Computers Math. Applic. Vol. 25, No. 3, pp. 43–57, 1993
Printed in Great Britain.

0097-4943/93 $5.00 + 0.00

ANALYSIS OF THE MULTIGROUP METHOD FOR MANIPULATING MULTIPLE STACKS*

BEEN-CHIAN CHIEN† AND RONG-JAYE CHEN
Institute of Computer Science and Information Engineering
National Chiao Tung University, Hsinchu, Taiwan 300, R.O.C.
WEI-PANG YANG
Department of Computer and Information Science
National Chiao Tung University, Hsinchu, Taiwan 300, R.O.C.

(Received February 1992)

Abstract—The multiple stacks problem is that a number of stacks have to be manipulated in a finite continuous memory, simultaneously. Many applications need the support of multiple stacks, e.g., the parallel computing with shared memory. There are two approaches mentioned in [1] for manipulating multiple stacks. In this paper, we present and analyze the Multigroup method which combines the concept of coexisting. By the concept of coexisting, storage sharing in a linear data structure is possible. We show that the new method has better performance than the two previous approaches, and improves the extra manipulating time.

1. INTRODUCTION

Stack is a kind of linear list. It is a simple and useful data structure. A stack usually has two operations, *Push* and *Pop*. The simplest and most natural way of keeping a stack inside a computer is to put items in sequential locations. It is quite convenient to deal with one stack by sequential locating. However, software developers frequently encounter programs which involve multiple stacks, each of which has dynamically varying size. In such a situation, keeping multiple stacks in a common area which is sequential in allocation will cause some troubles. Because the sequential locations are shared by many stacks, each stack must keep two pointers, the top and bottom addresses of the stack, just belonging to itself. However, developers would hate to impose a maximum size on each stack, since the size is usually unpredictable. Another problem is *overflow*. An overflow situation will occur when the stack is already full, yet there are still more items that ought to be put in. A solution for overflow is *memory reallocation*, making room for the overflowed stacks by taking some spaces from stacks that are not yet filled. This operation is called *repack*; it may move a few items to their proper locations in order to keep correctness of *push* operations coming later.

Knuth [1] proposed an intuitive algorithm to reallocate memory by *move* operation when overflow occured. Suppose that there are n stacks, and that m items are pushed into the n stacks randomly. As is shown in the analysis of Knuth [1], the reallocation needs

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

move operations in the average case. The detailed algorithm and analysis can be found in [1]; we do not describe it here.

An improved algorithm in [1], so-called **Algorithm G**, was proposed by J. Garwick, who suggested a *complete* repacking of memory when overflow occurs. The detail of Algorithm G is

*This research was partly supported by the National Science Council of Taiwan, R.O.C. under Contract: NSC81-0408-E009-19(1991).

†The author to whom all correspondence should be sent.

also shown in [1]. Algorithm G is more complicated than the previous Knuth's method but the performance of Algorithm G is better than Knuth's method. The analysis of Algorithm G can be found in [2].

In this paper, an idea called *coexisting* is applied to the manipulation of multiple stacks. By the concept of coexisting, an improved algorithm [3], called *Multigroup method*, is presented. We will also analyze the performance of the Multigroup method here. The results of analysis can estimate the algorithm's performance and explain its properties correctly. We will also show that the performance of our method is better than the previous two methods, both theoretically and practically. In Section 2, the concept of coexisting and the Multigroup method are stated. The models of analysis are described in Section 3. Section 4 outlines the theoretical values and experimental results and gives a comparison. In addition, we also compare the Multigroup method with the previous two methods in [1]. In conclusion, we suggest an approach which makes the Multigroup method work flexibly.

2. THE MULTIGROUP METHOD

A stack is a last-in-first-out (LIFO) data structure which always grows towards one direction. If the stack is given a fixed size, the number of items in the stack will not exceed the capacity of the stack; otherwise, an overflow will occur. The concept of coexisting is to set two stacks so that they grow towards each other. The capacities of the two stacks can be shared. If the number of items in one of the two stacks exceeds the capacity of a single stack, the capacity of the other stack can be used to store the exceeding items of the neighbor stack. This idea is shown in Figure 1. The Multigroup method applies the idea to the multiple stacks by dividing the total stacks into several groups and growing the even numbered stack and odd numbered stack in opposite directions.

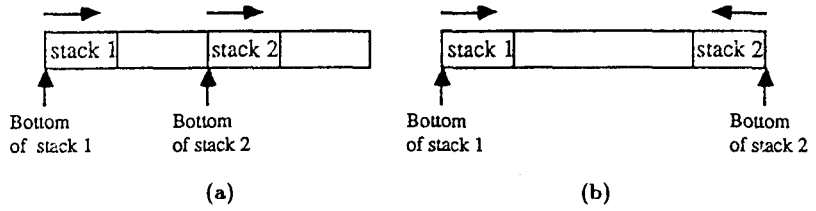

Figure 1. (a) Growing stacks in the same direction. (b) Growing stacks towards each other.

The detail of the Multigroup method is described as follows. Assume that there are n stacks and these stacks all share the common memory area consisting of all locations p with $P_0 < p \leq P_\infty$, where P_0 and P_∞ represent the lowest and highest locations of the common memory area. For each stack i, we shall use $BASE[i]$ and $TOP[i]$ to represent the position one less than the bottom and the position of the top of stack i, respectively. $CONTENTS[p]$ stands for the content of memory address p, $P_0 < p \leq P_\infty$. Let n' be defined as follows:

$$\begin{cases} n' = \dfrac{n}{2} & \text{if } n \text{ is even,} \\ n' = \dfrac{(n+1)}{2} & \text{if } n \text{ is odd.} \end{cases}$$

Initially, we divide the n stacks into n' groups. If n is odd, a pseudo stack $n+1$ will be added for processing easily. Each group consists of one odd numbered stack $2i-1$ and one even numbered stack $2i$, $1 \leq i \leq n'$. The two stacks in the same group grow towards each other. We start out by giving each group equal space, $\lfloor (P_\infty - P_0)/n' \rfloor$. The bottom and top positions of stacks begin with

$$\left.\begin{aligned} BASE[2i-1] = TOP[2i-1] &= \left\lfloor \left(\tfrac{i-1}{n'}\right)(P_\infty - P_0) \right\rfloor + P_0, \\ BASE[2i] = TOP[2i] &= \left\lfloor \left(\tfrac{i}{n'}\right)(P_\infty - P_0) \right\rfloor + P_0 + 1, \end{aligned}\right\} \quad 1 \leq i \leq n'.$$

If $P_\infty - P_0$ is divisible by n', all groups have the same available space. For example, if $P_\infty - P_0 = 20$ and $n = 4$, the initial step is shown as Figure 2, and the detailed algorithm is listed as follows:

```
Initial(n)
begin
   n' ← (⌈n/2⌉ * 2)/2;
   for i := 1 to n' do
      begin
         BASE[2i − 1] ← ⌊((i−1)/n') (P∞ − P0)⌋ + P0;
         TOP[2i − 1] ← ⌊((i−1)/n') (P∞ − P0)⌋ + P0;
         OLDTOP[2i − 1] ← ⌊((i−1)/n') (P∞ − P0)⌋ + P0;
         BASE[2i] ← ⌊(i/n') (P∞ − P0)⌋ + P0 + 1;
         TOP[2i] ← ⌊(i/n') (P∞ − P0)⌋ + P0 + 1;
         OLDTOP[2i] ← ⌊(i/n') (P∞ − P0)⌋ + P0 + 1;
      end;
end;
```

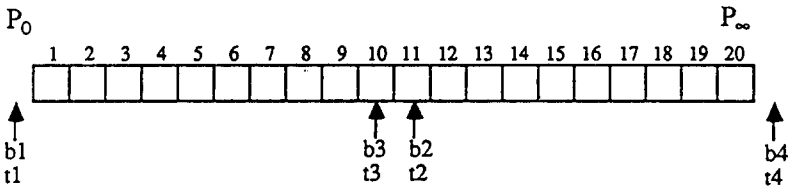

Figure 2. The initial step of the Multigroup method.

After the initial step, the *Push* and *Pop* operations are similar to the situation of a single stack, except that the odd numbered stacks always grow towards higher locations and the even numbered stacks grow towards lower locations. We show them in the following:

```
Push(i, Y)    (* Push the item Y into stack i *)
begin
   if i is odd
     then begin
             TOP[i] ← TOP[i] + 1;
             if TOP[i] ≥ TOP[i + 1]
               then Overflow(i)
          end
     else begin
             TOP[i] ← TOP[i] − 1;
             if TOP[i] ≤ TOP[i − 1]
               then Overflow(i)
          end;
   CONTENTS[TOP[i]] ← Y;
end;

Pop(i, Y)    (* Pop the item Y from stack i *)
begin
   if TOP[i] = BASE[i]
     then Underflow(i)
```

```
    else begin
           Y ← CONTENTS[TOP[i]];
           if i is odd
             then TOP[i] ← TOP[i] − 1
             else TOP[i] ← TOP[i] + 1;
         end;
end;
```

While pushing an item into stack i and finding $TOP[i] \geq TOP[i+1]$ or $TOP[i] \leq TOP[i-1]$, we say that overflow is occurring at the group which contains the stack i. The memory reallocation is needed at this time. The reallocation strategy is stated as follows:

1. Find the total amount of available space left.
2. Check whether all the available space is used up.
3. Compute the new $BASE$ address of each stack in accordance with the following principle:
   - Set two reallocation parameters α and β, where $0 \leq \alpha, \beta \leq 1$ and $\alpha + \beta = 1$.
   - Approximately $\alpha \times 100$ percent of available space is shared equally among the n' groups, and the other $\beta \times 100$ percents will be reallocated in proportion to the ratio of growth of individual stack pair since the last overflow.
4. Shift up or down each stack to the accurate position.
5. Adjust related pointers of each stack.

In the example of Figure 2, there are some items pushed into stacks and the stacks become Figure 3a. The left group containing stack 1 and stack 2 is already full at this time. Now, if an item belonging to stack 2 is pushed again. The left group will cause overflow. Before the new item which belongs to stack 2 is pushed, the number of items in the left group and the right group are 10 and 5, respectively; and the total number of available space is 5. Let $\alpha = 0$ and $\beta = 1$ be the predefined reallocation parameters. After reallocation, the number of available space in the left group is $5 \times 10/15 \approx 3$, the right group is $5 \times 5/15 \approx 2$. Then we can push the new item into stack 2, as shown in Figure 3b.

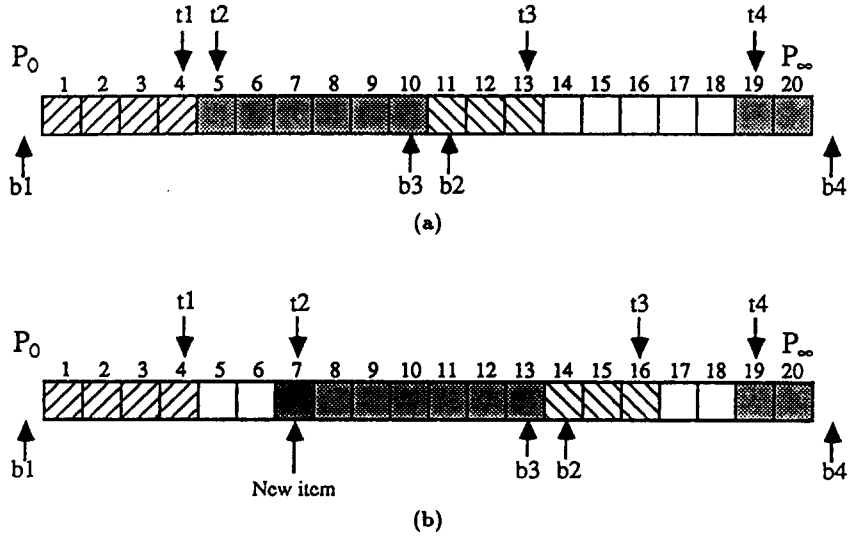

Figure 3. (a) An overflow; (b) the solving strategy of the Multigroup method.

The *Overflow* algorithm contains two additional arrays called $OLDTOP[i]$ and $NEWBASE[i]$, $1 \leq i \leq n$. They are used to store the value of $TOP[i]$, in which previous time memory was

allocated, and the new $BASE[i]$ just after reallocation. Further, a procedure called *Repack* is used to shift the items down or up. The algorithm is listed in as follows:

```
Overflow(i)    (* Solve the i^th stack's overflow *)
begin
   Sum ← P∞ − P0; (* Sum equal to the total amount of memory space left *)
   Inc ← 0;          (* Inc equal to the total amount of increases in stacks
                          size since the last allocation *)
   for j := 1 to n do (* Compute the total number of available space *)
       if j is odd then
         begin
            Sum ← Sum − (TOP[j] − BASE[j]);
            if TOP[j] > OLDTOP[j]
              then begin
                      D[j] ← TOP[j] − OLDTOP[j];
                      Inc ← Inc + D[j];
                   end
              else
                   D[j] ← 0;
         end
       else   begin
            Sum ← Sum − (TOP[j] − BASE[j]);
            if TOP[j] < OLDTOP[j]
              then begin
                      D[j] ← OLDTOP[j] − TOP[j];
                      Inc ← Inc + D[j];
                   end
              else
                   D[j] ← 0;
         end;
   if  sum ≤ 0 (* There is no remained available space *)
       then Systemfail;
   Ave ← α * ⌊Sum/n'⌋;
   Weight ← β * ⌊Sum/Inc⌋;
   NEWBASE[1] ← BASE[1];
   NEWBASE[2n'] ← BASE[2n'];
   σ ← 0;
   for j := 1 to n' − 1 do (* Set new bottom address of stacks *)
       begin
          τ ← σ + D[j − 1] * Weight + Ave;
          NEWBASE[2j] ← NEWBASE[2j − 1] + (TOP[2j − 1] − BASE[2j − 1])
                              +⌊τ⌋ − ⌊σ⌋ + (TOP[2j] − BASE[2j]) + 1;
          NEWBASE[2j + 1] ← NEWBASE[2j] − 1;
          σ ← τ;
       end;
   if  i is odd
      then TOP[i] ← TOP[i] − 1
      else  TOP[i] ← TOP[i] + 1;
   Repack;
   if  i is odd
      then TOP[i] ← TOP[i] + 1
      else  TOP[i] ← TOP[i] − 1;
   for j := 1 to n do (* Save the old TOP[j] for computing the increasing items *)
       OLDTOP[j] ← TOP[j];
end;
```

```
Repack    (* Move stacks' items *)
begin
  j ← 1;
  while j ≤ n do
    begin
    if NEWBASE[j] < BASE[j]    (* Shift down *)
      then begin
             δ ← BASE[j] − NEWBASE[j];
             if j is odd then
               for p := BASE[j] + 1 to TOP[j] do
                   CONTENTS[p − δ] ← CONTENTS[p]
             else
               for p := TOP[j] to BASE[j] − 1 do
                   CONTENTS[p − δ] ← CONTENTS[p];
             BASE[j] ← NEWBASE[j];
             TOP[j] ← TOP[j] − δ;
           end;
      j ← j + 1;
    end;
  while j ≠ 1 do    (* Shift up *)
    begin
      if NEWBASE[j] > BASE[j]
        then begin
               δ ← NEWBASE[j] − BASE[j];
               if j is odd then
                 for p := TOP[j] downto BASE[j] + 1 do
                     CONTENTS[p + δ] ← CONTENTS[p]
               else
                 for p := BASE[j] − 1 downto TOP[j] do
                     CONTENTS[p + δ] ← CONTENTS[p];
               BASE[j] ← NEWBASE[j];
               TOP[j] ← TOP[j] + δ;
             end;
        j ← j − 1;
    end;
end;

Underflow(i)    (* Stack i occurs underflow *)
begin
  writeln('Stack ',i,' is empty.');
  (* print the messages of stack empty *)
end;
```

3. THE PROBABILISTIC ANALYSIS OF THE MULTIGROUP METHOD

The time complexities of the Multigroup method are analyzed as follows. The procedure *Initial* takes $O(n)$ time to initialize the stacks' bottom and top addresses. The operation *Push* (or *Pop*) will spend $O(1)$ time to push an item into one of the stacks (or pop an item from one of the stacks), if the operation is successful. Otherwise, when the operations fail, the *Push* operation will reallocate memory and execute the *Overflow* procedure, and the *Pop* operation will execute the *Underflow* procedure. The *Underflow* procedure only needs $O(1)$ time to reply a message to users that there is no item in the stack. On the other hand, the procedure *Overflow* is much more complicated than the *Underflow*. In the *Overflow* procedure, the first **for** loop is used to compute the total number of available space, taking $O(n)$ time, while the overflow is occurring. The second and the third **for** loop is to set the new bottom addresses and save the old top addresses for every stack, taking $O(n)$ time, too. The only unknown time is the procedure

Repack. The *Repack* procedure, contained in *Overflow*, is used to move data items from old $BASE[i]$ to $NEWBASE[i]$ when overflow occurs. The item movements are very costly in time and take up most of the time of manipulating stacks. But, we do not know how many items will be moved indeed when overflow occurs. Since the number of item movements cannot be observed from the procedure directly, we proposed a probability model estimating the expectation of item movements and finding the overflow probability when k items are pushed into the stacks. Of course, k should not be larger than the size of memory.

Assume that the memory size used to manipulate all stacks is $P_\infty - P_0 = m$, the number of stacks is n, and n' is defined as in Section 2. An experiment which pushes items into n different stacks contained by n' groups will be done. For the sake of easy understanding, in the experiment, the m memory cells can be thought of as m balls and the n' groups as n' boxes. The n' boxes are labeled $1, \ldots, n'$. The random variable Y_j, $1 \le j \le m$, is defined as the label of the box into which the j^{th} ball is placed. $\{Y_j = i\}$ denotes the event that the experiment pushes the j^{th} ball into the i^{th} box. Let k pushes be a k-dimensional random vector $(Y_1, \ldots, Y_k)$, where $1 \le k \le m$. Here, the random variables $Y_1, \ldots, Y_k$ are mutually independent.

Each experiment begins with empty boxes. Then the balls are placed into the boxes and form a random vector $(Y_1, \ldots, Y_j, \ldots)$. After the m balls are all pushed into the boxes, the pushing will be stopped and the experiment is finished once. The finished experiment generates an m-dimensional random vector. Each random vector has at most m dimensions, when the number of balls is m.

We define X_i^j to be the random variable of the number of balls in i^{th} box after j balls are pushed, where $1 \le j \le m$ and $1 \le i \le n'$. Since the boxes in our experiment correspond to the groups in the Multigroup method, the items pushed into stack $2i-1$ or stack $2i$ should be placed to group i, i.e., box i. The difference between groups and boxes is that the capacity of each group is bounded and each box almost has no limitation. That is, when the number of balls is larger than the capacity of groups, overflow will occur. Let t_i, $1 \le i \le n'$, represent the number of available space of group i. The overflow will occur at the smallest j with $X_i^j > t_i$ for $1 \le i \le n'$.

We now discuss the expectation of total pushed items when overflow occurs. For simplicity, only the first overflow situation is considered here. At the initial step, assume the total memory size m is divisible by the number of groups n', and each group is of the same size m/n'. We denote $X_{(n')}^j$ to be the maximum of $X_1^j, \ldots, X_{n'}^j$, i.e., $X_{(n')}^j = \max\{X_1^j, \ldots, X_{n'}^j\}$. The first overflow will occur at the smallest j with $X_{(n')}^j > m/n'$, where $1 \le j \le m$. $\{X_{(n')}^k > t\}$ is the event that the maximum of the items in n' groups is more than t after k items are pushed. To compute the probability of the event $\{X_{(n')}^k > t\}$ more easily, we define $N(t)$ to be the smallest j with $X_{(n')}^j > t$; i.e., $N(t) = \inf\{j; X_{(n')}^j > t\}$. It is clear that the event $\{X_{(n')}^k > t\}$ is equivalent to the event $\{N(t) \le k\}$.

Therefore, when the k^{th} item is pushed, the probability of overflow will be

$$\begin{aligned}\Pr\{N(t) = k\} &= \Pr\{N(t) \le k\} - \Pr\{N(t) \le k-1\} \\ &= \Pr\{X_{(n')}^k > t\} - \Pr\{X_{(n')}^{k-1} > t\}.\end{aligned}$$

Now, if we place k balls into n' boxes, a k-dimensional random vector $(Y_1, Y_2, \ldots, Y_k)$ will be generated. Let the probability of placing a ball into the box i be $p'_i = \Pr\{Y_j = i\}$, $1 \le i \le n'$. The joint density of the numbers of balls in the n' boxes $X_1^k, \ldots, X_{n'}^k$ is as follows.

$$\Pr\{X_1^k = y_1, \ldots, X_{n'}^k = y_{n'}\} = \begin{cases} \dfrac{k!}{(y_1!)\cdots(y_{n'}!)} p'^{y_1}_1 \cdots p'^{y_{n'}}_{n'}, & y_i \text{ are nonnegative integers,} \\ & \text{such that } y_1 + \cdots + y_{n'} = k; \\ 0, & \text{elsewhere.} \end{cases}$$

Let

$$P_k(t) = \Pr\{X_1^k \le t, \ldots, X_{n'}^k \le t\}, \tag{1}$$

$$P_k(t) = \sum_{0 \le y_1 \cdots y_{n'} \le t} \frac{k!}{(y_1!)\cdots(y_{n'}!)} p'^{y_1}_1 \cdots p'^{y_{n'}}_{n'}, \tag{2}$$

where $y_1, \ldots, y_{n'}$ are nonnegative integers, such that $y_1 + y_2 + \cdots + y_{n'} = k$. We have

$$\begin{aligned} \Pr\{X_{(n')}^k > t\} &= 1 - \Pr\{X_{(n')}^k \le t\} \\ &= 1 - \Pr\{X_1^k \le t, \ldots, X_{n'}^k \le t\} \\ &= 1 - P_k(t). \end{aligned}$$

Therefore, when the k^{th} item is pushed, the overflow probability $\Pr\{N(t) = k\}$ can be represented as

$$\begin{aligned} \Pr\{N(t) = k\} &= \Pr\{X_{n'}^k > t\} - \Pr\{X_{n'}^{k-1} > t\} \\ &= 1 - P_k(t) - (1 - P_{k-1}(t)) \\ &= P_{k-1}(t) - P_k(t). \end{aligned} \tag{3}$$

If m is divisible by n', $\Pr\{N(m/n') = k\}$ is the probability of the first overflow occurring at the time when the k^{th} item is pushed after the n stacks are initialized. However, in general, the available space of each group are not the same in the second, third and later overflows. In order to suit these overflows, the equations (1)–(3) are generalized as follows. Let $t_1, t_2, \ldots, t_{n'}$ represent the number of available space of group 1, group 2, ..., group n', respectively. $N(t_1, t_2, \ldots, t_{n'})$ stands for the smallest j that one of the conditions $X_1^j > t_1, X_2^j > t_2, \ldots, X_{n'}^j > t_{n'}$ is satisfied; i.e., $N(t_1, t_2, \ldots, t_{n'}) = \inf\{j; X_i^j > t_i, 1 \le i \le n'\}$. And we define

$$P_k(t_1, \ldots, t_{n'}) = \Pr\{X_1^k \le t_1, \ldots, X_{n'}^k \le t_{n'}\}.$$

We have the following lemma.

LEMMA 1. *In the Multigroup method, after k items are pushed into n stacks, the probability of no overflow is*

$$P_k(t_1, \ldots, t_{n'}) = \sum_{0 \le y_1 \le t_1} \cdots \sum_{0 \le y_{n'} \le t_{n'}} \frac{k!}{(y_1!)\cdots(y_{n'}!)} p'^{y_1}_1 \cdots p'^{y_{n'}}_{n'},$$

where $y_1, \ldots, y_{n'}$ are nonnegative integers, such that $y_1 + y_2 + \cdots + y_{n'} = k$.

PROOF. Since the items belonging to stack $2i - 1$ stack $2i$ are pushed into group i, where $1 \le i \le n'$. Let p_j be the probability of pushing an item into stack j, and x_j be the number of items pushed into stack j, $1 \le j \le n$. Therefore, we have

$$\begin{aligned} p'_1 &= p_1 + p_2, \\ p'_2 &= p_3 + p_4, \\ &\vdots \\ p'_{n'} &= p_{2n'-1} + p_{2n'}. \end{aligned}$$

and

$$\begin{aligned} y_1 &= x_1 + x_2, \\ y_2 &= x_3 + x_4, \\ &\vdots \\ y_{n'} &= x_{2n'-1} + x_{2n'}. \end{aligned}$$

If n is odd, $p_{2n'} = 0$ and $x_{2n'} = 0$. Since $P_k(t_1, \ldots, t_{n'})$ is the general form of the equation (1), the lemma can be proved by applying the equation (2). ∎

If items are pushed into a multiple stacks system with n stacks, and $t_1, \ldots, t_{n'}$ are the numbers of available space of the n' groups, respectively, since the last reallocation, then we have the following lemma.

LEMMA 2. *When the k^{th} item is pushed, the probability of overflow is $P_{k-1}(t_1, \ldots, t_{n'}) - P_k(t_1, \ldots, t_{n'})$.*

PROOF. The overflow probability is similar to the equation (3),

$$\Pr\{N(t_1, \ldots, t_{n'}) = k\} = \Pr\{N(t_1, \ldots, t_{n'}) \leq k\} - \Pr\{N(t_1, \ldots, t_{n'}) \leq k-1\},$$

and

$$\begin{aligned}\Pr\{N(t_1, \ldots, t_{n'}) \leq k\} &= 1 - \Pr\{X_1^k \leq t_1, \ldots, X_{n'}^k \leq t_{n'}\} \\ &= 1 - P_k(t_1, \ldots, t_{n'}).\end{aligned}$$

Thus,

$$\Pr\{N(t_1, \ldots, t_{n'}) = k\} = P_{k-1}(t_1, \ldots, t_{n'}) - P_k(t_1, \ldots, t_{n'}). \qquad \blacksquare$$

Suppose the total number of available space so far is $t_1 + \cdots + t_{n'} = m$, n is the number of stacks, and n' is the number of groups. We define $EX_{m,n}$ to be the expectation of pushed items when overflow occurs.

LEMMA 3.

$$EX_{m,n} = \sum_{j=0}^{m} P_j(t_1, \ldots, t_{n'}).$$

PROOF. By Lemma 2,

$$\begin{aligned}EX_{m,n} &= \sum_{j=1}^{m+1} j \cdot \Pr\{N(t_1, \ldots, t_{n'}) = j)\} \\ &= \sum_{j=1}^{m+1} j \cdot (P_{j-1}(t_1, \ldots, t_{n'}) - P_j(t_1, \ldots, t_{n'})) \\ &= P_0(t_1, \ldots, t_{n'}) + P_1(t_1, \ldots, t_{n'}) + \cdots + P_m(t_1, \ldots, t_{n'}) - (m+1) \cdot P_{m+1}(t_1, \ldots, t_{n'}).\end{aligned}$$

Since the total number of available space is m, the $(m+1)^{\text{th}}$ push will cause overflow definitely. We have $P_{m+1}(t_1, \ldots, t_{n'}) = 0$, thus

$$EX_{m,n} = \textstyle\sum_{j=0}^{m} P_j(t_1, \ldots, t_{n'}). \qquad \blacksquare$$

When overflow occurs, the total items in n stacks must be repacked. Since the number of items in each group is different, various numbers of data movements are required. However, item movements are not necessary for all stacks, for the new bottom addresses $NEWBASE[i]$, $1 \leq i \leq n$, are determined by the ratio of each stack's items and the two parameters α and β. If the old bottom address $BASE[i]$ is equal to the new bottom address $NEWBASE[i]$, the items in i^{th} stack will avoid being moved. Moreover, since the bottom of stack $2i$ is connected with the bottom of stack $2i+1$ for $1 \leq i \leq n'-1$, the movement of stack $2i$ and stack $2i+1$ will occur at the same time.

We observe the addresses $BASE[i]$ and $NEWBASE[i]$ which are the bottom addresses of stack i before and after repacking, for $1 \leq i \leq n$. Let $t_1, t_2, \ldots, t_{n'}$ represent the number of available space in n' groups and x_i be the number of items in stack i when the last reallocation was done, $x_1 + \cdots + x_n = k$. Assume the bottom address of stack 1 is zero, i.e., $BASE[1] = 0$. Then, the bottom addresses of stacks n will be $BASE[n] = m+1$ if n is even (or $BASE[n+1] = m+1$

if n is odd), where m is the total number of memory cells. The bottom addresses of the other stacks are

$$\left.\begin{aligned} BASE[2i] &= \sum_{j=1}^{i}(x_{2j-1}+x_{2j}+t_j)+1 \\ BASE[2i+1] &= \sum_{j=1}^{i}(x_{2j-1}+x_{2j}+t_j) \end{aligned}\right\}, \qquad \text{for } 1 \le i \le n'-1.$$

From the last reallocation on, there are x'_i items pushed into stack i for $1 \le i \le n$, and $x'_1 + \cdots + x'_n = k'$. If overflow occurs at the k^{th} pushed item, the procedure *Overflow* will be executed as follows. First, it calculates the total number of available space in the n' groups and sets the new bottom address $NEWBASE[i]$; then the procedure *Repack* is started. Let $t'_1, t'_2, \ldots, t'_{n'}$ represent the number of available space remained in n' groups after the all k' items are pushed. Except the stack 1 and stack $2n'$, the new bottom addresses of stacks should be

$$\left.\begin{aligned} NEWBASE[2i] &= \sum_{j=1}^{i}(x_{2j-1}+x_{2j}+x'_{2j-1}+x'_{2j}+t'_j)+1 \\ NEWBASE[2i+1] &= \sum_{j=1}^{i}(x_{2j-1}+x_{2j}+x'_{2j-1}+x'_{2j}+t'_j) \end{aligned}\right\}, \qquad \text{for } 1 \le i \le n'-1.$$

The following condition must be satisfied in order to avoid moving items.

LEMMA 4. *The necessary and sufficient condition of* $BASE[2i] = NEWBASE[2i]$ *or* $BASE[2i+1] = NEWBASE[2i+1]$ *in the Multigroup method is*

$$\sum_{j=1}^{2i} x'_j \cdot \left(1 + \left\lfloor \frac{m-(k+k')}{k'} \right\rfloor \times \beta\right) = \alpha \times i \cdot \left(\left\lfloor \frac{m-k}{n} \right\rfloor - \left\lfloor \frac{m-(k+k')}{n} \right\rfloor\right) + \beta \times \left(\left\lfloor \frac{m-k}{k} \right\rfloor \cdot \sum_{j=1}^{2i} x_j\right),$$

where $1 \le i \le n'-1$ *and* α, β *are the parameters in procedure Overflow.*

PROOF. If $BASE[2i] = NEWBASE[2i]$ or $BASE[2i+1] = NEWBASE[2i+1]$, $1 \le i \le n'-1$, the addresses of the stacks before and after overflow should satisfy the condition

$$\sum_{j=1}^{i}(x_{2j-1}+x_{2j}+t_j) = \sum_{j=1}^{i}(x_{2j-1}+x_{2j}+t'_j+x'_{2j-1}+x'_{2j}),$$

except that the bottom addresses of stack 1 and stack $2n'$ are always 0 and $m+1$. So,

$$\sum_{j=1}^{i}(x_{2j-1}+x_{2j}+t_j) = \sum_{j=1}^{i}(x_{2j-1}+x_{2j}+t'_j+x'_{2j-1}+x'_{2j}), \qquad \sum_{j=1}^{2i} x'_j = \sum_{j=1}^{i}(t_i - t'_i). \quad (4)$$

Since the available space is determined by the number of items in the stacks and the parameters α and β, the values of t_j and t'_j are as follows:

$$t_j = \left\lfloor \frac{m-k}{n'} \right\rfloor \times \alpha + (x_{2j-1}+x_{2j}) \cdot \left\lfloor \frac{m-k}{k} \right\rfloor \times \beta, \quad (5)$$

$$t'_j = \left\lfloor \frac{m-(k+k')}{n'} \right\rfloor \times \alpha + (x'_{2j-1}+x'_{2j}) \cdot \left\lfloor \frac{m-(k+k')}{k'} \right\rfloor \times \beta. \quad (6)$$

We replace the t_j and t'_j in equation (4) with (5) and (6), and get

$$\sum_{j=1}^{2i} x'_j = \alpha \times i \cdot \left(\left\lfloor \frac{m-k}{n'} \right\rfloor - \left\lfloor \frac{m-(k+k')}{n'} \right\rfloor\right) + \beta \times \left(\left\lfloor \frac{m-k}{k} \right\rfloor \cdot \sum_{j=1}^{2i} x_j - \left\lfloor \frac{m-(k+k')}{k'} \right\rfloor \cdot \sum_{j=1}^{2i} x'_j\right).$$

Thus,

$$\sum_{j=1}^{2i} x'_j \cdot \left(1 + \left\lfloor \frac{m-(k+k')}{k'} \right\rfloor \times \beta\right) = \alpha \times i \cdot \left(\left\lfloor \frac{m-k}{n'} \right\rfloor - \left\lfloor \frac{m-(k+k')}{n'} \right\rfloor\right) + \beta \times \left(\left\lfloor \frac{m-k}{k} \right\rfloor \cdot \sum_{j=1}^{2i} x_j\right).$$

If the above condition is satisfied, the stack $2i$ and stack $2i+1$ will not be moved during the repacking process. ∎

In the condition of Lemma 4, the $x_1 + \cdots + x_n = k$ is the total number of items of the last reallocation, and m is the initial number of available space. Let the random variable X_i' be the number of pushed items since the last reallocation. The probability of stack i which has no movement is defined as q_i.

LEMMA 5.

$$q_i = 1, \qquad \text{for } i = 1 \text{ and } i = 2n' \text{ if } i \text{ is even}; \tag{7}$$

$$\begin{aligned} q_{2i} = q_{2i+1} = \sum_{k'=1}^{m-k+1} \Bigg[& \binom{k'-1}{c_i} \left(\frac{2i}{n}\right)^{c_i} \left(\frac{n-2i}{n}\right)^{k'-c_i} \\ & \times [P_{c_i}(t_1, \ldots, t_i) \cdot (P_{k'-1-c_i}(t_{i+1}, \ldots, t_{n'}) - P_{k'-c_i}(t_{i+1}, \ldots, t_{n'}))] \\ & + \binom{k'-1}{c_i - 1} \left(\frac{2i}{n}\right)^{c_i} \left(\frac{n-2i}{n}\right)^{k'-c_i} \\ & \times [P_{k'-c_i}(t_{i+1}, \ldots, t_{n'}) \cdot (P_{c_i-1}(t_1, \ldots, t_i) - P_{c_i}(t_1, \ldots, t_i))] \Bigg], \end{aligned} \tag{8}$$

$$\text{for } 1 \le i \le n' - 1,$$

where

$$c_i = \left\lfloor \frac{\alpha \times i \cdot \left(\left\lfloor \frac{m-k}{n'} \right\rfloor - \left\lfloor \frac{m-(k+k')}{n'} \right\rfloor \right) + \beta \times \left(\left\lfloor \frac{m-k}{k} \right\rfloor \cdot \sum_{j=1}^{2i} x_j \right)}{1 + \left\lfloor \frac{m-(k+k')}{k'} \right\rfloor \times \beta} \right\rfloor .$$

PROOF. We have known that the condition of the i^{th} stack will not be moved as described in Lemma 4. Since the numbers of items are all integers in our discussions, the event

$$\sum_{j=1}^{2i} X_j' = \frac{\alpha \times i \cdot \left(\left\lfloor \frac{m-k}{n'} \right\rfloor - \left\lfloor \frac{m-(k+k')}{n'} \right\rfloor \right) + \beta \times \left(\left\lfloor \frac{m-k}{k} \right\rfloor \cdot \sum_{j=1}^{2i} x_j \right)}{1 + \left\lfloor \frac{m-(k+k')}{k'} \right\rfloor \times \beta}$$

will be replaced by

$$\sum_{j=1}^{2i} X_j' = \left\lfloor \frac{\alpha \times i \cdot \left(\left\lfloor \frac{m-k}{n'} \right\rfloor - \left\lfloor \frac{m-(k+k')}{n'} \right\rfloor \right) + \beta \times \left(\left\lfloor \frac{m-k}{k} \right\rfloor \cdot \sum_{j=1}^{2i} x_j \right)}{1 + \left\lfloor \frac{m-(k+k')}{k'} \right\rfloor \times \beta} \right\rfloor .$$

Let

$$c_i = \left\lfloor \frac{\alpha \times i \cdot \left(\left\lfloor \frac{m-k}{n'} \right\rfloor - \left\lfloor \frac{m-(k+k')}{n'} \right\rfloor \right) + \beta \times \left(\left\lfloor \frac{m-k}{k} \right\rfloor \cdot \sum_{j=1}^{2i} x_j \right)}{1 + \left\lfloor \frac{m-(k+k')}{k'} \right\rfloor \times \beta} \right\rfloor .$$

Then, we have

$$q_{2i} = q_{2i+1} = \Pr \left\{ \sum_{j=1}^{2i} X_j' = c_i \right\}, \qquad \text{for } 1 \le i \le n' - 1.$$

Assume k items are pushed into the n stacks during the time that from the initial step to the last reallocation, and the overflow occurs when the k'^{th} item are pushed since the last reallocation. When the condition of no movement is satisfied, there are c_i items which will be pushed into the stack $1, \ldots,$ stack $2i$, and the others will be pushed into the stack $2i + 1, \ldots,$ stack n. It forms a binomial distribution. Let the overflow occur at the group containing stack $1, \ldots,$ stack $2i$,

$$\begin{aligned} q_{2i} = q_{2i+1} = \sum_{k'=1}^{m-k+1} \Bigg[& \binom{k'-1}{c_i - 1} \left(\frac{2i}{n}\right)^{c_i} \left(\frac{n-2i}{n}\right)^{k'-c_i} \\ & \times [P_{k'-c_i}(t_{i+1}, \ldots, t_{n'}) \cdot (P_{c_i-1}(t_1, \ldots, t_i) - P_{c_i}(t_1, \ldots, t_i))] \Bigg]. \end{aligned}$$

The other situation is that the overflow occurs at the group containing stack $2i+1, \ldots,$ stack n. In this case, we have

$$q_{2i} = q_{2i+1} = \sum_{k'=1}^{m-k+1} \left[\binom{k'-1}{c_i} \left(\frac{2i}{n}\right)^{c_i} \left(\frac{n-2i}{n}\right)^{k'-c_i} \times [P_{c_i}(t_1, \ldots, t_i) \cdot (P_{k'-1-c_i}(t_{i+1}, \ldots, t_{n'}) - P_{k'-c_i}(t_{i+1}, \ldots, t_{n'}))] \right].$$

Since stack 1 and stack $2n'$ are not moved always, $q_1 = q_{2n'} = 1$. The probabilities of the other stacks are the summation of the above cases. ∎

What we want to discuss now is the number of moved items when overflow occurs. Clearly, the expectation of item movements is the expectation of the pushed items when overflow occurs subtracting the expectation of items without movement in the n stacks. We conclude the above results in the following theorem. If in the system there remains m' available space after $j-1$ overflows, the number of groups is n', and EX^{j-1} is the sum of expectation of the number of items in the stacks after $j-1$ overflows, since the system was started. Let $j \geq 0$ and $EX^0 = 0$. We have $EX^j = EX^{j-1} + EX_{m',n}$.

THEOREM 1. *The expectation of item movements in j^{th} overflow is*

$$EX^j \cdot \left(1 - \sum_{i=1}^{n} p_i \cdot q_i\right).$$

PROOF. Since the stacks' experiments is a multinomial distribution, the expectation of items in each stack is $(EX^{j-1} + EX_{m',n}) \cdot p_i$ when j^{th} overflow occurs at the expectation $EX_{m',n}$. We know that $EX^j = EX^{j-1} + EX_{m',n}$. Therefore, the expectation of the number of item movements becomes

$$EX^j - \sum_{i=1}^{n}(EX^j \cdot p_i \cdot q_i) = EX^j \cdot \left(1 - \sum_{i=1}^{n} p_i \cdot q_i\right). \quad ∎$$

$P_k(t_1, \ldots, t_{n'})$ is the probability of no overflow after k items are pushed. $P_{k-1}(t_1, \ldots, t_{n'}) - P_k(t_1, \ldots, t_{n'})$ is the probability of overflow when k^{th} item is pushed. Therefore, the probability for the number of overflow occurring after k items are pushed will be found by the recurrence function in the following theorem. Let $Q_i(k, n')$ be the probability of the Multigroup method whose overflow occurs i times. We have

THEOREM 2.

$$Q_i(k, n') = \sum_{j=1}^{k} [(P_{j-1}(t_1, \ldots, t_{n'}) - P_j(t_1, \ldots, t_{n'})) \cdot P_{k-j}(t'_1, \ldots, t'_{n'})], \qquad \text{for } i = 1;$$

$$Q_i(k, n') = \sum_{j=1}^{k} [(P_{j-1}(t_1, \ldots, t_{n'}) - P_j(t_1, \ldots, t_{n'})) \cdot Q_{i-1}(k-j, n')], \qquad \text{for } i \geq 2.$$

PROOF. The probability of only one overflow after k items pushed is that the first overflow occurs at the j^{th} pushed item and the remained $k-j$ items do not cause overflow. Let $t_1, \ldots, t_{n'}$ and $t'_1, \ldots, t'_{n'}$ be the number of available space in n' groups before and after overflow occurring. So, let $t_1 + \cdots + t_{n'} = j$ and $t'_1 + \cdots + t'_{n'} = k-j$,

$$Q_1(k, n') = \sum_{j=1}^{k} [\Pr\{N(t_1, \ldots, t_{n'}) = j\} \cdot P_{k-j}(t'_1, \ldots, t'_{n'})]$$

$$= \sum_{j=1}^{k} [(P_{j-1}(t_1, \ldots, t_{n'}) - P_j(t_1, \ldots, t_{n'})) \cdot P_{k-j}(t'_1, \ldots, t'_{n'})].$$

By recurrences, we can get

$$Q_i(k,n') = \sum_{j=1}^{k} [(P_{j-1}(t_1,\ldots,t_{n'}) - P_j(t_1,\ldots,t_{n'})) \cdot Q_{i-1}(k-j,n')], \qquad \text{for } i \geq 2. \quad \blacksquare$$

4. THE SIMULATIONS AND COMPARISONS

In Section 3, we analyze the number of item movements. In this section we will simulate the Multigroup method and compare with its theoretical values. Our simulations are made by giving a sequence of random data, which assumes the probabilities of items pushed into each stack are the same. Moreover, the two parameters α and β are set to 1 and 0. At the beginning of our simulations, the number of available space in each group is $\lfloor m/n' \rfloor$, for $1 \leq i \leq n$. After an overflow occurred, the number of available space in each group is determined by the reallocation parameters and the number of items in the stacks at that time. By the formulas of Section 3, we compute the theoretical values of the first overflow and compare with experimental average values when the first overflow occurs. The experimental values are averaged by running 1000 different cases.

Table 1. The comparison of the first overflow's theoretical values and experimental results.

m	n	$EX_{m,n}$	Experi.	$\sum_{i=1}^{n} q_i$	Experi.
20	2	21.0000	21.0000	2.0000	2.0000
	4	18.2999	17.6781	2.0000	2.0000
	5	13.8081	13.8159	1.3794	1.4185
40	4	36.8598	36.2438	2.0000	2.0000
	5	29.7458	29.5888	1.2701	1.2708
	10	29.1027	29.2042	3.1229	3.1151
50	4	46.2739	45.4364	2.0000	2.0000
	5	36.7200	36.8208	1.1833	1.1902
	10	37.2426	37.3387	3.0232	3.0301
60	5	46.0593	46.1512	1.1939	1.1761
	6	51.7092	51.6531	2.3974	2.4252
	10	45.5479	45.6486	2.9446	2.9690
80	4	74.7968	74.3549	2.0000	2.0000
	5	60.2033	60.3363	1.1123	1.1101
	8	65.9726	66.3240	2.5998	2.5878
100	4	93.9615	93.0033	2.0000	2.0000
	5	76.8048	76.7397	1.0964	1.0721
	10	79.8300	79.9670	2.7446	2.8368

In Table 1, we list two values: one is the mean of the items when the first overflow occurs and the other is the mean of the total number of stacks having no movement. The experimental results are close to the theoretical values. The difference between the theoretical values and experimental results is caused by the experimental variants. If the number of experiments approaches infinite, the experimental results will almost match the theoretical values.

Figure 4 shows the numbers of item movements of the three methods under the memory size is 100 and the same pushed items. From the figure, we know that the numbers of item movements of Garwick's method are much smaller than Knuth's methods, but it does not mean that the Garwick's method is always better. The reason is that the Garwick's method needs to spend extra time to calcualate the number of available space and preset the addresses $NEWBASE[i]$.

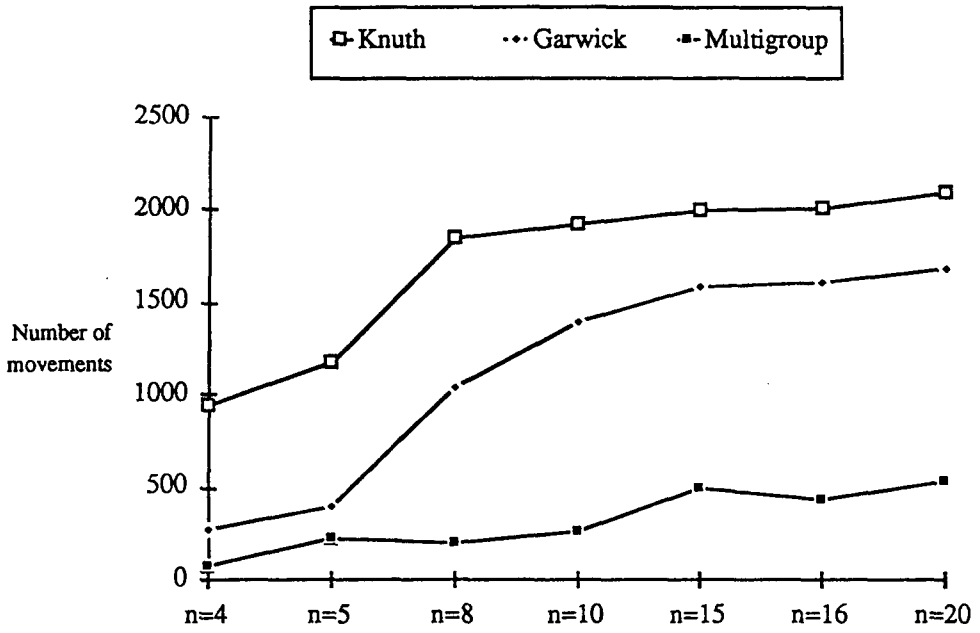

Figure 4. Figure 4. The comparison of the number of item movements with the two methods in [4]; the memory size = 100 and n is the number of stacks.

If n is large, the additional computation will consume much time. It is a great disadvantage for the Garwick's method. In general, the critical ratio of the number of stacks n and the memory size m, n/m, is about 1/12 from our observation. If the ratio of n/m is larger than 1/12, Knuth's method will be better than Garwick's method. Therefore, we had better know the memory size and the number of stacks in advance and keep the ratio n/m under the critical ratio to guarantee the performance.

The Multigroup method not only improves the overhead of extra computations but also reduces the number of item movements, since the number of groups is only a half of the number of stacks approximately. In Figure 4, we can find the number of item movements in Multigroup method is better more than a half of Garwick's method's when n is even. When n is odd, it will cause the worse cases, but the performance is still better than the Garwick's method.

5. CONCLUSIONS

In this paper, we analyze the performance of the Multigroup method under push operations. The probability model can be used to explain the properties of the algorithms. By our analysis, the accurate estimation will be found. It can support many types of information when we design a multiple stacks system. We also show that our Multigroup method is a good method for the multiple stacks systems. It improves the previous two methods and obtains a better performance. Further, it uses less extra storage space and computation time than Garwick's method. During our simulations, the probability of data are known in advance, since the data are generated by a uniform generator. If we know the push probability of each stack p_i in advance, the values of α and β can be decided to be $\alpha = 1$, $\beta = 0$ or $\alpha = 0$, $\beta = 1$ simply, but we usually do not know the distribution of pushed items. So, we suggest that the values of α and β be not fixed, and leave them as two variables. Initially the two values can be set to $\alpha = 1$ and $\beta = 0$. We can change them when the *Push* operation push items into the stacks. The strategy of change can depend on the variance of input data. Of course, this will complicate the Multigroup method.

Basically, we can also use the results of our analyses to decide the load factor of a multiple stacks system. Since increasing the load factor and decreasing the processing time is a trade-off, a high load factor will derive a costly processing time. By the analyses of this paper, the best load factor for the multiple stacks system can be found in advance, and the performance will be improved in practice.

REFERENCES

1. D.E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA, (1973).
2. B.-C. Chien, R.-J. Chen and W.-P. Yang, The probability analysis of Garwick's method for multiple stacks manipulation, Manuscript, (1991).
3. W.-P. Yang, T.C. Chiu, W.C. Lee and S.S. Tung, A new strategy for multiple stacks manipulation, *The Computer Journal*, (to appear) (1992).
4. P.G. Hoel, S.C. Port and C.J. Stone, *Introduction to Probability Theory*, Houghton Mifflin Company, Boston, MA, (1971).
5. H.A. David, *Order Statistics*, Wiley, (1981).
6. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Rockville, MD, (1982).