PARALLEL REDUCTION

FIGURE 2-7 EREW PRAM algorithm to sum n elements using $\lfloor n/2 \rfloor$ processors.

```
SUM (EREW PRAM)
Initial condition: List of n ≥ 1 elements stored in A[0 ··· (n − 1)]
Final condition: Sum of elements stored in A[0]
Global variables: n, A[0 ··· (n − 1)], j
begin
 spawn (P0, P1, P2, ···, P⌊n/2⌋−1)
 for all Pi where 0 ≤ i ≤ ⌊n/2⌋ − 1 do
  for j ← 0 to ⌈log n⌉ − 1 do
   if i modulo 2^j = 0 and 2i + 2^j < n then
      A[2i] ← A[2i] + A[2i + 2^j]
   endif
  endfor
 endfor
end
```

PREFIX SUM

FIGURE 2-9 PRAM algorithm to find prefix sums of an n-element list using $n - 1$ processors.

```
PREFIX.SUMS (CREW PRAM):
Initial condition: List of n ≥ 1 elements stored in A[0 ··· (n − 1)]
Final condition: Each element A[i] contains A[0] ⊕ A[1] ⊕ ··· ⊕ A[i]
Global variables: n, A[0 ... (n − 1)], j
begin
 spawn (P1, P2, ..., Pn−1)
 for all Pi where 1 ≤ i ≤ n − 1 do
  for j ← 0 to ⌈log n⌉ − 1 do
   if i − 2^j ≥ 0 then
      A[i] ← A[i] + A[i − 2^j]
   endif
  endfor
 endfor
end
```

LIST RANKING

```
LIST.RANKING (CREW PRAM):
 Initial condition: Values in array next represent a linked list
 Final condition: Values in array position contain original distance
                  of each element from end of list
 Global variables: n, position[0...(n-1)],next[0...(n-1)], j
 begin
  spawn (P0, P1, P2, ..., Pn-1)
  for all Pi where 0 ≤ i ≤ n - 1 do
   if next[i] = i then position[i] ← 0
   else position[i] ← 1
   endif
   for j ← 1 to ⌈log n⌉ do
    position[i] ← position[i] + position[next[i]]
    next[i] ← next[next[i]]
   endfor
  endfor
 end
```

MERGE TWO SORTED LISTS

```
MERGE.LISTS (CREW PRAM):
Given: Two sorted lists of n/2 elements each, stored in
       A[1] ··· A[n/2] and A[(n/2) + 1] ··· A[n]
       The two lists and their unions have disjoint values
Final condition: Merged list in locations A[1] ··· A[n]
Global  A[1 ··· n]
Local   x, low, high, index
begin
 spawn (P1, P2, ..., Pn)
 for all Pi where 1 ≤ i ≤ n do
  { Each processor sets bounds for binary search }
  if i ≤ n/2 then
   low ← (n/2) + 1
   high ← n
  else
   low ← 1
   high ← n/2
  endif
  { Each processor performs binary search }
  x ← A[i]
  repeat
   index ← ⌊(low + high)/2⌋
   if x < A[index] then
    high ← index - 1
   else
    low ← index + 1
   endif
  until low > high
  { Put value in correct position on merged list }
  A[high + i - n/2] ← x
 endfor
end
```

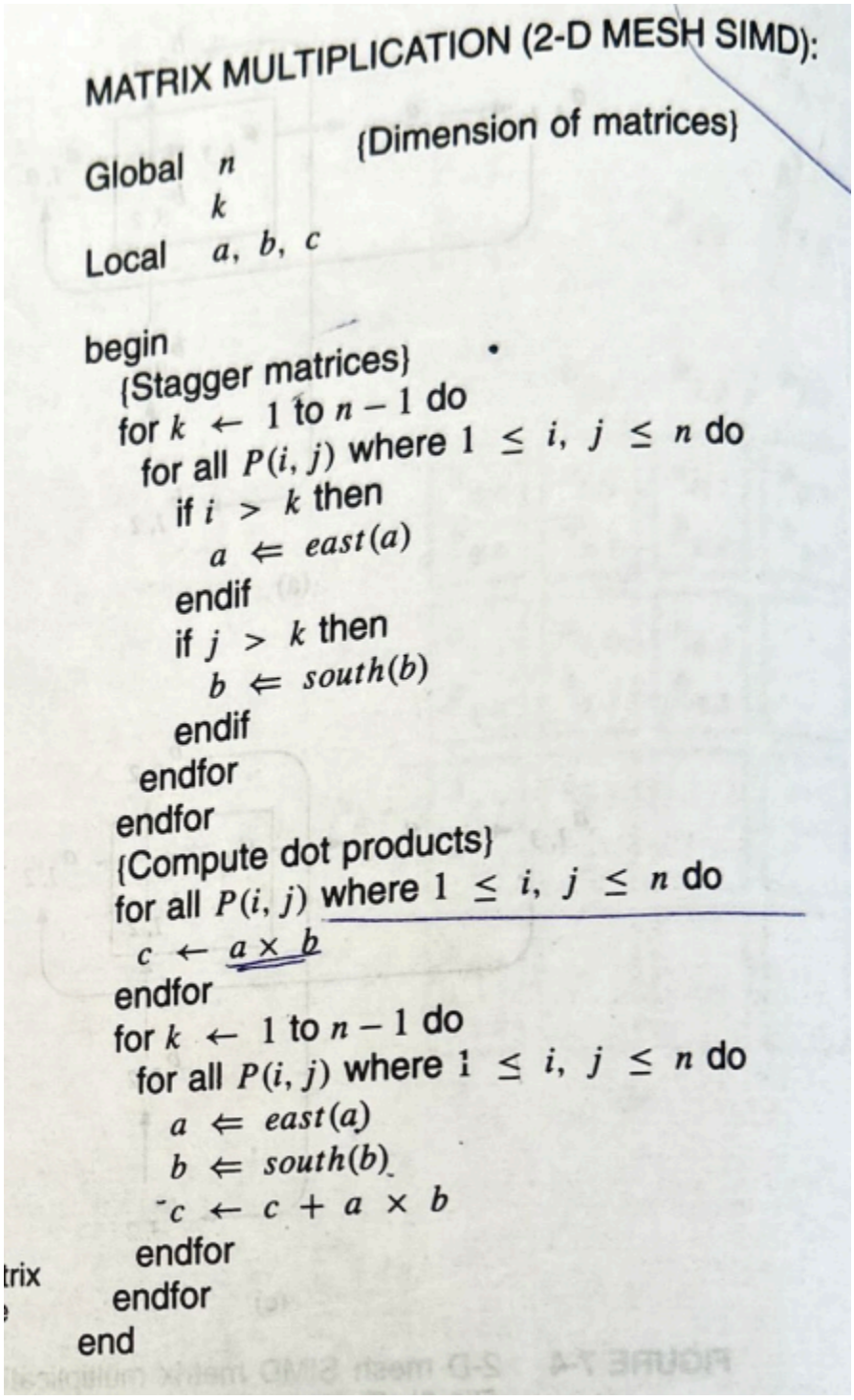
URE 2-17 PRAM algorithm to merge two sorted lists. The two lists and their unions have disjoint values.

GRAPH COLORING

```
GRAPH.COLORING (CREW PRAM):
  Global  n                                  {Number of vertices}
          c                                  {Number of colors}
          A[1...n][1...n]                    {Adjacency matrix}
          candidate[1...c][1...c]···[1...c]  {n-dimensional boolean matrix}
          valid                              {Number of valid colorings}
          j, k
begin
  spawn (P(i_0, i_1, ..., i_{n-1})) where 0 ≤ i_v < c for 0 ≤ v < n
  for all P(i_0, i_1, ..., i_{n-1}) where 0 ≤ i_v < c for 0 ≤ v < n do
    candidate[i_0, i_1, ..., i_{n-1}] ← 1
    for j ← 0 to n − 1 do
      for k ← 0 to n − 1 do
        if A[j][k] and i_j = i_k then
          candidate[i_0, i_1, ..., i_n] ← 0
        endif
      endfor
    endfor
    valid ← Σ candidate   {Sum of all elements of candidate}
  endfor
  if valid > 0 then print "Valid coloring exists"
  else printif "Valid coloring does not exist"
  endif
end
```

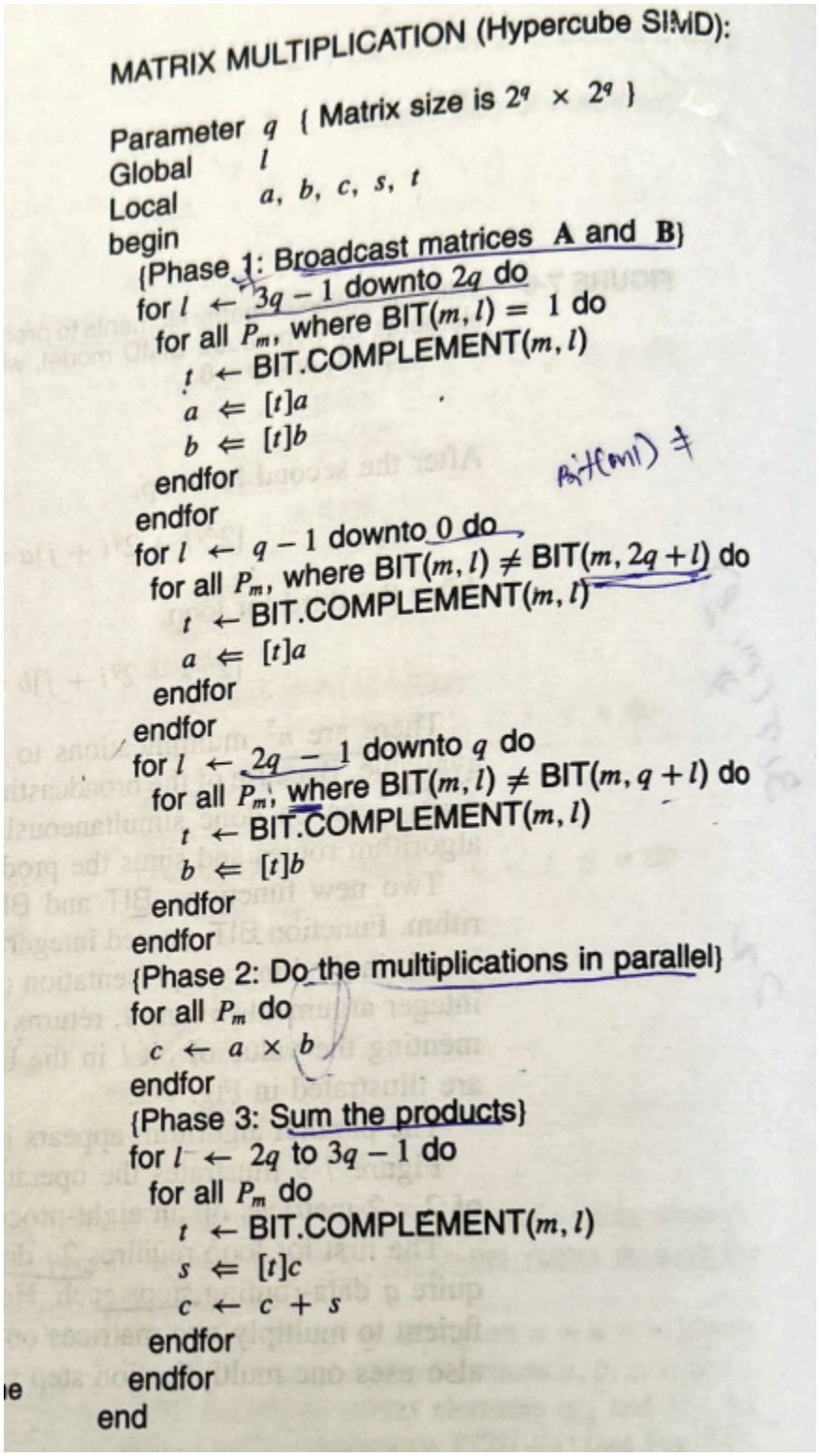
2-19 CREW PRAM algorithm to determine if a graph with n vertices can be colored with c colors.

MATRIX MULTIPLICATION - 2D MESH



```
MATRIX MULTIPLICATION (2-D MESH SIMD):

Global  n                {Dimension of matrices}
        k
Local   a, b, c

begin
  {Stagger matrices}
  for k ← 1 to n − 1 do
    for all P(i, j) where 1 ≤ i, j ≤ n do
      if i > k then
        a ⇐ east(a)
      endif
      if j > k then
        b ⇐ south(b)
      endif
    endfor
  endfor
  {Compute dot products}
  for all P(i, j) where 1 ≤ i, j ≤ n do
    c ← a × b
  endfor
  for k ← 1 to n − 1 do
    for all P(i, j) where 1 ≤ i, j ≤ n do
      a ⇐ east(a)
      b ⇐ south(b)
      c ← c + a × b
    endfor
  endfor
end
```

MATRIX MULTIPLICATION - HYPERCUDE SIMD MODEL



```
MATRIX MULTIPLICATION (Hypercube SIMD):

Parameter q  { Matrix size is 2^q × 2^q }
Global    l
Local     a, b, c, s, t
begin
  {Phase 1: Broadcast matrices A and B}
  for l ← 3q − 1 downto 2q do
    for all P_m, where BIT(m, l) = 1 do
      t ← BIT.COMPLEMENT(m, l)
      a ⇐ [t]a
      b ⇐ [t]b
    endfor
  endfor
  for l ← q − 1 downto 0 do
    for all P_m, where BIT(m, l) ≠ BIT(m, 2q + l) do
      t ← BIT.COMPLEMENT(m, l)
      a ⇐ [t]a
    endfor
  endfor
  for l ← 2q − 1 downto q do
    for all P_m, where BIT(m, l) ≠ BIT(m, q + l) do
      t ← BIT.COMPLEMENT(m, l)
      b ⇐ [t]b
    endfor
  endfor
  {Phase 2: Do the multiplications in parallel}
  for all P_m do
    c ← a × b
  endfor
  {Phase 3: Sum the products}
  for l ← 2q to 3q − 1 do
    for all P_m do
      t ← BIT.COMPLEMENT(m, l)
      s ⇐ [t]c
      c ← c + s
    endfor
  endfor
end
```

ENUMERATION SORT

```
ENUMERATION SORT (SPECIAL CRCW PRAM):

  Parameter   n                          {Number of elements}
  Global      a[0...(n − 1)]             {Elements to be sorted}
              position[0...(n − 1)]      {Sorted positions}
              sorted[0...(n − 1)]        {Contains sorted elements}
begin
  spawn (P_{i,j}, for all 0 ≤ i, j < n)
  for all P_{i,j}, where 0 ≤ i, j < n do
    position[i] ← 0
    if a[i] < a[j] or (a[i] = a[j] and i < j) then
      position[i] ← 1
    endif
  endfor
  for all P_{i,0}, where 0 ≤ i < n do
    sorted[position[i]] ← a[i]
  endfor
end
```

ODD-EVEN TRANSPOSITION SORT

```
ODD-EVEN TRANSPOSITION SORT (ONE-DIMENSIONAL
MESH PROCESSOR ARRAY):

  Parameter   n
  Global      i
  Local       a      {Element to be sorted}
              t      {Element taken from adjacent processor}
begin
  for i ← 1 to n/2 do
    for all P_j, where 0 ≤ j ≤ n − 1 do
      if j < n − 1 and odd(j) then
                                          {Odd-even exchange}
        t ⇐ successor(a)                  {Get value from successor}
        successor(a) ⇐ max(a, t)          {Give away larger value}
        a ← min(a,t)                      {Keep smaller value}
      endif
      if even(j) then
                                          {Even-odd exchange}
        t ⇐ successor(a)                  {Get value from successor}
        successor(a) ⇐ max(a,t)           {Give away larger value}
        a ← min(a,t)                      {Keep smaller value}
      endif
    endfor
  endfor
end
```

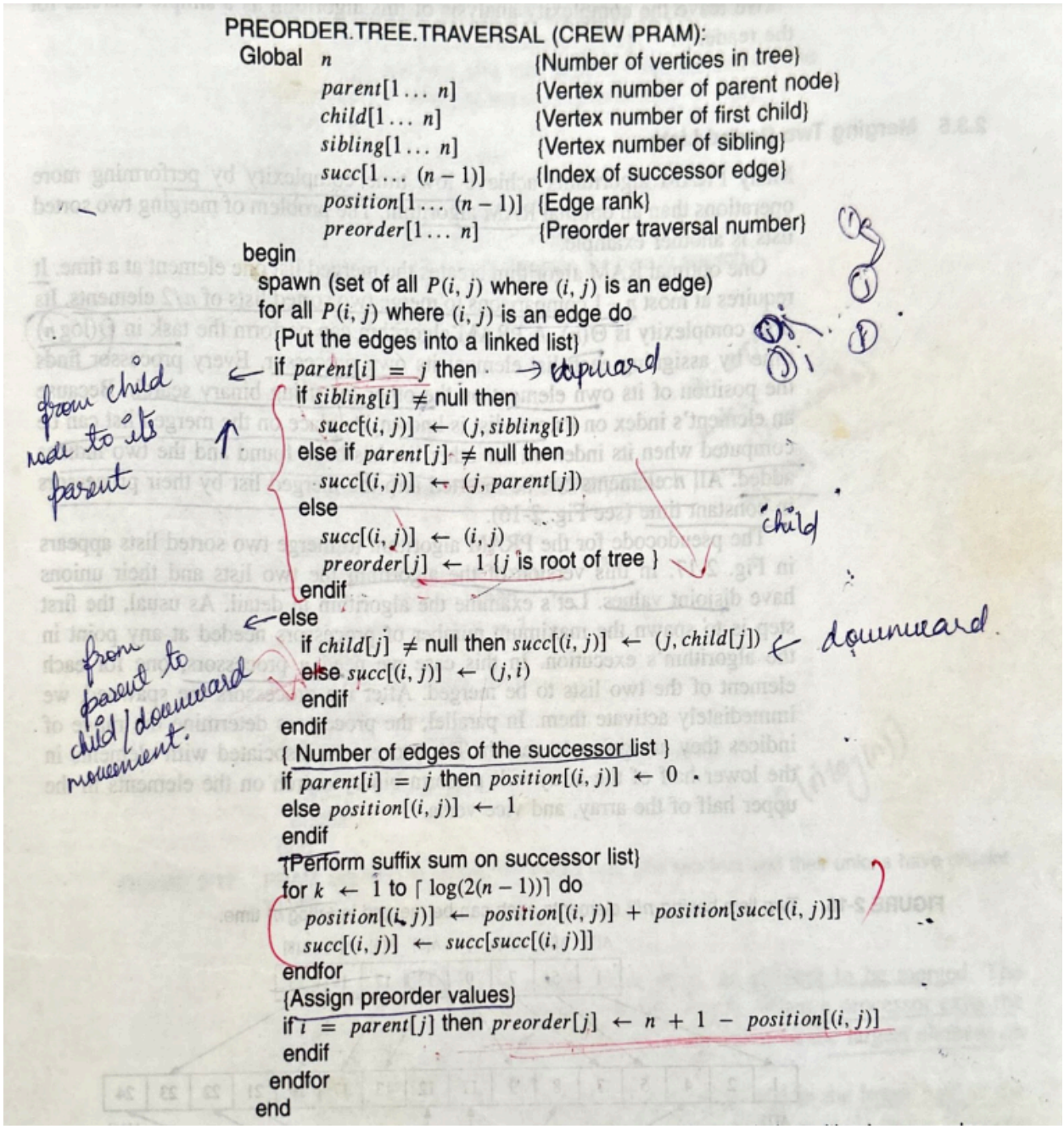
PARALLEL QUICKSORT

```
QUICKSORT (UMA MULTIPROCESSOR):

  Global   n              {Size of array of unsorted elements}
           a[0...(n - 1)] {Array of elements to be sorted}
           sorted         {Number of elements in sorted position}
           min.partition  {Smallest subarray that is partitioned rather than
                           sorted directly}
  Local    bounds         {Indices of unsorted subarray}
           median         {Final position in subarray of partitioning key}
begin
  sorted ← 0
  INITIALIZE.STACK()

  for all P_i, where 0 ≤ i < p do
    while (sorted < n) do
      bounds ← STACK.DELETE()

      while (bounds.low < bounds.high) do
        if (bounds.high - bounds.low < min.partition) then
          INSERTION.SORT (a, bounds.low, bounds.high)
          ADD.TO.SORTED (bounds.high - bounds.low + 1)
          exit while
        else
          median ← PARTITION (bounds.low, bounds.high)
          STACK.INSERT (median + 1, bounds.high)
          bounds.high ← median - 1

        if bounds.low = bounds.high then
          ADD.TO.SORTED (2)
        else
          ADD.TO.SORTED (1)
        endif
      endif
    endwhile
  endwhile
endfor
end
```

HYPERQUICKSORT

```
HYPERQUICKSORT (HYPERCUBE MULTICOMPUTER):
  Global   n            {Initial number of elements per processor}
           d            {Dimension of hypercube}
           i            {Dimension number of current hypercube}
  Local    logical.num  {Unique processor number}
           partner      {Processor's partner in the exchange}
           root         {Root processor of current hypercube}
           splitter     {Median of root processor's sorted list}
begin
  for all P_j, where 0 ≤ j < 2^d do
    Sort n values using sequential quicksort algorithm
    if d > 0 then
      for i ← d downto 1 do
        root ← root of the binary i-cube containing processor logical_num
        if logical.num = root then
          splitter ← median of the sorted list held by processor logical_num
        endif
        Processor root broadcasts splitter to other processors in binary i-cube
        Use splitter to partition sorted values into low list, high list
        partner ← logical_num ⊗ 2^(i-1)      { Bitwise exclusive "or" }
        if logical.num > partner then
          Send low list to processor partner
          Receive another high list from processor partner
        else {logical.num < partner}
          Send high list to processor partner
          Receive another low list from processor partner
        endif
        Merge two lists into a single sorted list of values
      endfor
    endif
  endfor
end
```

PREORDER TREE TRAVERSAL



```
PREORDER.TREE.TRAVERSAL (CREW PRAM):
 Global  n                          {Number of vertices in tree}
         parent[1 ... n]            {Vertex number of parent node}
         child[1 ... n]             {Vertex number of first child}
         sibling[1 ... n]           {Vertex number of sibling}
         succ[1 ... (n - 1)]        {Index of successor edge}
         position[1 ... (n - 1)]    {Edge rank}
         preorder[1 ... n]          {Preorder traversal number}
 begin
  spawn (set of all P(i, j) where (i, j) is an edge)
  for all P(i, j) where (i, j) is an edge do
   {Put the edges into a linked list}
   if parent[i] = j then
    if sibling[i] ≠ null then
     succ[(i, j)] ← (j, sibling[i])
    else if parent[j] ≠ null then
     succ[(i, j)] ← (j, parent[j])
    else
     succ[(i, j)] ← (i, j)
     preorder[j] ← 1 {j is root of tree }
    endif
   else
    if child[j] ≠ null then succ[(i, j)] ← (j, child[j])
    else succ[(i, j)] ← (j, i)
    endif
   endif
   { Number of edges of the successor list }
   if parent[i] = j then position[(i, j)] ← 0
   else position[(i, j)] ← 1
   endif
   {Perform suffix sum on successor list}
   for k ← 1 to ⌈ log(2(n - 1))⌉ do
    position[(i, j)] ← position[(i, j)] + position[succ[(i, j)]]
    succ[(i, j)] ← succ[succ[(i, j)]]
   endfor
   {Assign preorder values}
   if i = parent[j] then preorder[j] ← n + 1 - position[(i, j)]
   endif
  endfor
 end
```