2.2.2. Alocação Seqüencial

O jeito mais simples e mais natural de manter uma lista linear dentro de um computador é colocar os itens da lista em locais seqüenciais, um elemento após o outro. Então nós teremos:

$$\text{LOC}(X[j + 1]) = \text{LOC}(X[j]) + c,$$

onde c é o número de palavras por elemento. (Geralmente $c = 1$. Quanto $c > 1$, é às vezes mais conveniente dividir uma lista única em c listas "paralelas, então a k-ésima palavra do elemento $X[j]$ é armazenado a uma distância fixa do local da primeira palavra de $X[j]$. Nós vamos continuamente supor, no entanto, que grupos adjacentes de c palavras formam um único elemento.) Em geral,

$$\text{LOC}(X[j]) = L0 + cj, \quad (1)$$

onde L_0 é uma constante chamada de *endereço da base*, o local de um elemento artificialmente suposto X[0].

Esta técnica para representar uma lista linear é tão óbvia e bem conhecida que parece não haver necessidade de utilizá-la em nenhum comprimento. Mas nós veremos muitos outros métodos "mais sofisticados" de representação depois nesse capítulo, e é uma boa idéia examinar o caso simples primeiro para ver até aonde podemos ir com isso. É importante entender as limitações tanto quanto o poder do uso da alocação seqüencial.

Alocação seqüencial é bastante conveniente ao lidarmos com uma pilha. Nós simplesmente temos uma variável T chamada de *indicador da pilha*. Quando a pilha está vazia, fazemos T = 0. Para colocarmos um novo elemento Y no topo da pilha, fazemos

$$T \leftarrow T + 1; X[T] \leftarrow Y. \quad (2)$$

E quando a pilha não está vazia, podemos igualar Y ao elemento do topo e remover aquele elemento revertendo as ações de (2):

$$Y \leftarrow X[T]; T \leftarrow T - 1. \quad (3)$$

(Em computadores é geralmente mais eficiente manter o valor c_T ao invés de T, por causa do (1). Como é fácil fazer as modificações, iremos continuar nossa discussão como se c = 1.)

A representação de uma fila ou uma fila dupla mais geral é um pouco mais engenhosa. Uma solução óbvia é manter dois ponteiros, suponha F e R (para o começo (front) e o fim(rear) da fila), com F = R = 0 quando a fila está vazia. Então inserir um elemento no final da fila poderia ser

$$R \leftarrow R + 1; X[R] \leftarrow Y; \quad (4)$$

remover o elemento do começo (F aponta uma casa antes do começo) poderia ser

$$F \leftarrow F + 1; Y \leftarrow X[F]; \text{ se } F = R, \text{ então } F \leftarrow R \leftarrow 0. \quad (5)$$

Mas note o que pode acontecer: Se R ficar sempre na frente de F (então sempre existe pelo menos um elemento na fila), as entradas usadas serão X[1], X[2],..., X[1000],..., ao infinito, e isso é horrivelmente esbanjador de espaço de armazenamento. O método simples (4), (5) deve ser usado somente na situação em que sabemos que F alcança R muito regularmente (por exemplo, se todas as remoções vêm de uma vez, o que esvazia a fila).

Para evitar o problema da fila exceder a memória, nós podemos tomar M elementos X[1],...,X[M] arrumados implicitamente em um círculo, com X[1] em seguida do X[M]. Então os processos (4) e (5) acima se tornam (para R = F = M inicialmente)

$$\text{Se } R = M \text{ então } R \leftarrow 1, \text{ cc } R \leftarrow R + 1; X[R] \leftarrow Y. \quad (6)$$
$$\text{Se } F = M \text{ então } F \leftarrow 1, \text{ cc } F \leftarrow F + 1; Y \leftarrow X[F]. \quad (7)$$

A discussão acima foi muito irrealista, pois supomos sutilmente que nada poderia dar errado. Quando removemos um elemento de uma pilha ou fila, supomos que havia ao menos um elemento presente. Quando inserimos um elemento em uma pilha ou fila, supomos que existe espaço para ele na memória. Mas claramente o método (6) (7) permite no máximo M elementos na fila completa, e os métodos (2) (3) (4) (5) permitem que T e R alcancem apenas uma certa quantidade máxima dentro de qualquer programa de computador dado. As seguintes especificações mostram como as ações acima devem ser reescritas para um caso geral, quando não supomos que essas restrições estão automaticamente satisfeitas:

$X \Leftarrow Y$ (inserir na pilha): $T \leftarrow T + 1$; se $T > M$, então OVERFLOW; $X[T] \leftarrow Y$. (2a)

$Y \Leftarrow X$ (remover da pilha): se $T = 0$, UNDERFLOW; $Y \leftarrow X[T]$; $T \leftarrow T - 1$. (3a)

$X \Leftarrow Y$ (inserir na fila): se $R = M$, então $R \leftarrow 1$, cc $R \leftarrow R + 1$; se $R = F$, então OVERFLOW; $X[R] \leftarrow Y$. (6a)

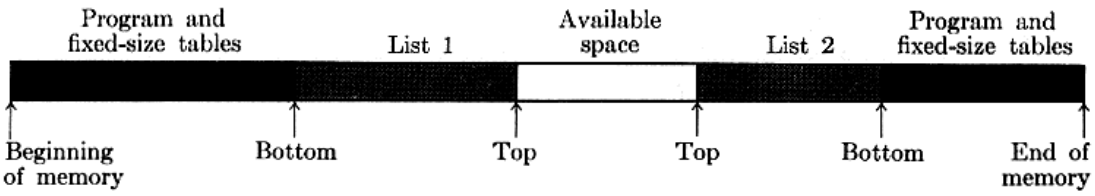
$Y \Leftarrow X$ (remover da fila): se $R = F$, então UNDERFLOW; Se $F = M$, então $F \leftarrow 1$, cc $F \leftarrow F + 1$; $Y \leftarrow X[F]$. (7a)

Aqui supomos que X[1],...,X[M] é todo o espaço disponível para a lista; OVERFLOW e UNDERFLOW significam um excesso ou deficiência de itens.

A próxima questão é, "O que faremos quando acontecer um OVERFLOW ou um UNDERFLOW?". No caso de UNDERFLOW, tentamos remover um item inexistente; isso geralmente significa uma condição – não uma situação de erro – que pode ser usada para conduzir o andamento do programa, i.e, podemos remover itens repetidamente até que aconteça um UNDERFLOW. Uma situação de OVERFLOW, porém, é geralmente um erro; significa que a tabela já está cheia, e há ainda mais informação que deve ser inserida. A política comum no caso de OVERFLOW é informar relutantemente que o programa não pode continuar porque sua capacidade de armazenamento foi excedida, e terminar o programa.

É claro que não gostamos de desistir em uma situação de OVERFLOW quando apenas uma lista ficou muito grande, enquanto outras listas do mesmo programa podem muito bem ter uma grande quantidade de espaço disponível. Na discussão acima estávamos inicialmente pensando em um programa com apenas uma lista. Porém, freqüentemente encontramos programas que envolvem muitas pilhas, cada uma com seu tamanho variando dinamicamente. Em cada situação devemos evitar impor um tamanho máximo em cada pilha, por que geralmente o tamanho é algo imprevisível; e sempre se um tamanho máximo é determinado para cada pilha, raramente encontramos todas as pilhas cheias simultaneamente.

Quando há apenas duas listas de tamanho variável, elas podem muito bem se manter juntas se fizermos com que elas cresçam uma em frente da outra:

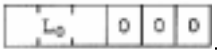


Aqui a lista 1 (list 1) cresce para a direita, e a lista 2 (list 2) (armazenada na ordem inversa) cresce para a esquerda. O OVERFLOW não irá ocorrer a menos que o tamanho total de ambas as listas preencha todo o espaço da memória. As listas podem independentemente aumentar ou diminuir de forma que o tamanho máximo efetivo de cada uma pode ser significativamente mais que a metade do espaço disponível. O esboço acima do espaço na memória é usado muito freqüentemente.

O leitor pode facilmente se convencer, porém, que não há jeito de armazenar três ou mais tabelas seqüenciais de tamanho variável na memória de forma que (a) OVERFLOW ocorra somente quando o tamanho total de todas as listas exceda o espaço total, e (b) cada lista tenha uma localização fixa para seu elemento

"inferior". Quando há, digamos, dez ou mais tabelas de tamanho variável – e isso não é raro – o problema de alocação de espaço se torna muito significante. Se desejarmos satisfazer a condição (a), devemos desistir da condição (b); isto é, devemos permitir que os elementos "inferiores" das listas mudem suas posições. Isso significa que a localização L_0 da Eq. 1 não é mais constante; nenhuma referência à tabela deve ser feita a um endereço absoluto de memória, todas as referências devem ser relativas ao endereço da base L_0. No caso do MIX, o código para trazer o elemento de 1 palavra no registro A é mudado:

```
de      LD1  I              para, por exemplo,   LD1  I
        LDA  L0, 1                               LDA  BASE(0:2)        (8)
                                                 STA  *+1(0:2)
                                                 LDA  *,1
```

onde BASE contém 

.
Esse endereçamento relativo é evidentemente mais lento do que quando a base estava fixa, embora vemos que seria apenas levemente mais lento se o MIX tivesse um recurso de "endereçamento indireto" (veja exercício 3).

Um caso especial importante ocorre quando cada uma das listas de tamanho variável é uma pilha. Então, desde que apenas o elemento do topo de cada pilha é relevante em qualquer tempo, podemos proceder tão eficientemente quanto antes. Suponha que temos n pilhas; os algoritmos de inserção e remoção acima se tornara os seguintes, se BASE[i] e TOP[i] são as variáveis de ligação para a i-esima pilha:

Inserção: TOP[i] $\leftarrow$ TOP[i] + *c*; se TOP[i] > BASE[i+1], então OVERFLOW;
cc ELEM(TOP[i]) $\leftarrow$ Y. (9)

Remoção: se TOP[i] = BASE[i], então UNDERFLOW; cc Y $\leftarrow$ ELEM[TOP[i]),
TOP[i] $\leftarrow$ TOP[i] – *c*. (10)

Aqui BASE[i + 1] é a localização do começo da (i + 1)-ésima pilha. A condição TOP[i] = BASE[i] significa que a pilha *i* está vazia.

Na situação acima, OVERFLOW não é mais tão crítico quanto antes; podemos realocar memória, liberando espaço para a tabela que estourou (OVERFLOW) pegando algum espaço vago das tabelas que ainda não foram preenchidas. Devido ao número de jeitos possíveis fazer isso, e já que esses algoritmos de realocação são muito importantes na conexão com a alocação seqüencial de tabelas lineares, não iremos explorar esse problema em detalhe. Iremos começar dando o mais simples desses métodos, e então iremos considerar alguns dos métodos alternativos.

Suponha que existem n pilhas, e os valores BASE[i] e TOP[i] estão sendo manipulados como em (9), (10). Essas pilhas estão todas compartilhando a área comum de memória consistindo em todas as localizações L com $L_0 \le L < L_\infty$ (Aqui L_0 e L_∞ são constantes que especificam o número total de sentenças disponíveis para uso; $L_\infty - L_0$ é um múltiplo de c). Devemos iniciar com todas as pilhas vazias, e BASE[i] = TOP[i] = L_0 – c, para todo i. Usamos também BASE[n + 1] $\equiv L_\infty$ – c de forma que (9) vai funcionar propriamente para i = n. Agora a qualquer momento em que uma pilha particular, exceto a pilha n, adquire mais elementos do que tinha antes, o OVERFLOW vai ocorrer.

Quando a pilha i estoura (overflow), há três possibilidades:

a) Encontramos o menor k para o qual $i < k \le n$ e TOP[k] < BASE[k+1], se existir algum k. Agora movemos tudo uma casa acima:

Faça CONTEÚDO(L + c) $\leftarrow$ CONTEÚDO(L), para TOP[k] + c > L $\ge$ BASE[i + 1] + c.

(Note que isso deve ser feito para diminuir, e não aumentar, os valores de L para evitar a perda de informação. É possível que TOP[k] = BASE [i + 1], no caso em que nada precisa ser movido).

Faça BASE[j] $\leftarrow$ BASE[j] + c, TOP[j] $\leftarrow$ TOP[j] + c, para $i < j \le k$.

b) Nenhum k pode ser encontrado como em (a), mas pode ser encontrado o maior k para o qual $1 \le k < i$ e TOP[k] < BASE[k + 1]. Agora mova tudo para baixo uma casa:

Faça CONTEÚDO(L) ← CONTEÚDO(L + c), para BASE[k + 1] ≤ L < TOP[i].

(Note que isso deve ser feito para incrementar os valores de L.)

Faça BASE[j] ← BASE[j] – c, TOP[j] ← TOP[j] – c, Para k < j ≤ i.

c) Temos TOP[k] = BASE[k + 1] para todo k ≠ i. Então obviamente não conseguimos encontrar espaço para uma nova entrada na pilha, e devemos desistir.

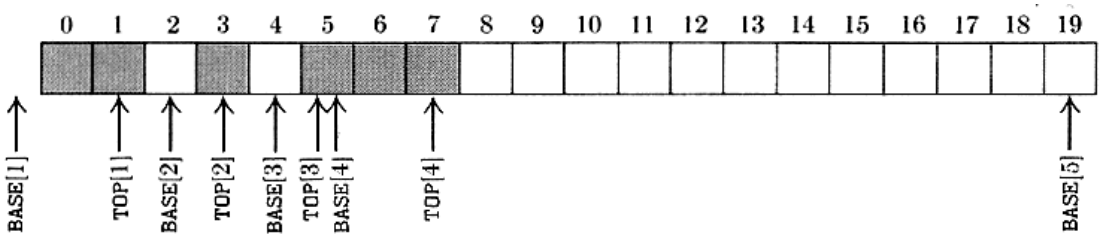


Fig 4. Exemplo de configuração de memória após algumas inserções e remoções.

A figura 4 ilustra a configuração da memória para o caso n = 4, c = 1, $L_0 = 0$, $L_\infty = 20$, após a ação sucessiva

$$I_1^* \; I_1^* \; I_4 \; I_2^* \; D_1 \; I_3^* \; I_1 \; I_1^* \; I_2^* \; I_4 \; D_2 \; D_1.$$

(Aqui I_j e D_j se referem à Inserção e à remoção na pilha j, e um asterisco se refere à uma ocorrência de OVERFLOW, supondo que nenhum espaço está inicialmente alocado para as pilhas 1, 2 e 3).

É claro que muitos dos primeiros estouros de pilha que ocorrem com esse método podem ser eliminados se escolhermos nossas condições iniciais sabiamente, ao invés de alocarmos todo o espaço inicial para a n-esima pilha como sugerido acima. Por exemplo, se esperarmos que cada pilha tenha o mesmo tamanho, poderemos começar com

$$\texttt{BASE}[j] = \texttt{TOP}[j] = c\left\lfloor \left(\frac{j-1}{n}\right)\left(\frac{L_\infty - L_0}{c}\right)\right\rfloor + L_0 - c. \qquad (11)$$

Experiências de operação com um programa particular podem sugerir os melhores valores iniciais; porém, não importa quão bem a alocação inicial é montada, ela pode prevenir no máximo um número fixo de overflows, e o efeito é notável somente nos primeiros estágios da execução do programa.

Outra possível melhora no método acima seria liberar espaço para mais de uma nova entrada cada vez que a memória é realocada. O deslocamento das tabelas na memória é uma operação que consume tempo, e conseguimos ganhar velocidade deslocando até 2 ou 3 de uma vez só ao invés de deslocar em 1 algumas vezes.

Esta idéia foi explorada por J. Garwick, que sugere uma completa realocação de memória quando o overflow ocorre, baseado na mudança do tamanho de cada pilha desde a última realocação. Este algoritmo usa um vetor adicional, chamado OLDTOP[i], 1 ≤ i ≤ n, que guarda o valor que TOP[i] tinha logo antes do último momento em que a memória foi alocada. Inicialmente, as tabelas são feitas como antes, com OLDTOP[i] = TOP[i]. O novo algoritmo procede como se segue:

Algoritmo G (*Realocação de tabelas seqüenciais)*. Suponha que um OVERFLOW ocorreu na pilha i, de acordo com (9). Após o Algoritmo G ser executado, ou encontraremos a capacidade de memória excedida ou a memória foi rearrumada de forma que a ação ELEM(TOP[i]) ← Y possa ser feita. (Note que TOP[i] já foi incrementado em (9) antes do Algoritmo G acontecer).

G1. [Inicialização] Faça SUM ← L_∞ - L_0, INC ← 0. Então faça o passo G2 para 1 ≤ j ≤ n. (O efeito será fazer com que SUM seja igual ao total de espaço na memória disponível, e INC igual ao total de incrementos no tamanho da tabela desde a última alocação). Depois que isso for feito, vá para o passo G3.

G2. [Coletando estatísticas.] Faça SUM ← SUM – (TOP[j] – BASE[j]). Se TOP[j] > OLDTOP[j], faça D[j] ← TOP[j] – OLDTOP[j] e INC ← INC + D[j]; cc faça D[j] ← 0.

G3. [A memora está cheia?] Se SUM < 0, não podemos continuar.

G4. [Computar causas da alocação] Faça $\alpha = (0.1/n)(SUM/c)$, $\beta = (0.9/INC)(SUM/c)$. (Aqui α e β são frações, não inteiros, que estão sendo computadas para precisão razoável. No próximo passo, cada lista terá 10% mais espaço disponível que agora tem; os outros 90% desse espaço serão divididos proporcionalmente à quantidade do aumento no tamanho da tabela desde a alocação anterior).

G5. [Computar novos endereços de base]. Faça NEWBASE[1] ← BASE[1]; então para j = 2, 3,..., n faça

$$\texttt{NEWBASE}[j] \leftarrow \texttt{NEWBASE}[j-1] + \texttt{TOP}[j-1] - \texttt{BASE}[j-1] + c\lfloor\alpha\rfloor + c\lfloor \texttt{D}[j-1]\beta\rfloor.$$

G6. [Realocar] Executar o Algoritmo R abaixo, e então fazer OLDTOP[j] ← TOP[j] para 1 ≤ j ≤ n. ♦

É visto com prazer já que $\lfloor\alpha\rfloor \le \alpha$ e $\lfloor \texttt{D}[j-1]\beta\rfloor \le \texttt{D}[j-1]\beta$, as formas de realocação do passo G5 nunca darão resultados anormais. Talvez a parte mais interessante de todo esse algoritmo é o processo de realocação geral, que agora descrevemos. Realocação não é trivial, já que algumas porções de memória se deslocaram pra frente, e outras pra trás; é obviamente importante não sobrescrever nenhuma das informações boas da memória enquanto tudo está sendo movido.

Algoritmo R (Realocar tabelas seqüenciais). Para 1 ≤ j ≤ n a informação especificada por BASE[j] e TOP[j] de acordo com as convenções estabelecidas acima é movida para novas posições especificadas por NEWBASE[j], e os falores de BASE[j] e TOP[j] são adequadamente ajustados.

R1. [Inicialização] Faça j ← 1. (Note que a pilha 1 nunca precisa ser movida, então para eficiência o programador deve colocar a maior pilha primeiro, se ele souber qual delas será a maior).

R2. [Achar o começo do deslocamento] Incrementar j de 1 em 1 até que aconteça
a) NEWBASE[j] < BASE[j]: vá para R3; ou
b) NEWBASE[j] > BASE[j]: vá para R4; ou
c) j > n; o algoritmo termina.

R3. [Mover pra trás] Faça δ ← BASE[j] – NEWBASE[j]. Faça CONTEÚDO(L - δ) ← CONTEÚDO[L], para L = BASE[j] + c, BASE[j] + c + 1,..., TOP[j] + c – 1. (Note que é possível que BASE[j] seja igual a TOP[j], nesse caso nenhuma ação é necessária). Faça, BASE[j] ← NEWBASE[j], TOP[j] ← TOP[j] - δ. Vá para R2.

R4. [Achar o topo do deslocamento] Ache o menor k ≥ j tal que NEWBASE[k + 1] ≤ BASE[k + 1]. (Nota: NEWBASE[n + 1] deveria igualar BASE[n + 1], de forma que um k sempre irá existir.) Então faça o passo R5 para t = k, k – 1,..., j; finalmente faça j ← k e vá para R2.

R5. [Mover pra frente] Faça δ ← NEWBASE[t] – BASE[t]. Set CONTEÚDO(L + δ) ← CONTEÚDO(L), para L = TOP[t] + c – 1, TOP[t] + c -2,..., BASE[t] + c. (Note que como no passo R3, nenhuma ação é necessária aqui.) Faça BASE[t] ← NEWBASE[t], TOP[t] ← TOP[t] + δ. ♦

Nos Algoritmos G e R fizemos intencionalmente com que seja possível termos

$$\texttt{OLDTOP}[j] \equiv \texttt{D}[j-1] \equiv \texttt{NEWBASE}[j]$$

para 1 ≤ j ≤ n + 1, isto é, estas três tabelas podem compartilhar os locais comuns da memória já que seus valores nunca são necessários ao mesmo tempo. Será necessário executar o passo G2 para decrementar os valores de j e fazer NEWBASW[n + 1] ← BASE[n + 1] no passo G5 quando usarmos sua sobreposição.

Descrevemos esses algoritmos de realocação para pilhas, mas é claro que eles podem ser adaptados para quaisquer tabelas de endereçamento relativo nas quais as informações atuais estão contidas entre BASE[j] e TOP[j]. Outros ponteiros (por exemplo, INÍCIO[j], FIM[J]) podem também ser adicionados nas listas, fazendo

com que o algoritmo funcione também para uma fila ou fila dupla. Veja o exercício 8 que considera o caso de uma fila em detalhe.

A análise matemática dos algoritmos de alocação-armazenamento dinâmico é extremamente difícil. Talvez alguns dos resultados matemáticos da "teoria das filas" podem ser aplicados a este estudo, embora a orientação dessa teoria pareça ser diferente.

Como um exemplo da teoria que pode ser derivado, suponha que consideremos o caso quando as tabelas crescem apenas por inserção (remoções e inserções subseqüentes que cancelam seus efeitos são ignorados), e suponhamos além disto que é esperado que cada tabela se preencha na mesma velocidade. Esta situação pode ser modelada imaginando-se uma seqüência de m operações de inserção $a_1, a_2, \ldots, a_m$, onde cada a_i é um inteiro entre 1 e n (representando uma inserção no topo da pilha a_i). Por exemplo, a seqüência 1, 1, 2, 2, 1 significa 2 inserções na pilha 1, seguida de 2 na pilha 2, seguida de outra na pilha 1. Podemos observar que cada uma das n^m possíveis especificações $a_1, a_2, \ldots, a_m$ como igualmente prováveis, e então podemos calcular a média do número de vezes que é necessário mover uma palavra de um local para outro durante as operações de realocação a medida que a tabela completa é construída. Para o primeiro algoritmo, começando com todo o espaço disponível dado na n-esima pilha, vemos que a média de movimentos pedida é

$$\frac{1}{2}\left(1-\frac{1}{n}\right)\binom{m}{2}. \qquad (12)$$

Desse modo, como devíamos esperar, o número de movimentos é proporcional ao quadrado do número de itens das tabelas. Se não começamos com todo o espaço dado na n-ésima pilha, mas de preferência expandira cada uma das primeiras n – 1 pilhas t células antes que o overflow ocorra, temos

$$\frac{1}{2n^t}\sum_{0\le k\le m-t-2}\binom{t-1+k}{k}\binom{m-t-k}{2}\left(1-\frac{1}{n}\right)^{k+1}, \quad \text{for} \quad t\ge 0. \qquad (13)$$

que é melhor que o (12), mas ainda é proporcional a m^2. Se colocarmos $n = \infty$ no sum, (13) se reduz a

$$\frac{1}{2n^t}\binom{m}{t+2},$$

que é um limitante superior não muito informativo; um limitante inferior é dado pelo termo k = 0, nominalmente

$$\frac{1}{2n^t}\left(1-\frac{1}{n}\right)\binom{m-t}{2}.$$

A moral da história parece ser que um número muito grande de movimentos vai ser feito se um número razoavelmente grande de itens for posto nas tabelas. Este é o preço que devemos pagar pela habilidade de empacotar um número grande de tabelas seqüenciais juntas. Nenhuma teoria foi desenvolvida para analisar as características do Algoritmo G, e não é razoável que qualquer modelo simples é apto para descrever as características das tabelas da vida real em cada um ambiente de qualquer modo.

Experiências mostram que quando a memória é apenas carregada pela metade (i.e, o espaço disponível é igual à metade do espaço total), precisamos rearranjar um pouco menos as tabelas com o Algoritmo G; o importante é talvez que o algoritmo se comporta bem no caso meio-cheio e que isso ao menos nos dá as respostas corretas no caso quase cheio.

Mas vamos pensar sobre o caso quase cheio mais cuidadosamente: Quando as tabelas quase preenchem a memória, o Algoritmo R demora geralmente bastante tempo para executar seu trabalho, e para fazer as coisas piorarem, o overflow é muito mais freqüente, já que o espaço na memória está quase todo usado. Há pouquíssimos programas que vao chegar perto de encher a memória sem estourar rapidamente antes; e estes que estouram a memória vão provavelmente gastar enormes quantidades de tempo nos Algoritmos G e R, logo antes da memória exceder. Infelizmente, um programa não depurado vai freqüentemente estourar a capacidade de memória. Para evitar gastar todo este tempo, uma sugestão possível seria parar o Algoritmo G no passo G3 se SUM é menor que S_{min}, com S_{min} escolhido pelo programador para prevenir realocações excessivas. Quando há muitas tabelas seqüenciais de tamanho variável, não devemos esperar fazer uso de 100% do espaço da memória antes do armazenamento ser excedido.