANCE is a subroutine which supplies
*   a stack frame of the specified length.
*
*      Call is:
*          BAL  R15,ADVANCE
*
*      On entry:
*          R2 = required frame length
*           (must not be 0)
*
*      On return:
*          R0 has an undefined value
*          R1 = addr of stack frame
*
ADVANCE  LM    R0,R1,ANCHOR+4  Ending addr, "available"
         SLR   R0,R1           Length of the free space

         CLR   R0,R2           Sufficient room in chunk?
         LA    R0,0(R1,R2)     Tentative new "available"
         ST    R0,ANCHOR+8     Optimistically set anchor
         BNLR  R15             Return very quickly if so

         ST    R1,ANCHOR+8     Repair the damaged anchor

    (Arrive here if there is insufficient free space
    in current chunk, in order to obtain new chunk.)

    Save necessary registers in private save area.

    If current chunk is empty (i.e. if "available"
    pointer is equal to ending address of prefix),
      copy prefix of current chunk to 1st two words
       of anchor: this describes prior chunk if any;
      return entire current chunk to memory alloc-
       ator, including its prefix;
    else
      return unused tail of current chunk to memory
       allocator.
      copy "available" pointer to ANCHOR+4: this
       is ending address of last surviving frame.

    Obtain new chunk from memory allocator: new chunk must
    be bigger than prior chunk (whose length still resides
    at ANCHOR+12), and big enough for the requested frame.
    Terminate program if insufficient memory is available.

    Copy 1st two words of anchor to prefix of new chunk:
    these contain beginning and ending address of sur-
    viving frames from prior chunk.

    Update anchor to describe new current chunk.

    Reload saved registers and go back to ADVANCE.
```

```
*
*   RETREAT is a subroutine which vacates the
*   memory occupied by the given stack frame.
*
*      Call is:
*          BAL  R15,RETREAT
*
*      On entry:
*          R1 = addr of stack
*           frame to be vacated
*
*      On return:
*          R1 has undefined value
*
RETREAT  ST    R1,ANCHOR+8     Optimistically set anchor
         SL    R1,ANCHOR       Offset of the given frame
         CL    R1,ANCHOR+12    Is frame in current chunk?
         BLR   R15             Return if so (common case)

         L     R1,ANCHOR+8     Regain given frame address
```

```
(Arrive here if frame being vacated does not reside
in current chunk, in order to return given frame to
memory allocator as isolated piece.)

Save necessary registers in private save area.

Set "available" pointer to ending address of prefix
in current chunk: this repairs damage done above when
optimistically setting anchor.  (Since given frame be-
longs in prior chunk, current chunk must be empty, so
free space starts immediately after prefix.)

Obtain beginning and ending address of prior chunk
from prefix of current chunk, and derive length of
given frame (i.e. ending address of chunk - given
address of frame).

If this is only surviving frame from prior chunk
(i.e. if frame address = ending address of prefix),
  copy prefix of prior chunk to prefix of current
   chunk: this describes the penultimate chunk if
   any;
  return entire remains of prior chunk to memory
   allocator, including its prefix;
else
  return given frame to memory allocator as an
   isolated piece;
  store given frame address in 2nd word of prefix
   in current chunk: this is new ending address
   of last surviving frame in prior chunk.

Reload saved registers and return to caller.
```

5. NOTES AND OBSERVATIONS

1. The short path in "advance" contains 6 instructions, or 7 on a RISC which does not support a general "Load Multiple" instruction (see for example [3]). This is the same number as in the corresponding sequence for Conventional Method 2.

   The short path for "retreat" contains 4 instructions, or 6 on a RISC which does not support Addition or Comparison with the contents of memory. I believe this is one more instruction than the corresponding sequence for Conventional Method 2, or two more on a RISC.

   (In Conventional Method 2 the given frame address can be compared for equality with the ending address of the prefix in the current chunk -- which can easily be included in the anchor -- and a subtraction is therefore not necessary.)

   So in the common case the extra cost of the self-adjusting stack is one extra instruction when vacating a frame (or 2 on a RISC). The advantage is that the amortized performance will often be better. Also the performance will be more reproducible and exhibit a smaller variance.

2. The self-adjusting stack requires that the system memory allocator be capable of deallocating the tail of a piece of allocated memory without affecting the contents (or the address) of the surviving head. This does not usually present any difficulty in S/370 operating systems; but there are varying traditions in other environments. In "malloc", for example (which is the allocator used by the C runtime environment), a piece of allocated memory retains an identity and must be handled as a unit; and while it is possible to "reallocate" an existing piece and thus increase or decrease its length, the address may be changed as a side-effect. This presents serious difficulties when implementing a stack.

   Some implementations of "malloc" guarantee that the address will not be changed during a reallocation provided the new length is less than the old (e.g. AIX release 3.2 on the RS/6000: see [4]). This permits a self-adjusting stack to be implemented.

3. The implementation given above permits the use of the following idiom, which is useful in some contexts:

   Obtain stack frame of ample size;

prepare contents and incidentally
 derive the exact size required;

vacate the unwanted tail.

In terms of the actual code, this might be written thus:

```
LA    R0,128          Obtain ample stack frame
BAL   R15,ADVANCE
...
LA    R2,80(,R1)      Address of unwanted tail
ST    R2,ANCHOR+8     Reset "available" pointer
```

4. Sometimes it is convenient if the linkage code in a program can vacate several stack frames at once. This may happen after certain exceptions, or if a "goto" statement passes control from a deeply nested subroutine to a label that resides in a routine that is "higher" in the calling chain.

   It turns out that the "retreat" subroutine shown above could easily be adapted to handle this -- without affecting the fast path. R1 on entry would now contain the address of the oldest frame to be vacated. If the given frame resides in the current chunk, "retreat" will handle the situation correctly without ever realizing that something odd is going on. If the given frame resides in a prior chunk, "retreat" must identify this chunk (which will not necessarily be the immediately preceding one), and deallocate its tail. It must also deallocate all the intervening chunks in toto. This involves chaining back through the prefix areas of the prior chunks (details omitted).

5. My colleague Stephen Watt made the following observation and the following suggestion.

   Consider a program which is not iterative but is deeply recursive. Such a program has two phases. First it "winds up", during which it acquires stack frames (without vacating any); then it "winds down", during which it vacates these frames (without acquiring new ones). At the peak, the stack depth may be large (perhaps thousands).

   When the program starts to wind down, a substantial proportion of its stack frames will lie outside the current chunk. The exact proportion depends on the amount by which each new chunk is bigger than the prior one, and the fullness of the current chunk at the transition -- but in any event there may be many such frames, perhaps occupying only a few words each. It would be foolish to return each of these frames individually to the memory allocator, since

the cost of doing so might exceed the entire cost of everything else in the program.

This problem can be avoided by introducing a threshold for the minimum frame size to be deallocated as an individual piece. Recall that the "retreat" subroutine, when passed a frame that lies in a prior chunk, computes the length of the frame by subtracting the given frame address from the ending address of the last surviving frame in this (same) prior chunk. Well, if this length is less than the threshold, the deallocation can be deferred (and of course the ending address of the last surviving frame is not adjusted). Then when the next frame from the same chunk is vacated, its apparent length will include that of the frame whose deallocation was deferred. Eventually the apparent length will reach the threshold (or the last surviving frame in the chunk will be vacated), and then the accumulated frames will be deallocated as one piece.

The value of the threshold is not at all critical. Practical values might perhaps lie in the range 4K bytes to 64K bytes, depending on the size of the memory and the performance of the allocator. The maximum length of memory that can at any time be tied up due to deferred deallocation is bounded by the value of the threshold.

6. In the code sequences above, I have not worried about alignment issues. In practice, each stack frame must be aligned on a word boundary or a double-word boundary (depending upon the underlying machinery). If the caller of "advance" always requests a length that is a multiple of this alignment, no extra work is required. But if the caller is allowed to request a length of (say) 11 bytes, a few extra instructions must be inserted at the beginning of "advance" in order to round up the given length.

6. ALTERNATIVE IMPLEMENTATION WHICH PERMITS DYNAMIC TRACING

The implementation described above was selected for its speed, i.e. for the minimum number of instructions in the short paths.

While I was preparing this paper, it struck me that there is an alternative implementation, which requires only one more instruction in the fast path for "advance", and which has the following interesting property. It permits dynamic tracing of "call" and "return" statements with no extra overhead (other than the one instructions), except when tracing is actually enabled. Furthermore, tracing can be enabled and disabled dynamically, and asynchronously, while the program is running.

Since program designers are often willing to sacrifice a little speed in order to provide optional "call" tracing, it is interesting to find that, when a self-adjusting stack is used, this facility can be provided for a very modest cost.

The trick, of course, is to combine the test for tracing with the existing size check (in "advance") and the existing address check (in "retreat"). Then when tracing is disabled, and there is nothing else unusual, the short path will be followed. But when tracing is enabled, and/or something unusual is afoot, a longer path will be followed. The longer path is more complicated than in the first implementation -- but its performance is of no great importance.

For this alternative implementation we will use a slightly different anchor, as shown in Fig. 5. The alternative anchor does not contain an "available" pointer; instead it contains the offset of the free space. (Note however that the chunk prefix is the same as before, and still contains the beginning and ending address of the prior chunk.)

Normally, the third word of the anchor (which is shown as "K" in Fig. 5) contains the same value as the fourth word (which is shown as "L" and contains the length of the current chunk).

To enable tracing, "K" is set to 0. To disable tracing, it is reset to the contents of the fourth word. These changes may be made by an interruption handler while the program is running -- and possibly while the anchor is in the process of being updated for a new chunk (see below for details).

Note that the alternative "advance" subroutine uses slightly different linkage rules. The required length is now passed in R0 instead of in R2. (This is unadulterated expediency: I simply want the short path to come out as prettily as possible.)

Here, then, are the alternative subroutines.

```
*
*   ALTADVA ("alternative advance") is a subroutine
*   which supplies a stack frame of the specified
*   length, and traces the call statement if trac-
*   ing is enabled.
*
*      Call is:
*          BAL   R15,ALTADVA
*
*      On entry:
*          R0 = required frame length
*            (must not be 0)
*
*      On return:
*          R1 = addr of stack frame
*          R2 has an undefined value
*
ALTADVA  LM    R1,R2,ALTANCH  Chunk addr, free offset
         ALR   R1,R2          Tentative stack frame addr

         ALR   R2,R0          Tentative new free offset
         ST    R2,ALTANCH+4   Optimistically update anchor
         BC    3,*+10         Goto "SLR R2,R0" if no room

         CL    R2,ALTANCH+8   Complete the size check
         BNHR  R15            Return quickly if room

         SLR   R2,R0          Regain available offset
         ST    R2,ALTANCH+4   Repair damaged anchor

    (Arrive here if there appears to be insufficient
    free space in current chunk, in order to obtain
    new chunk and/or handle tracing, as appropriate.)

    Save necessary registers in private save area.

    If ALTANCH+8 = 0, repeat the assembler sequence
    above using ALTANCH+12 in place of ALTANCH+8, and
    "BNH tracecall" in place of "BNHR R15". This dis-
    tinguishes between (a) current chunk is adequate
    and tracing is enabled, and (b) current chunk is
    inadequate (with tracing enabled or disabled).

    (Arrive here if there is insufficient free space
    in the current chunk.)

    If current chunk is empty (i.e. if offset
    of free space is equal to length of prefix),
      copy prefix of current chunk to 1st two words
        of anchor: this describes prior chunk if any;
      return entire current chunk to memory allocat-
        or, including its prefix;
    else
      return unused tail of current chunk to memory
        allocator.
      temporarily set ALTANCH+4 to ending address of
        surviving part of current chunk (see below).
```

Obtain new chunk from memory allocator: new chunk must be bigger than prior chunk (whose length still resides at ALTANCH+12), and big enough for the requested frame. Terminate program if insufficient memory is available.

Copy 1st two words of anchor to prefix of new chunk: these contain beginning and ending address of surviving frames from prior chunk.

Update anchor to describe new current chunk. This requires synchronization with interruption handler that clears or sets ALTANCH+8 to enable or disable tracing. The following sequence shows how this can be done with Compare-and-Swap (without looping). When control reaches the Compare-and-Swap instruction (CS), ALTANCH+8 will contain one of the following possible values:

0	Tracing enabled
Length of old current chunk	Tracing disabled
Length of new current chunk	Tracing has just been disabled

Here is the update sequence:

```
Set ALTANCH, ALTANCH+4 for new current chunk
Set ALTANCH+12 to length of new current chunk

Load R3 with the length of the old current chunk
Load R4 with the length of the new current chunk

CS   R3,R4,ALTANCH+8   Update ALTANCH+8 if stale
```

(This last instruction can be read as follows. "If R3 equals the value at ALTANCH+8, store R4; else place the value from memory in R3, without changing the contents of memory". The entire thing is executed as an atomic operation.)

Reload saved registers and go back to ADVANCE.

tracecall:

Save R1 (address of new stack frame).

Trace the call (details are omitted).

Reload saved registers, reload R1 (the new stack frame address), and return to caller.

```
*
*   ALTRETR ("alternative retreat") is a subroutine
*   which vacates the memory occupied by the given
*   stack frame, and traces the "return" statement
*   if tracing is enabled.
*
*      Call is:
*          BAL  R15,ALTRETR
*
*      On entry:
*          R1 = addr of stack frame to be vacated
*
*      On return:
*          R1 has undefined value
*
ALTRETR  SL    R1,ALTANCH      Tentative new avail offset
         ST    R1,ALTANCH+4    Optimistically set anchor
         CL    R1,ALTANCH+8    Is frame in current chunk?
         BLR   R15             Return if so (common case)

         AL    R1,ALTANCH      Regain given frame address
```

```
   (Arrive here if frame being vacated does not appear
   to reside in current chunk, in order to return given
   frame to memory allocator and/or handle tracing, as
   appropriate.)

   Save necessary registers in private save area.

   If contents of ALTANCH+4 < contents of ALTANCH+12,
   goto "tracereturn", since given frame resides in
   current chunk after all.

   Set ALTANCH+4 to length of prefix: this repairs damage
   done above when optimistically setting anchor.  (Since
   given frame belongs in prior chunk, current chunk must
   be empty, so free space starts immediately after pre-
   fix.)

   Obtain beginning and ending address of prior chunk
   from prefix of current chunk, and derive length of
   given frame (i.e. ending address of chunk - given
   address of frame).

   If this is only surviving frame from prior chunk
   (i.e. if given frame address = ending address of
   prefix),
     copy prefix of prior chunk to prefix of current
      chunk: this describes the penultimate chunk if
      any;
     return entire remains of prior chunk to memory
      allocator, including its prefix;
   else
     return given frame to memory allocator as an
      isolated piece;
     store given frame address in 2nd word of prefix
      in current chunk: this is new ending address
      of last surviving frame in prior chunk.
```

If ALTANCH+8 ¬= 0, reload saved registers and return to caller (else fall through to "tracereturn").

tracereturn:

Trace return statement (details are omitted).

Reload saved registers and return to caller.

Historical note

The self-adjusting stack was devised in 1989 and has been eploited in the YMS editor "Ed" since 1992.

REFERENCES

[1] Virtual Machine/System Product, Exec 2 Reference, SC24-5219, IBM (1980)

[2] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

[3] May, C. et al. (editors), The PowerPC Architecture, 2nd edition, IBM and Morgan Kaufmann (1994)

[4] AIX Version 3.2, Technical reference, Base operating system and extensions, Vol. 1, SC23-2382-02, IBM (1993)

CJS, 1995-01-05.

CALENDRIC PROGRAMMING

Joint work with James H. Davenport* and Paul R. Kosinski†

1. BACKGROUND

"The ancient Roman year commenced with March, as is indicated by the names September, October, November, December, which the last four months still retain. July and August, likewise, were anciently denominated Quintilis and Sextilis ..." (from [1]).

At the time of Julius Caesar, the months contained 29 and 30 days alternately. Every second year (with some exceptions), an additional short month was inserted between February 23 and February 24 in order to keep the civil and solar years more or less in step.

In 47 B.C. Julius Caesar revised the lengths of the months, as shown in Fig. 1a, eliminated the occasional short month, and decreed that there should be one extra day in February every 4 years. Note that each alternate month contained 31 days. It was sensible and regular.

Unfortunately, in recognition of this calendric contribution, Quintilis was renamed in honour of Julius Caesar. This was harmless in itself, but it had the following regrettable repercussion. Some years later, Augustus resolved a minor confusion concerning the extra day in February; and following precedent, Sextilis was renamed for him. But Augustus was miffed that "his" month was shorter than the one named after Julius; so the months were resized, as shown in Fig. 1b, to the lengths that we still use.

The next part of the story that concerns us here took place in the 16th century, when Pope Gregory XIII made a correction to the frequency of leap years, and established the calendar we follow today. According to his rule, February in year Y contains the following number of days:

$$D = 28 + (Y \% 4 \ \textbf{xor} \ Y \% 100 \ \textbf{xor} \ Y \% 400),$$

where "P % Q" evaluates to 1 if P is exactly divisible by Q (or to 0 otherwise).

---

*Present address: School of Mathematical Sciences, Bath University, Bath, England

†Present address: Digital Equipment Corporation, 334 South Street, Shrewsbury, Mass.

Month	Days
March	31
April	30
May	31
June	30
July	31
August	30
September	31
October	30
November	31
December	30
January	31
February	29 or 30

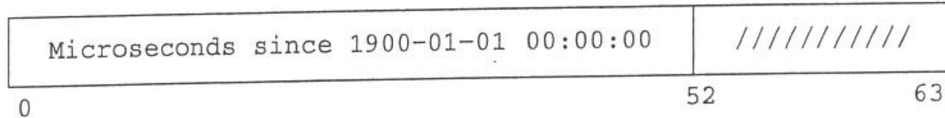
Fig. 1a. The lengths of the months as defined under Julius Caesar. The picture uses an origin of March in order to expose the repeating pattern and to place the variable month at the end.

Month	Days
March	31
April	30
May	31
June	30
July	31
August	31
September	30
October	31
November	30
December	31
January	31
February	28 or 29

Fig. 1b. The lengths of the months as redefined under Augustus.

2. COMPUTER TIME STAMPS

Most computers maintain a clock which can be read by the program. In S/370, for example, a clock reading comprises a 64-bit unsigned integer in which the high-order 52 bits represent the number of microseconds that have elapsed since A.D. 1900-01-01 00:00:00.000000 GMT (see [2]):



The low-order 12 bits do not necessarily represent accurate fractions of microseconds. (Some of these bits may be set in order to guarantee that clock readings on any machine are unique within the epoch.)

It is usually convenient to store internal time stamps in the format supplied by the clock. When recording the time-of-last-change for a file, for example, the current clock reading can simply be stored in the file directory. This is more compact than a human-readable date and time, and for most programming purposes it is more convenient.

There is however a potential performance problem, e.g. when listing files. When files are listed, their names, sizes and time stamps are obtained from the file directory and laid out in human-readable format. The results are placed in (say) an edit file. The program that handles this should certainly be capable of formatting the information for thousands of files in a fraction of a second.

Similar requirements exist for records in data bases, event logs, etc.

It is therefore important that conversion of an internal time stamp to a human-readable date and time should run fast.

3. OBTAINING MONTH AND DAY-IN-MONTH FROM DAY-IN-YEAR

Suppose we have computed the day-of-the-year, and we want the month and the day-of-the-month.

At first glance you might think that Augustus made things difficult for us. If the months had retained their lengths as assigned under Julius Caesar, it would be possible, by adjusting the origin to March 1, to determine the month-pair, and the day in the month-pair, from the quotient and remainder of an integer division (by 61). Thence the precise month and the day-of-

the-month would emerge after one comparison and a conditional subtraction -- without any looping.

But as it happens, Augustus did us no great harm. If you examine Fig. 1b, you will see that, using the same origin of March 1, the year is divided into just under 2.4 regular 5-month intervals, each of which exhibits the following pattern:

31, 30, 31, 30, 31.

As a result, a division operation could be used to select the correct 5-month interval and also the day within that interval. Then we would be left with the problem of choosing the correct month in the group of 5, and the day in that month, using the technique that worked for Julius' calendar.

But things get even better. By choosing an appropriate "quasi-reciprocal", it is possible to avoid the division operations and perform multiplications instead. These run substantially faster on most machines.

Division by a constant can be converted to multiplication by a quasi-reciprocal as follows. Suppose we wish to divide X (a variable) by U (a constant). We can instead multiply X by V, where:

$$V = 2^k/U \quad \text{approximately}$$

and extract the (w-k) high-order bits, where w is the total number of bits in the result. There is some flexibility in the value of k, but it should be chosen so that:

$$\lg U < k < w - \lg Q,$$

where Q is the maximum quotient.

In some situations the k low-order bits of the result are also useful. These comprise the remainder, in the form of a fraction.

This trick can be used only when the range of values for X is suitably restricted -- but that is indeed the situation we face when handling calendric numbers.

The word "approximately" is of course significant. Except in the uninteresting case where U is a power of 2, V cannot be exact. It is therefore necessary to find an approximation for V that produces the desired result for all relevant values of X. A thorough discussion of this topic in given in [3].

In the present application X is the day-of-the-year, so there are only 366 possible values, and it is easy to find a suitable value for V by performing a search.

While making this search, my colleagues and I found to our delight that Augustus did us no harm at all! By adjusting the origin by half a day it is possible to find a constant which breaks up the day-of-the-year into the month-of-the-year and the day-of-the-month, just as easily as if Julius had held sway. In the end we need one multiplication by $2^{16}/61$ approximately, and one multiplication by 61 exactly (the number of days in a 2-month interval), along with a little bookkeeping. This is how it can be written in S/370 assembly language:

```
*
*    R1 = day of year (0 = Mar1, 1 = Mar2, ...).
*
        LA    R1,1(R1,R1)      Adjust origin half a day
        MH    R1,=Y(1071)      Mult by 2**16/61 approx

        LR    R0,R1            Extract high-order bits
        SRL   R0,16            R0 := month (0,1,2,...)

        N     R1,=A(2**16-1)   Extract low-order bits
        MH    R1,=Y(61)        Mult by days-in-2-months
        SRL   R1,17            Discard the surplus bits
*
*    Now R0 = month number (0=Mar,1=Apr,...),
*    and R1 = day-of-the-month (0,1,2,3,...).
*
```

The second multiplication (by 61) can be replaced if desired by a shift and three subtractions (since 61 = 64-3).

4. OBTAINING YEAR AND DAY-IN-YEAR FROM DAYS-SINCE-1900

The previous section assumes that we have already computed the day-of-the-year. This section describes how to do that.

The epoch of the S/370 clock runs from A.D. 1900-01-01 00:00:00.000000 to A.D. 2042-09-17 23:53:47.370495. This includes one centennial year that is not a leap year (1900), and one centennial year that is a leap year (2000). The fact that the latter is a leap year simplifies matters considerably, for it means that after February 1900 each four-year interval contains the same number of days.

The best way to handle this part of the calculation is to bias the origin to 29 February 1896. Then every four-year interval will contain the same number of days, and the only remaining nuisance will be that dates in January and February 1900 will be off by one.

The same trick can be used as before to avoid a division operation. Here however we need more than 32 bits of information from the first multiplication, so we must use a fullword

multiplication instruction, which supplies a 64-bit result in two consecutive registers:

```
*
*   R1 = days since 1 Jan 1900 (0,1,2,...)
*
        LA    R1,(4*365+1)-(31+28)+1(,R1) Orig = 1896-02-29
        M     R0,=A(11758976)      Times 2**32/365.25 approx

        SRL   R1,12                Multiply the remainder
        MH    R1,=Y(2922)           by 365.25 (exactly)
        SRL   R1,23

        S     R0,=A(4)             Years since 1900-03-01
        BNL   *+6                  Skip if after Feb 1900
        BCTR  R1,0                 Adjust for Jan-Feb 1900
*
*   Now R0 = whole years since 1900-03-01 (-1,0,1,...),
*   and R1 = day-of-the-year (0 = Mar1, 1 = Mar2, ...).
*
```

5. CONVERSION FROM S/370 CLOCK TO YYYY-MM-DD HH:MM:SS.MMMMMM

Appendix A contains a S/370 subroutine that does the entire job of converting a S/370 clock reading to a human-readable date and time. It employs the methods described above, although the detailed tactics are different -- in order, for example, to obtain "01" for January and "03" for March (rather than treating January as month 10 and March as month 0 as in the fragments above).

When deriving the time-of-day (in hours, minutes, seconds), division operations can be replaced by multiplications in the same way as when computing the date. It is a little less devious in this case, since the time units are uniform.

In the end, the subroutine contains 60 instructions (including those which save and restore the caller's registers). There are no branches except for the "return" statement. Among the potentially expensive instructions there are 2 divisions, 12 multiplications, 3 instances of the "Convert to Decimal" instruction and 3 instances of "Unpack". If desired, 8 of the 12 multiplications can easily be replaced by shifts and subtractions; this is advantageous on machines that do not support fast multiplication (and it is almost never harmful).

On an IBM 9021 mainframe, model 941, the execution time of the subroutine as given is under 1.3 microseconds.

Appendix B contains informal descriptions of the S/370 "Convert to Decimal" and "Unpack" instructions for readers who are not familiar with them.

Historical note

The method described here for converting clock readings to human-readable date and time has been used in YMS since 1979-11-05 10:34:26.402003 GMT.

APPENDIX A

Subroutine to convert S/370 clock reading to date and time

```
*
*       GREG is a subroutine which converts a S/370
*       clock reading to date and time in characters.
*
*       Call is:
*             L      R15,=A(GREG)
*             BALR   R14,R15
*
*       On entry:
*             R0 = addr of 64-bit clock reading
*             R1 = addr of a 26-byte reply area
*
*       On return:
*             The date and time are stored in
*              the reply area, laid out thus:
*
*                 ....:....1....:....2....:.
*                 yyyy-mo-dd hr:mi:ss.mmmmmm
*
*              where "mmmmmm" represents the
*              residual microseconds (within
*              the second)
*
*             R0-R15 are unchanged from entry
*             The condition code is undefined
*
*       Notes:
*
*       1.    The constant multipliers from which the year
*             and the day-in-the-year are derived produce the
*             correct results through 2100-02-28.  If, there-
*             fore, the S/370 clock were to be extended (with
*             an additional high-order bit), this subroutine
*             could easily be modified to use it. After 2100-
*             02-28, the results will initially be off by one
*             day.  (The behaviour beyond this date has not
*             been investigated.)
*
*       2.    The result area (addressed by R1) is not mod-
*             ified until after the complete answer has been
*             computed.  Therefore the areas addressed by R0
*             and R1 may overlap.
*
*       3.    This routine requires a 6-word register save
*             area and a double-word named SCRATCH. Address-
```

```
*              ability to these is regarded as a packaging
*              issue and is not specified here.
*
GREG     DS    0H              (The entry point)
         USING *,R15           Local program cover

         STM   R2,R7,...       Save needed registers

         LR    R2,R0           Addr of given time stamp
         LM    R2,R3,0(R2)     Put the value into R2,R3
         LR    R5,R3           Save the low-order part

         SRDL  R2,12+12        Remove low-order bits
         D     R2,=A((24*60*60/64)*(1000000/64))

         LR    R4,R2           Restore discarded bits
         SLL   R5,8             and hence set R4,R5 to
         SRDL  R4,8+12          microsecs since midnight

         D     R4,=A(1000000) R5 = secs since midnight
         LR    R7,R4           R7 = remaining microsecs
*
*      Derive time-of-day in hours, minutes and seconds.
*      At this point:
*
*            R3 = whole days since 1900-01-01 (0,1,...)
*            R5 = whole secs since midnight (0,1,...)
*            R7 = the residual microseconds (0,1,...)
*
         M     R4,=X'00123460'  Times 2**32/3600 approx
         LR    R6,R4           Whole hours since midnight
         MH    R6,=Y(1000)     (Prepare for the CVD instr)

         SRL   R5,1            Make sure remainder is +ve
         M     R4,=A(2*60)     R4 = whole mins since hour
         ALR   R6,R4           Add in the residual minutes
         MH    R6,=Y(1000)     (Prepare for the CVD instr)

         SRL   R5,1            Make sure remainder is +ve
         M     R4,=A(2*60)     R4 = whole secs since min
         ALR   R6,R4           Add in the residual secs
*
*      Derive the year number and the day-of-the-year,
*      by using an origin of 1896-02-29. At this point:
*
*            R3 = whole days since 1900-01-01 (0,1,...)
*            R6 = 10**6 x hour + 10**3 x minute + second
*            R7 = the residual microseconds (0,1,2,...)
*
         LA    R5,(4*365+1)-(31+28)+1(,R3)  Orig = 1896-02-29
         M     R4,=X'00B36D80'  Multiply by 2**32/365.25 approx
         SL    R4,=A(4)          Yrs since 1900-03-01 (-1,0,...)

         SRL   R5,3            Mult remainder by 365.25 exactly
         LA    R3,8*365+2
         MR    R2,R5           R2 = tentative day (0=Mar1,...)

         LR    R5,R4           Years since 1900-03-01 (-1,0,...)
         SRL   R4,31           R4 = 1 if it is before 1900-03-01
         SLR   R2,R4           Decr day-of-year if Jan,Feb 1990
```

```
*
*     Derive the year (1900,1901,...), month (01,02,...)
*     and day-of-the-month (01,02,...).  At this point:
*
*           R2 = the day-of-the-year (0=Mar1,1=Mar2,...)
*           R5 = whole years since 1900-03-01 (-1,0,...)
*           R6 = 10**6 x hour + 10**3 x minute + second
*           R7 = the residual microseconds (0,1,2,...)
*
         ALR   R2,R2            Perform scaling (fishy business)
         LA    R3,4*61+3(R2,R2)  Fudge origin, scale some more
         M     R2,=X'002C9E00'   Mult by 2**32/(24x61) approx
         LA    R5,1900(R2,R5) Add century, incr if Jan or Feb
         MH    R5,=Y(1000)      (And prepare for the CVD instr)

         SRL   R3,1             Make sure the remainder is +ve
         M     R2,=A(2*12)      R4=month (0=Jan,1=Feb,2=Mar,...)
         LA    R5,1(R2,R5)      Ensnare month (1=Jan,2=Feb,...)
         MH    R5,=Y(1000)      (And prepare for the CVD instr)

         SRL   R3,1             Make sure the remainder is +ve
         M     R2,=A(61)        R2 = day of the month (0,1,...)
         LA    R2,1(,R2)        Adjusted day of month (1,2,...)
         ALR   R5,R2            Into low bits of R5 (see below)
*
*     Store the answer in the caller's reply area.
*     At this point:
*
*           R5 = 10**6 x year + 10**3 x month + the day
*           R6 = 10**6 x hour + 10**3 x minute + second
*           R7 = the residual microseconds (0,1,2,...)
*
*     Also:
*           R0,R1 are unchanged from entry
*
         CVD   R5,SCRATCH       Convert the date to decimal
         OI    SCRATCH+7,15     (And fix up the lousy sign)
         UNPK  0(10,R1),SCRATCH(8)  YYYY*MO*DD (*=junk)
         MVI   4(R1),C'-'       Insert pretty separators
         MVI   4+3(R1),C'-'

         MVI   10(R1),C' '      Separate date from time

         CVD   R6,SCRATCH       Convert time to decimal
         OI    SCRATCH+7,15
         UNPK  11(8,R1),SCRATCH(8) HR*MI*SS (*=junk)
         MVI   11+2(R1),C':'         Smooth and polish
         MVI   11+2+3(R1),C':'

         MVI   19(R1),C'.'      Separate secs,microsecs

         CVD   R7,SCRATCH       Handle the microseconds
         OI    SCRATCH+7,15
         UNPK  20(6,R1),SCRATCH(8)

         LM    R2,R7,...        Reload caller's regs
         BR    R14               and return happily
```

APPENDIX B

Informal description of the S/370 CVD and UNPK instructions

Convert to Decimal

CVD gpr,loc — Convert the 2's-complement value in the given GPR to an 8-byte "packed" decimal number and store the result at the given location. A packed decimal number contains 2 digits per byte except for the rightmost digit (which shares a byte with an encoded sign nibble).

Unpack

UNPK loc1(length1),loc2(length2)

Unpack the packed decimal number that resides at loc2 (having length length2) and store the result at loc1 (with length length1). An unpacked decimal number occupies one byte per digit. It is human-readable except possibly for the rightmost digit (which shares its byte with the encoded sign nibble).

REFERENCES

[1] Encyclopaedia Britannica, 11th edition, Vol. IV, "Calendar", Cambridge University Press (1910)

[2] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

[3] Warren, Henry S., Jr., Changing division by a constant to multiplication in two's-complement arithmetic, IBM Research Report RC 18601 (1992)

CJS, 1995-04.

FIVE CODING TECHNIQUES

This paper comprises notes on the following coding techniques:

1. Page crossing
2. The navel pointer
3. Interleaved arrays
4. Eccentric loop control
5. Deriving a single-bit mask

PAGE CROSSING

In the low-level parts of operating systems, the following situation quite often arises:

> A program knows the virtual address and the length of an area of memory in some address space, and it needs:
>
> a. to find out whether the area involves more than one page frame, and if it does:
>
> b. the length that projects beyond the first page boundary.

This can arise when copying data from one address space to another, or when performing an IO operation to or from an address space that is mapped to discontiguous page frames.

There is a beautiful way of handling this, which works elegantly on many computers.

Suppose Rx contains the real or virtual address of the area (it does not matter which), and Ry contains the length of the area. Using S/370 notation (see [1]), and the symbol PAGE for the page-size, we can write:

```
O      Rx,=A(-PAGE)      -Length to end of 1st page
ALR    Rx,Ry             Hence the length that spills
BO     CROSSES           Branch if more than one page
```

The conditional branch ("BO") is taken if (a) the "Add logical" instruction sets carry and (b) the result (in Rx) is non-zero. When the branch is taken, Rx contains the length that spills beyond the first page.

This method depends upon the fact that the page-size is a power of 2, and that addresses are assigned (both in real and virtual address spaces) so that pages are aligned on a multiple of their size. (The proof will not be given here; it emerges straightforwardly from the properties of binary number representation.)

Note that, despite the comment on the OR instruction above ("O"), the address and the length are both treated as unsigned numbers, and their high-order bit may be 1.

Architectural note

Unfortunately some modern machines do not support any equivalent of the "BO" instruction, and require the carry bit and

the arithmetic result to be tested separately. On the PowerPC, for example (see [2]), six instructions are needed, e.g. thus:

```
li     Rz,-PAGE        # -(Length per page frame)
or     Rx,Rx,Rz        # -Length to end of 1st page
addc.  Rx,Rx,Ry        # Hence the length that spills
mcrxr  2               # Place carry bit in cond reg
bne    cr2,$+8         # Skip if only 1 page touched
bne    CROSSES         # Branch if more than 1 page
```

Historical note

This technique was devised circa 1976 and has been used extensively in EM since then.

References

[1] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

[2] May, C. et al. (editors), The PowerPC Architecture, 2nd edition, IBM and Morgan Kaufmann (1994)

CJS, 1994-12.

THE NAVEL POINTER

There are various ways of terminating a linked list of control blocks. A popular convention is to store a null pointer (or 0) in the last forward pointer field. In some situations, however, it can be more convenient to store a pointer that points to itself.

I call this a "navel" pointer, by allusion to the object of contemplation.

Assume that the pointer field resides at the beginning of the control block. A navel pointer can then be set without requiring any registers other than the one that contains the address of the block itself. Let us use S/370 assembler notation (see [1]), and suppose that R1 contains the address of a control block. Then the following instruction terminates the list:

```
ST    R1,0(,R1)       Terminate the linked list
```

It is equally easy to test whether the current block is the last in the list. Suppose as before that R1 contains the address of the block:

```
CL    R1,0(,R1)       The last block in the list?
BE    NOMORE          Branch if so (add another)
```

Finally, it is easy to find the end of the list. This too requires only one register. Suppose R1 contains the address of any block in the list -- possibly the last one; then the following three-instruction loop finds the last block in the list and places its address in R1:

```
L     R1,0(,R1)       Chain to next block maybe
CL    R1,0(,R1)       The last block in the list?
BNE   *-8             Loop if not (back to Load)
```

(As a general rule, it is preferable to record the address of the last block in a list separately rather than searching for it. There are however situations where the chain is known to be short and it is not worth maintaining a separate datum. An example arises in file systems, when handling the "levels" of a file tree, where it may be known _a priori_ that the tree depth does not exceed 4 or 5.)

Architectural note

Some of the advantages of the navel pointer disappear on a RISC, where memory accesses are confined to Load and Store

instructions, and a second register is always required to make a comparison.

Historical note

Navel pointers have been used in the EM file system since 1975.

Reference

[1] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

CJS, 1995-01-15.

INTERLEAVED ARRAYS

Suppose two integer arrays of equal size are allocated dynamically and enlarged as necessary to handle the most demanding situation that arises during execution. Consider for example the job of constructing the "delta2" table for a Boyer-Moore string search in a text editor (see [1], appendix). When the editor begins execution, a modest area of memory is allocated, sufficient for two integer arrays containing (say) 32 entries each. When this proves insufficient, a larger area is established, and retained thereafter (unless there is a shortage of memory). The array origins, and their current size "k", can be maintained in a static anchor, as depicted in Fig. 1.

If the program is to exhibit the best possible performance, it must, when using the arrays, devote a separate register to each array origin. Assume that the element width is 4 bytes; then on a byte-addressed machine a "fetch" operation from (say) B[j] involves the following steps:

> derive the offset of B[j] from B[0] by multiplying j by the array width (4 in this case);
>
> fetch word residing at B[0] + derived offset.

If the program were to maintain only one pointer (say to A[0]), then every time it required an element of B, it would need to add the array length (4×k) to the offset. I am here assuming that the machinery allows at most two registers to participate directly in the formation of an operand address.

The need for two separate "origin" registers can be obviated by interleaving the arrays, as shown in Fig. 2. Now the program can maintain a single pointer, to A[0]; and a "fetch" operation from B[j] can be handled as follows:

> derive the offset of B[j] from B[0] by multiplying j by the combined array width (8 in this case);
>
> fetch word residing at A[0] + 4 + derived offset.

A fetch from A[j] is similar (with the 4 omitted).

The same trick can be used with three or more arrays. It is not confined to integers, or even to arrays which have the same width. The only requirement is that the arrays all have the same number of entries.

Obviously this technique can be used only if the arrays are private to the program that creates and uses them. An

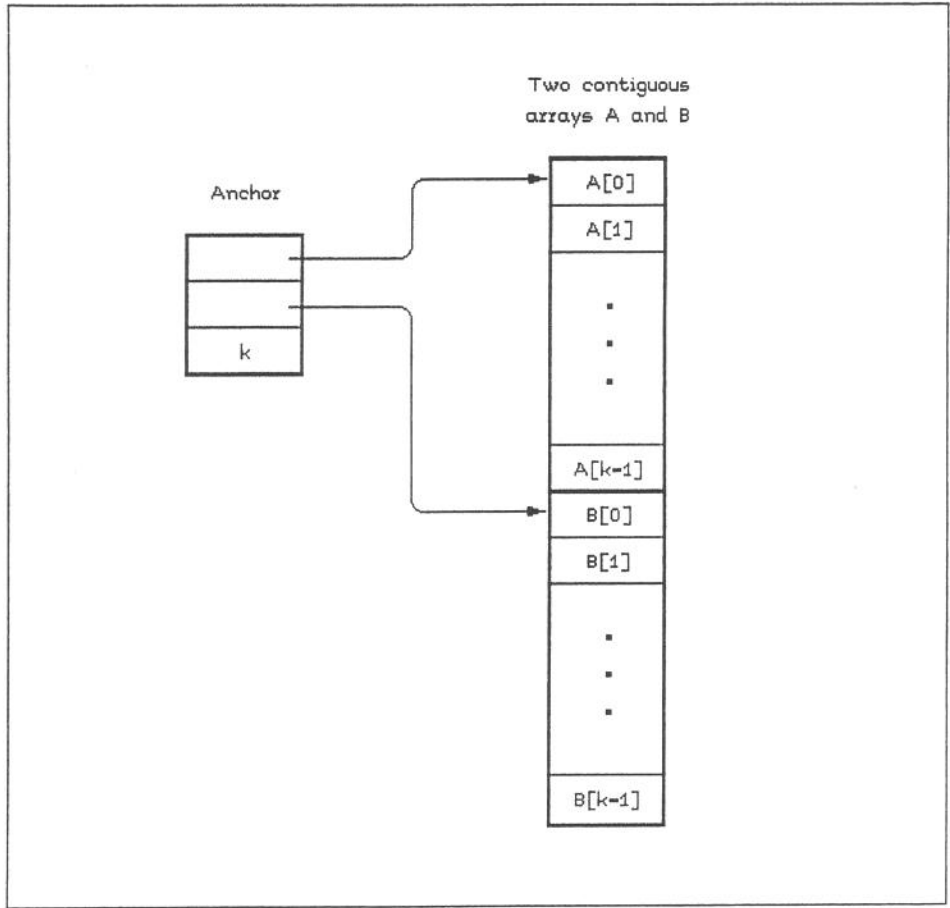


Fig. 1. Two arrays A and B of equal size placed contiguously in memory.

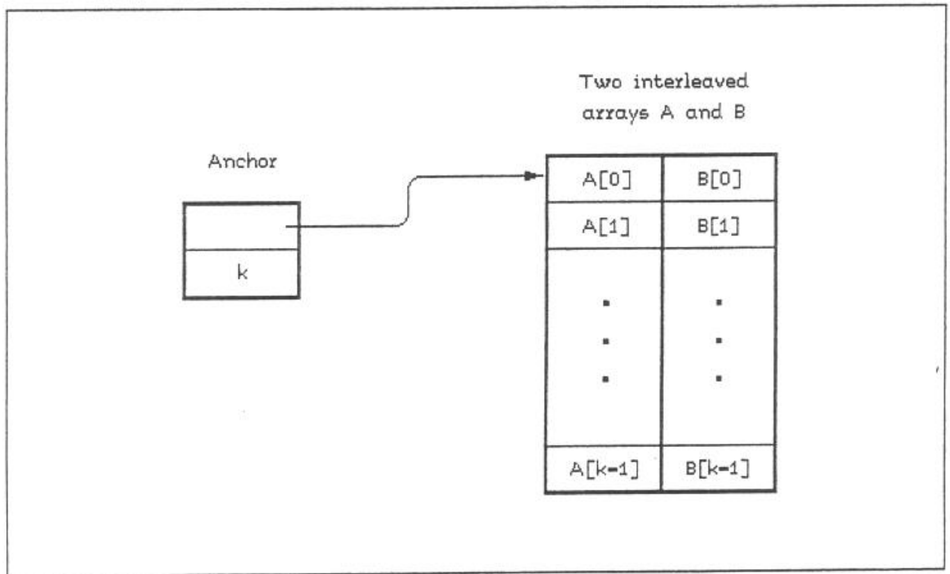


Fig. 2. The same two arrays interleaved in memory.

interleaved array cannot very well be passed to a Fortran program.

Architectural note

This technique is beneficial on a machine that allows two registers and a displacement to participate directly in the formation of an operand address. This is a property of many traditional architectures, including S/370 (see [2]). It is not always true on a RISC, however. In the case of PowerPC, for example, the operand address can be formed from two registers, or from a register and a displacement, but not from all three (see [3]). On the other hand, PowerPC has plenty of general-purpose registers ... so perhaps it has less need.

Historical note

I devised this technique in 1996 while preparing a note on the latency of Boyer-Moore search (see [1]).

References

[1] Stephenson, C.J., On the latency of Boyer-Moore search, IBM Research Report RC 20542 (1996)

[2] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

[3] May, C. et al. (editors), The PowerPC Architecture, 2nd edition, IBM and Morgan Kaufmann (1994)

CJS, 1996-08-29.

In S/370, the instruction "BALR Ra,Rb" transfers control to the address contained in Rb (provided b ¬= 0), and then places in Ra the ending address of the "BALR" (see [1]). This instruction is intended for calling subroutines.

There is a widely known trick for executing a sequence of code exactly twice:

```
        BALR  Rx,0            Rx = head of 2-times loop

        Code sequence

        BALR  Rx,Rx           Loop back once, reset Rx
```

There is a related trick which sandwiches a second sequence (executed once) between two executions of a first sequence:

```
        BAL   Rx,SEQA         Rx = head, and enter loop

        Sequence B            Executed once (inner layer)

SEQA    Sequence A            Executed twice (outer layers)

        BALR  Rx,Rx           Loop back once (and reset Rx)
```

This has the same effect as:

```
        Sequence A            Layer 1
        Sequence B            Layer 2
        Sequence A            Layer 3
```

The idea can be extended to a many-layered sandwich. Suppose for example that the following flow is desired:

```
        Sequence A            Layer 1
        Sequence B            Layer 2
        Sequence A            Layer 3
        Sequence C            Layer 4
        Sequence A            Layer 5
```

This can be programmed as:

```
        BAL   Rx,SEQA         Rx = 1st head, enter loop

        Sequence B            Layer 2 (is executed once)

        BAL   Rx,SEQA         Rx = 2nd head, reenter loop

        Sequence C            Layer 4 (is executed once)

SEQA    Sequence A            Layers 1, 3 and 5 (3 times)

        BALR  Rx,Rx           Loop back twice, reset Rx
```

The same effect could have been obtained by writing Sequence A as a discontiguous subroutine and calling it three times:

```
         BAL   Rx,SEQA        Execute layer 1
         Sequence B           Execute layer 2
         BAL   Rx,SEQA        Execute layer 3
         Sequence C           Execute layer 4
         BAL   Rx,SEQA        Execute layer 5
         ...   ...
SEQA     Sequence A           Execute Sequence A and
         BR    Rx              then return to caller
```

Sometimes, however, it is structurally undesirable to separate related sequences. Also this last version occupies one more word than the previous version.

Historical note

These techniques were devised during the 1970s.

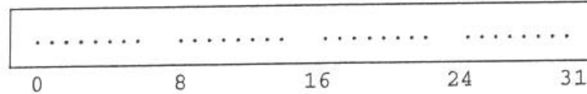
Reference

[1] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

CJS, 1994-12.

DERIVING A SINGLE-BIT MASK

Suppose a machine word contains W bits, and the bits in the word are numbered 0,1,..,W-1 from the left, big endian style. If W = 32, for example, the bits are numbered thus:



Now suppose Rx contains the number of a bit in a word, and imagine that a mask is required containing a 1 in this position and 0's elsewhere. If W = 32 and Rx = 10, for example, the mask would be:

```
00000000 00100000 00000000 00000000
```

On machines with parameterizable shift instructions, the mask can be derived from the bit number, and placed in (say) Ry, by setting bit 0 of Ry to 1 (and the rest of Ry to 0), and then shifting Ry to the right by the value in Rx. Consider S/370, for example, which has a 32-bit word (see [1]). The mask can be derived thus:

```
L     Ry,=A(2**31)      Bit 0=1, other bits 0
SRL   Ry,0(Rx)          Ry = the required mask
```

Now here is a variant which puts the mask in Rx, replacing the given bit number, without involving any other registers:

```
O     Rx,=A(2**31)      Bit 0=1, rest unchanged
SRL   Rx,0(Rx)          Rx = the required mask
```

This has the desired effect because on S/370 only the low-order 6 bits of the second operand are used by the machinery to compute the shift amount. So the high-order bit is shifted, without its presence affecting the amount by which it is shifted. Meanwhile the low-order 1-bits (which appear in the shift amount) are shifted out and discarded. This is true for all valid values of Rx. (Proof. For a shift amount exceeding 0, it is true because the number of bits required to represent the shift amount never exceeds the shift amount itself since N > lg N. For a shift amount of 0, it is true because there are no 1-bits in the first place.)

The technique can easily be extended to a 64-bit double-word. In this case we need one extra register (but not two). Suppose Rx is an even register, and Rx+1 contains the bit-number (0 <= Rx+1 <= 63). The following sequence puts the mask in Rx || Rx+1:

```
         L      Rx,=A(2**31)       Bit 0=1, other bits 0
         SRDL   Rx,0(Rx+1)         Rx||Rx+1 = needed mask
```

There is a closely related trick which computes the mask for a bit within a byte. In this case we do not even need a literal (or any memory reference). Suppose Rx contains the bit number (0 <= Rx <= 7). The following sequence puts the mask in the low-order 8 bits of Rx, replacing the bit number:

```
         LA     Rx,128(,Rx)        Bit 24=1, rest unchanged
         SRL    Rx,0(Rx)           Now bits 24..31 = mask
```

This can be useful when handling the block allocation map in a file system, or the page allocation map for a backing store.

Architectural note

These methods do not work if the bits are numbered from the right, little endian style.

Historical note

I believe this technique was devised in 1975, but it may have been two or three years later.

Reference

[1] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

CJS, 1994-12.

PRACTICAL METHODS FOR HANDLING SELF-ADJUSTING BINARY SEARCH TREES

This paper describes and reviews four different methods for constructing binary search trees. The second, third and fourth methods build "self-adjusting" trees. The original papers on these techniques were published in 1980, 1983 and 1985, by me, by Vuillemin, and by Sleator and Tarjan. My intention here is to elucidate and compare the methods, to describe them in practical terms, and to show some experimentally obtained performance data.

1. BACKGROUND

A binary tree is a structure of nodes which are connected such that all but one of them possess a father, and all of them have zero, one or two sons. (If A is B's father, B is a son of A.) The node which does not possess a father is called the "root". A node that possesses no sons is called a "leaf". An empty tree contains no nodes. In a tree of size 1, the root is a leaf.

Any node in a tree can be regarded as the root of the "subtree" comprising it and its descendants.

A binary "search" tree is a binary tree in which each node possesses a value (sometimes called its "key"), and in which the connections satisfy the following constraints:

> For each node X,
>
> the value of X equals or exceeds that of its descendants on the left (if any), and
>
> the value of X does not exceed that of its descendants on the right (if any).

The values are not necessarily distinct.

Binary search trees are useful for sorting (especially when the number of items is not known in advance), for symbol tables, for dynamic memory allocation, and for various other applications.

Let all nodes possess the following three named fields:

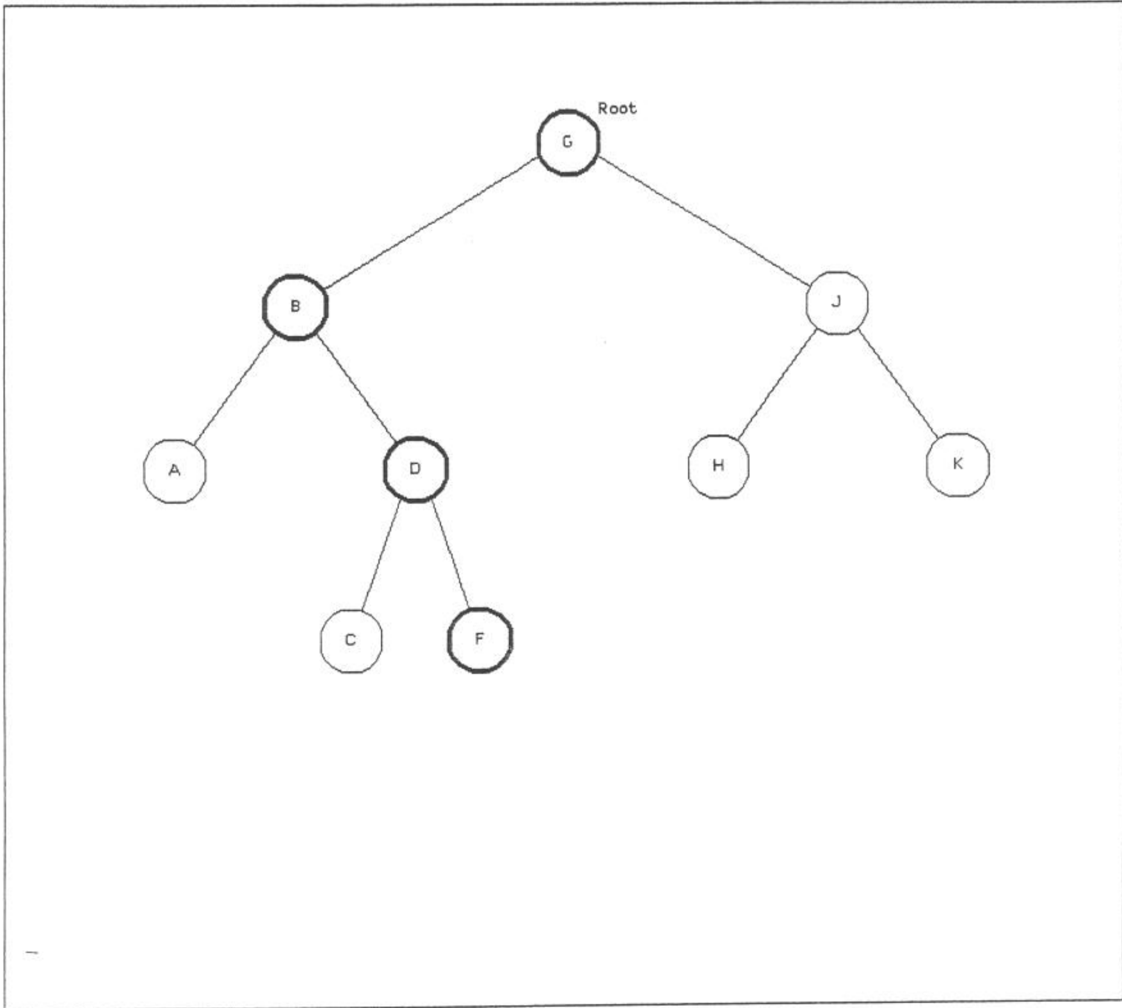


Fig. 1. An example of a binary search tree. The highlighted nodes are the ones whose values are examined if the tree is searched for a node with the value "E".

left	pointer to the left son (null if son is missing)
right	pointer to the right son (null if son is missing)
value	the "value" of this node

Note that a node does not possess a pointer to its father.

Using these definitions, we can search a binary search tree for a node possessing any desired value (say "V") by descending from the root, as follows:

```
X := addr of root;

while X ¬= null do
  begin
    if value(X) = V then
      goto found;         [X = addr of winning node]
    if value(X) > V then
      X := left(X)
    else
      X := right(X)
  end;

goto notfound;           [There is no winning node]
```

As an example, imagine searching for the value "E" in the tree shown in Fig. 1. The search will examine the highlighted nodes (G,B,D,F), and then fail.

In this paper I use an Algol-like notation in which:

a. the function "addr of" yields the address of its argument;

b. the notation "name(X)" refers to the "name" field in the node that is addressed by the pointer "X"; and

c. the notation "0(X)" refers to the field (of any name) that is addressed by the pointer "X".

Here is how a binary search tree is used to sort a sequence of items. The tree starts empty. As each item arrives, a new node is obtained, the value of the item is copied to the node, and the node is inserted into the tree. Finally the tree contains one node for each item.

A binary search tree can also be used to support a symbol table. In this application the usual operation

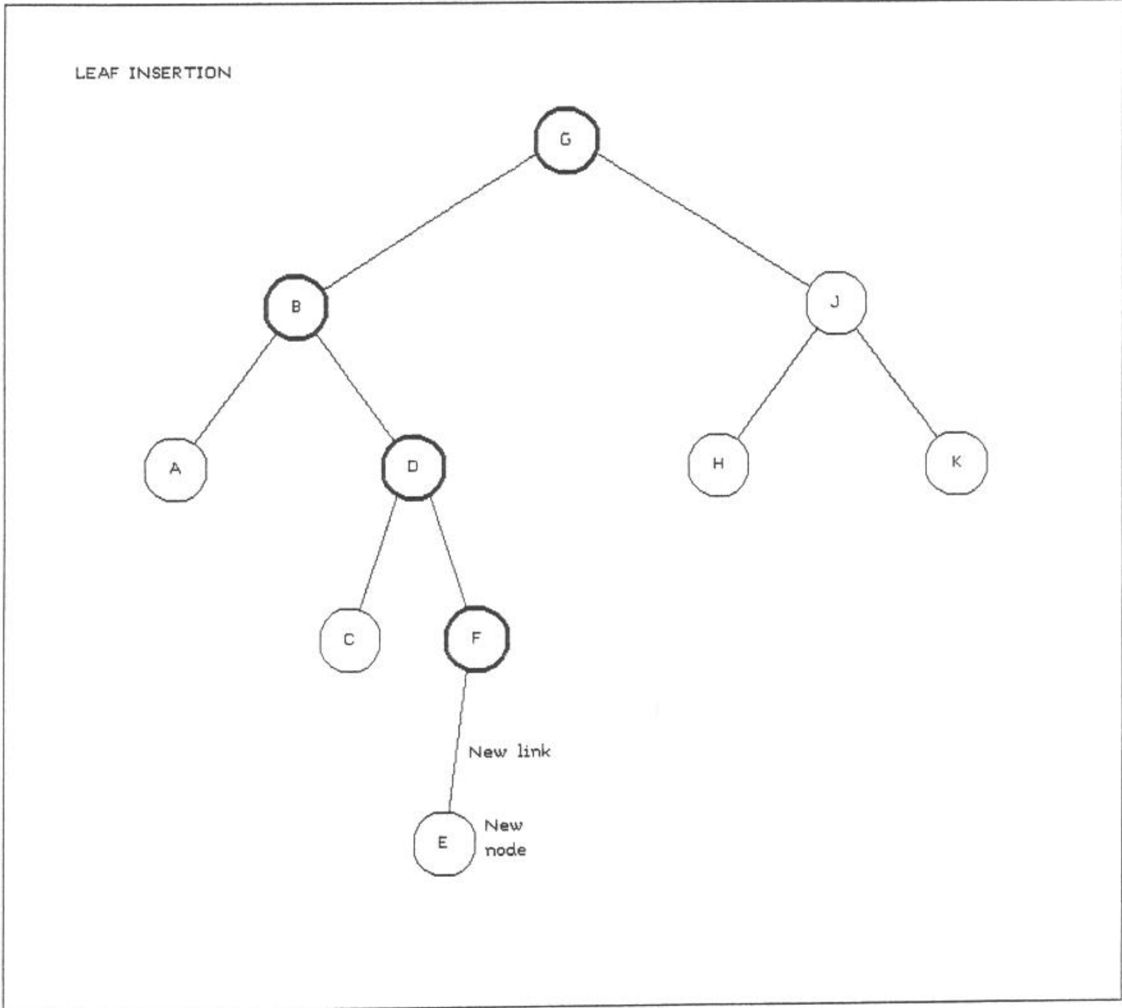


Fig. 2. The node "E" is inserted as a leaf into the tree shown in Fig. 1.

is "find or insert". If the given symbol is already present, the effect is to find the existing node (without inserting a new one); otherwise the effect is to insert a new node (and supply its address). Finally the tree contains one node for each distinct symbol.

2. METHODS OF NODE INSERTION

Clearly there are numerous valid ways of building a binary search tree from a given set of nodes. For 3 items with distinct values there are 5 legal trees; for 7 items, there are 429 trees; for 1000 items, there are more than 2.046×10^{597} trees! So the question arises as to how best to build the tree, i.e. where each node should be placed.

Method 1 -- Leaf Insertion

The easiest way to insert a node is to descend the existing tree from the root, pretending to search for a matching node, but actually treating any matches as if the old matching node possessed a marginally smaller value. When this search "fails", the new node is attached to the last old node visited, becoming its left or right son (as appropriate). See for example [1,2]. I call this method "Leaf Insertion", because each new node starts out as a leaf.

Suppose that memory for a new node has already been allocated, and the "value" field has been set. Let the static pointer variable "anchor" contain the address of the existing root (null if none). We can then perform a leaf insertion thus:

```
Z := addr of new node;
Y := addr of anchor;
X := 0(Y);

while X ¬= null do
  begin
    if value(X) > value(Z) then
      Y := addr of left(X)      [Descend to the left]
    else
      Y := addr of right(X);    [Descend to the right]
    X := 0(Y)
  end;

0(Y) := Z;                      [Attach the new node]
left(Z)  := null;               [Clear pointer fields]
right(Z) := null;
```

Fig. 2 shows the result of inserting the node "E" as a leaf into the tree that is shown in Fig. 1. The highlighted nodes are the ones whose values are examined during the insertion.

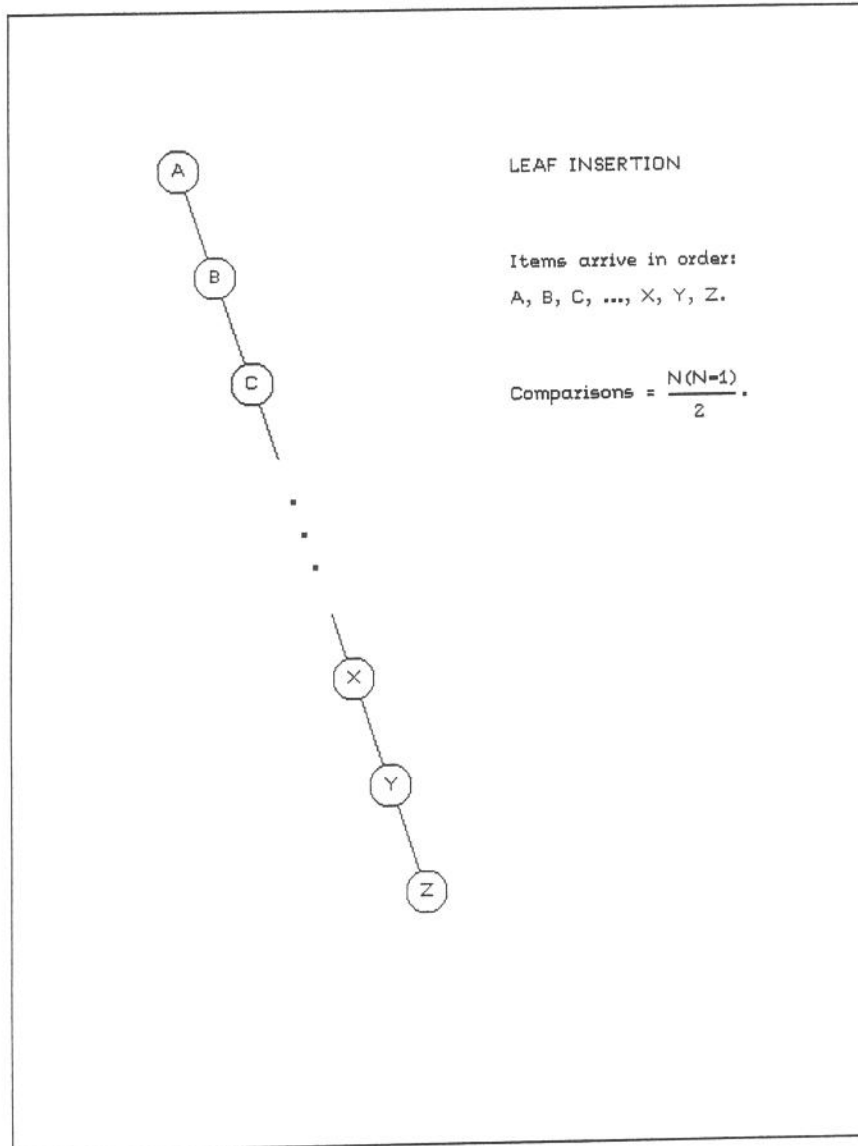


Fig. 3. Bad case for Leaf Insertion (items already sorted).

This method is simple, and has good "average" performance. To sort N randomly ordered items, the expected number of comparisons approaches 2N ln N (for large N), which is within a factor of 1.39 of the information-theoretic minimum. It has several disadvantages, however:

1. In the worst case, the number of comparisons required to sort N items is N(N-1)/2. This (or a number close to it) occurs when the items are already sorted or reverse sorted (or nearly so). Unfortunately such cases arise often in practice.

   Fig. 3 shows what happens when the items A,B,C,...,X, Y,Z arrive in order. For each new node inserted, all the previous ones must be examined.

   Sadly, there is no complementary "good" sequence which performs much better than average. The best case occurs when the order of arrival is such that the tree is perfectly balanced -- and then the performance is only 1.39 times better than for an average random sequence.

2. The first few nodes to be inserted are placed at or near the root and are never moved during the life of the tree. Therefore the number of comparisons required to insert the later nodes is strongly influenced by the order of arrival of the first few. Suppose for example that when sorting 1000 items, the 100 lowest-valued ones arrived first (in order). Then the piece of the tree containing them would degenerate to a list (with each node possessing a right son only), and every insertion (after the 100th) would involve 100 futile comparisons before reaching the first interesting node. Because of this, the variance is large.

3. When Leaf Insertion is used to maintain a symbol table, the overall performance is strongly influenced by the order in which the symbols are first encountered. If a frequently used symbol "X" is declared after numerous seldom-used symbols, it will be placed low in the tree, and every reference to "X" will give rise to many comparisons.

Method 2 -- Root insertion

It is possible to place a new node at the root (see [3]). Doing so is only slightly more complicated than placing it as a leaf.

The method of Root Insertion can be described as follows. The node to be inserted becomes the new root, and a descent is made through the old tree, starting at the old root, searching

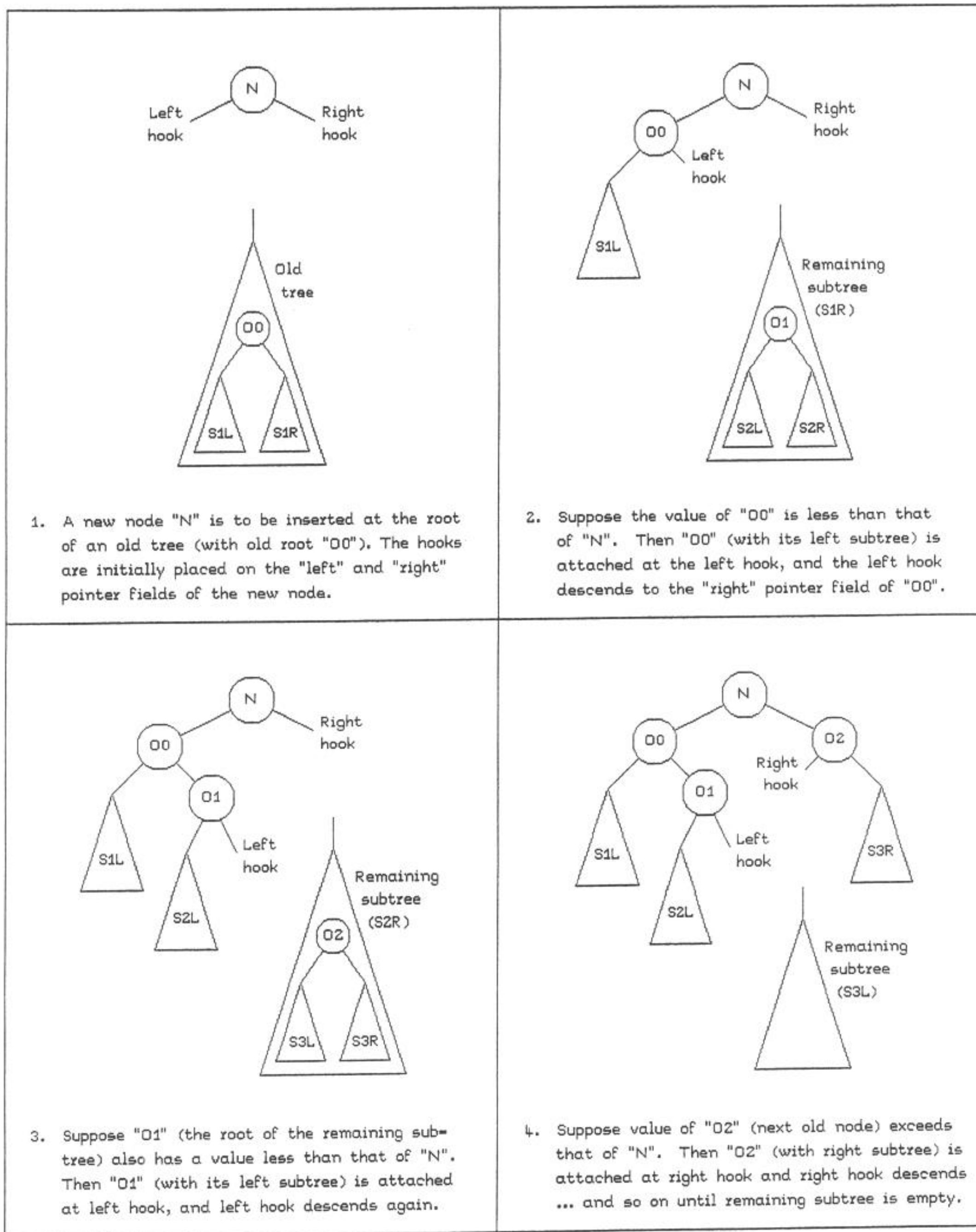


Fig. 4. Root insertion in progress.

for a matching node. Each node visited is treated as follows. If the node has a value exceeding that of the new root, the node (with its descendants on the right) is attached on the right-hand side of the tree, and its left son becomes the next node to be visited; otherwise the node (with its descendants on the left) is attached on the left-hand side of the tree, and its right son becomes the next node to be visited. The two points of attachment (on the left and right of the tree) move down during the insertion so that the new tree is properly ordered.

The points of attachment are maintained as pointer variables named "left_hook" and "right_hook". When an insertion begins, they point to the "left" and "right" fields of the new node. As the insertion proceeds, the hooks move down the inner edges of the new developing tree. To be precise, after a node has been attached on the left (i.e. connected to the left hook), the left hook descends to the "right" field in the node that was just attached, and vice versa.

When the next node to be visited does not exist, the fields addressed by the two hooks are set to null. The new tree is then complete.

The operation is illustrated in Fig. 4. Here is the algorithm:

```
Z := addr of new node;
X := anchor;                    [The addr of the old root]
anchor := Z;                    [New node becomes new root]

left_hook := addr of left(Z);          [Initialize hooks]
right_hook := addr of right(Z);

while X ¬= null do
  if value(X) > value(Z) then
    begin
      0(right_hook) := X;       [Attach old node on right]
      right_hook := addr of left(X);      [Maintain hook]
      X := left(X)              [Descend to next old node]
    end
  else
    begin
      0(left_hook) := X;        [Attach old node on left]
      left_hook := addr of right(X);      [Maintain hook]
      X := right(X)             [Descend to next old node]
    end;

0(left_hook) := null;           [Complete the new tree]
0(right_hook) := null;
```

Fig. 5 shows the result of inserting the node "E" at the root of the tree shown in Fig. 1. As before, the highlighted nodes

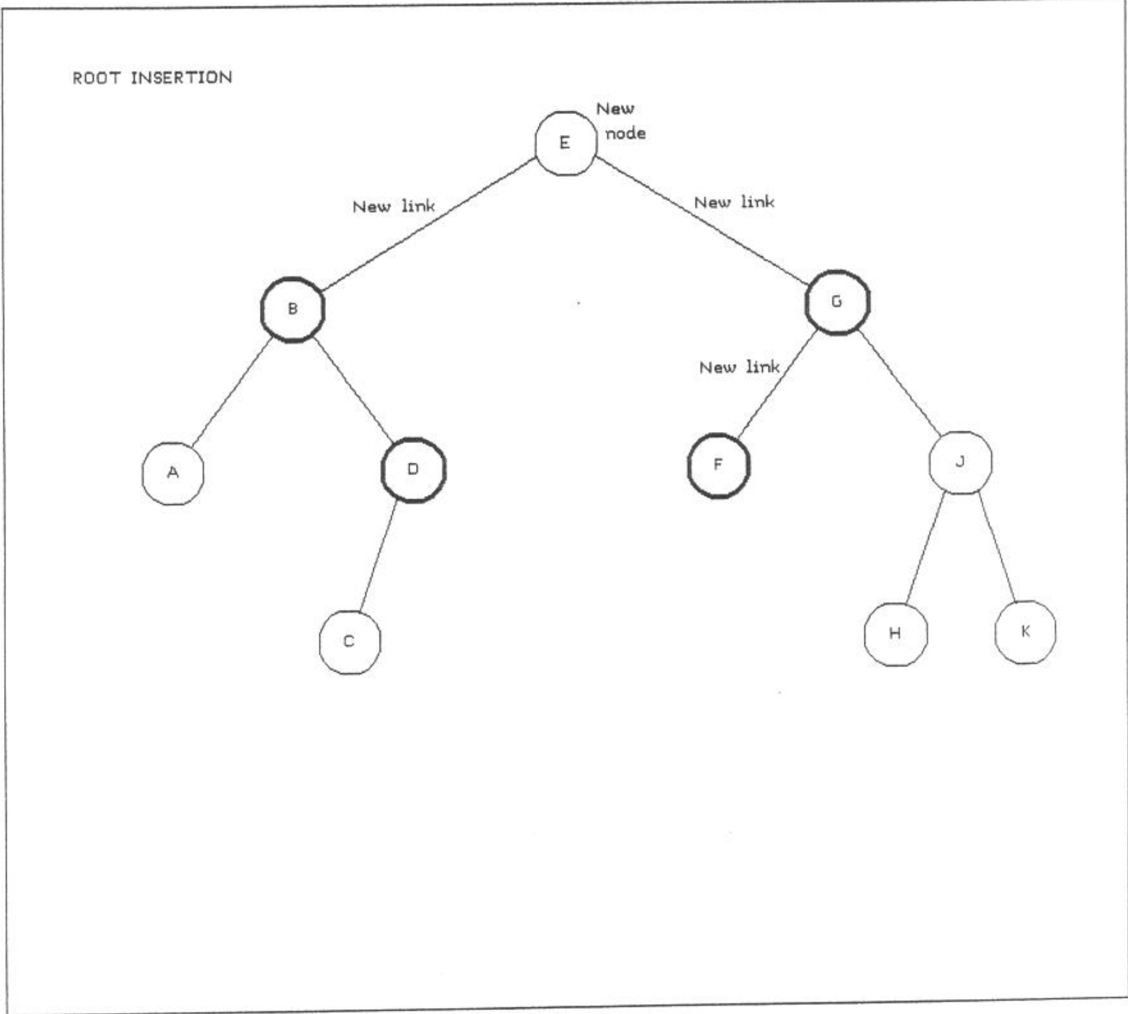


Fig. 5. The node "E" is inserted at the root of the tree shown in Fig. 1.

indicate the ones whose values are examined during the insertion.

For random data, this method has exactly the same average performance as leaf insertion (see [3]). The limiting cases are however quite different:

1. When the items are already sorted or reverse sorted, the method works at its best, and requires only N-1 comparisons.

   Fig. 6 shows what happens when the items A,B,C,...,Z arrive in order. For each node inserted (after the first), one comparison in required.

   This is nice in two ways. (a) The popular case of sorted or partly sorted data elicits unusually good performance instead of unusually bad, and (b) the good performance is extraordinarily good -- instead of being only slightly better than average.

   Sadly, there is still a worst case which requires $O(N^2)$ comparisons. It occurs when the sequence comprises a so-called "cut deck". The situation is illustrated in Fig. 7. When N is large, the number of comparisons required to sort a cut deck is $N^2/4 + O(N)$.

2. Since the tree is constantly being rearranged, the first few nodes to be inserted do not dominate the subsequent performance. Because of this, the variance is much smaller than for Leaf Insertion.

   Fig. 8 shows experimentally obtained histograms for sorting 1000 random items using Leaf Insertion and Root Insertion. The means are the same, but clearly the variances are very different.

3. When Root Insertion is used to maintain a symbol table, the "find or insert" operation is implemented as follows.

   Assume that the given symbol has not been seen before, and start to insert a new node for it at the root of the tree. If an old matching node is not encountered during the insertion, the assumption was correct, and the insertion is completed. On the other hand, if an old matching node is encountered, the old node is promoted to the root, the new node is thrown away, and the fields addressed by the two hooks are set to null.

   In either case, the act of referring to a symbol has the side-effect of placing its node at the root of the tree.

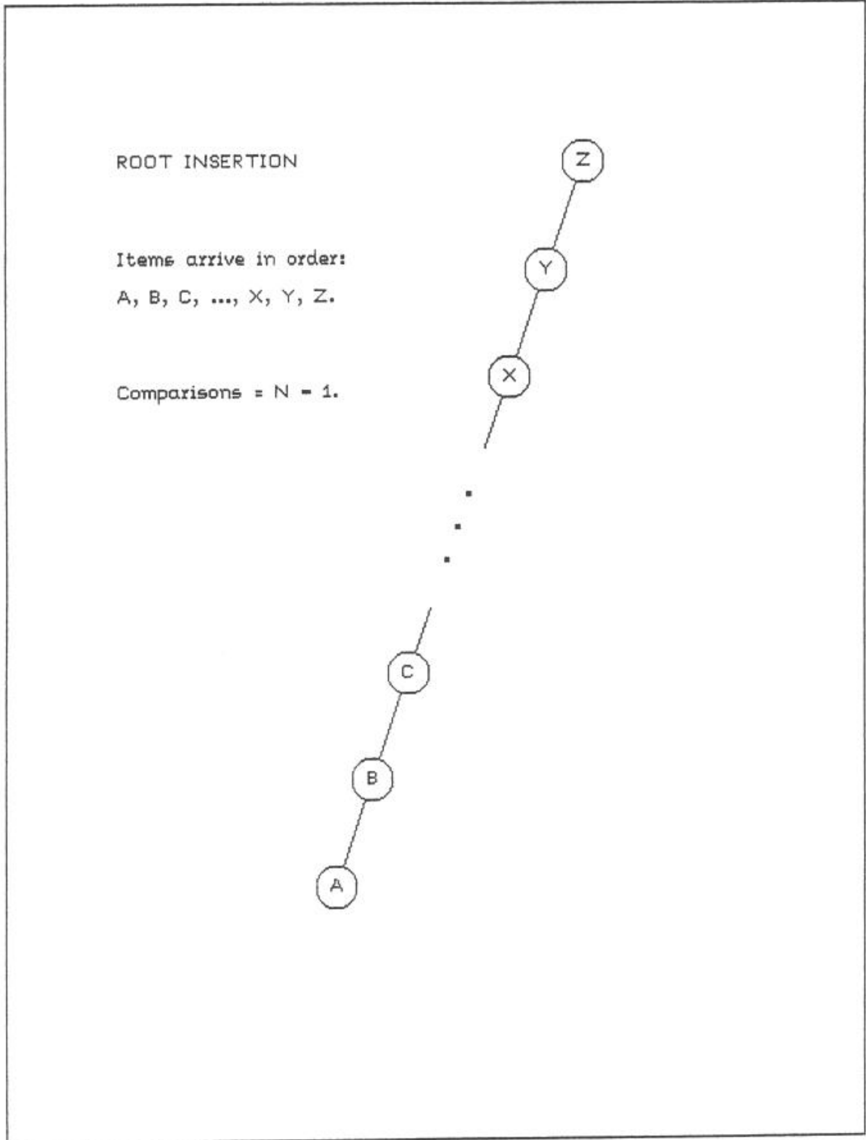


Fig. 6. Good case for Root Insertion (items already sorted).

ROOT INSERTION

Items form a "cut deck":

N,O,P,...,X,Y,Z,A,B,C,...,K,L,M.

$$\text{Comparisons} = \frac{N^2}{4} + O(N).$$

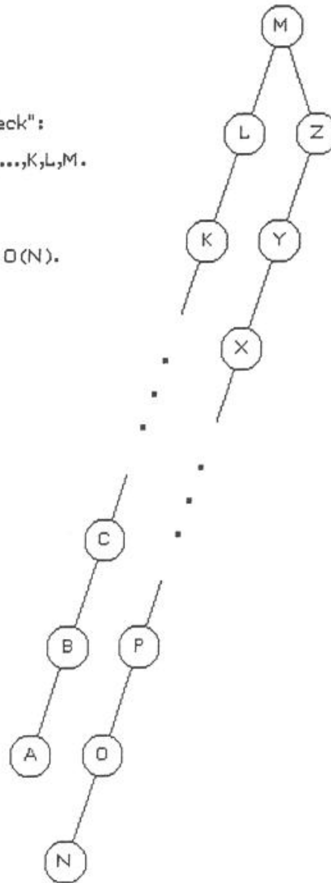


Fig. 7. Bad case for Root Insertion (cut deck).

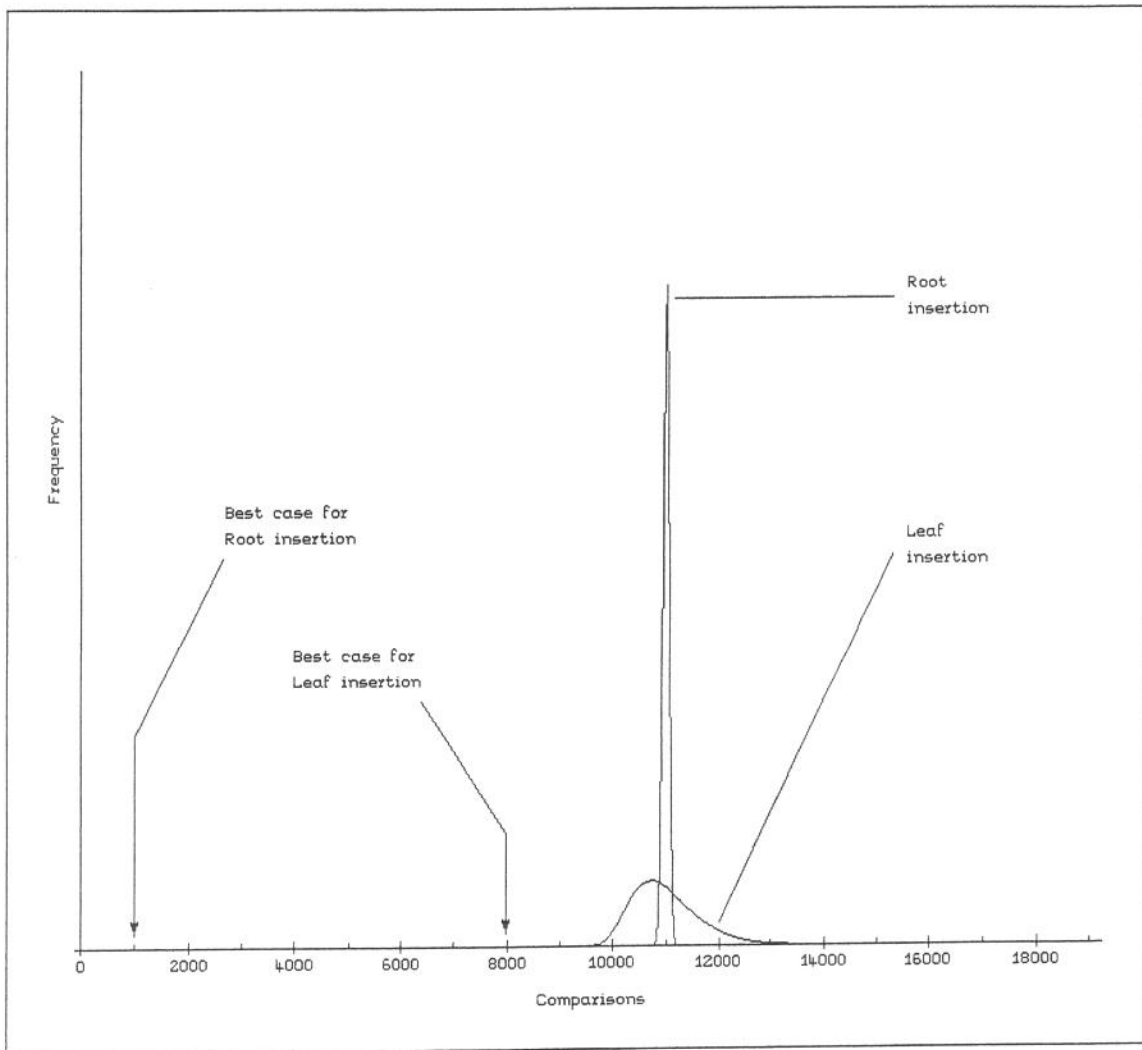


Fig. 8. Experimentally obtained histograms for sorting 1000 random values using two different tree-based mathods. Theoretical best cases are also shown.

Therefore the tree adjusts itself automatically so that popular symbols gather near the root, and the initial order has little effect on the overall performance. This is highly desirable.

A pause to take stock

In a tree that is built by Leaf Insertion, the nodes are ordered vertically such that later arrivals lie "below" earlier arrivals (or, to be precise, sons are always younger that their fathers). In a tree that is built by Root Insertion, the opposite is true: even though the tree is constantly rearranged, an older node will never become an ancestor of a newer one (see [3]).

As a result of these properties, a combination of Leaf Insertion and Root Insertion provides the ability to control the vertical position of each node -- so that the height of a node need not be purely a function of the order of arrival or the order of reference. This observation was made by Vuillemin (see [4]), who coined the term "cartesian" tree for a binary search tree in which the vertical order is significant. The insertion of a node at an intermediate depth can be accomplished simply by starting the insertion as if the new node was to be attached as a leaf; then changing strategies part way down, and inserting the new node at the root of the remaining subtree!

Fig. 9 shows the three intermediate points where the node "E" can be inserted in the tree of Fig. 1.

Cartesian trees have proved useful in dynamic memory allocation (see [5]). The available pieces of memory can be connected such that the "values" of the nodes are simply their addresses, and the nodes are ordered vertically such that no son is longer than its father. The vertical ordering helps the job of allocation, since it allows a suitable piece to be identified without visiting any nodes of inadequate size (which are often numerous). The horizontal ordering helps the job of deallocation, since the neighbours of the deallocated piece can be found by performing a binary search (which does not usually need to visit many nodes).

So the fact that the existing nodes in a tree do not reverse their relative vertical positions as a side-effect of inserting a new node can be a valuable property.

Some applications, however, do not benefit from the property. Examples are sorting and symbol tables.

The following question therefore arises. If we were willing to give up the property, and allow nodes to migrate up and down the tree, would there be some other benefit we could extract in exchange?

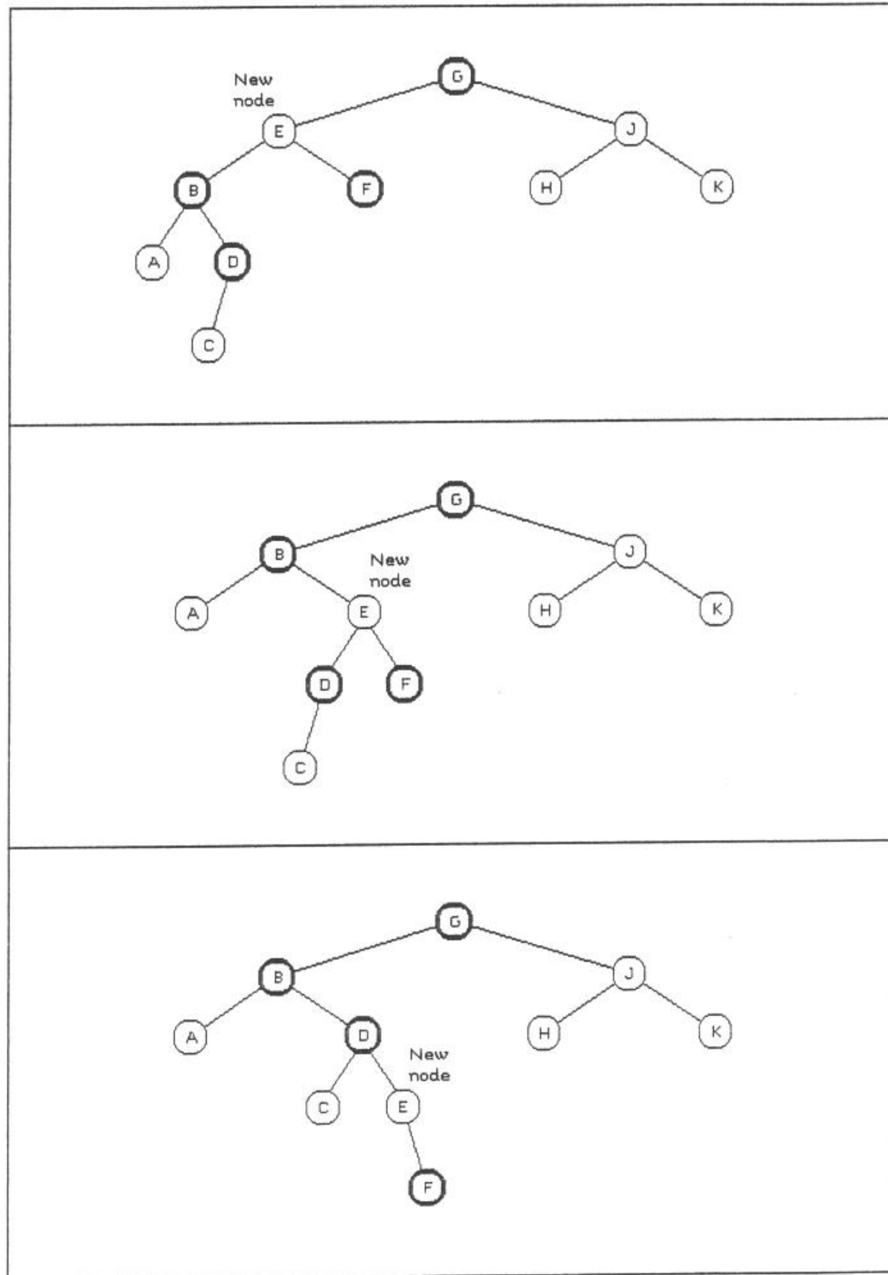


Fig. 9. Three intermediate positions where the node "E" can be inserted in the tree of Fig. 1 by using a combination of Leaf Insertion and Root Insertion.

The reason I know this is a good question to ask is because the answer has already been provided by the invention of "Splay Trees", due to Sleator and Tarjan [6,7,8].

Method 3 -- Splayed Root Insertion

I approach Splay Trees from a somewhat different starting point from that of Sleator and Tarjan. Their ingenious analysis assesses the changes in state of the tree by working from the bottom up. Here I will consider the variety of Splay which they call "Top-Down Simple Splay". This operates during a descent from the root -- without needing to climb back up the tree or perform a second descent. It is most easily described (and programmed) as a variant of Root Insertion.

The node to be inserted becomes the new root, and a descent is made through the old tree, starting at the old root, searching for a matching node. To a first approximation each node visited is treated in the same way as for ordinary Root Insertion. If the node has a value exceeding that of the new root, the node (with its descendants on the right) is attached on the right-hand side of the tree, and its left son becomes the next node to be visited; otherwise the node (with its descendants on the left) is attached on the left-hand side of the tree, and its right son becomes the next node to be visited. The two points of attachment (on the left and right of the tree) move down during the insertion so that the new tree is properly ordered.

The points of attachment are maintained as pointer variables (left_hook and right_hook) which are adjusted in the same way as for ordinary Root Insertion. When an insertion begins, they point to the "left" and "right" fields in the new node. As the insertion proceeds, the hooks slide down the inner edges of the new developing tree.

When the next node to be visited does not exist, the fields addressed by the two hooks are set to null. The new tree is then complete.

The difference between Splayed Root Insertion and ordinary Root Insertion is as follows. When two consecutively visited old nodes belong on the same side of the developing tree, an extra step is performed. This is called a "Splay" step.

Suppose that two consecutively visited nodes (C1 and C2) both belong on the left of the new node. Then the Splay step consists of the following local rearrangement:

C1, C2 and the left son of C2 are "rotated".

In this process, C2 is promoted to the position previously

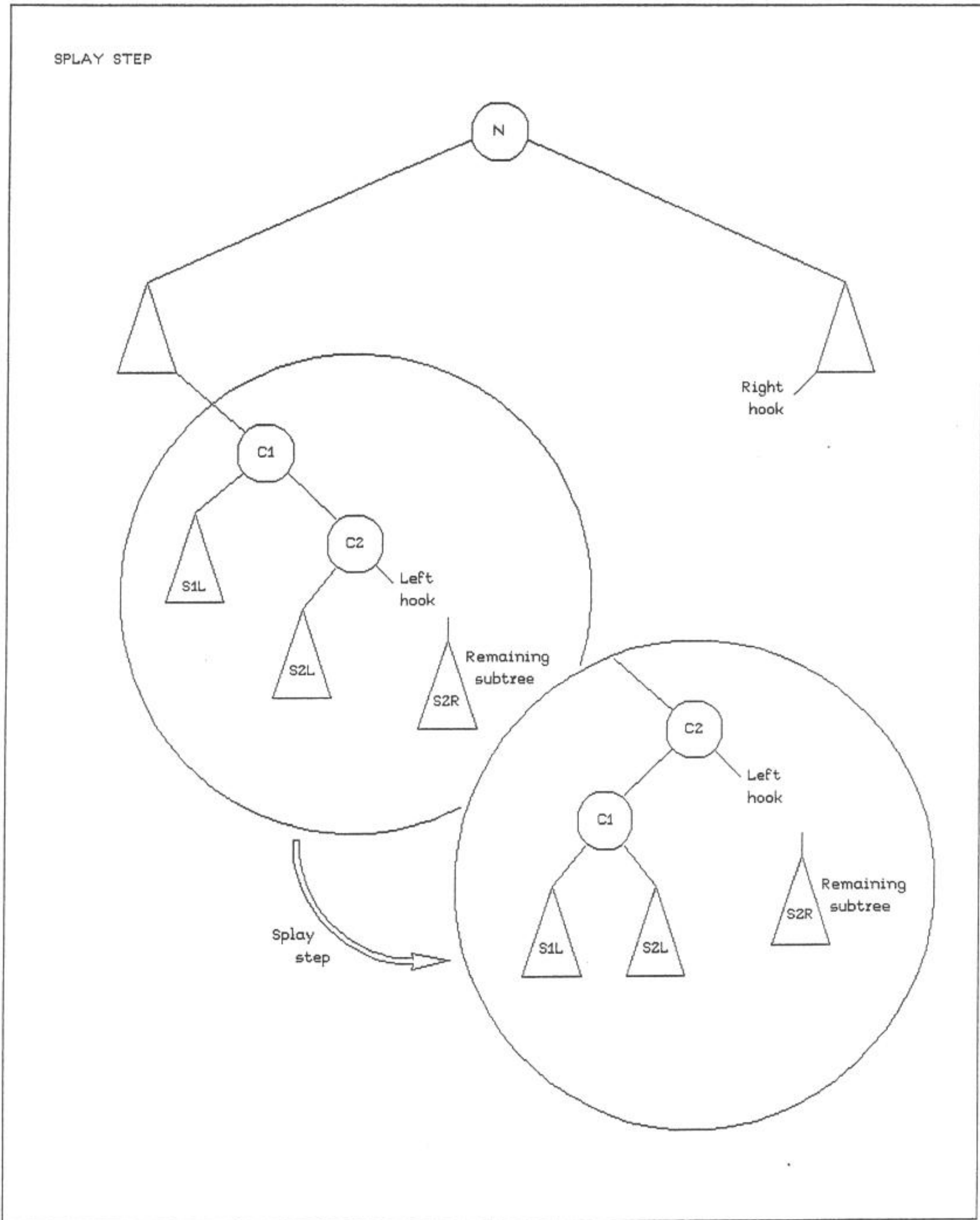


Fig. 10. A Splay step is performed when two consecutive nodes (C1 and C2) belong on the same side with respect to the new node (N).

occupied by C1; C1 (with its left subtree if any) is demoted so it becomes the left son of C2; and the old left son of C2 becomes the new right son of C1.

The hooks are not affected by the Splay step. In this example, the left hook is placed on the right pointer field of C2 -- which is where it would have been if the Splay step had not been performed.

The Splay step is illustrated in Fig. 10. The triangular subtrees are of arbitrary shapes and sizes, and in particular any or all of them may be empty.

A similar description, mutatis mutandis, applies to the case in which two consecutively visited nodes belong on the right.

During an insertion, each node visited participates in at most one Splay step. If the son of a promoted node belongs on the same side of the tree as its father, it is not deemed to be consecutive with its father.

The net effect is that some of the nodes visited during the course of an insertion move away from each other, and the tree becomes "splayed".

Here is one way of writing the algorithm. We need two more variables than for ordinary Root Insertion. These are named "left_father" and "right_father", and when necessary hold the addresses of the fathers of the nodes to which the hooks are currently attached. These variables are set when a non-splaying attachment is made and referred to when a Splay step is performed. (Since there is at least one of the former before the first of the latter, these variables do not need to be initialized.)

```
Z := addr of new node;

left(Z) := null;                  [Clobber pointer...]
right(Z) := null;                 [...fields in new node]

X := anchor;                      [The addr of the old root]
anchor := Z;                      [New node becomes new root]

left_hook := addr of left(Z);         [Initialize the hooks]
right_hook := addr of right(Z);

while X ¬= null do
  if value(X) > value(Z) then     [If node belongs on right]
    if 0(right_hook) ¬= X then    [Unless 2nd consecutive...]
      begin                       [...perform Root Insertion]
        0(right_hook) := X;
        right_father := right_hook;    [Remember old hook]
        right_hook := addr of left(X);
        X := left(X)
      end
    else
      begin                           [Perform a Splay step]
        0(right_hook) := right(X);
        right(X) := 0(right_father);
        0(right_father) := X;
        right_hook := addr of left(X);     [Maintain hook]
        X := left(X);             [Descend to next old node]
        0(right_hook) := null     [And clobber this pointer]
      end
  else                            [Same thing on the left]
    if 0(left_hook) ¬= X then
      begin
        0(left_hook) := X;
        left_father := left_hook;
        left_hook := addr of right(X);
        X := right(X)
      end
    else
      begin
        0(left_hook) := left(X);
        left(X) := 0(left_father);
        0(left_father) := X;
        left_hook := addr of right(X);
        X := right(X);
        0(left_hook) := null
      end;

0(left_hook) := null;             [Complete the new tree]
0(right_hook) := null;
```

In this description of Splay the test for whether two consecutively visited nodes belong on the same side of the tree is made by examining the pointer in the appropriate hook. There are other ways of accomplishing this. (The test can in fact

be eliminated altogether by "unrolling" the loop and examining two nodes at a clip.)

Fig. 11 shows the result of "splaying" the node "E" into the tree shown in Fig. 1. As before, the highlighted nodes indicate the ones whose values are examined during the insertion.

Why is this extra complexity desirable? What does it buy?

1. The amount of work required to build a tree using Splay has the following beautiful and remarkable property. Although an individual insertion may involve many comparisons, the total number of comparisons required for the entire sequence of N insertions (in any order) is provably no greater than O(N log N).

   So the "average" performance for sorting N random items is O(N log N), and the worst case performance is also O(N log N).

   Furthermore, the best case is N-1. This is the same as for ordinary Root Insertion, and it occurs for the same sequences (i.e. when the items are already sorted or reverse sorted).

2. Like Root Insertion, Splay constantly rearranges the tree. The first few nodes to be inserted are incapable of dominating the subsequent performance.

   Fig. 12 shows experimentally obtained histograms for sorting 1000 random items using Leaf Insertion, Root Insertion and Splay. (The first two are the same as in Fig. 8.) On average, Splayed Root Insertion requires about 5 per cent more comparisons than Leaf Insertion or ordinary Root Insertion. In most programming situations this small loss in average performance is however less important than the guarantee that there are no very bad cases.

3. When Splay is used to maintain a symbol table, the "find or insert" operation is implemented as follows.

   Assume that the given symbol has not been seen before, and start to insert a new node for it at the root (using "Splay"). If an old matching node is not encountered during the insertion, the assumption was correct, and the insertion is completed. On the other hand, if an old matching node is encountered, the old node is promoted to the root, the new node is thrown away, and the fields addressed by the two hooks are set to null.

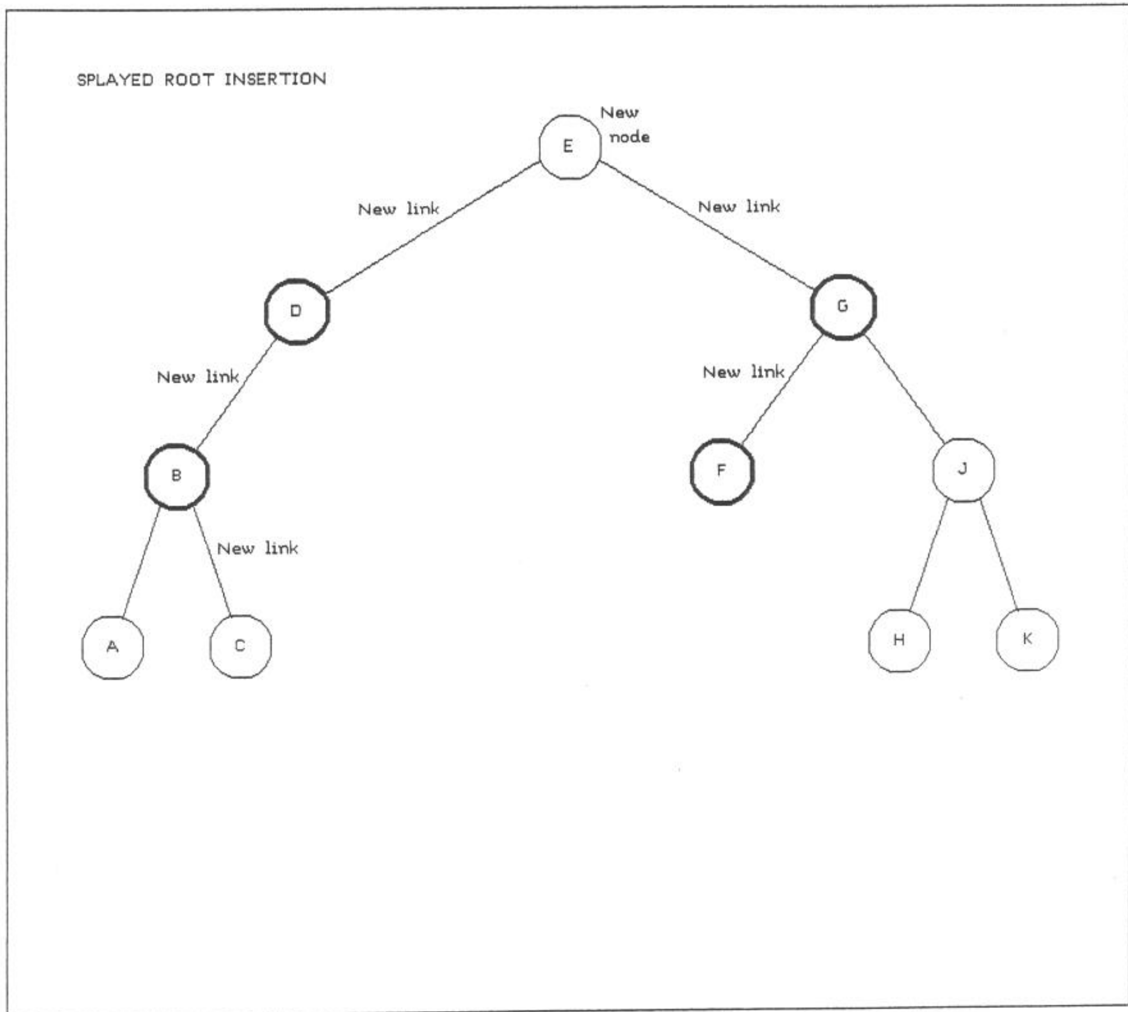


Fig. 11. The node "E" is "splayed" into the tree shown in Fig. 1.

In either case, the act of referring to a symbol has the side-effect of placing its node at the root of the tree. Therefore the tree adjusts itself automatically so that popular symbols gather at or near the root, and the initial order cannot affect the subsequent performance very much.

Method 4 -- Spliced Root Insertion

The method I call "Spliced Root Insertion" (or "Splice" for short) is related to Splay, and was inspired by it. I believe it is algorithmically equivalent to the variety of Splay that Sleator and Tarjan refer to as "Top-Down Semisplay" [7,8].

As with Splay, the procedure for inserting a node is the same as for ordinary Root Insertion until two consecutively visited nodes belong on the same side of the developing tree. When this happens, a "Splice" step is performed. What takes place during a Splice step is not however the same as what takes place during a Splay step. In the case of Splice, the rearrangement is not purely local.

Suppose that two consecutively visited nodes (C1 and C2) both belong on the left side of the new node. Then the resulting Splice step performs the following rearrangement:

> The new node (which was placed at the root) is moved down and attached to the right hook;
>
> the left son of C2 becomes the right son of C1;
>
> the right son of C2 is detached and comprises the remaining subtree;
>
> C2 is promoted to the root of the tree; and
>
> a brand new root insertion is started in order to insert the new node in the remaining subtree.

If during the newly started insertion two consecutively visited nodes both belong on the same side, the entire procedure is repeated, and so on, until the tree has been completely rebuilt (i.e. until the next node to be visited does not exist). The fields addressed by the two hooks are then set to null.

The Splice step is illustrated in Fig. 13. The triangular subtrees are of arbitrary shapes and sizes, and in particular any or all of them may be empty.

A similar description, _mutatis mutandis_, applies to the

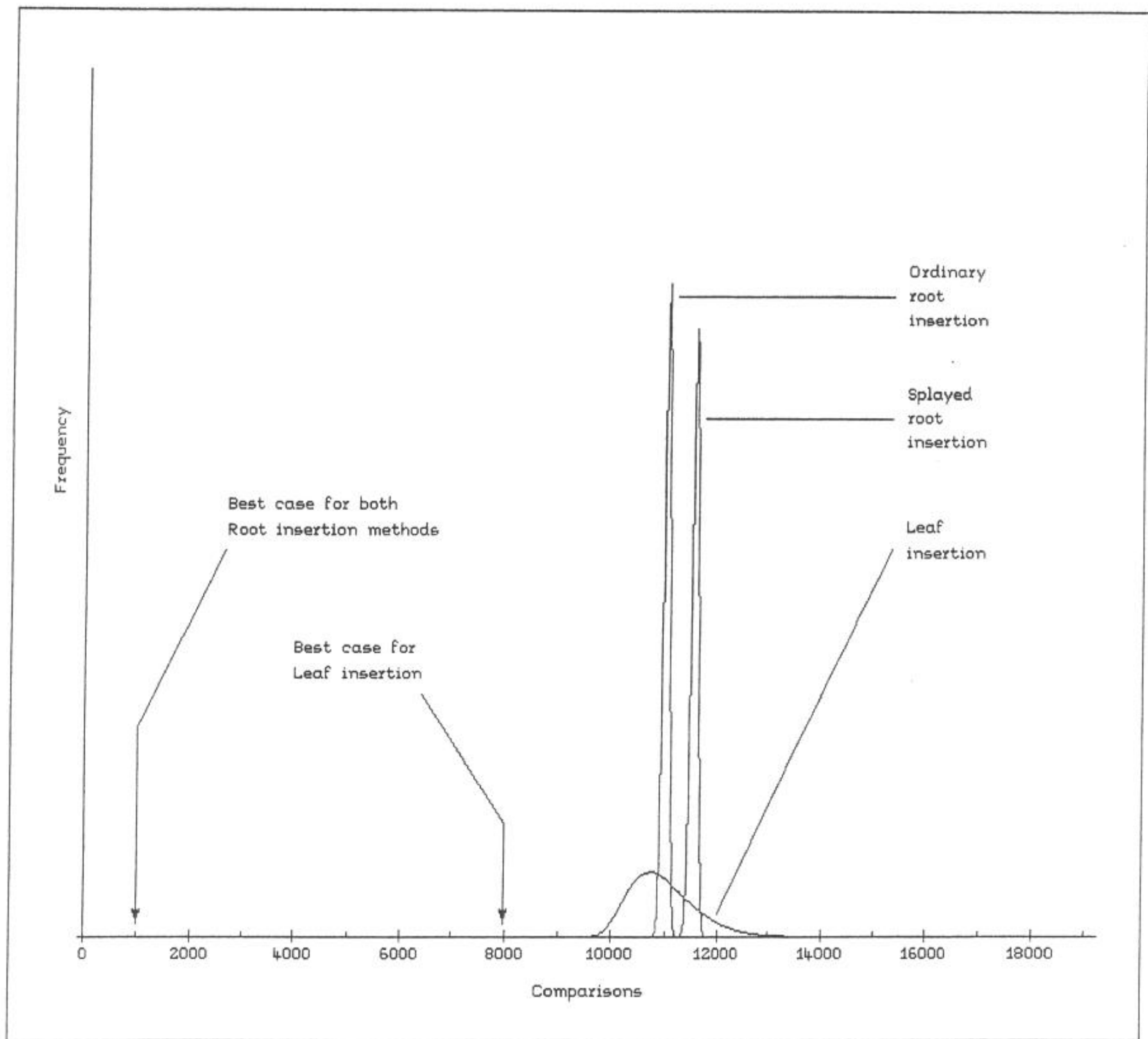


Fig. 12. Experimentally obtained histograms for sorting 1000 random values using three different tree-based mathods. Theoretical best cases are also shown.

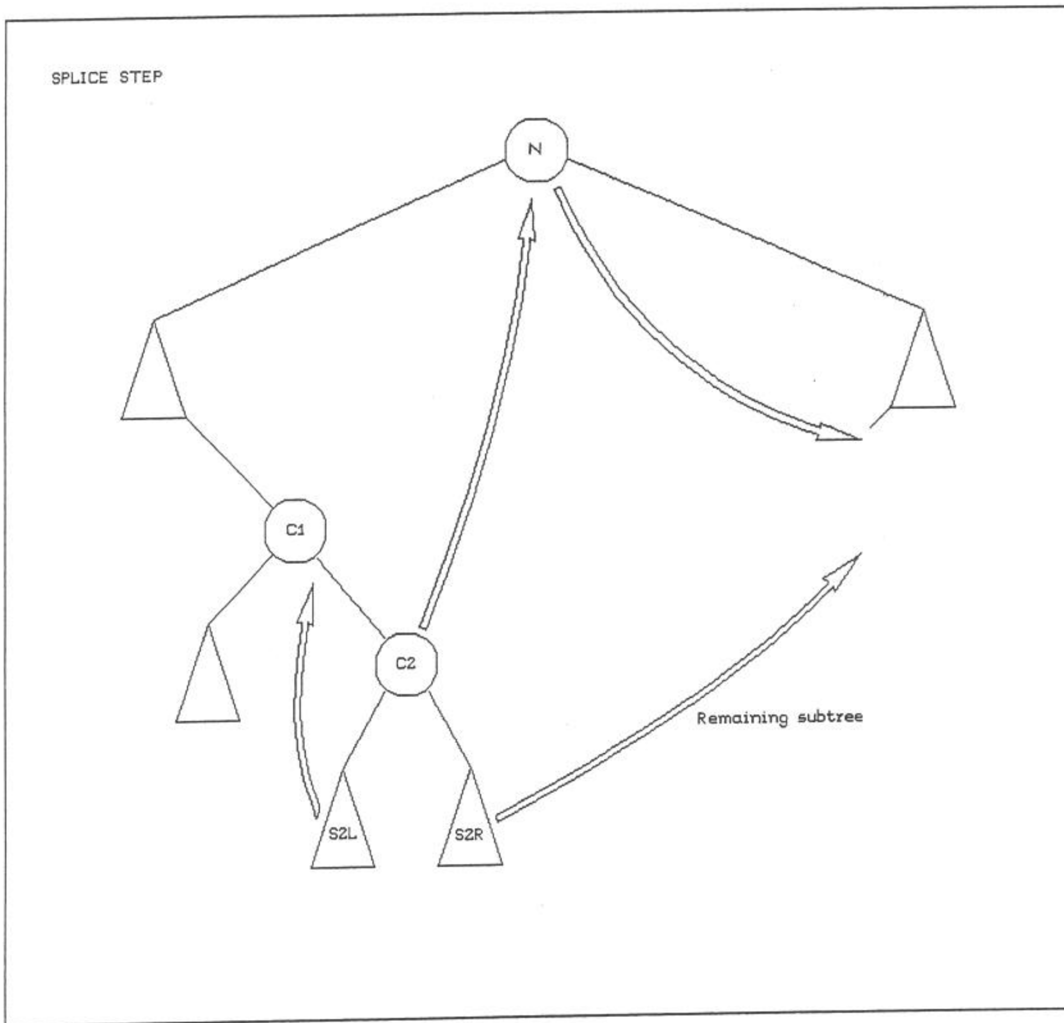


Fig. 13. A Splice step is performed when two adjacent nodes (C1 and C2) belong on the same side with respect to the new node (N).

case in which two consecutively visited nodes belong on the right.

The net effect is that some of the old nodes visited in the course of inserting a new node are "spliced" into higher levels of the tree.

Here is one way of writing the algorithm. We need two more variables than for ordinary Root Insertion. One is named "father": this holds the address of the point of attachment for the subtree currently being constructed; it is initialized to the address of the anchor, and descends after every Splice step. The other is a (very) temporary variable named "Y".

```
Z := addr of new node;

left(Z) := null; right(Z) := null;       [Clobber pointers]

X := anchor; father := addr of anchor;

left_hook := addr of left(Z);          [Initialize the hooks]
right_hook := addr of right(Z);

while X ¬= null do
  if value(X) > value(Z) then        [If node belongs on right]
    if O(right_hook) ¬= X then       [Unless 2nd consecutive...]
      begin                          [...perform Root Insertion]
        O(right_hook) := X;
        right_hook := addr of left(X);
        X := left(X)
      end
    else
      begin                          [Perform a Splice step]
        O(right_hook) := right(X); Y := left(X);
        left(X) := left(Z); right(X) := right(Z);
        O(father) := X; father := left_hook; [Promote "C2"]
        left_hook := addr of left(Z); [Start new insertion]
        right_hook := addr of right(Z);
        if father = left_hook then      [Handle null subtree]
          father := addr of left(X);
        X := Y                          [Left son of "C2"]
      end
  else                               [Same thing on the left]
    if O(left_hook) ¬= X then
      begin
        O(left_hook) := X;
        left_hook := addr of right(X);
        X := right(X)
      end
    else
      begin
        O(left_hook) := left(X); Y := right(X);
        left(X) := left(Z); right(X) := right(Z);
        O(father) := X; father := right_hook;
        left_hook := addr of left(Z);
        right_hook := addr of right(Z);
        if father = right_hook then
          father := addr of right(X);
        X := Y
      end;

O(father) := Z;                      [Attach the new node]
O(left_hook) := null; O(right_hook) := null;
```

In this description of Splice the test for whether two consecutively visited nodes belong on the same side of the tree is made by examining the pointer in the appropriate hook. (This is the same as in Splay above, and it can be eliminated in the same

way if desired.) In addition, the Splice step tests whether the subtree is empty on the other side of the tree from "C1"; since if it is, the point of attachment for the next phase of the insertion must be adjusted so that it lies in the promoted node, not the new node. (These tests can also be eliminated if desired, by using two alternate temporary nodes as "working" roots, and postponing copying the tree pointers to "C2" until after the next phase of the insertion. This requires several extra variables, however.)

Fig. 14 shows the result of "splicing" the node "E" into the tree shown in Fig. 1. As before, the highlighted nodes indicate the ones whose values are examined during the insertion.

What are the properties of Splice?

1. The amount of work required to build a tree using Spliced Root Insertion has the same beautiful property as Splayed Root Insertion, viz. that although an individual insertion may involve many comparisons, the total number of comparisons required for the entire sequence of N insertions (in any order) is provably no greater than O(N log N).

   The best case is N-1. This is the same as for ordinary Root Insertion and for Splay, and occurs for the same sequences (i.e. when the items are already sorted or reverse sorted).

2. Like ordinary Root Insertion and Splay, "Splice" constantly rearranges the tree. The first few nodes to be inserted are incapable of dominating the subsequent performance.

   Fig. 15 shows experimentally obtained histograms for sorting 1000 random items using all four methods described here. (The first three are the same as in Fig. 12.) Notice that whereas Splayed Root Insertion requires (on average) about 5 per cent more comparisons than Leaf Insertion or Root Insertion, Spliced Root Insertion requires (on average) about 8 per cent <u>fewer</u> comparisons. The practical benefit of this depends upon the relative costs of making a comparison and the other operations involved (see below). The variance of Spliced Root Insertion is clearly even smaller than that of ordinary Root Insertion or Splayed Root Insertion.

3. When Splice is used to maintain a symbol table, the "find or insert" operation is implemented as follows.

   Assume that the given symbol has not been seen before, and start to insert a new node for it (using "Splice"). If an old matching node is not encountered during the insertion, the assumption was correct, and the insertion is completed. On the other hand, if an old matching node is encountered,

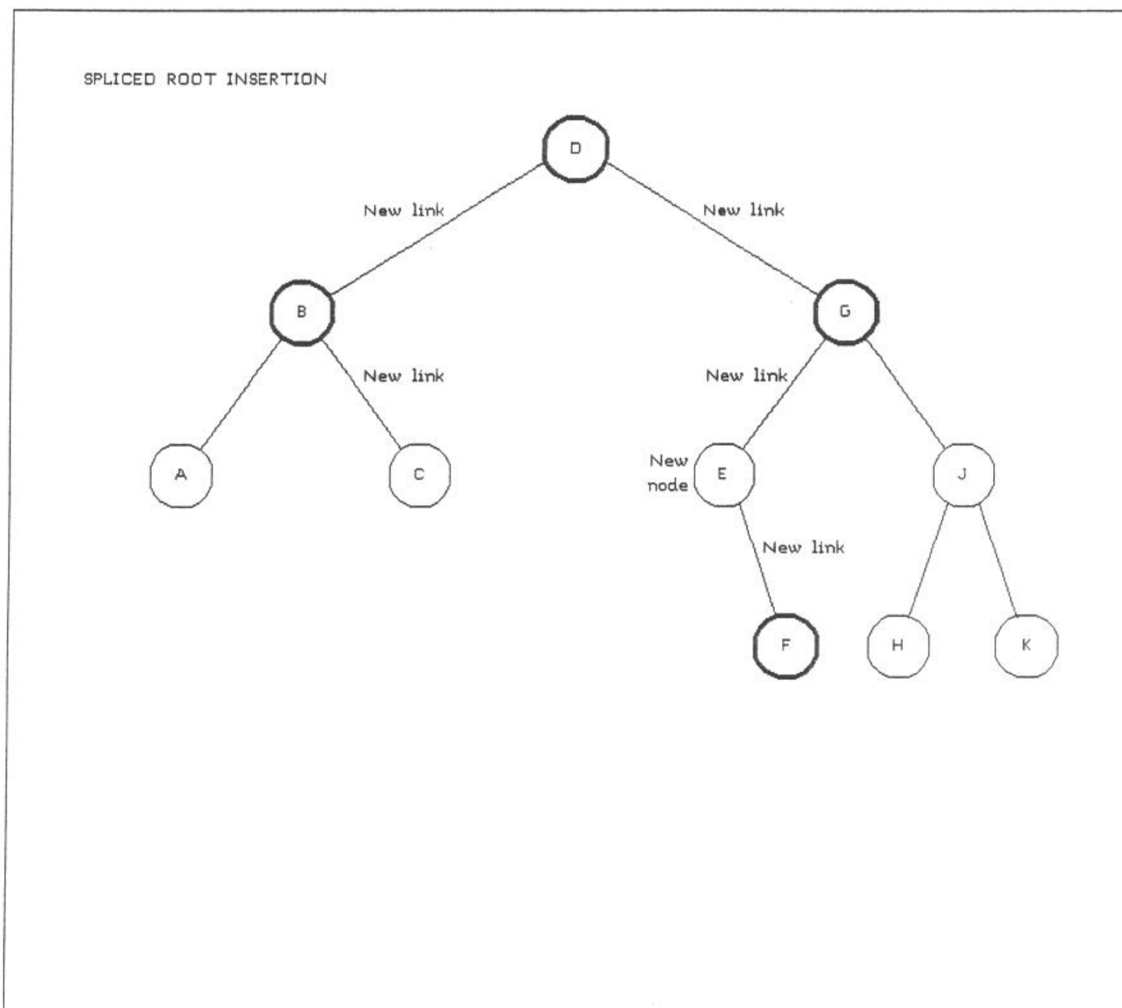


Fig. 14. The node "E" is "spliced" into the tree shown in Fig. 1.

the old node is promoted to the root of the current subtree, the new node is thrown away, and the fields addressed by the two hooks are set to null.

In this way the tree adjusts itself automatically so that popular symbols eventually gather near the root, though in general it requires more than one reference to a symbol before it reaches this privileged area. As with Splay, the initial order cannot affect the overall performance very much.

Whether this behaviour is more or less desirable than that of Splay probably depends upon the reference pattern. I have not investigated this point in detail.

Table I summarizes the characteristics of the four methods described in this paper.

3. OBSERVATIONS AND FINE POINTS

1. When used for sorting, all the methods discussed in this paper are "stable". This means that if several items have the same value, their arrival order is preserved. The arboreal rearrangements never give rise to a reversal of node positions, left-to-right.

1. In this paper I have not discussed how to use the three varieties of Root Insertion to search a tree without also being prepared to insert a new node.

   The good performance guarantees of (say) Splayed Root Insertion will not provide much advantage if search-without-insertion is a common operation and the classical method of search is used.

   Suppose for example that we use Splayed Root Insertion to construct a tree from the already-sorted sequence B,C, D,...,X,Y,Z. Only 24 comparisons will be executed while building the tree. But if what follows is 1000 (failed) searches for "A", the total number of comparisons executed during the entire sequence of operations will be $24 + 1000 \times 25 = 25024$.

   This is easily remedied. When searching for a value, behave as though the intention was to insert it, i.e. place a temporary node for it at the root, and proceed to insert this into the old tree (using whichever method seems the most suitable). If a match is found, the old winning node is promoted and the temporary node is discarded: this is the same as for "find or insert". If a match is not

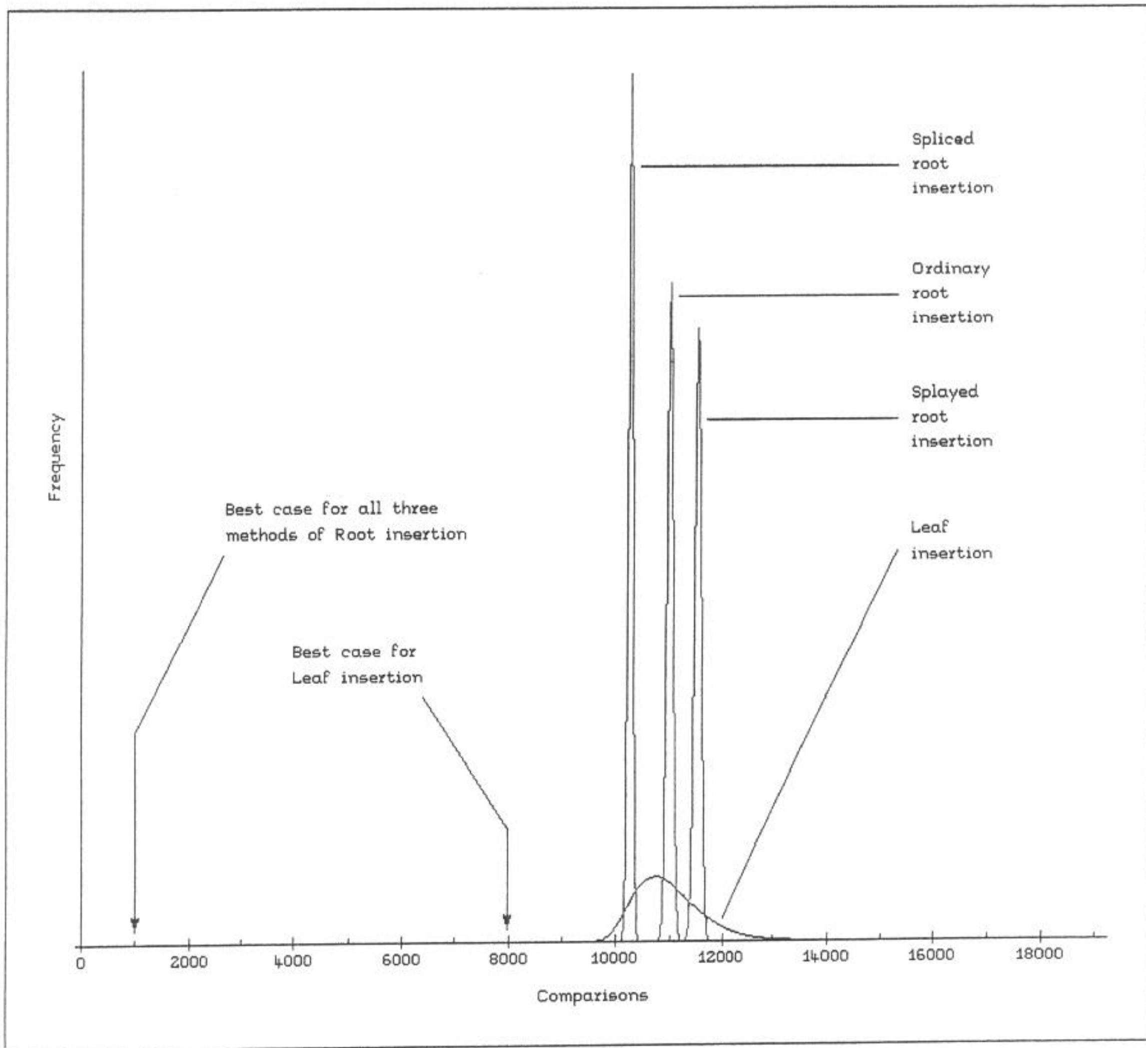


Fig. 15. Experimentally obtained histograms for sorting 1000 random values using four different tree-based mathods. Theoretical best cases are also shown.

Method	Best case for sorting	Average for sorting	Worst case for sorting	Other properties
Leaf insertion	C = O(NlogN) Occurs when items are "perfectly" distributed	C = O(NlogN)	$C = N^2/2+O(N)$ Occurs when items already sorted or reverse sorted	Preserves vertical order: a son is always younger than its father
Ordinary root insertion	C = O(N) Occurs when items already sorted or reverse sorted	C = O(NlogN)	$C = N^2/4+O(N)$ Occurs for "cut deck"	Inserts at root (or promotes to root) Preserves vertical order: a son will always have been referred to less recently than its father
Splayed root insertion	C = O(N) Occurs when items already sorted or reverse sorted	C = O(NlogN)	C = O(NlogN)	Inserts at root (or promotes to root) Does not preserve vertical order
Spliced root insertion	C = O(N) Occurs when items already sorted or reverse sorted	C = O(NlogN)	C = O(NlogN)	Node is inserted at (or promoted to) a level that is roughly half the depth of the search that was required for the insertion (or reference) Does not preserve vertical order

Table I. Characteristics of four different methods for constructing binary search trees. The symbol "C" stands for "number of comparisons".

found, the last node visited is promoted to the root, and its son (it has at most one) is attached in its old place.

This requires one extra pointer variable, to maintain the address of the father of the current node (so that if the current node turns out to be the last one visited, its son can be attached in its place).

The number of comparisons executed for the awkward example is then:

```
      24  (while constructing the tree)
    + 25  (for 1st unsuccessful search)
   + 999  (for remaining 999 searches)
  = 1048  (total).
```

These counts apply to ordinary Root Insertion and Splayed Root Insertion. Splice requires 26 extra comparisons; for in this case several unsuccessful searches for "A" are needed before "B" migrates all the way to the root.

3. The average number of comparisons required to sort N randomly ordered items with Leaf Insertion or Root Insertion is:

$$2N(\ln N - 2 + g) + O(\log N)$$

where g is Euler's constant (appproximate value 0.577); see [3].

The average number of comparisons required to sort N randomly ordered items with Splay or Splice is not known precisely. Judging from the experimentally obtained histograms, Splayed Root Insertion requires about 5 per cent more comparisons than Root Insertion, and Splice requires about 8 per cent fewer comparisons than Root Insertion. I have no evidence that these differences disappear -- or even change very much -- with increasing values of N.

If these ratios hold constant for large N, then the experimental evidence suggests that the average number of comparisons required for sorting by Splice lies within a factor of about 1.28 of the information-theoretic minimum.

Whether such constant factors are important depends upon the circumstances. A spliced insertion involves slightly more bookkeeping than a splayed insertion. Sometimes the cost of this exceeds the cost of the comparisons, and then Splay would probably be preferable to Splice. But if the values comprise long strings, comparisons between them may dominate the running time. On some mainframe computers, a

long string comparison costs 100 times as much as an ordinary instruction such as Load or Branch -- even if there is a mismatch on the first character [9]. In such cases, Splice would usually beat Splay. (Actually this is an extreme situation, which should preferably be attacked by other means, such as avoiding the badly performing instructions altogether.)

When a tree is constructed for a purpose other than sorting, there may of course be a good reason for wanting to place the "winning" node at the root (use Splay), or for not doing so (use Splice).

4. In the 1960's and 1970's there was a lot of work on AVL trees and other fairly complicated methods for explicitly rearranging trees to maintain balance -- or even bias (see reviews in [2,8]). In my opinion there is little remaining reason to use these techniques, which require extra fields in the nodes, and additional bookkeeping. Self-adjusting trees constructed with Splay or Splice yield an amortized performance that is close to what is obtainable with explicit balancing, and they yield even better performance when the sequence exhibits order or a skewed reference pattern. Similar advice is given in [8].

Historical notes

Ordinary root insertion was devised by me in 1976 and was immediately used in the YMS sorting program "Sort". My colleague Walter Daniels promptly ported the program to VM/CMS, where it became widely used in IBM, under the name "TSORT". Eventually (in 1993) the program was made available to customers. Some things take a long time.

Ordinary root insertion has also been used for the symbol table in the YMS program binder since 1979.

The use of cartesian trees for dynamic memory allocation was originated by me in 1980, and has been used in YMS since 1981. Subsequently this method of memory allocation has became widely adopted, and it (or variants of it) are used by Sun Microsystems; in IBM's runtime support for OS/400, MVS, AIX and OS/2; in VM networking; in the research program "Tierra", which studies evolution of digital organisms; and in several other experimental systems.

Spliced root insertion has been used in the YMS editor since 1985, both for sorting the lines of an edit file and for maintaining edit "variables". In 1986 the YMS sorting program was rewritten to use the same method.

A combination of Spliced root insertion and hashing is used for the symbol tables in "Phantasm", an experimental S/370 and S/390 assembler written by me in 1994.

REFERENCES

[1] Knuth, D.E., The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley (1968)

[2] Knuth, D.E., The Art of Computer Programming, Vol. 3, Sorting and Searching, Addison-Wesley (1973)

[3] Stephenson, C.J., A method of constructing binary search trees by making insertions at the root, Int. J. Comp. Inf. Sci. 9, 1, p. 15 (1980)

[4] Vuillemin, J., A unifying look at data structures, CACM 23, 4, p. 229 (1980)

[5] Stephenson, C.J., Fast Fits - New methods for dynamic storage allocation, Symposium on operating system principles, Bretton Woods (1983)

[6] Sleator, D.D. and Tarjan, R.E., Self-adjusting binary trees, Proceedings of the 15th Annual ACM Symposium on Theory of Computing, p. 235, ACM (1983)

[7] Tarjan, R.E., Private communication (1983-84)

[8] Sleator, D.D. and Tarjan, R.E, Self-adjusting binary search trees, JACM 32, 3, p. 652 (1985)

[9] Hack, M., Private communication (1994)

CJS, 1995-01.

ON TRAVERSING AND DISMANTLING BINARY TREES

Joint work with Paul R. Kosinski*

Let each node in a binary tree possesses the following fields, in addition to those containing the "value" of the node and any other associated information:

left	pointer to the left son (null if son is missing)
right	pointer to the right son (null if son is missing)
tag	flag bit (normally 0)

Such a tree can be traversed, without the use of additional memory, by the technique known as "pointer reversal".

CONVENTIONAL METHOD

Suppose we want to visit the nodes of a binary tree from left to right (technically called "postorder"). While descending to the left, the "left" pointer field is taken over temporarily to record the address of the father; and when descending to the right, the "right" pointer field is taken over for this purpose. In the latter case, the flag is also set. The flag is examined when climbing back up, to find out whether the ascent is from the left son (in which case we repair the "left" pointer, handle this node, and descend to the right), or from the right son (in which case we repair the "right" pointer and continue the ascent).

Here is the method Knuth gives for doing this, in [1] page 562, transcribed to an Algol notation (see [2]). The static pointer named "anchor" contains the address of the root, and the subroutine named "handle" does whatever is appropriate (for the particular situation) when visiting a node. The comments are mine.

* Present address: Digital Equipment Corporation, 334 South Street, Shrewsbury, Mass.

```
  X := anchor;                          [Addr of the root]

  if X = null then                      [Do nothing if...]
    goto finis;                         [...tree is empty]

  Y := addr of anchor;                  [The root's "father"]

descend_to_left:                        [Find leftmost descendant]

  Z := left(X);                         [The addr of my left son]

  while Z ¬= null do                    [While there are still...]
    begin                               [...descendants on my left]
      left(X) := Y;                     [Save my father's addr...]
      Y := X;                           [...and step down to left]
      X := Z;
      Z := left(X)
    end;

visit:                                  ["Visit" the current node]

  handle(X);                            [* Do whatever is needed *]
  Z := right(X);                        [The addr of my right son]

  if Z ¬= null then
    begin                               [Descend to the right]
      right(X) := Y;                    [Save father's addr...]
      tag(X) := 1;                      [...and set needed flag]
      Y := X;                           [I am my rt son's father]
      X := Z;                           [Descend to my right son]
      goto descend_to_left              [Loop back to handle this]
    end;

ascend:                                 [Climb back up the tree]

  if Y = addr of anchor then            [Leave the loop if the...]
    goto finis;                         [...traversal is complete]

  if tag(Y) = 0 then                    [If climbing from the left]
    begin
      Z := left(Y);                     [Ascend from left son...]
      left(Y) := X;                     [...and repair left ptr]
      X := Y;
      Y := Z;
      goto visit                        [Go back to handle node]
    end;

  Z := right(Y);                        [Ascend from right son...]
  right(Y) := X;                        [...and repair right ptr...]
  tag(Y) := 0;                          [...and clear the flag here]
  X := Y;
  Y := Z;
  goto ascend;                          [And loop back for more]

finis:                                  [End of the traversal]
```

Knuth offers the conjecture that it is impossible to traverse a binary tree without using at least one flag bit per node -- in addition to the two pointers. As far as I know this conjecture has not been refuted.

SIMPLIFIED METHOD

The purpose of this paper is to point out that a simpler method of pointer reversal exists for left-to-right (or "post-order") traversal, provided we relax the rules and allow the original tree structure to be destroyed as a side-effect of the traversal. This may seem a preposterous condition; but in practical situations it is often acceptable or even desirable. Here are two examples:

1. A binary search tree has been used to sort a sequence of items. When the sort is complete, the nodes must be visited from left to right, and the values emitted in sorted order. We have no use for a node once its value has been emitted. (In fact, if the memory for the node was allocated as an individual piece, it can be returned to the memory allocator at the same time -- as soon as the value has been emitted; then when the traversal is complete, there will be nothing left at all, which is exactly what we want.)

2. A set of binary search trees has been used to construct a symbol table. (The roots of the trees may be addressed from a pointer array which is indexed with a hash code derived from the symbol: see for example [3].) When the symbol table is complete, the trees must be merged so that the symbols can be emitted in alphabetical order. The easiest way to do this (and probably the most efficient way) is to transform each tree to a linked list, to merge the lists, and then to step through the final combined list. We have no interest in the shapes of the trees after their nodes have been chained into linked lists.

The simpler method makes use of the following observation. The only reason for returning from a right son to its father is to repair the father's right pointer field. If this pointer does not need to be repaired, we can bypass the immediate father and proceed directly to the father of G, where G is the closest ancestor that is a left son. This assumes (of course) that we know the address of this node. Well, it turns out that, when descending to a right son, it is just as easy to supply the address of this (remote) node as to supply the address of the immediate father. In fact we do not need to tamper with the "right" pointer fields at all, unless we want to use them for something else, as in Version 2 below. Also we do not require a flag.

Here are two versions of the simpler method. The first version visits the nodes in order and destroys the tree. As before, the static pointer named "anchor" contains the address of the root, and the subroutine "handle" does whatever is necessary when visiting a node, which may include deallocating the memory it occupies.

Version 1 -- Visit nodes in order and destroy tree

```
    X := null;                [Root has no father]
    Y := anchor;              [Address of the root]
    goto middle;              [Enter the loop below]
  top:                        [Head of the outer loop]
    handle(X);                [* Do whatever is reqd *]
    Y := right(X);            [Descend to right son...]
    X := left(X);             [...and ascend to father]
  middle:                     [Enter outer loop here]
    while Y ¬= null do        [Find leftmost descendant]
      begin
        Z := left(Y);         [Step down to left and...]
        left(Y) := X;         [...save my father's addr]
        X := Y;               [Tentative current node]
        Y := Z                [New paternal address]
      end;
    if X ¬= null then         [Loop around unless the...]
      goto top;               [...traversal is complete]
```

The second version (below) visits the nodes in order, and also transforms the tree to an ordered doubly-linked list. In each node, the "left" pointer field finally contains the address of the next node in the linked list and the "right" field contains the address of the "left" pointer field in the previous node. The anchor, which initially contains the address of the root, finally points to the first node in the linked list. (The back-pointer in the first node points to the anchor, and the forward-pointer in the last node is set to null.)

Fig. 1 shows an example of the transformation.

The subroutine "handle" does whatever is necessary when visiting a node -- which may be nothing at all. Obviously in this case the subroutine must not deallocate the memory occupied by the node.

Version 2 -- Visit nodes in order and linearize tree

```
      W := addr of anchor;          [Chain list from here]
      X := null;                    [Root has no father]
      Y := anchor;                  [Address of the root]
      goto middle;                  [Enter the loop below]

    top:                            [Head of the outer loop]

      handle(X);                    [Omit if nothing to do]
      Y := right(X);                [Descend to right son]
      right(X) := W;                [Omit for one-way list]
      W := addr of left(X);         [This is new predecessor]
      X := left(X);                 [Now ascend to my father]

    middle:                         [Enter outer loop here]

      while Y ¬= null do            [Find leftmost descendant]
        begin
          Z := left(Y);             [Step down to left and...]
          left(Y) := X;             [...save my father's addr]
          X := Y;                   [Tentative current node]
          Y := Z                    [New paternal address]
        end;

      0(W) := X;                    [Chain me to predecessor]

      if X ¬= null then             [Loop around unless the...]
        goto top;                   [...traversal is complete]
```

The storing of the back-pointer ("right(X) := W") may be omitted if a singly-linked list is desired. The "right" pointer fields would then have undefined contents after the traversal.

OBSERVATIONS AND FINE POINTS

1. Pointer reversal (of any variety) can be used only when it is acceptable for the tree to be in a mutilated state for the duration of the traversal. The method is not always suitable for "real-time" programs that require predictable performance for search or insertion operations.

2. The simplified method described here is easier to program than the general method. It is also slightly more efficient. For one thing, the "left" pointer field does not need to be repaired, and the "right" pointer field does not need to be reversed (or repaired). Also, in Knuth's general method, a node is "touched" P times, where:

$$P = 1 + L + R.$$

Here L = 1 if the node possesses a left son (or 0 if it does not), and R = 1 if the node possess a right son

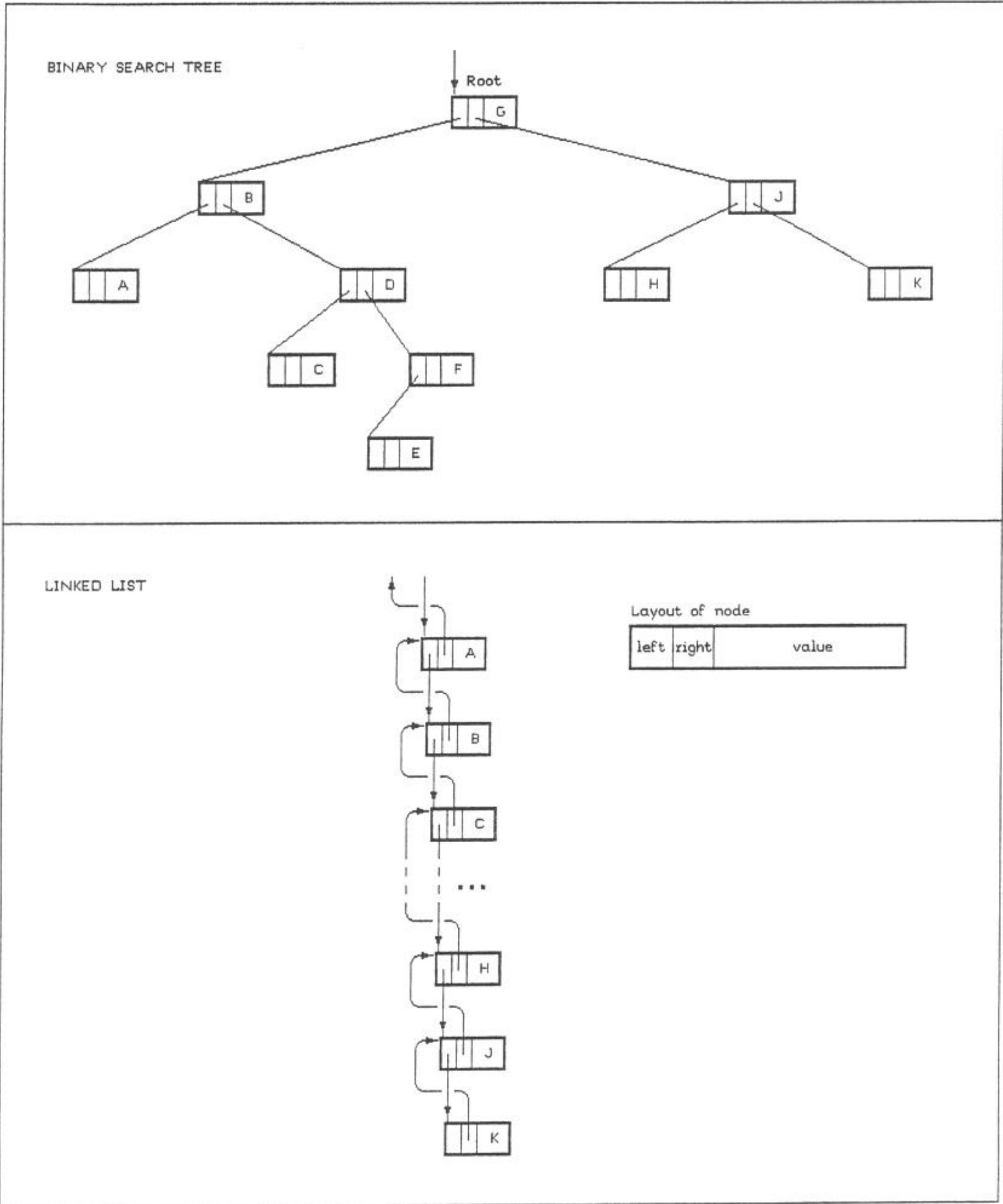


Fig. 1. Transformation from binary search tree to doubly-linked list.

(or 0 if it does not). In the simplified method, a node is touched only Q times, where:

$$Q = 1 + L.$$

3. As a curiosity, note that it is possible to traverse a binary search tree by the simplified method and end up with a valid binary search tree -- though the resulting tree will usually have a radically different shape from before. This can be accomplished by transforming the tree to a list, as in Version 2 above, except that the "right" pointer fields are used for the forward pointers, and the "left" pointer fields are set to null. This list comprises a degenerate binary search tree in which all the nodes (except for a single leaf) have a right son and no descendants on the left. This is not usually a desirable shape for a tree. If however the tree is maintained by Root Insertion, "Splay" or "Splice" (see [2]), it will usually return to a jumbled state quite soon -- typically after "about" log N search or insertion operations (where N = total number of nodes).

Historical note

The ideas described here were devised during the period 1976-1985. The first version was incorporated in the YMS sorting program at the earlier date. The second version has been incorporated in numerous YMS programs since the later date.

REFERENCES

[1] Knuth, D.E., The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley (1968)

[2] Stephenson, C.J., Practical methods for handling self-adjusting binary search trees, IBM Research Report RC 20542 (1996)

[3] Stephenson, C.J., Warped hashing, IBM Research Report RC 20542 (1996)

CJS, 1995-01-15.

Empty

THE SKEWED FILE TREE

This paper reviews the symmetrical N-way tree that is conventionally used in file systems, and describes an alternative "skewed" structure which has certain advantages.

CONVENTIONAL STRUCTURE

When storing files (or file directories) in blocks on a disk, a symmetrical N-way tree is sometimes used. The internal nodes of the tree comprise pointer blocks, and the leaves contain the data.

Fig. 1 illustrates such a tree with a depth of 3. In the figure, pointer blocks are shown as deep rectangles, and blocks containing data are shown as narrow horizontal ones. This is purely for reasons of exposition; all the blocks usually have the same size.

The disk blocks referred to in this paper are actually "logical" blocks. A logical block comprises one or more contiguous physical blocks which are read or written as a unit.

In the diagram, an unrealistically small blocksize is depicted. In practice a block is usually large enough to hold many pointers, not just 8 as shown.

The "pointers" in a pointer block are not of course ordinary pointers, to locations in memory. Instead, they are "disk" pointers. A disk pointer is simply a logical block number; it may have a value of (say) 12 or 3456789, which identifies block number 12 or block number 3456789. Pointers that lead to unoccupied regions (e.g. beyond the end of a file) are represented by a reserved value such as 0.

This structure is used in (for example) IBM's Conversational Monitor System for the "Extended Disk Format" file system (circa 1980). Unfortunately I cannot give a useful reference, since the documentation of the system internals has never been made generally available.

An important reason for using a tree (rather than a list) is so that a randomly selected byte in a file (or file directory) can be read or written without having to read all the preceding bytes in the file (or directory) to find its location on disk. The number of blocks that must be read to reach any given byte is equal to 1 + the depth of the tree (D = 0,1,2,...), which is a logarithmic function of file size, thus:

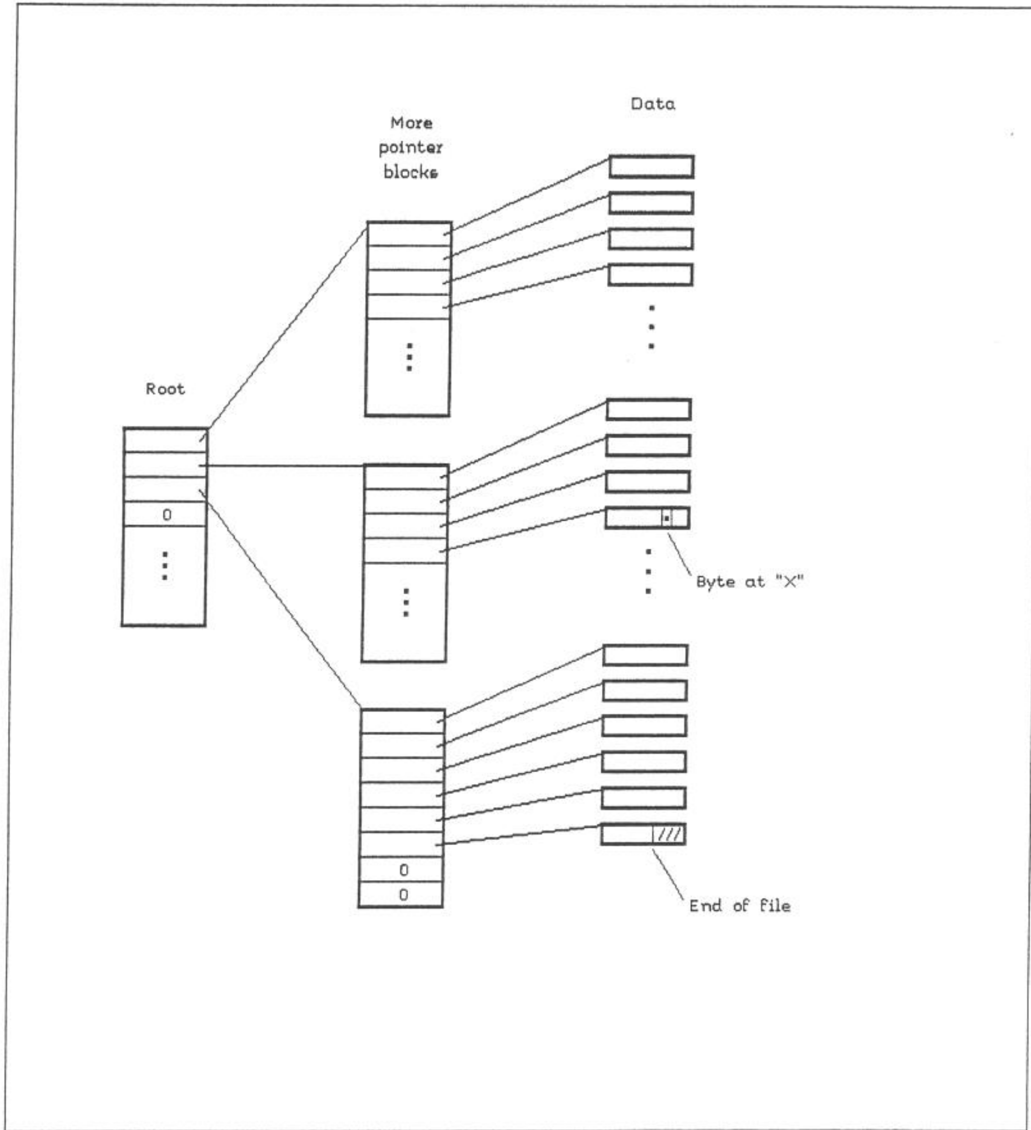


Fig. 1. Conventional tree structure for a file system.

$$D = \lceil \log_p S \rceil,$$

where p is the number of pointers per pointer block, and S is the size of the file in blocks, i.e. the number of blocks occupied (or partially occupied) by data (S = 1,2,...).

The tree depth is not usually large. Consider a blocksize of 1K bytes, and suppose disk pointers occupy 4 bytes each. We then get:

File size in bytes	Depth of tree
1 .. 1K	0
1K+1 .. 256K	1
256K+1 .. 64M	2
64M+1 .. 16G	3

(In practice not all file systems employ such regular structures. In Unix, for example, the "i-node" for a file contains pointers to a short list of blocks containing data, followed (as necessary) by a single pointer to a pointer block, a single pointer to a second-level pointer block, and a single pointer to a third-level pointer block (see for example [1]). So a file is mapped as a short list followed if necessary by one, two or three trees having different (but fixed) depths.)

READING A RANDOMLY SELECTED BYTE

Let the bytes of data in a file be numbered sequentially 0,1,..., N-1. This is called the "file address". Now suppose an application program wants to see the byte at file address X. Let us examine how the file system sets about getting it.

This can be regarded as is a two-step process. Step 1 constructs a "path" from the root to the byte at X. The path is a list of offsets into disk blocks. Deriving it is a purely computational procedure; it is a function of the tree shape and the value of X -- but it has nothing to do with the contents of the file. Step 2 uses the path, together with a disk pointer to the root, to read the necessary disk blocks, chaining through pointer blocks until it reaches the data.

To be specific, consider the situation depicted in Fig. 1. Let us imagine (unrealistically) that all the blocks have a length of 8 bytes and disk pointers occupy 1 byte each. Then p = 8, and X as marked has a file address of 93. Step 1 would compute the path by dividing X by p, D+1 times, and storing the remainders, from right to left. The result would be the following path:

1, 3, 5.

The first number (1) is the offset into the root from which the

next disk pointer is to be fetched; the second number (3) is the offset into this next pointer block from which the next disk pointer is to be fetched; and the last number (5) is the offset of byte X in the datum block.

Now suppose the (untyped) variable Y holds the block number of the root. Then Step 2 could obtain the needed byte as follows:

```
read block Y from disk to buffer;
D := first number from the path;
while path is not exhausted do
  begin
    Y := ptr from offset D of buffer;
    read block Y from disk to buffer;
    D := next number from the path
  end;
Y := byte from offset D of buffer;
```

Y now contains the byte at file address X.

(Actually this general method is not used for most file read operations. In practice most file operations are sequential, i.e. the byte or bytes read or written logically succeed the previous byte or bytes read or written. File systems handle these cases specially, bypassing the general method. This is not however relevant to the issues discussed here.)

A MESSY COMPLICATION

This all seems reasonably straightforward. There is however a complication that I have glossed over. The path that is contructed during Step 1 is a function of the file size, as well as the blocksize and the values of p and X. For small files, there will be only 1 or 2 numbers in the path; for large files, there will be 4 or more. This is an intrinsic property of the symmetrical tree.

Suppose that 11 more blocks of data are appended to the file shown in Fig. 1. It will then be necessary to increase the depth of the tree. This can be done by creating a new root, demoting the old root to be the first offspring of the new one (effectively shifting the entire old picture to the right), and then allocating two new pointer blocks and one new datum block for the new part of the tree. Thereafter (until the depth changes again) the path to X is:

0, 1, 3, 5,

even though X has not moved at all!

The fact that the path is a function of the file size can be a nuisance when a file grows and shrinks. Consider a long write operation that enlarges a file. Suppose it turns out that the tree depth must be increased. A new root is created and the tree is adjusted. Now suppose a problem arises that prevents the operation from completing, such as an insufficiency of free blocks. Correctly aborting the write operation can be complicated. Obviously any new pointer blocks that have been optimistically allocated but end up unused must be pruned from the right. But depending on exactly when the insufficiency occurs, it may also be necessary to remove the new root and restore the tree to its old shape and size. (In the case of very long write operations, it may be necessary to remove more than one new root, though this depends on the atomicity of file operations and the detailed semantics of the file system.) All this is possible; but it can be messy.

The changing tree depth also interferes with the optimum assignment of blocks on disk. As a general rule, it is desirable to place logically proximate blocks as close as possible to each other, so that they can be read back (sequentially) with the minimum of "seek" delays. When the tree depth grows, and a new file root is created, the file system will try to place the new root near the prior root. But by the time this happens, the blocks near the prior root will almost always be occupied already, so the file system will be forced to place the new root far away. (It would not be a good idea to leave a gap, ready for a few later roots, since this would fragment the block-space when files are small, which is common in practice.)

Note in passing that the offsets that comprise the "path" to a byte are somewhat analogous to the digits of a number in Arabic notation. If we continue with the unrealistic one-byte pointers postulated above, and let p = 8, the path to a byte in a file of depth 6 comprises an "octal" list such as:

0, 0, 0, 5, 3, 4.

Notice however that, unlike the situation with an Arabic number, we are not entitled to elide leading zeros. The length of the path depends on the file size, and all the digits are significant.

The directions you follow to find the 7th apple in a row of apples depends on the total number of apples. If there are 8 or fewer, all you will need is the number "7". But if there are more than 8, you will need "07", or "007", ...

AN ALTERNATIVE STRUCTURE

It is possible to map the bytes of a file (or file directory) into a regular but skewed tree instead of a symmetrical one. The

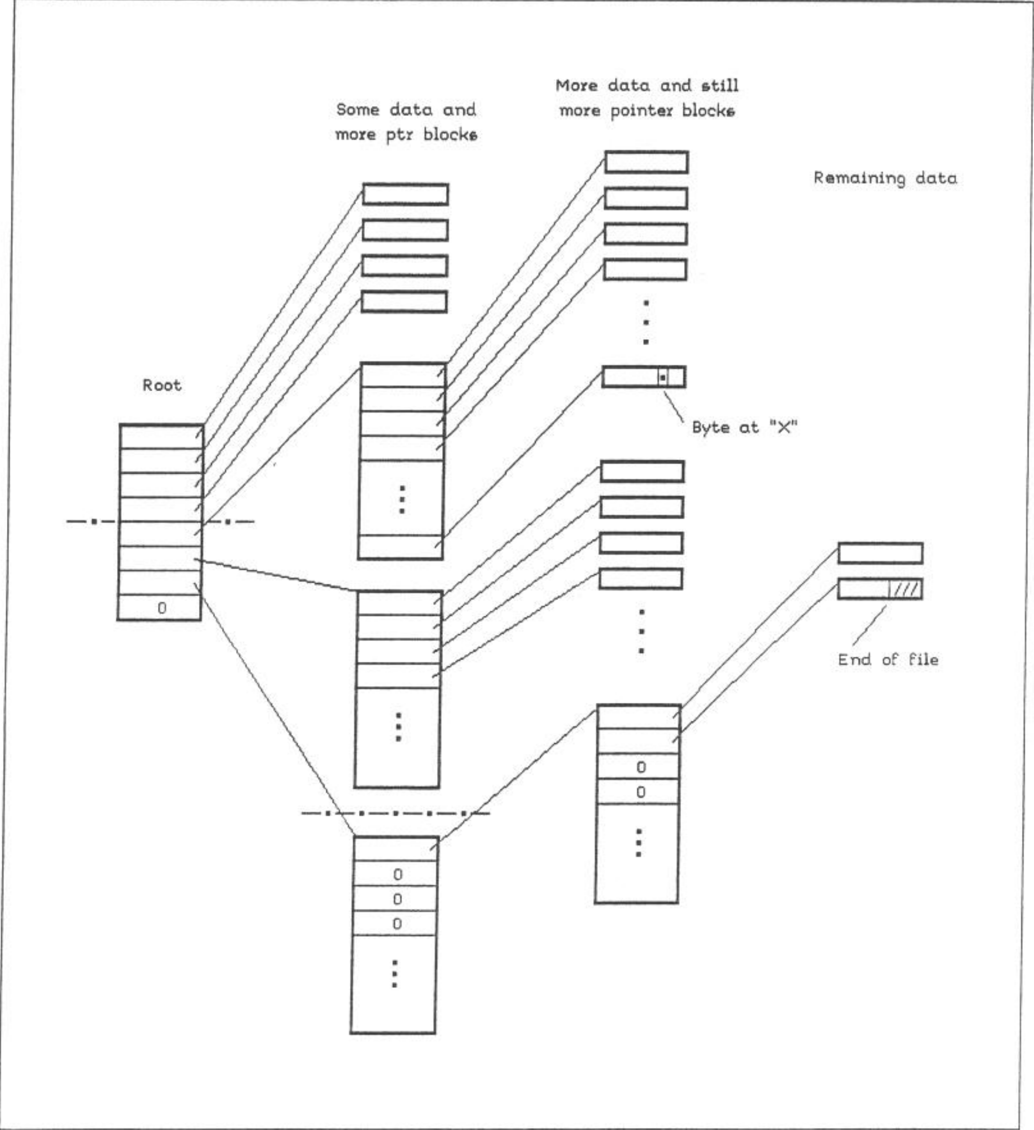


Fig. 2. Skewed tree structure for a file system.

skewed tree has roughly the same desirable performance for random accesses in large objects, but there are some interesting differences.

The skewed tree can be described as follows.

> The root of the tree is said to be at level 0. The root is divided into two equal parts: the first half contains disk pointers to the first generation of datum blocks (in level 1 of the tree), and the second half contains pointers to a second generation of pointer blocks (also in level 1).
>
> The second generation of pointer blocks (in its entirety) can be regarded as a pointer-space, which is conceptually divided into two equal parts: the first half contains pointers to a second generation of datum blocks (in level 2 of the tree), and the second half contains pointers to a third generation of pointer blocks (also in level 2).
>
> This third level of pointer blocks (in its entirety) can be regarded as a pointer-space, which is conceptually divided into two equal parts, and so on.

Fig. 2 illustrates such a tree, showing the same file (and blocksize) as Fig. 1.

In practice the maximum depth is not much greater than when a symmetrical tree is used. Consider as before a blocksize of 1K bytes and disk pointers that occupy 4 bytes each. We then get the following capacities for the "layers" in the tree:

Capacity	Depth
128K	1
16M	2
2G	3
256G	4

The skewed structure favours the low-numbered bytes in the file, which always reside at a tree depth of 1. This may or may not be desirable in its own right, depending on the reference patterns. But the interesting point is this. The path to a byte of data is a function of the file address (and of course the blocksize and pointer size), but it is independent of the file size.

To be specific, consider the situation depicted in Fig. 2. Imagine (as before) that all the disk blocks have a length of 8 bytes and disk pointers occupy 1 byte each; so $p = 8$ and $X = 93$. The path to byte X is now:

4, 7, 5,

and this remains constant for as long as the byte at X survives.

EVALUATION

This alternative structure is certainly not a panacea for file system designers. Compared with the symmetrical structure, it has some advantages and some disadvantages; but in any case its impact is insufficient to make file systems "easy".

The principal advantages are the ones I have already alluded to. The root never needs to be reassigned just because a file grows or shrinks. Pruning a tree is consequently straightforward. Pointer blocks with no descendants can be discarded; others must be kept. The root does not have to be treated as a special case.

(There may of course be other reasons for reassigning the root, having nothing to do with the structure that is used. In a "safe" file system, any update to an existing file requires that all the pointer blocks lying on the path to an updated datum block be reassigned, so that no blocks that appear in the "old" view will be overwritten by the "new" one. A skewed tree does not magically make this unnecessary.)

The fact that the path to a byte is independent of the size of the tree can be particularly beneficial when the structure is used for a file directory (as distinct from a file). As a general rule it is desirable to write only those parts of a directory tree which have been changed since they were last written. (Otherwise it would be expensive to make a small change to a small file in a large directory.) But when the directory was last written, it may have had a different size -- as well as different contents. It is hard to figure out which blocks need to be written if the path to an old leaf node is subject to change as a side-effect of directory growth or shrinkage.

The skewed tree has a few disadvantages.

The computation of a "path" to a byte at file address X is more complicated than with a symmetrical tree, since the data are in general distributed among several "layers" of the tree.

Another snag is that very small files require 2 blocks (rather than 1), since the root always contains pointers only. This can of course be circumvented by handling one-block files as a special case, and storing them on disk without a root. But special cases bring their own costs and complications, and a system designer is almost never entitled to claim they are "free".

Note in passing that a path to a byte in a skewed tree has

(like an Arabic number) the property that leading zeros will not occur (except in the case of a path of length 2 to the first block of data). The directions you follow to find the 7th apple no longer depend on the number of apples.

The analogy with numbers is not however exact. In a path through a skewed tree, the first number in the path already provides some information about the length of the path: offsets that reach into the second half of the root cannot for example appear at the beginning of a path of length 2.

Historical bote

The skewed tree described here has been used in the EM file system since 1975.

REFERENCE

[1] Bach, M., "The Design of the Unix Operating System", Prentice-Hall (1986)

CJS, 1995-01-18; revised 1996-09-13.

BULK HASHING OF FILE DIRECTORIES

This paper describes a method for fast searching of a file directory which does not impose any requirements on the format of the directory itself.

BACKGROUND

A file directory consists of a set of file directory "entries". An entry describes a file, and usually has a length of around 50 to 100 bytes. It contains the name of the file, its size, flags, time stamp, a disk pointer to its root, and so on.

A file directory usually occupies several or many "logical" disk blocks. A logical disk block typically has a length of between 512 and 4K bytes, and consists of one or more consecutive physical disk blocks which are read and written as a unit. Henceforth in this paper, the word "block" implies "logical block".

A file directory is written on disk, along with the files that are described in it.

At some stage, during the course of its operations, the file system reads a file directory from disk to memory. The directory may thereafter remain in memory, or the copy in memory may be discarded and the directory read anew each time it is needed. In either case the directory will from time to time need to be searched (to satisfy enquiries), and it may also need to be updated and written back to disk (if changes are made to the files).

When a file directory is updated, it is usually desirable to write only those blocks whose contents have changed since they were last written. Otherwise it would be expensive to change a small file in a large directory. This argues for retaining the block structure (as on the disk) when a directory is read into memory, and for "treading lightly" when making changes, e.g. avoiding unnecessary rearrangement when an old file is erased or a new file is created.

And of course it argues against keeping a file directory in sorted order, since the entire thing would then have to be written to disk every time a file whose name began with the letter "a" was created or erased!

But now the following question arises. If a file direct-

ory is not (and should not be) maintained in sorted order, how should it be searched?

SEARCH METHOD 1 -- SUPERIMPOSED BINARY SEARCH TREE

It would be possible to construct a binary search tree in memory, containing one node for each file in the file directory. A node would contain "left" and "right" tree pointers, and also a couple of words for the location of the associated entry in the file directory (e.g. the address of the directory block and the offset of the entry in it). The tree would be ordered by file name.

Then the expected search time would be O(log N), where N is the number of files in the directory.

Obviously this makes sense only if we can afford to keep a copy of the directory permanently in memory. Otherwise we would be unable to search the tree.

Provided we can afford the memory -- for the directory itself, plus 4 words per file for the tree -- this scheme would probably work quite well. If a file is erased, the corresponding node must of course be removed from the tree (in addition to the entry being removed from the directory itself). And if the last entry in the directory is then moved up, to fill the vacated slot, we must find the node for this file too, so we can update the node, to point to the new location. None of this is particularly difficult or expensive.

Although this proposal seems workable, I do not know of any system that employs it.

SEARCH METHOD 2 -- HASH MASKS

Here is a completely different method, which uses hashing.

When the file system first reads a file directory, it allocates and clears a hash mask for each directory block. For the time being, let us say there are about are 10 bits in a mask per file in the directory block; so if a directory block contains about 50 file entries on average, the masks might contain about 500 bits each. (More on this later.) Now for each file in the block, the system computes several independent hash codes with values in the range 0 through K-1, where K = the number of bits in the mask, and for each of these hash codes it sets the corresponding bit in the mask. Let us designate the number of hash codes per file as H, and for the time being let us say that H has a value of 4. (More on this later.) If none of the hash

codes were used more than once, about 40 per cent of the bits would finally be set; but in reality some hash codes will be repeated, and typically about a third of the bits will be set.

Now the copy of the file directory in memory may be discarded if necessary; but the file system keeps the hash masks in memory, and records which directory blocks they refer to.

Fig. 1 shows the relationship between the hash masks and the associated directory blocks.

When the file directory needs to be searched, this is what the file system does.

The system computes H hash codes for the file name that is desired, using the same hash function. Then for each hash mask, it inspects the bits corresponding to these hash codes. If any of these bits are zero, it immediately knows that the associated directory block cannot possibly contain the desired file; but if all the bits it inspects are one, it must visit the directory block and examine the entries in it.

So a hash mask is a sort of "template". It contains information on all the file names in the directory block "in bulk". The masks can be used to filter out most of the directory blocks. Usually, by inspecting at most 4 bits, the file system can avoid examining any of the entries in a directory block. For the remaining blocks, nothing is saved, but little is lost.

The effect of this is to improve the speed of a search by a "mere" constant factor -- but the factor may be large, and should not be sneezed at.

If the hash codes are uniformly distributed, and we use the parameters suggested above, the proportion of false hits will be about $(1/3)^4$, or 1 block in 81.

Furthermore, since very few blocks need to be visited, the penalty for not keeping a copy of the directory permanently in memory may be acceptable -- for search performance anyway. The occasional directory block that needs to be visited can simply be read anew from disk when it is required. (There are however other good reasons for keeping a copy of the file directory in memory if it is at all feasible, e.g. to handle file list operations.)

PARAMETERS FOR A HASH MASK

Let us define the "leverage" provided by a set of hash masks as:

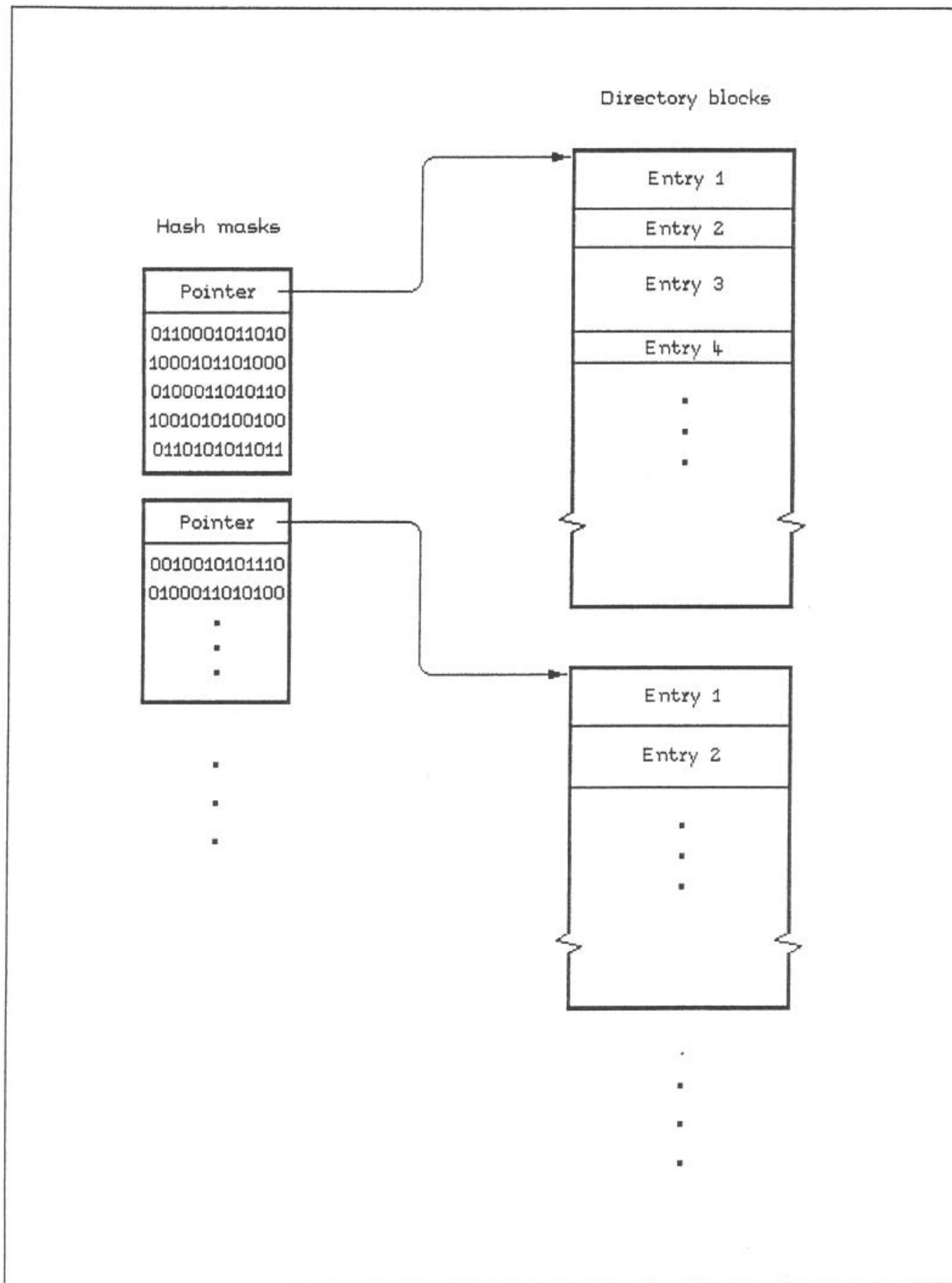


Fig. 1. Relationship between hash masks and their associated directory blocks. The pointers may be ordinary memory pointers (if a copy of the directory is maintained in memory) or disk pointers (otherwise).

$$L = \frac{D}{E},$$

where: D = the number of blocks in the directory (or the number of blocks preceding the one containing the desired file name, if this exists),

E = the number of false hits, i.e. the number of directory blocks whose contents are examined (but which do not contain the desired file name).

So a hash mask that provides no useful information has a leverage of 1; and a hash mask that works perfectly exhibits an infinite leverage.

In the previous section I suggested the following parameters as a rough guide:

$$B = 10,$$
$$H = 4,$$

where: B = number of bits in hash mask per file in the directory block,

H = number of hash codes per file;

and I stated that with these values:

$$L = 81 \text{ approximately.}$$

This assumes (or course) that the hash codes are distributed uniformly.

Let us now look more closely at how the leverage depends on the values we choose for B and H.

For any given value of B, the maximum leverage obtains for that value of H which ends up setting half the bits in the mask. Using a larger value of H is counterproductive, since (a) it reduces the leverage, and (b) it demands more work -- to compute the additional hash codes, and to set and examine the additional bits.

In fact it is usually desirable to use a value of H which ends up setting somewhat fewer than half the bits, since the loss in leverage can be small, and less work is involved.

Fig. 2 shows how the leverage of a hash mask depends on the values of B and H -- assuming as before that the hash

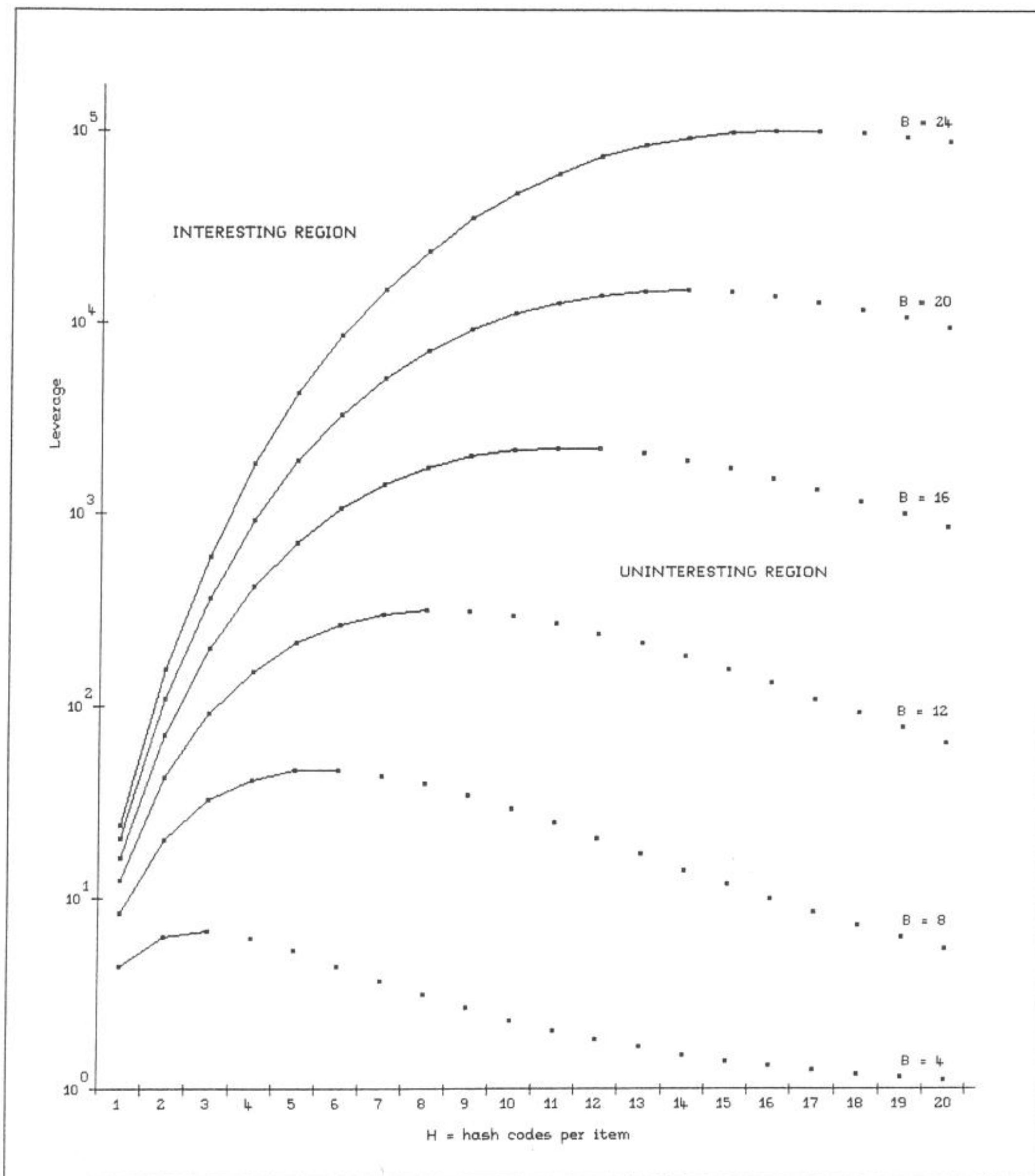


Fig. 2. The leverage of a hash mask as a function of B (bits in mask per item) and H (hash codes per item) assuming uniformly distributed hash codes.

codes are uniformly distributed.

Consider the case in which B = 8 bits per file. The maximum leverage (of about 46) occurs when H is 6. But little is lost if H is 4, which yields L = 42.

Similar conclusions apply to the other cases. If B = 16 bits per file, the maximum leverage (of about 2175) occurs when H is 11. But the leverage is almost as good if H is 8, which yields L = 1737.

Actually, if a leverage around this value is desired, it might be even better to set B = 24 and H = 4, which yields L = 1799. (This assumes memory is made of silicon, not gold, and an extra byte per file will not break the bank.)

OBSERVATIONS AND FINE POINTS

1. To be effective, bulk hashing requires a good hash function which yields a reasonably uniform ditribution of hash codes. A universal hash function would be desirable (see [1]).

2. In practice it may not be necessary to compute H separate hash codes for each file name. Suppose for example that the number of bits in a hash mask is 1024; then the "short" hash values required to set (or test) the bits will be only 10 bits long. If the hash function yields (say) a 32-bit number, up to 3 short hash values can be extracted from a single result.

3. Hash masks can be constructed discontiguously from the file directory blocks, and do not depend on the detailed structure of the directory. They can be used with almost any existing file directory, without requiring changes to the layout on disk or the handling of the directory itself.

4. A potential disadvantage of bulk hashing (as so far described) is that the entire mask for a directory block must be recomputed from scratch after a change to any file name in the associated directory block.

   On most machines the time required to recompute the hash mask for a directory block is less than the time required to update the directory on disk, so the maintenance of the hash mask is not normally a performance issue.

   If however the file system is capable of supplying a private "read-only" view of a directory, and it allows

files to be "erased" from this view, or "renamed", without affecting the persistent picture, then there exists the possibility that many such erasures or rename operations might be performed without reading or writing the disk. In these circumstances a noticeable proportion of the time required to perform these operations could be spent maintaining the hash masks.

This disadvantage can be alleviated by allowing the hash masks to become slightly "stale". When a file is renamed, or when entries have to be shuffled around and a new name is brought into a directory block, the hash codes for the new name are computed, and the corresponding bits are set in the mask; but no attempt is made to clear the bits for the old name (the one that is no longer present). Gradually, as such changes are made, the hash masks will become increasingly polluted, and the leverage will suffer. A pollution count can be maintained, and the mask can be recomputed from scratch occasionally, before the leverage falls to an unacceptable level.

Historical note

Bulk hashing has been used in the EM file system since 1976.

REFERENCE

[1] Carter, J.L. and Wegman, M.N., Universal classes of hash functions, J. Computer and System Sciences, 18, 2 (1979)

CJS, 1995-01-26.

WARPED HASHING

Hashing can be "abused" so that it selects among homonyms in addition to playing its usual role of improving search performance.

BACKGROUND

A good way to organize a symbol table is to use a combination of hashing and binary search trees. The hash table consists of an array of pointers to the roots of a set of disjoint binary search trees. To find (or enter) a symbol, a cheap hash code is computed for it, and used to index into the array. This selects the appropriate tree, which is then searched by one of the tree methods that yields good amortized performance, such as "Splay" or "Splice" (see [1]).

It is desirable, but not necessary, for the hash codes to be uniformly distributed. The binary search trees can be made to perform well even when there are many symbols in the same class. The hashing may be regarded as an "almost free" bonus, which usually distributes the symbols among several trees of moderate size (instead of having them all in one big tree). It provides a useful performance benefit, with little computational penalty.

For example, the hash code may simply comprise the low-order k bits of the sum of the EBCDIC or ASCII characters comprising the symbol name, where k = 5 or 6 (say). This can be computed cheaply, while the symbol is being parsed.

Finally, if the entire symbol table is required in alphabetical order (e.g. for inclusion in a program listing), the trees can be transformed to ordered lists, which can then be merged into a single linked list (see [2]). An explicit sort is never required.

HOMONYMS

Sometimes the same name can stand for more than one thing. Here is an example.

In S/370 and S/390 assembler language, a "blank" name can be used for a control section, a dummy section and a common area (see [3,4]). In fact all three of these "symbols" may appear in the same program. The language is defined such that when a reference is made to a blank name, it is known from context which of the three entities is being referred to.

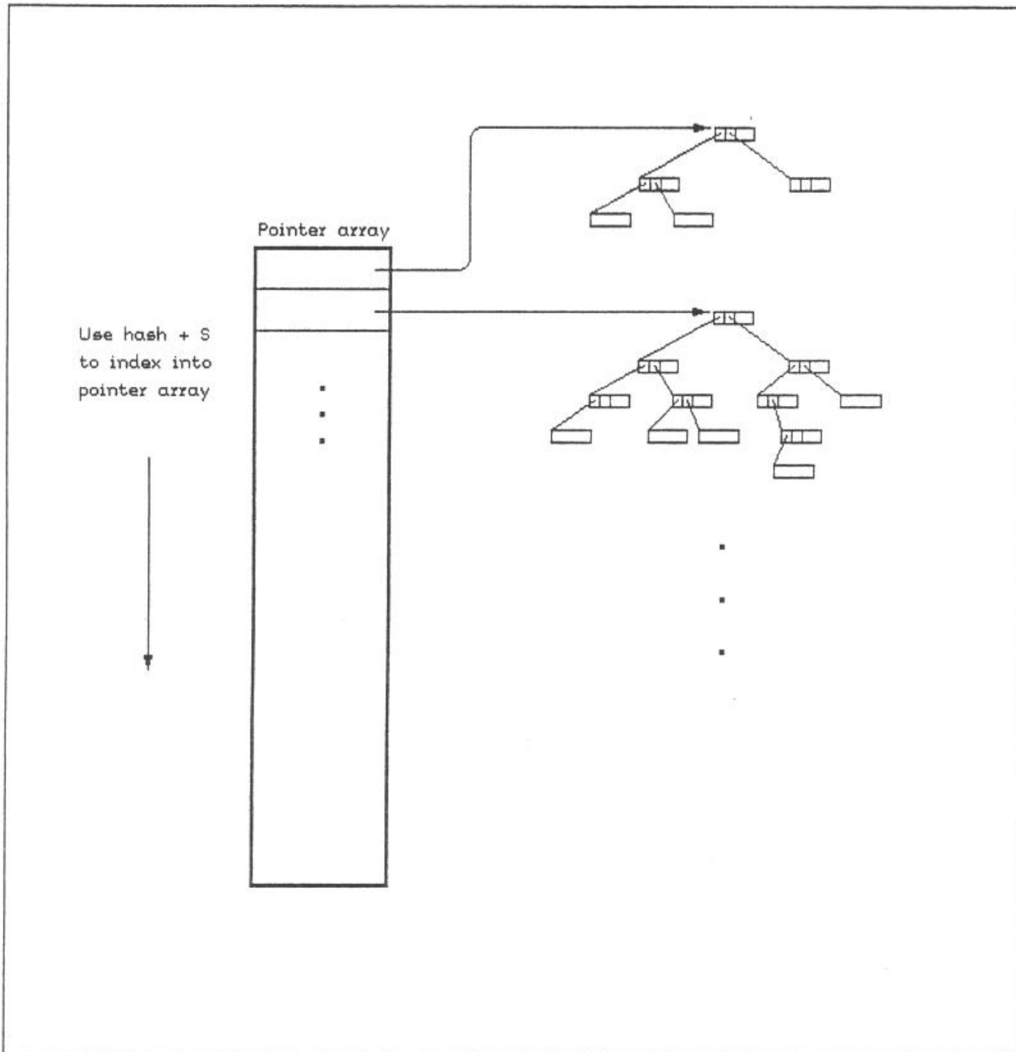


Fig. 1. "Warped" hashing. "S" is the selection code.

How should an assembler handle such homonyms in its symbol table?

It would not do to have three separate symbol tables, one for control sections, a second for dummy sections and a third for common areas, since most symbols (such as X) can be used for at most one entity, and the assembler would be constantly searching all three tables.

Perhaps the obvious answer is to handle a blank name as a special case, and maintain the information on the three entities in three symbol table entries which are not stored in the main symbol table. This would certainly work, but it would require special handling in several places. The most annoying one concerns the preparation of the symbol table for the listing. How and when should the blank names be put into the main symbol table so that they will be listed (correctly positioned) with the other symbols? Should we insert them on the fly, while linearizing or merging the trees? (Sounds expensive.) Or should we postpone the issue until the main symbol table has been linearized, and then employ a separate traversal to find the right place to insert the blank names at the last moment? (Sounds messy.)

There is a simpler method.

PERVERTING THE HASH CODE

The primary purpose of hashing is to direct attention to the tree that contains (or will contain) some required symbol, and away from the other trees, which cannot possibly contain it -- since all the symbols in these other trees have a different hash code.

But there is nothing to stop us from "perverting" the hash code, and deliberately putting some particular symbol into the wrong tree! Of course this will do us no good -- since the symbol will never be found -- unless we always pervert the hash code for this symbol in the same way, and always visit the same wrong tree.

This suggests an elegant solution to the "homonym" problem outlined above. We can add a "selection code" S to the hash code for a blank name, where (say) S = 0 for the blank control section, 1 for the blank dummy section, or 2 for the blank common section. Then as far as the symbol table is concerned, they will be treated as three different symbols, in three different trees. There will never be a clash.

Of course this requires that the number of hash classes be at least as great as the maximum number of homonyms for a name.

When the trees are linearized and merged, the blank names will automatically be merged with the other names, and will appear in the right order. The linearizing and merging procedures have no interest in the hash codes: they will not even notice that some names appear more than once, and some instances are in the "wrong" trees.

The technique is depicted in Fig. 1.

Historical note

Warped hashing has been exploited in "Phantasm", an experimental assembler which runs under EM-YMS, since 1994 (see [4]).

REFERENCES

[1] Stephenson, C.J., Practical methods for handling self-adjusting binary search trees, IBM Research Report RC 20542 (1996)

[1] Stephenson, C.J., On traversing and dismantling binary trees, IBM Research Report RC 20542 (1996)

[3] High Level Assembler for MVS & VM & VSE Language Reference SC26-4940, IBM (1995)

[4] Stephenson, C.J. and Hack, M., Phantasm: An experimental assembler for System/390; available on request (1995)

DOUBLE-ENDED MEMORY ALLOCATOR

Joint work with Paul R. Kosinski*

This paper describes an easy way of managing two logical pools of memory in a single linear address space.

POOLS OF MEMORY

Operating systems and complicated application programs often find it desirable to maintain more than one "pool" of dynamically allocated memory. An operating system may for example maintain one pool for satisfying requests from application programs (call this Pool A), and another for satisfying requests from the system itself (call this Pool B).

One reason for doing this is that the characteristic lengths and lifetimes tend to be quite different. A system control block may occupy a few hundred bytes and survive for months; whereas a compiler work area may occupy several megabytes and live for a few seconds. Allocating both these from the same pool could give rise to be unnecessary long-term fragmentation of memory.

In some situations two pools of memory are sufficient. When this is the case, an obvious way to manage the pools is to start one of them at the "high" end of memory (and work down), and to start the other at the "low" end (and work up). Between them lies a "wilderness" which is gradually consumed from the two ends. Neither of the pools need have a rigid limit to its growth; so the system will report an insufficiency only if (a) there is insufficient free space in the required pool, and (b) the remaining length of the wilderness is also insufficient.

When a piece of memory abutting the wilderness is deallocated (in either pool), the pool can shrink, allowing the wilderness to regain some of its lost ground.

Fig. 1 illustrates the arrangement.

This usually works tolerably well in practice. There are however pathological situations in which an insufficiency is reported when there is still plenty of available memory -- but in the wrong pool.

Let Pool A begin at 0 (and work down), and let Pool B end at 100 MB (and work up). Initially the wilderness occupies the entire space

* Present address: Digital Equipment Corporation, 334 South Street, Shrewsbury, Mass.

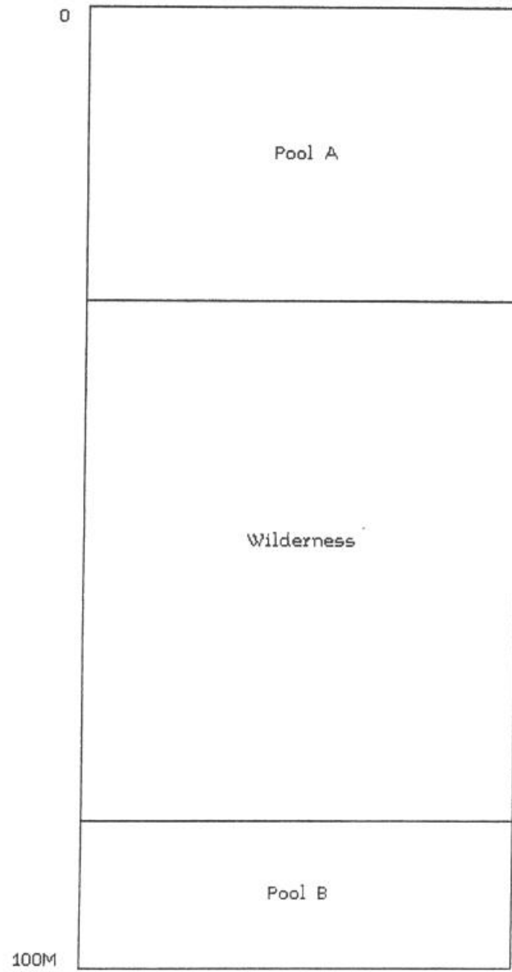


Fig. 1. Two pools of memory with intervening wilderness.

(from 0 to 100 MB). Suppose that 90 MB of short-term memory is allocated from Pool A (its address will be 0), followed by a separate piece of length 1 MB (its address will be 90M). The wilderness now occupies the region from 91 MB to 100 MB. Almost immediately the 90 MB piece is deallocated, leaving plenty of free space in Pool A. Next a request is made for 10 MB from Pool B -- which cannot be satisfied.

In principle it would be possible for the Pool B manager to make an emergency request to the Pool A manager for the needed space, but for a variety of reasons this is not always feasible. The two pools must usually remain distinct (so that the boundaries with the wilderness can be properly handled). Also the memory that the Pool B manager obtained from Pool A might end up near the beginning of Pool A, or in the middle of Pool A, depending on the condition of the pool at the time. Then even if all the memory that has been intentionally allocated from Pool A is deallocated, Pool A may develop more-or-less permanent "holes", caused by emergency requests from the Pool B manager ... so that ultimately the Pool A manager may also be unable to satisfy reasonable requests.

THE CARTESIAN TREE

A nice way to manage a pool of dynamically allocated memory is to maintain the available pieces of memory in a "cartesian" tree. A cartesian tree is a binary search tree in which the "values" of the nodes are ordered horizontally (as in any binary search tree), and their "weights" are ordered vertically such that no son is heavier than its father (see [1]). In this application the "values" of the nodes are simply their addresses, and their "weights" are their lengths (see [2]).

The vertical ordering helps the job of allocation, since it allows a suitable piece to be identified without visiting any nodes of inadequate size (which are often numerous). The horizontal ordering helps the job of deallocation, since the neighbours of the deallocated piece can be found by performing a binary search (which does not usually need to visit many nodes).

Fig. 2 shows a small example of such a tree.

It is possible to allocate the leftmost adequate piece from such a tree by sliding down the left-hand branch until reaching a node whose left son is insufficient. This is called "leftmost" fit; it is functionally equivalent to "first fit" from a list, but it is usually much faster in execution.

Alternatively (and just as easily) it is possible to allocate the rightmost adequate piece, by sliding down the right-hand branch until reaching a node whose right son is insufficient. We will call this "rightmost" fit; it is functionally equivalent to "first fit" from a list chained in reverse address order (but again it is much faster).

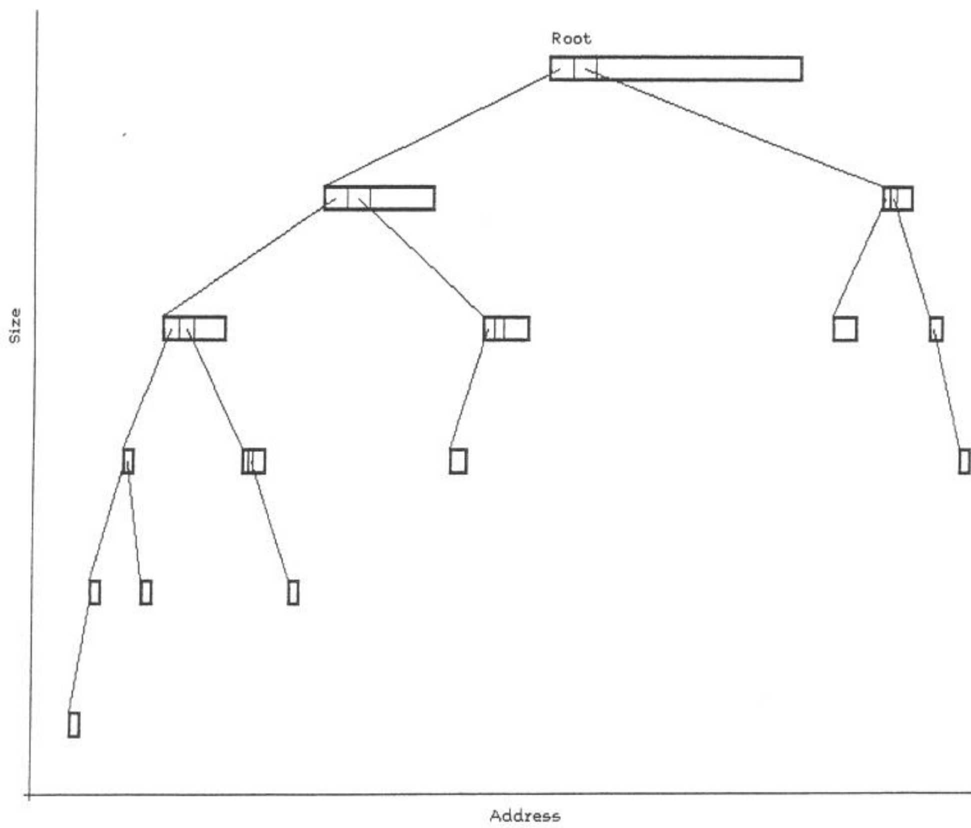


Fig. 2. Use of "cartesian" tree for dynamic memory allocation.

THE DOUBLE-ENDED ALLOCATOR

It is possible for two memory allocators to use the same cartesian tree, and allocate from opposite ends, using leftmost fit and rightmost fit.

There is no explicit wilderness. Initially all the memory resides in the root of the tree. The Pool A manager removes pieces as near as possible to the left-hand end, and the Pool B manager removes pieces as near as possible to the right-hand end. When pieces are deallocated, they are coalesced with their neighbours, if any, without regard to which Pool manager allocated them.

If memory gets tight, or becomes fragmented, it may happen that some of the pieces allocated by the Pool A manager have a larger address than some of the pieces allocated by the Pool B manager (and vice versa). But this will happen only when it is the sensible thing to do; and it will then happen without the two managers needing to take special action to achieve it -- or even being aware that it has happened.

And when this happens, it will cause the minimum of disruption. Consider the same scenario as before. When the Pool B manager is asked for 10 MB, it finds it at 80 MB, thereby keeping the pieces of memory it allocates far away from the end of the tree that is favoured by the Pool A manager.

In any case, an insufficiency is reported only if it is physically impossible to satisfy a memory request.

If this technique were used with an allocator that employed a list instead of a tree, the list would need to be doubly-linked (or employ exclusive-or pointers), so that it could be traversed from either end. (I do not recommend this. List-based allocators perform badly when the number of fragments is large: see [2].)

Historical note

A double-ended memory allocator as described here has been in use in YMS since 1985.

REFERENCES

[1] Vuillemin, J., A unifying look at data structures, CACM 23, 4, p. 229 (1980)

[2] Stephenson, C.J., Fast Fits - New methods for dynamic storage allocation, Symposium on operating system principles, Bretton Woods (1983)

CJS, 1995-01-15.

Empty

ON THE LATENCY OF BOYER-MOORE SEARCH

In 1977 Boyer and Moore described an ingenious algorithm for searching a string for the first instance of a pattern. Before beginning the search, the program inspects the pattern and prepares an integer array Q[0], Q[1], ..., Q[q-1], where q is the size of the alphabet. This array is used during the search to advance through the string with a variable "stride" whose size depends upon the characters encountered.

The algorithm usually avoids having to examine all the characters in the string preceding the match, and handsomely outperforms simpler methods. There is however a disadvantage. Even for short patterns and short strings, the program must prepare the entire array Q, which typically contains 256 entries. There are situations in which the time required for this preparation may exceed the time required to perform the entire search by a more straightforward method.

This paper describes a way of reducing the fixed overhead and thereby rendering the Boyer-Moore algorithm more widely applicable. It also contains several more or less unrelated observations on the implementation of Boyer-Moore search.

1. BACKGROUND

Here is a review of the basic ideas involved in a Boyer-Moore string search (see [1]).

Instead of examining the string from left to right, looking for a match with the head of the pattern and working forwards (as in a "brute force" search), a Boyer-Moore search darts to and fro in the string, first looking for a match with the tail of the pattern, and then when necessary scanning backwards to check the preceding characters. The beauty of this is that, as a side-effect of encountering a mismatching character in the string, the program can usually determine that there is no possibility of another tail-match in the next few characters (following the mismatch). Most of the time it can stride ahead without having to examine all the characters in the string, even once.

What follows comprises a slightly more formal description.

We will treat the string and the pattern as indexable arrays of characters. Specifically, we will search string[1], string[2], ..., string[stringlen] for the first instance of pattern[1], pattern[2], ..., pattern[patlen]. For simplicity we assume that 0 < patlen <= stringlen.

Phase 1. Preparation.

The integer array Q[0], Q[1], ..., Q[q-1] contains one entry for each character in the alphabet. If a character occupies a byte (as is typically the case), then q = 256.

Before a search, this array is prepared, from the pattern, so that its entries contain the lengths of the strides that will be taken during the search when the corresponding character is encountered in the string (see below).

Specifically, all the entries in the array Q are first set to patlen, the length of the pattern. Then those that correspond to characters that appear in the pattern are adjusted as follows.

We visit the characters pattern[j] from left to right (j = 1,2,...,patlen). For each one:

> the bits comprising the pattern character are treated as an unsigned radix 2 integer which is used to index into the array Q;
>
> the indexed entry is set to patlen-j.

An exception is made for the array entry that corresponds to the very last pattern character, which is set to the maximum unsigned integer (all ones). For the other pattern characters, the value in the array represents the distance of the character from the rightmost character of the pattern. If a character appears more than once in the pattern, the value in the array is determined by the position of the rightmost instance. For characters that do not appear in the pattern, the value in the array remains set to patlen.

Phase 2. Search.

In this phase we stride through the string, without usually visiting all the characters, and stop to check only when necessary.

We will use "i" to index into the string. We start with i = patlen; so the first character to be visited is string[patlen]. This is the first one that stands any chance of lining up with the last character of the pattern.

a. "Fast" loop. Stride boldly through string.

The array Q supplies the lengths of the "bold" strides that may safely be taken, without risking that a match might be missed. (In the previous literature, what I am calling a "bold stride" is usually referred to as "delta1".) Here

is how it works.

The bits comprising string[i] are treated as an unsigned radix 2 integer which is used to index into the array Q. The value of the indexed entry is added to i. The new string[i] is then visited and handled in the same way.

This "fast" loop continues until i > stringlen (or until the addition yields a "carry" out of the high-order bit position). This occurs when either (1) we run off the end of the string without having found a match (end of search), or (2) we visit a character in the string that matches the last character of the pattern. This is a "tail-match".

The latter situation is distinguishable by the fact that the last stride has the special value of all ones. As a result of the last addition (which yielded "carry"), i now contains the index of the character preceding the last one visited.

In this case we fall through to "b" below.

b. "Slow" loop. Check for complete match.

At this point we know that string[i+1] = pattern[patlen]. Now the possibility exists that string[i] = pattern[patlen-1], string[i-1] = pattern[patlen-2], and so on, yielding a complete match starting at string[i-patlen+2]. To check for this, we scan backwards through the string, beginning at string[i], comparing the characters one by one with the corresponding characters of the pattern. This "slow" loop stops when either (1) we establish that there is a complete match (end of search), or (2) we encounter a mismatching character.

In the latter case, we fall through to "c" below.

Note in passing that when striding forwards in the "fast" loop the array Q is used and the pattern itself is not referred to. When scanning backwards in the "slow" loop, examining the characters preceding a tail-match, the pattern is used and the array is not referred to.

c. Stride hesitantly and continue search.

Arrive here if a mismatching character is found when scanning backwards in the "slow" loop.

Before restarting the "fast" loop, we must adjust the string index i; otherwise we might repeatedly trip over the same tail-match, which we now know is bogus.

I will call this adjustment a "hesitant" stride (since it is preceded by the "slow" loop, which scans backwards).

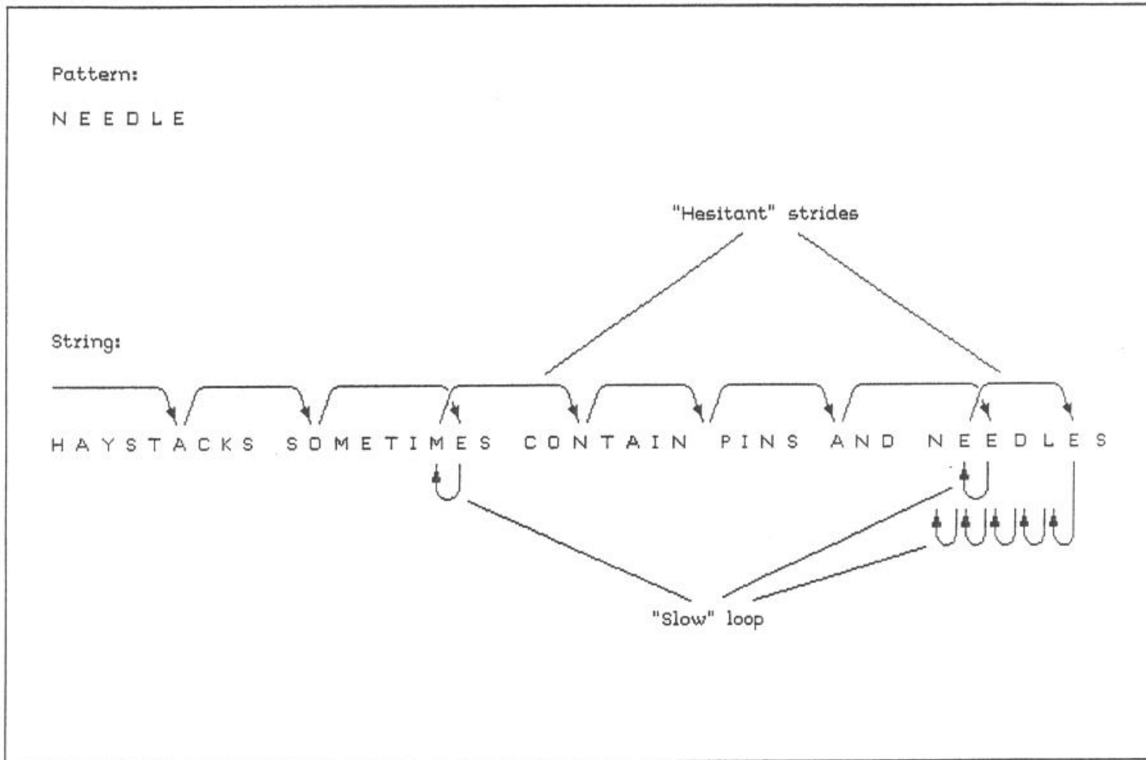


Fig. 1. Example of Boyer-Moore search. The first hesitant stride is obtained from the entry for "M" in the array Q. The second one is obtained from the 5th entry of the "delta2" table.

It is always permissible to set i to 1 + the index of the tail-match. This results in starting the "fast" loop on the following character. It is sometimes safe, however, to take a longer hesitant stride, which yields better average performance and helps some awkward cases (see below for details).

In any event, having adjusted i by the appropriate hesitant stride, we restart the "fast" loop (see "a" above).

Fig. 1 shows an example of a Boyer-Moore search in action.

Determination of hesitant stride after a bogus tail-match

The determination of the longest safe hesitant stride (following a bogus tail-match) is the most complicated and difficult part of the Boyer-Moore algorithm.

This part of the algorithm is only marginally relevant to the topic of this paper. The following information is included mainly for reasons of completeness.

Here are three strategies for determining the hesitant stride, in increasing order of complexity and efficacy.

Resume on character following tail-match

As noted above, it is acceptable to resume on the character following the tail-match, but it can be wantonly inefficient. Consider for example a search for "AAABB" in "XXX BA BA BA BA ...". It turns out that every A, B and blank in the string will be visited, the A's and the B's in the "fast" loop and the blanks in the "slow" loop.

Try to use the bold stride from the array Q

If the mismatching character in the string does not appear in the pattern (or if it appears, but not very close to the right end), we may be able to make better progress by pretending we encountered the mismatch in the "fast" loop instead of in the "slow" one.

Specifically, the bits comprising the mismatching character are treated as an unsigned radix 2 integer which is used to index into the array Q. The indexed entry supplies the length of the bold stride that would have been taken if the character had been encountered in the "fast" loop. If this stride, added to the index of the mismatching character, takes us beyond the character following the tail-match, then it makes sense to set i to this value.

The algorithm that uses this method of determining the hesitant

stride is sometimes called the "simplified" Boyer-Moore algorithm.

Now a search for "AAABB" in "XXX BA BA BA BA ..." visits only half the A's, B's and blanks in the string -- which represents a worthwhile improvement. This approach does not however yield good results for all awkward cases. Consider for example the search for "ABBBB" in "BBBBBBB...". The mismatching string character is always "B"; the bold stride in the array Q is unusable; and most of the characters in the string will be visited 5 times. In general the cost of such a search is O(patlen × stringlen), which is very bad.

Use "delta2" table (in addition to the array Q)

Consider the last example above. After the "slow" loop has found a "B" where an "A" was desired, it is clear to you the reader that it would be safe to set i to 5 + the index of the tail-match, for you know there are no instances of "A" in the substring that has been examined by the "slow" loop.

It is possible to codify this knowledge in an additional table containing patlen integers. Boyer and Moore called this "delta2". Like the array Q, this table is prepared ahead of time by examining the pattern. Entry j of the table (j=1,2,...,patlen-1) specifies how far it is safe to advance after finding a mismatch at pattern position j, irrespective of the string character that caused the mismatch. See the Appendix for more information on delta2.

Now the hesitant stride can be determined by examining the appropriate entry in the array Q (corresponding to the mismatching character), and also the appropriate entry in delta2 (corresponding to the point of mismatch), and using the one that moves further to the right.

This is the "complete" Boyer-Moore algorithm.

The time required to construct the delta2 table is O(patlen), and is independent of the alphabet size. This will be relevant later.

2. SEARCH TIME AND PREPARATION TIME

Notice that the expected speed of the search phase increases with the pattern length. If the pattern length is 10, the average bold stride may with luck be 8 or 9; but if the pattern length is 2, the bold stride will be at most 2. If the pattern length is 1, the method offers no algorithmic advantage over brute force.

Observe also that some patterns (and some strings) inherently favour Boyer-Moore, while others do not. The Boyer-Moore algorithm behaves very nicely if we search for:

"aaab" in "aaa-aaa-aaa";

but it is hard to beat brute force if we look for:

"abc" in "--bc"

or: "abbb" in "bbbbbbbb".

Happily, it has been proved that the worst running time for the search phase of the complete Boyer-Moore algorithm is O(stringlen): see Knuth, Morris and Pratt [2]; Guibas and Odlyzko [3]; and most recently Cole [4]. To be more precise, Cole showed that the maximum number of character-comparisons required for a failed string search is less than 3 × stringlen. (In this context, the use of a character to index into the array Q counts as a "comparison".) This kind of result is important, for it tells us that the search performance of the algorithm, when faced with the most unfavourable pattern and string, is comparable with that of the best other known methods (not counting those that preprocess the string). So the algorithm has excellent typical performance, and acceptable worst-case performance.

This is not, however, the whole story. Even in its simplified form, a Boyer-Moore search entails preparing the array Q[1..q-1], where q = alphabet size. The time required for this preparation is O(q + patlen). Typically q is 256. When many short searches are performed, with patlen << q, the accumulated time taken to prepare this array may exceed the total time required to locate the patterns (or to fail) by brute force.

There are two effects at work here. For one thing, short patterns derive the least benefit from the method. For another, short patterns are likely to occur more often in the string, and therefore require the shortest searches.

The original authors were aware of these points and warned that it may not be advisable to use the method when the search is expected to be short. There have been a number of subsequent performance studies, which inter alia have investigated the correlation between pattern length and search time, e.g. when using text editors [5,6,7,8]. These conclude that the Boyer-Moore algorithm (or a derivative) is to be preferred when the pattern length exceeds a threshold (such as 2 or 4), but they all express reservations about using it unconditionally, when the pattern is very short, because of the initial overhead.

The following section describes how the preparation time can be reduced to O(patlen), thereby eliminating these difficulties.

Note that the problem to be solved is to reduce the preparation time for the array Q. The delta2 table is more complicated, but requires only time O(patlen) for its preparation. We can be a little more precise. If the new algorithm given in the Appendix for preparing delta2 is hand-coded, it executes about a dozen instructions be-

fore entering the first loop and about two dozen × (patlen-1) instructions thereafter. (These are all fairly simple instructions. The counts are not much affected by the particular instruction set provided that a byte or a word can be fetched or stored in a single operation.) For a pattern length of 2 or 3 we therefore get a total of roughly 36 or roughly 60 instructions. This may not be negligible, but it is a lot cheaper than preparing the array Q for an alphabet of 256 -- which requires about 300 instructions, more or less, depending on how much we unroll the "store" loop.

Actually the previous literature suggests that many experimentalists eschew the complete Boyer-Moore algorithm, and use the simplified one (or a close relative), which does not use delta2. The main advantage of the complete algorithm is that it limits how bad the performance can be in pathologically awkward cases, which (it is said) do not arise very often in practice.

3. IMPLEMENTATION OF "FAST" LOOP

We must make sure that whatever changes we make to improve the latency do not interfere with the speed of the "fast" loop -- since this dominates the typical search performance. I will therefore approach the problem by examining the implementation of the "fast" loop.

Version 1. Straightforward transliteration of the Boyer-Moore "fast" loop.

In their paper, Boyer and Moore describe how the "fast" loop can be cunningly implemented in three instructions on DEC's PDP-10, which was a word-addressed machine. The following represents a straightforward transliteration to S/370, which is a byte-addressed architecture (see [9]). We need one more instruction, in order to quadruple the value of the character (treated as a radix 2 integer) and index into the integer array Q, which contains one word per entry.

Note that the string is addressed by a register that contains its <u>ending</u> address (i.e. the address+1 of the last byte) and indexed by a register which (if regarded as a 2's-complement integer) contains a negative value. This results in a "carry" out of the index register when the string is exhausted, or when a match is encountered with the last pattern character.

```
*
*    ---------  Fast loop, version 1  ---------
*
*    On arrival here:
*
*       0 <= R0 < 256    (in readiness for "IC")
*       0 <= R1 < 2**30 (in readiness for "SRDL")
*       R2 = (patlen - stringlen - 1) mod 2**32 =
*        offset (from right end of string) of the
*        1st character to be visited in the string
*       R3 = ending addr of string to be searched
*
*    Also:
*
*       Q[0..255] is an integer array containing
*        the bold strides for the particular pat-
*        tern (see text for details)
*
FAST1     IC     R0,0(R2,R3)      Character from string
          SRDL   R0,32-2          R0 := 0; R1 = 4 × char
          AL     R2,Q(R1)         Take a bold stride and
          BL     FAST1             loop unless "carry"
*
*    Fall from loop when either (a) we encounter a
*    character in the string that matches the last
*    pattern character (in which case Q(R1) is all
*    ones), or (b) the string is exhausted (and
*    Q(R1) contains some other value).
*
```

The only tricky thing here is the use of the "double shift" instruction, which multiplies the character value by 4, and simultaneously clears R0 in readiness for the next "Insert Character" instruction.

Unfortunately shift instructions, and particularly double shifts, are liable to take several machine cycles. Replacing the shift by an explicit multiplication would only make matters worse. This leads us to version 2.

Version 2. Use a byte array (instead of a word array).

If we change the array Q so that it contains a byte per entry (instead of a word), then we can index into it with the raw character code and avoid the multiplication.

There are two snags with using a byte-wide array. (a) We cannot store long strides for long patterns. I will postpone this issue and temporarily restrict the pattern length to 254. (b) We cannot store a dummy stride (in the entry corresponding to the last pattern character) that is large enough to force the addition operation in the loop to yield a "carry" out of the high-order bit position, and so set the condition code required for

loop control. The latter problem can be circumvented as follows.

We will prepare the table to that the entries contain 1 + the value of the bold stride (instead of the bold stride itself), and store zero in the entry that corresponds to the last pattern character (insead of all ones). So the entries for characters that do not appear in the pattern are set to patlen+1, and the entries for characters that do appear are set to d+1, where d = distance from the last pattern character of the rightmost instance of the character (d = 0,1,...,patlen-1); except that the entry corresponding to the last pattern character is set to 0 (instead of 1). It follows that a value of 1 appears nowhere in the table.

In the "fast" loop, we will subtract 1 from the contents of the array and so obtain the values we really want -- with 0 becoming all ones.

Although this version contains five instructions instead of four, it occupies the same space (since BCTR and ALR take only 2 bytes each), and it runs faster on most machines.

```
*
*    ---------  Fast loop, version 2  ---------
*
*    On arrival here:
*
*       0 <= R1 < 256   (in readiness for "IC")
*       R2 = (patlen - stringlen - 1) mod 2**32 =
*        offset (from right end of string) of the
*        1st character to be visited in the string
*       R3 = ending addr of string to be searched
*
*    Also:
*
*       Q[0..255] is a byte array containing 1 +
*        the bold strides for the particular pat-
*        tern (see text for details)
*
FAST2    IC    R1,0(R2,R3)     Character from string
         IC    R1,Q(R1)        Tentative bold stride+1
         BCTR  R1,0            R1 := R1-1 (bold stride)
         ALR   R2,R1           Take a bold stride and
         BL    FAST2            loop unless "carry"
*
*    Fall from loop when either (a) we encounter a char-
*    acter in the string that matches the last pattern
*    character (in which case R1 is all ones), or (b)
*    the string is exhausted (and 1 <= R1 <= 254).
*
```

Now let us return to the issue I postponed.

Patterns having a length exceeding 254 can be handled simply by pretending (for the purpose of preparing the array Q, and the

"fast" loop) that they have a length of 254. The array entries for characters that do not appear in the pattern are all set to 254+1, and only the rightmost 254 characters of the pattern are visited when setting the other entries. The only disadvantage is that the stride is now limited to 254, even for a very long pattern. In practice long patterns occur rarely; and even when they do occur, the typical bold stride is limited to around q, the size of the alphabet. This is an intrinsic property of the Boyer-Moore method, and in particular of using the "value" of a character as an index. Even when the pattern length exceeds q, bold strides that exceed q can be expected only when there is a peculiar distribution of characters, neither typical nor random, such that most characters in the string do not appear among the last q characters of the pattern.

It is advisable to retain a full integer per entry in the delta2 table (if this table is used); otherwise we will undermine the guarantee on worst-case performance.

Note in passing that the FAST2 loop above may if desired be written with the last two instructions replaced by a BXLE (Branch on Index Low or Equal). But doing this obscures the points of algorithmic interest, and the resulting code does not necessarily run faster.

The latency of Versions 1 and 2.

The word-wide array used in Version 1 is the same as the original array described by Boyer and Moore, and the cost of preparing it is therefore the same (see section 2 above).

The byte-wide array used in Version 2 can be prepared more cheaply, since it is smaller. The initial filling of the table (with values of patlen+1) could for example be handled 4 entries at a time with a "store" loop. Alternatively, in S/370, the MVC instruction can be used, with overlapping operands, to propagate a value through memory. The propagation of a single byte, being a popular case, is often handled especially fast by the hardware or microcode. On some models, a 256-byte array can be thus filled in about the same time it takes for 5 turns through a simple "store" loop. (The array could also be filled at about the same speed by using a pad character with the MVCL instruction.)

In many cases this may be "good enough". Algorithmically, however, it is not very satisfying. The improvement decreases the preparation cost by only a constant factor. The cost still contains a term that is proportional to the alphabet size (as distinct from the pattern length). And the fancy S/370 propagating MVC is no help on a RISC.

Version 3. Low-latency implementation.

It is possible to use a byte-wide array Q, but with a different numeric encoding, such that the array entries for characters that do not appear in the pattern always possess the same constant values. Then the preparation for any particular search needs to modify only the entries that correspond to the characters that do appear in the pattern.

The cost of preparation now increases with pattern length (as it must), but is independent of the alphabet size. Of course, the array must be "repaired" after a search, so that it is ready to be modified anew, for the next pattern; but this also involves touching only the entries for the characters that appear in the pattern.

Happily, it turns out that an encoding exists which (a) satisfies these goals, and (b) affects neither the size nor the speed of the "fast" loop. Here is how it works.

We initialize all the entries in the array Q to 255. This is done once, when the program is loaded, or when it begins execution. Then before a search we set the entries for characters that appear within the last 254 characters of the pattern to 255-e+d, where e = min(patlen,254), and (as before) d = distance from the last pattern character of the rightmost instance of the character (d = 0, 1,...,patlen-1); except that (also as before) the entry corresponding to the last pattern character is set to 1 less, i.e. to 254-e. (Another way of looking at this is that the entries are set to the low-order 8 bits of d-e-1, or -e-2 for the very last pattern character.) In the case of a long pattern, only the rightmost 254 characters are visited when preparing the array.

We can now write the "fast" loop as shown below.

```
*
*    ---------  Fast loop, version 3  ---------
*
*    On arrival here:
*
*       R0 = 255-e, where e = min(patlen,254)
*       0 <= R1 < 256   (in readiness for "IC")
*       R2 = (patlen - stringlen - 1) mod 2**32 =
*        offset (from right end of string) of the
*        1st character to be visited in the string
*       R3 = ending addr of string to be searched
*
*    Also:
*
*       Q[0..255] is a byte array (see above for
*        details)
*
```

```
FAST3    IC    R1,0(R2,R3)    Character from string
         IC    R1,Q(R1)       Entry from the array
         SLR   R1,R0          The next bold stride
         ALR   R2,R1          Take bold stride and
         BL    FAST3           loop unless "carry"
*
*    Fall from loop when either (a) we encounter a char-
*    acter in the string that matches the last pattern
*    character (in which case R1 is all ones), or (b)
*    the string is exhausted (and 1 <= R1 <= 254).
*
```

This is my recommended implementation of the Boyer-Moore "fast" loop on a byte-addressed machine.

4. MISCELLANEOUS OBSERVATIONS

This section contains several additional observations on the implementation of Boyer-Moore search. Only the first is concerned with latency.

Patterns of length 1

It is fairly clear the Version 3 of the "fast" loop can be employed with advantage for patterns of length 2 or more. Consider a pattern of length 2. Preparation involves setting (and later repairing) 2 entries in the array Q, and setting 2 entries in delta2 if this table is used. For this modest investment, we are typically rewarded with skipping almost every other character in the string that precedes the first match.

It is interesting to ask whether Version 3 is good enough to be employed with patterns of length 1 also. This would be nice, since it would obviate the need for special cases and the implementation of more than one method. We will consider two situations, having different programming constraints.

Situation 1

> The program performing the search is permitted to apply a temporary "patch" to the string in memory

In this case an instance of the character to be sought can be temporarily placed at the end of the string. Then the search is sure to succeed, and the inner loop can be reduced to something like this:

```
LOOP     IC    R0,0(R2,R3)    Character from string
         ALR   R2,R1          Increment index (R1=1)
         CR    R0,R4          Check char (R4=pattern)
         BNE   LOOP           Loop back unless match
```

On most machines this loop would find a one-character pattern somewhat faster than Version 3 of the "fast" loop. Depending on the organization of the particular machine, especially with respect to the resolution of operand addresses and speculative execution, it would probably run between 1.2 and 1.7 times the speed of FAST3.

These estimates assume that the string already resides at the top of the storage hierarchy (e.g. in the cache), or that the cost of moving it there piecemeal is negligible. If this is not the case, the performance would be reduced for both cases, and the relative advantage would be smaller.

In any event, the advantage is not overwhelming.

Situation 2

> The program performing the search is not permitted to apply a temporary patch to the string in memory

This situation obtains if the string is shared by concurrent processes, or if it is deemed undesirable to modify memory unnecessarily because of the impact on the performance of the memory hierarchy (such as paging).

In this case the inner search loop must test for the end of the string, as well as for a match, e.g. thus:

```
LOOP    IC    R0,0(R2,R3)    Character from string
        CR    R0,R4          Check char (R4=pattern)
        BE    FOUND          Branch if perfect match
        ALR   R2,R1          Increment index (R1=1)
        BL    LOOP           Loop unless exhausted
```

This usually offers little or no advantage over Version 3 of the "fast" loop. So in this situation there is no cause to handle a pattern of length 1 as a special case.

But these detailed analyses are not always relevant. There is a different and rather general argument which suggests that the performance of the search phase for a pattern of length 1 is not very important, and therefore it usually makes sense to employ the same method as for other pattern lengths -- even if there is some loss in search performance -- provided the method does not involve expensive preparation. The search performance is unimportant in this case because the search is likely to be short. For randomly distributed characters selected from an alphabet of size 256, the first match has an expected offset of 256. For English text, the typical offset is less than 50. This argument breaks down if the alphabet is large (e.g. 64K); if we are checking for a character we do not expect to find;

or if we are searching for a special character which is inserted between large blocks of text.

Unfavourable patterns

The speed of Boyer-Moore search is impaired if the string contains many instances of the last pattern character, since the "fast" loop will constantly yield control to the "slow" one. When searching English text, for example, it usually takes longer to find a pattern that ends with a blank than to find one of the same length (or even slightly shorter) which ends with a "q".

For this reason it may be desirable to trim trailing blanks from the pattern, before preparing the array Q and delta2 (if used). The trimmed blanks are ignored in the "fast" and "slow" loops, and checked only when the rest of the pattern has been matched. This trick is most beneficial when the simplified algorithm is used. When the complete algorithm is used, trimming blanks undermines the guarantee on worst-case performance, and may yield a _reduction_ in speed, e.g. when the given pattern contains many trailing blanks and the string contains many instances of the trimmed pattern.

In a system debugging tool which is used to search raw memory, it may be desirable to trim binary zeros as well as blanks.

This technique is not of course useful for the all-blank pattern (or the all-zero pattern). The best thing to do here is to keep all the blanks (and zeros) and trim nothing.

More sophisticated techniques for selecting desirable pattern characters for early examination are described in [10].

Backward search

The implementations described in section 3 can be adapted for searching a string backwards.

When preparing the array Q, the pattern is traversed from right to left; the values stored represent the distance from the left-most character of the pattern; and the exceptional case applies to the _first_ pattern character (instead of the last). Corresponding changes are required, _mutatis mutandis_, for the delta2 table (if used).

Before the "fast" loop, it is convenient to load the base register with the _beginning_ address of the string, and initialize the index to stringlen-patlen. In the loop, the index is repeatedly _decremented_ by the bold stride until the subtraction operation yields a _borrow_ from the high-order bit position.

The obvious corresponding changes must be made in the "slow"

loop and in the resolution of the hesitant stride.

In situations where a Yes-No answer is required ("Does that string contain this pattern?"), but the position of the first match (if any) is not required, it may be desirable to choose the direction of search based on the properties of the pattern. Suppose for example that the pattern comprises the 12 characters "the   ". If we trim trailing blanks and then search forwards, we will progress with the following "bold" strides:

Character	Stride
e	0
h	1
t	2
All others	3

But if we search backwards (and keep the blanks), we can expect to make substantially faster progress:

Character	Stride
t	0
h	1
e	2
blank	3
All others	12

Choice of hesitant stride after a mismatch in the "slow" loop

When delta2 is used, it may be desirable to obtain the hesitant stride (following a mismatch in the "slow" loop) from the delta2 table alone, without attempting to improve on it by also referring to the array Q. In my programs the cost of selecting between these exceeds the cost of a complete turn through the "fast" loop, and the latter usually makes more progress. Similar advice is given in [10]. Fortunately this simplification does not undermine the linear worst-case performance guarantee.

Fuzzy case search

The Boyer-Moore method can be adapted to support a fuzzy case search. Perhaps the nicest rule for this is to allow a lower-case pattern character (e.g. "a") to match a string character of either case (i.e. "a" or "A") but require that other pattern chars match exactly. This can be handled as follows.

When preparing the array Q, lower-case characters in the pattern are deemed to stand both for themselves and for their upper-case cousins. So the character "a" sets the array entry in Q for the letter "a" and also for the letter "A" -- whereas the character "A" or "2" sets the entry for "A" or "2" only. During the search,

the "fast" loop is oblivious to the chicanery, and proceeds as before.

Some subtle adjustments are required when preparing delta2, if this is used. Consider the pattern "xABCXABC". Suppose the "slow" loop finds a match with the final ABC, which is preceded in the string by an "x" (lower-case). The substring "xABC" is capable of matching the first four characters of the pattern; therefore the value of delta2[5] must be 7, and not 11 as it would be for an "exact" match (see Appendix). The effect is not commutative: for the pattern "XABCxABC", delta2[5] may safely be set to 11.

Unfortunately these changes undermine the guarantee on worst-case performance, and there exist patterns and strings which yield search time O(patlen × stringlen) when mixed-case patterns are used. I have been unable to improve on this without giving up useful function or building additional tables.

5. CONCLUSION

There is a variant of the Boyer-Moore search algorithm which has preparation time O(patlen), instead of O(q + patlen), where q is the size of the alphabet. This reduces or eliminates the penalty of Boyer-Moore when faced with short patterns or short strings, and it has no known disadvantages. It makes the algorithm more widely applicable and almost always obviates the need for handling a short pattern as a special case.

HISTORICAL NOTES

A low-latency implementation of Boyer-Moore search was written by me in 1983 for the YMS text editor "Ed". (The exact form given here was not however devised until this paper was being prepared!) Fuzzy-case search and trimming of trailing blanks from the pattern were devised at the same time.

Reverse search has been used by me since 1986 in Examine (a read-only file editor), and Plod (a diagnostic tool for disks). Since 1991 it has also been used in the debugging program Prowl.

To the best of my knowledge, the idea of basing the direction of search on the properties of the pattern has not been exploited.

The material in the Appendix was assembled in 1996 while preparing this paper.

APPENDIX

Preparation of the Boyer-Moore "delta2" table

There has been a fair bit of muddle in the literature over the preparation of the integer array "delta2". This may be one of the reasons why experimentalists have tended to concentrate on the simplified Boyer-Moore algorithm.

The delta2 table (hereafter referred to as "d2") is used in the complete Boyer-Moore algorithm when determining the "hesitant" stride after a failed tail-match. The table entry d2[j], j = 1,2,...,patlen, represents the maximum stride that can safely be taken after encountering a mismatch in the "slow" loop at pattern position j (without regard to the identity of the mismatching character in the string).

Boyer and Moore [1] gave a procedural definition of the values required in the table. A straightforward implementation of their definition has running time O(patlen^2). For an efficient method of preparation, they referred to the contemporaneous paper by Knuth, Morris and Pratt [2] -- which in turn alluded to an earlier version of the paper by Boyer and Moore that had been distributed as a technical report.

In section 8 of [2], an algorithm is given for preparing the d2 table in time O(patlen). It involves three outer loops (one of which contains an inner loop), and uses a scratch integer array (of size patlen). Unfortunately this algorithm gives wrong results for some patterns, including those consisting of a single repeated character such as "aaaa".

The error was reported to Knuth by K. Mehlhorn, in a letter dated 20 Oct 1977 [11]. Mehlhorn devised a repair, involving an additional pair of nested loops which are appended to the original algorithm in [2]. Eventually Mehlhorn's repair appeared in print, in the 1982 paper by Smit [6].

Meanwhile, in 1980, Rytter, without apparently knowing about Mehlhorn's contribution, published a different repair which is appreciably more complicated (see [12]).

In 1991 Hume and Sunday reported that Mehlhorn's repair (which they attributed to Smit) does not give optimal shifts, and implied that Rytter's repair is to be preferred (see [10]). They did not however offer any evidence for this observation. I have been unable to reproduce it; and at the time of writing (20 Sep 1996) I have not succeeded in obtaining from them an example of a pattern that demonstrates it. I believe the two repairs always give the same (correct) results.

As you may by now have surmised, the underlying reason for all this confusion is that the efficient but imperfect algorithm given in [2] is hard to understand.

* * *

Here is a procedural definition of d2. Following Boyer and Moore, but using different terminology, we describe the values required thus:

```
            d2[j]  =  x  +  y, for 1 <= j <= patlen,

        where:  x = patlen-j = length of tail, and

                y = offset of tail from its nearest
                    quasi-tail-replica.
```

The "tail" is the piece of the pattern that lies <u>beyond</u> pattern[j].

A "quasi-tail-replica" is a substring in the pattern which (a) comprises a replica of the tail, and (b) is preceded by a character that is <u>not</u> a replica of the character preceding the tail. It may overlap the tail.

For the purpose of identifying quasi-tail-replicas, the pattern is deemed to be preceded by patlen wild cards (residing at pattern[0], pattern[-1], ...). A wild card has the property that it reliably replicates any character in the tail and it reliably <u>fails</u> to replicate the character preceding the tail. Therefore, for all tails in all patterns, a quasi-tail-replica exists whose last character lies at pattern[0]; but of course this is not always the <u>nearest</u> quasi-tail-replica.

The null tail is replicated at all points. This is relevant only for j = patlen.

This procedural definition is expressed formally in section 8 of [2] by means of the equation:

```
d2[j] =
   min{patlen-j+s|s>=1 and (s>=j or pattern[j-s]¬=pattern[j])
     and ((s>=i or pattern[i-s]=pattern[i]) for j<i<=patlen)}.
```

(In that paper d2 is designated dd′, and patlen is designated m.) A straightforward implementation of either the procedural or formal definition yields execution time O(patlen^2). All the ensuing muddle has been concerned with the desire for a more efficient realization.

* * *

Here, for ease of reference, are the three existing algorithms for preparing d2[1..patlen] in time O(patlen). I have transcribed them to a uniform Algol notation, and changed some of the variable names so that each algorithm obtains the pattern from the character array pattern[1..patlen] and places its results in the integer array d2[1..patlen].

Original (incorrect) algorithm, given on page 342 of [2], 1977

```
for k := 1 step 1 until patlen do
  d2[k] := 2 × patlen - k;

j := patlen;
t := patlen + 1;

while j > 0 do
  begin

    f[j] := t;

    while t          <= patlen      and
          pattern[j] ¬= pattern[t] do
      begin
        d2[t] := min(d2[t],patlen-j);
        t := f[t]
      end;

    t := t - 1;
    j := j - 1

  end;

for k := 1 step 1 until t do
  d2[k] := min(d2[k],patlen+t-k);
```

Mehlhorn's repair (1977), shown on page 62 of [6], 1982

This comprises a complete copy of the original algorithm followed by a new assignment statement and a new nest of loops:

```
copy of original algorithm in toto (see above);

[------- What follows is new material -------]

tp := f[t];

while t <= patlen do
  begin
    while t <= tp do
      begin
        d2[t] := min(d2[t],patlen+tp-t);
        t := t + 1
      end;
    tp := f[tp]
  end;
```

Rytter's repair, given in [12], 1980

This includes most (but not all) of the original algorithm. The last loop of the original algorithm is replaced by two new outer loops (each with an inner loop). There are several new variables, too. When necessary, the first new loop performs a second traversal of the pattern (or some of it), and builds a second temporary integer array f1[1..patlen] (or some of it). This second array can if desired use the same space as the first temporary array; therefore extra array space is not required. Nonetheless this algorithm is longer and more complicated than Mehlhorn's, and it does slightly more work:

```
for k := 1 step 1 until patlen do
  d2[k] := 2 × patlen - k;

j := patlen;  t := patlen + 1;

while j > 0 do
  begin
    f[j] := t;
    while t          <= patlen      and
         pattern[j] ¬= pattern[t] do
      begin
        d2[t] := min(d2[t],patlen-j);
        t := f[t]
      end;
    t := t - 1;  j := j - 1
  end;

[----- End of unmodified part: what follows is new -----]

q := t;  t := patlen + 1 - q;  q1 := 1;

j1 := 1;  t1 := 0;

while j1 <= t do
  begin
    f1[j1] := t1;
    while t1           >= 1            and
         pattern[j1] ¬= pattern[t1] do
      t1 := f1[t1];
    t1 := t1 + 1;
    j1 := j1 + 1
  end;

while q < patlen do
  begin
    for k := q1 step 1 until q do
      d2[k] := min(d2[k],patlen+q-k);
    q1 := q + 1;
    q := q + t - f1[t];
    t := f1[t]
  end;
```

* * *

Since I seem to be deeply involved in this, I propose to add to the confusion by giving a slightly simpler algorithm for preparing the d2 table in time O(patlen). The new algorithm has only two outer loops, each containing one inner loop. Like the other algorithms, it requires a scratch integer array (of size patlen). The first outer loop combines the first two outer loops of the original algorithm; and the second combines the last outer loop of the original with Mehlhorn's appendage. I do not have a proof of correctness -- and if I did you would be wise to treat my proof with extreme caution. I have however tested the algorithm with 50 million randomly generated patterns, having lengths in the range 1 to 16, selected from small biased alphabets. It has consistently given the same results as Mehlhorn's repair, and the same as a straightforward implementation that has running time O(patlen^2).

<u>New algorithm for d2</u>

```
t := patlen;
f[patlen] := t + 1;
d2[patlen] := patlen;

for j := patlen-1 step -1 until 1 do
  begin

    f[j] := t;
    d2[j] := 2 × patlen - j;
    pattern[patlen+1] := pattern[j];

    while pattern[j] ¬= pattern[t] do
      begin
        d2[t] := min(d2[t],patlen-j);
        t := f[t]
      end;

    t := t - 1

  end;

j := 1;

while t < patlen do
  begin
    for k := j step 1 until t do
      d2[k] := min(d2[k],patlen+t-k);
    j := t + 1;
    t := f[t]
  end;
```

Like the other algorithms, this one obtains the pattern from the character array pattern[1..patlen], and places the results in the integer array d2[1..patlen]. The last entry, at d2[patlen], is not actually required during the search, but excluding it from the pre-

paration is more trouble than it is worth. The sequence as written uses the character location at pattern[patlen+1] in a devious ploy (of no algorithmic importance) to simplify the inner "while" loop.

It can fairly easily be shown that the algorithm has running time O(patlen):

First outer loop

Clearly the outer "for" loop is executed exactly patlen-1 times.

Next I will prove that the first statement in this loop always sets f[j] to a value that exceeds j.

As long as the inner "while" loop is not executed, t remains in step with j (having a value of j+1); so f[j] is set to j+1.

If and when the inner loop is first executed, t is loaded from f[t] one or more times, and each time the value of t increases by 1 (to a maximum of patlen+1). Consequently, in subsesequent turns through the outer loop, t[j] is set to a value that exceeds j+1; and in subsequent turns through the inner loop, the value of t increases by 1 or more every time it is loaded.

The argument is iterative; and it follows that f[j] is always assigned a value that exceeds j.

Now let p = the total number of times the inner loop is executed.

On entering the outer loop, t has the value patlen; and on leaving, it has a value of at most patlen. Inside, it is incremented by at least p, and decremented by exactly patlen-1. Therefore p <= patlen-1.

Second outer loop

Each turn around the outer "while" loop, t increases by 1 or more, and the loop terminates when t reaches or exceeds patlen. Since t >= 1 on entry, the outer loop is executed at most patlen-1 times.

Finally, each time around the outer loop, the increment in j is equal to the number of turns through the inner "for" loop. Since on entry j = 1, and finally j <= patlen, the inner loop is also executed at most patlen-1 times.

This proof has been confirmed informally by counting iterations while processing 1 million randomly generated patterns.

* * *

To end, here are some examples of d2. Example 1 is given as a starting reference point: this pattern contains no repeated characters, and is therefore uninteresting. Example 2 contains plenty of repetition, but it has no effect on d2 since it does not involve the tail of the pattern. The other examples show the effects of repetition that does involve the tail. The last example is taken from [2] and (to quote from that paper) "illustrates most of the subtleties of the algorithm".

The examples can be checked by hand using any of these methods:

(a) by reference to the functional definition of d2 (see [1], or the description above),

(b) by evaluating the formal equation for d2 (see [2], also shown above),

(c) by stepping through the algorithm as repaired by Mehlhorn (see [6], or "Mehlhorn's repair" above),

(d) by stepping through the algorithm as repaired by Rytter (see [12], or "Rytter's repair" above),

(e) by stepping through the new algorithm (see "New algorithm for d2" above), and

(f) by devising yet another algorithm and stepping through that.

I recommend the first exercise, and the last two.

Example 1.

index	1	2	3	4	5	6	7
pattern	a	b	c	d	e	f	g
d2	13	12	11	10	9	8	1

Example 2.

index	1	2	3	4	5	6	7
pattern	a	a	a	a	a	a	b
d2	13	12	11	10	9	8	1

Example 3.

index	1	2	3	4	5	6	7
pattern	a	b	c	d	e	e	e
d2	13	12	11	10	3	3	3

Example 4. (*)

index	1	2	3	4	5	6	7
pattern	a	a	a	a	a	a	a
d2	7	7	7	7	7	7	7

(*) This is an example of a pattern for which the original algorithm in [2] gives wrong results.

Example 5.

index	1	2	3	4	5	6	7	
pattern	a	b	a	b	a	b	a	(*)
d2	8	7	8	7	8	7	1	

Example 6.

index	1	2	3	4	5	6	7	8	9
pattern	c	d	b	c	d	a	b	c	d
d2	15	14	13	12	11	7	9	10	1

Example 7.

index	1	2	3	4	5	6	7	8	9	
pattern	a	b	c	a	b	c	a	b	c	(*)
d2	11	10	9	11	10	9	11	10	1	

Example 8.

index	1	2	3	4	5	6	7	8	9	
pattern	a	b	a	d	e	f	a	b	a	(*)
d2	14	13	12	11	10	9	10	3	1	

Example 9.

index	1	2	3	4	5	6	7	8	9	10	11
pattern	b	a	d	b	a	c	b	a	c	b	a
d2	19	18	17	16	15	8	13	12	8	12	1

REFERENCES

[1] Boyer, R.S. and Moore, J.S., A fast string searching algorithm, CACM 20, 10, p. 762 (1977)

[2] Knuth, D.E., Morris, J.H. and Pratt, V.R., Fast pattern matching in strings, SIAM J. Comput. 6, 2, p. 323 (1977)

[3] Guibas, L.J. and Odlyzko, A.M., A new proof of the linearity of the Boyer-Moore string searching algorithm, Proc. 18th IEEE Symp. Foundations of Computer Science, p. 189 (1977)

[4] Cole, R., Tight bounds on the complexity of the Boyer-Moore string matching algorithm, SIAM J. Comput. 23, 5 p. 1075 (1994)

[5] Horspool, R.N., Practical fast searching in strings, Software 10, 6, p. 501 (1980)

[6] Smit, G.V., A comparison of three string matching algorithms, Software 12, 1, p. 57 (1982)

[7] Davies, D.J.M., String searching in text editors, Software 12, 8, p. 709 (1982)

(*) These are examples of patterns for which the original algorithm in [2] gives wrong results.

[8] Davies, G. and Bowsher, S., Algorithms for pattern matching, Software 16, 6, p. 575 (1986)

[9] IBM System/370 Principles of Operation, GA22-7000, IBM (1972..1987)

[10] Hume, A. and Sunday, D., Fast string searching, Software 21, 11, p. 1221 (1991)

[11] Knuth, D., Private communication (1996)

[12] Rytter, W., A correct preprocessing algorithm for Boyer-Moore string-searching, SIAM J. Comput. 9, 3 p. 509 (1980)

CJS, 1996-09-20.