number,

2. The successor of any natural number is unique (e.g., no two natural numbers share the same successor), and

3. No natural number has 1 as a successor.

Property (1) provides the basic mechanism: the successor of 1 is 2; the successor of 2 is 3, and so on. Properties (2) and (3) insure an infinite sequence of numbers which never wraps over itself.

Recursive mathematics became fashionable during the 1930's with the study of "computable" functions, particularly in association with Alan Turing's formal prototype of a programmable computer (1936-7; see also the discussion by Kleene, 1952). Turing was a mathematician, not an engineer, whose stated purpose was to distill mechanical computations to their simplest atoms. He postulated a machine whose actions are completely determined by two numbers: one stored in an internal register, the other situated on a magnetic tape at the reading-and-writing head. Each individual action can alter the register, change the number on the tape, and/or move the tape left or right by one unit. Since each action potentially depends on values stored in the register or written on the tape during one or more previous

actions, the machine's behavior is clearly recursive. Simplistic as Turing's design may seem, it can (given enough time, a large enough register and a long enough tape) implement any calculation executable on a real-life computer.

Recursion has also played fundamental roles in theoretic disciplines other than mathematics. Examples include:

1. Logic: Kurt Goedel's startling proof (1931) that many logical propositions are themselves unprovable

2. Artificial Intelligence: Claude Shannon's proposals for automatic chess-playing (1950),

3. Linguistics: Noam Chomsky's taxonomy of grammars (1957),

4. Theory of Art: George Stiny's and James Gip's formulations of "shape grammars" in painting and sculpture (1972),

5. Theory of Music: the analytic procedures of Lorenz (1921), Schenker (1935) Meyer and Cooper (1960), Tenney (1964), Jackendoff and Lerdahl (1983).

Among computer programmers, "recursion" usually refers specifically to programs which are capable of 'invoking themselves' using methods which will be described in this chapter. Recursion in this sense is an extremely powerful technique of programming with applications to such diverse tasks as sorting large amounts of data, processing grammatic structures (for example, compiling algebraic expressions into sequences of machine instructions), and searching algorithms.

10.1 SELF-INVOKING PROGRAMS

To understand what it means for a program to "invoke" itself, consider a simple application: computing terms of the Fibonacci series. This series was formulated in 1202 by Leonardo Fibonacci to estimate how many progeny a pair of rabbits can produce over a designated number of breeding periods. If we denote the ith Fibonacci as F(i) and establish F(1)=1 and F(2)=2, then Equation 10-1 gives a recursive formula which may be used to derive all subsequent terms:

$$F(i) = F(i-1) + F(i-2) \qquad \text{(Equation 10-1)}$$

F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	F(8)	F(9)	F(10)
1 + 2 = 3									
	2 + 3 = 5								
		3 + 5 = 8							
			5 + 8 = 13						
				8 + 13 = 21					
					13 + 21 = 34				
						21 + 34 = 55			
							34 + 55 = 89		

Fig. 10-1

Given F(1) and F(2), we may then use the procedure detailed in Figure 10-1 to compute subsequent terms.

Figure 10-1: Calculating terms of the Fibonacci series. The vertical alignment illustrates how successive terms serve in turn as input data for later calculations.

Program FIBON1 illustrates how a programmer might implement this process using a subroutine called TERM which invokes itself. TERM is hypothetic because most implementations of FORTRAN '77 disallow self-invoking subroutine calls. The reason for this restriction is that when a subroutine invokes itself, it must be able initially to set aside its variables and later to restore these variables into force once the sub-invocation has run its course. Subroutines encoded in such languages as ALGOL and its descendents incorporate this ability implicitly in their calling sequences, but FORTRAN subroutines do not. We shall see that the extra sophistication in language is not really necessary -- indeed, the practice actually abuses the capabilities of an ALGOL-like language, which really have a different purpose -- but for the moment it will be instructive to see what a self-invoking subroutine call might look like:

Ex 10-1

```
1           program FIBON1
2     C
3     C     Hypothetical program with a self-invoking subroutine
4     C     for computing terms of the Fibonacci series
5     C
6           data MLEV/12/
7           LEVEL = 1
8           call TERM(MLEV,1,1,0)
9           end
```

~~Ex 10-2~~

```
1           subroutine TERM(MLEV,LEVEL,I1,I2)
2     C     Compute term
3           ITERM = I1 + I2
4           print *, LEVEL,ITERM
5     C     Advance to next level
6           if (LEVEL.ge.MLEV) stop
7           call TERM(MLEV,LEVEL+1,ITERM,I1)
8           return
9           end
```

-- Programming example 10-1: program FIBON1 --

TERM's call to itself takes care of the shift in parameters, so that what was originally I1 becomes I2 in the new invocation while what was originally ITERM becomes I1. Only one line of subroutine TERM (line 3) is actually devoted to computing the Fibonacci number. Another line (line 6) keeps track of the level of recursion, denoted by the variable LEVEL, in order to stop the program when its task is complete. The level of recursion indicates the number of nested invocations, or equivalently, the number of returns which must be executed by TERM before control is restored to the main program. Notice, however, that TERM never gives itself a chance to execute a return; it either calls itself or stops the program. The resulting process is a simple feedback loop.

Program FIBON2 illustrates how a programmer would implement the same process without a self-invoking subroutine. The algorithm resolves into an iterative loop with LEVEL serving as the index. The major difference between FIBON1 and FIBON2 is that where FIBON1 managed the shift in Fibonacci terms and the computation of levels implicitly trough nested calls to TERM, FIBON2 manages the same operations explicitly through direct transfers.

Ex 10-2

```
1           program FIBON2
2     C
3     C     Iterative loop for computing terms of the Fibonacci series
4     C
5           data MLEV/12/,LEVEL/1/,I1/1/,I2/0/
6     C
7           do
8     C       Compute term
9               ITERM = I1 + I2
10              print *, LEVEL,ITERM
11    C         Determine I1 and I2 for next iteration
12              I2 = I1
13              I1 = ITERM
14    C         Advance to next level
15              if (LEVEL.ge.MLEV) stop
16              LEVEL = LEVEL + 1
17          repeat
18          end
```

-- Programming example 10-2: program FIBON2 --

The full power of recursive programming becomes manifest when programs are capable of backtracking to earlier levels of recursion. To understand backtracking, consider the problem of enumerating all the distinct ways of choosing N items from a collection of M. This elementary problem has many applications to musical decision-making, including such problems as enumerating all of the distinct triads in the major scale (N=3; M=7) or enumerating all of the distinct seven-note scales in the chromatic scale (N=7; M=12).

Figure 10-2 enumerates all of the ways of choosing 3 items out of 7. In order to exclude redundant combinations, the program must constrain the enumerating process so that 1) the second choice is always larger than the first and 2) the third choice is always larger than the second. The enumerating process is recursive in the sense that it derives each combination by selecting an item and appending it to a combination containing one fewer items. The level of recursion corresponds to the number of items. Thus, the shallowest level enumerates singlets, the median level enumerates ways of appending items to each singlet to produce duplets, and the deepest level enumerates ways of appending items to each duplet to produce triplets.

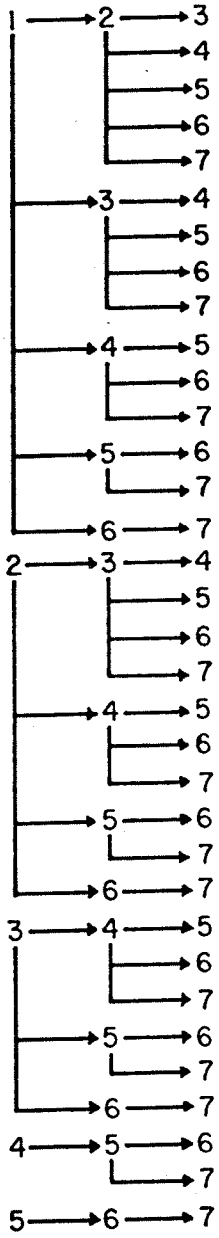

1 → 2 → 3
4
5
6
7
3 → 4
5
6
7
4 → 5
6
7
5 → 6
7
6 → 7
2 → 3 → 4
5
6
7
4 → 5
6
7
5 → 6
7
6 → 7
3 → 4 → 5
6
7
5 → 6
7
6 → 7
4 → 5 → 6
7
5 → 6 → 7

Fig. 10-2

Figure 10-2: Choosing 3 items out of 7 - The leftmost column of numbers shows choices of the first item in a triplet, while the next two columns are each derived by selecting one additional item and appending it to the partial results obtained in the preceding column. Branching arrows indicate where one choice serves for multiple triplets.

Program COMBN1 illustrates how a programmer might implement this process using a self-invoking subroutine called PICK which incorporates the ability to backtrack. Like its counterpart subroutine TERM in program FIBON1, PICK is unacceptable to most FORTRAN '77 compilers. It serves here an illustrative purpose only. Each nested invocation of PICK chooses one item for a combination; the maximum number of recursive levels is the number of items per combination, N.

-- Programming example 10-3: program COMBN1 --

Program COMBN2 illustrates how a programmer would implement the same process without a self-invoking subroutine call. Notice that in place of COMBN1's single variable INDX, COMBN2 requires an array (also called INDX) to store indices for items chosen at each level of recursion. This array accomplishes explicitly what

Ex 16-3

```
1           program COMBN1
2     C
3     C     Hypothetical program with a self-invoking subroutine
4     C     for enumerating all distinct ways of choosing N out of
5     C     M items
6     C
7           data M/7/,N/3/,LEVEL/1/
8           call PICK(LEVEL,M,N,0)
9           stop
10          end
```

```
1           subroutine PICK(LEVEL,M,N,I)
2           do (INDX=I+1,M-N+LEVEL)
3             print *, LEVEL,INDX
4             if (LEVEL.lt.N) call PICK(LEVEL+1,M,N,INDX)
5           repeat
6           LEVEL = LEVEL - 1
7           return
8           end
```

Ex 10-4

```
      program COMBN2
C
C     Iterative loop for enumerating all distinct ways of
C     choosing N out of M items.
C
      parameter (M=7,N=3)
      dimension INDX(N)
      LEVEL = 1
      LIM = M - N
      INDX(LEVEL) = 0
C     Main loop
      do
        I = INDX(LEVEL) + 1
        print *, LEVEL,I
        if (I.ge.LIM+LEVEL) then
C         Backtrack to preceding level
          LEVEL = LEVEL - 1
          if (LEVEL.lt.1) stop
        else
          INDX(LEVEL) = I
          if (LEVEL.lt.N) then
C           Advance to next level
            LEVEL = LEVEL + 1
            INDX(LEVEL) = I
          end if
        end if
      repeat
      end
```

FORTRAN '77 would have had to do -- but could not have done -- implicitly for INDX in subroutine PICK.

-- Programming example 10-4: program COMBN2 --

10.2 DATA STRUCTURES

The FIBON and COMBN programs demonstrate two varieties of recursion which have markedly different uses. The essential difference between these two varieties lies in the conditions under which either process comes to a halt. The first variety terminates upon acheiving a goal (or terminal) level: in the case of the FIBON programs, a designated Fibonacci term. The second variety of recursion bounces many times off of a goal (terminal) level: in the case of the COMBN programs, the last item in a combination. This second variety does not terminate until the process returns to the original level.

It is appropriate to call the first variety horizontal recursion since most of its applications deal with sequential decisions involving feedback -- Markov chains (Chapter 6) and cumulative feedback (Chapter 7) are elementary examples. Horizontal recursion often penetrates to extremely remote levels

(note 1). A characteristic of horizontal recursion is that as later levels of recursion are encountered, the earlier levels become increasingly less relevant to the process at hand. More elaborate implementations of horizontal recursion than we have seen with the FIBON programs may include capabilities for backtracking (Chapter 14), but this characteristic holds firm nonetheless.

We shall call the second variety vertical recursion because many of its applications deal with hierarchic structures in which the levels of recursion correspond to levels of generality. Here the notion of recursive "depth" retains meaning. Vertical recursion tends to remain fairly close to the original level for the simple reason that the number of potential goals increases exponentially with the number of recursive levels. Unlike with horizontal recursion, the original levels are periodically consulted and updated.

Before we delve any further into recursive processes, we must first gain greater understanding as to how recursive programs are able to distinguish between similar items of information generated at different levels. The requirements of horizontal and vertical recursion differ. Horizontal recursion requires information only pertaining to the most recent levels; prior to a certain point, information may either be discarded or stored in a sequential file. The necessary data structure for

such a process is called a queue. Vertical recursion requires information pertaining to all levels up to and including the present one; memory locations abandoned during a backtrack may be reused when the level advances again. The necessary data structure for vertical recursion is called a stack.

10.2.1 Queues

Queues are useful for deferring tasks which need to be performed in a certain order and for retaining a limited history of past actions. They are sequential lists of information processed on a "first-in-first-out" basis. Illustrative of queues are the roped-in pathways we now commonly find in banks. Whenever a new customer arrives, he takes his position at the tail of the line. Each time a teller becomes available, she services the first customer at the head. The remaining customers then move forward to fill the empty position.

10.2.1.1 Implementation - We can implement a queue in a computer program declaring an array (or perhaps several parallel arrays if

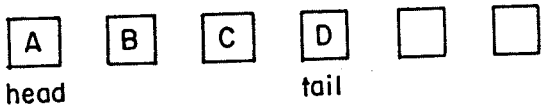
several elements of information pertain to each item) large enough to accomodate all the items with which the computer must conceivably store. Since it is time-consuming to shift each of the items in an array whenever something is extracted, we can implement a circular queue which leaves each item in one position and keeps track of the head and tail by pointers. Figure 10-3 illustrates how these pointers "wrap around" whenever the physical boundary of the array is reached.

Figure 10-3: Implementation of circular queues - Boxed letters depict filled positions in the queue, blank boxes depict empty positions. Insertions draw letters from the "input" stream and place them in the queue; extractions draw letters from the queue and place them in the "output" stream.

The library subroutine PUT inserts single values into a queue, while its counterpart, the library subroutine GET, serves to extract such values. Each utility serves in conjunction with the other, and each requires five arguments:

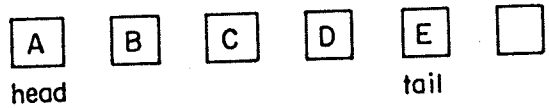
1. ELEMNT - Value to be either placed in queue by PUT or extracted from queue by GET. ELEMNT may be either a real or an integer.

Initial state:



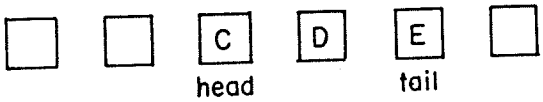
Input: E, F, G, H, I
Output:

Insert one item:



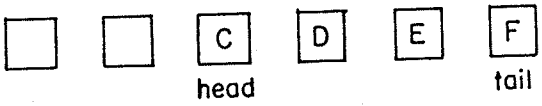
Input: F, G, H, I
Output:

Extract two items:



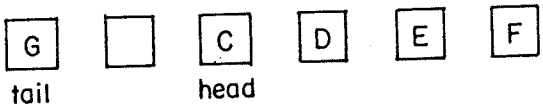
Input: F, G, H, I
Output: A, B

Insert one item:



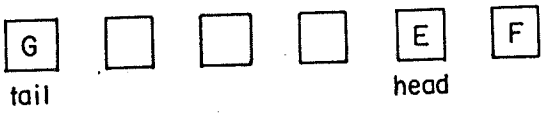
Input: G, H, I
Output: A, B

Insert one item (wrap around):



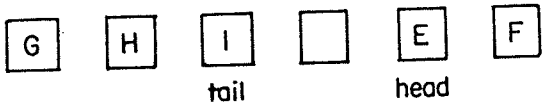
Input: H, I
Output: A, B

Extract two items:



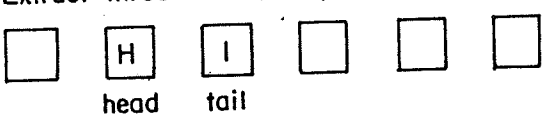
Input: H, I
Output: A, B, C, D

Insert two items:



Input:
Output: A, B, C, D

Extract three items (wrap around):



Input:
Output: A, B, C, D, E, F, G

Fig 16-3

Ex 5-5

```
 1          subroutine PUT(ELEMNT,HEAD,TAIL,QUEUE,ICNT,LENGTH)
 2          dimension QUEUE(1)
 3          integer HEAD,TAIL
 4          ICNT = ICNT + 1
 5          if (ICNT.gt.LENGTH) stop 'Queue overflow in PUT.'
 6          TAIL = TAIL + 1
 7          if (TAIL.gt.LENGTH) TAIL = TAIL - LENGTH
 8          QUEUE(TAIL) = ELEMNT
 9          return
10          end
```

```
 1          subroutine GET(ELEMNT,HEAD,TAIL,QUEUE,ICNT,LENGTH)
 2          dimension QUEUE(1)
 3          integer HEAD,TAIL
 4          ICNT = ICNT - 1
 5          if (ICNT.lt.0) stop 'Queue underflow in GET.'
 6          ELEMNT = QUEUE(HEAD)
 7          HEAD = HEAD + 1
 8          if (HEAD.gt.LENGTH) HEAD = HEAD - LENGTH
 9          return
10          end
```

2. TAIL - Pointer to tail of queue. TAIL must be an integer.

3. HEAD - Pointer to head of queue. HEAD must be an integer.

4. QUEUE - Queue. QUEUE must be an array of dimension LENGTH whose type matches that of ELEMNT.

5. ICNT - Number of elements stored in queue. When ICNT=0 the queue is empty; when ICNT=LENGTH, the queue is full. ICNT must be an integer.

6. LENGTH - Maximum number of positions in queue.

PUT manipulates TAIL; GET manipulates HEAD.

-- Programming example 5-5: subroutines PUT and GET --

10.2.1.2 Application: An 8th-Order Markov Chain - To illustrate the use of queues, we consider the problem of implementing an

8th-order Markov chain of degrees selected from an F major scale. Here, the selection of a degree depends on its eight most immediate predecessors. With such a high order it becomes highly impractical to implement Markov matricies (an 8-dimensional matrix with seven elements per "side" would contain well over five million entries!), so it becomes necessary to compute transition probabilities by formula.

Program DEGCHN implements such a formula. It assigns weights so that the two immediately preceding degrees have no likelihood of selection, while earlier degrees increase in likelihood with their position in the queue. Array SCLSCL gives the repertory of scale degrees required by the library subroutine SELECT, which in this application maps the index ISCL into itself. Array SCLQUE holds the eight most recently used scale degrees; array WGTQUE holds the weights assigned to each degree on the basis of its most recent position in SCLQUE. Each iteration of the main loop (lines 13-38) selects a scale degree (lines 18-33), prints it (line 34), and places it in the queue (lines 36-37; GET serves no purpose in this application beyond making room in the queue for the next item). DEGCHN computes weights of selection by determining the most recent occurance of each degree in the queue (lines 18-23) and storing a weight associated with this position in array WGTSCL (lines 25-31).

Table 10-1 traces the status of the queue and the weightings

Ex 10-6

```
 1            program DEGCHN
 2            parameter (MSCL=7,MQUE=8)
 3            integer SCLSCL(MSCL),SCLQUE(MQUE),HEAD,TAIL
 4            real WGTSCL(MSCL),WGTQUE(MQUE)
 5            data SCLSCL/1,2,3,4,5,6,7/, HEAD/1/, TAIL/1/, ICNT/0/
 6            data WGTQUE/0.,0.,1.,1.64,2.70,4.44,7.30,12.00/
 7      C
 8      C     Fill queue with null scale degrees
 9      C
10            do (MQUE times)
11              call PUT(0,HEAD,TAIL,SCLQUE,ICNT,MQUE)
12            repeat
13      C
14      C     Main loop
15      C
16            do (20 times)
17              SUM = 0.
18              do (ISCL=1,MSCL)
19      C         Locate most recent position of ISCLth scale degree
20                IPTR = TAIL
21                do (IDXQUE=1,MQUE)
22                  if (SCLQUE(IPTR).eq.ISCL) exit
23                  IPTR = IPTR - 1
24                  if (IPTR.le.0) IPTR = IPTR + MQUE
25                repeat
26      C         Store weight associated with this position
27                if (IDXQUE.le.MQUE) then
28                  WGTSCL(ISCL) = WGTQUE(IDXQUE)
29                else
30                  WGTSCL(ISCL) = WGTQUE(MQUE)
31                end if
32                SUM = SUM + WGTSCL(ISCL)
33              repeat
34      C       Choose a scale degree
35              call SELECT(ISCL,SCLSCL,WGTSCL,SUM,MSCL)
36              print *, ISCL
37      C       Place scale degree in queue
38              call GET(NULL,HEAD,TAIL,SCLQUE,ICNT,MQUE)
39              call PUT(ISCL,HEAD,TAIL,SCLQUE,ICNT,MQUE)
40            repeat
41            stop
42            end
```

```
Queue:  --  --  --  --  --  --  --  --
Degrees and weights:   F:12.0   G:12.0   A:12.0  Bb:12.0   C:12.0   D:12.0   E:12.0
Sum: 84.00   RANF: .5686   Scaled value: 47.76  Selected degree: Bb

Queue:  --  --  --  --  --  --  --  Bb
Degrees and weights:   F:12.0   G:12.0   A:12.0  Bb:  .0   C:12.0   D:12.0   E:12.0
Sum: 72.00   RANF: .7238   Scaled value: 52.11  Selected degree:  D

Queue:  --  --  --  --  --  --  Bb  D
Degrees and weights:   F:12.0   G:12.0   A:12.0  Bb:  .0   C:12.0   D:  .0   E:12.0
Sum: 60.00   RANF: .8441   Scaled value: 50.64  Selected degree:  E

Queue:  --  --  --  --  --  Bb  D  E
Degrees and weights:   F:12.0   G:12.0   A:12.0  Bb: 1.0   C:12.0   D:  .0   E:  .0
Sum: 49.00   RANF: .2870   Scaled value: 14.06  Selected degree:  G

Queue:  --  --  --  --  Bb  D  E  G
Degrees and weights:   F:12.0   G:  .0   A:12.0  Bb: 1.6   C:12.0   D: 1.0   E:  .0
Sum: 38.64   RANF: .7358   Scaled value: 28.43  Selected degree:  C

Queue:  --  --  --  Bb  D  E  G  C
Degrees and weights:   F:12.0   G:  .0   A:12.0  Bb: 2.7   C:  .0   D: 1.6   E: 1.0
Sum: 29.34   RANF: .2973   Scaled value:  8.72  Selected degree:  F

Queue:  --  --  Bb  D  E  G  C  F
Degrees and weights:   F:  .0   G: 1.0   A:12.0  Bb: 4.4   C:  .0   D: 2.7   E: 1.6
Sum: 21.78   RANF: .7355   Scaled value: 16.02  Selected degree: Bb

Queue:  --  Bb  D  E  G  C  F  Bb
Degrees and weights:   F:  .0   G: 1.6   A:12.0  Bb:  .0   C: 1.0   D: 4.4   E: 2.7
Sum: 21.78   RANF: .1087   Scaled value:  2.36  Selected degree:  A

Queue:  Bb  D  E  G  C  F  Bb  A
Degrees and weights:   F: 1.0   G: 2.7   A:  .0  Bb:  .0   C: 1.6   D: 7.3   E: 4.4
Sum: 17.08   RANF: .3316   Scaled value:  5.66  Selected degree:  D

Queue:  D  E  G  C  F  Bb  A  D
Degrees and weights:   F: 1.6   G: 4.4   A:  .0  Bb: 1.0   C: 2.7   D:  .0   E: 7.3
Sum: 17.08   RANF: .8636   Scaled value: 14.75  Selected degree:  E

Queue:  E  G  C  F  Bb  A  D  E
Degrees and weights:   F: 2.7   G: 7.3   A: 1.0  Bb: 1.6   C: 4.4   D:  .0   E:  .0
Sum: 17.08   RANF: .3776   Scaled value:  6.45  Selected degree:  G

Queue:  G  C  F  Bb  A  D  E  G
Degrees and weights:   F: 4.4   G:  .0   A: 1.6  Bb: 2.7   C: 7.3   D: 1.0   E:  .0
Sum: 17.08   RANF: .2290   Scaled value:  3.91  Selected degree:  F

Queue:  C  F  Bb  A  D  E  G  F
Degrees and weights:   F:  .0   G:  .0   A: 2.7  Bb: 4.4   C:12.0   D: 1.6   E: 1.0
Sum: 21.78   RANF: .8525   Scaled value: 18.56  Selected degree:  C

Queue:  F  Bb  A  D  E  G  F  C
Degrees and weights:   F:  .0   G: 1.0   A: 4.4  Bb: 7.3   C:  .0   D: 2.7   E: 1.6
Sum: 17.08   RANF: .7111   Scaled value: 12.14  Selected degree: Bb

Queue:  Bb  A  D  E  G  F  C  Bb
Degrees and weights:   F: 1.0   G: 1.6   A: 7.3  Bb:  .0   C:  .0   D: 4.4   E: 2.7
Sum: 17.08   RANF: .6358   Scaled value: 10.86  Selected degree:  D

Queue:  A  D  E  G  F  C  Bb  D
Degrees and weights:   F: 1.6   G: 2.7   A:12.0  Bb:  .0   C: 1.0   D:  .0   E: 4.4
Sum: 21.78   RANF: .6726   Scaled value: 14.65  Selected degree:  A

Queue:  D  E  G  F  C  Bb  D  A
Degrees and weights:   F: 2.7   G: 4.4   A:  .0  Bb: 1.0   C: 1.6   D:  .0   E: 7.3
Sum: 17.08   RANF: .9238   Scaled value: 15.77  Selected degree:  E

Queue:  E  G  F  C  Bb  D  A  E
Degrees and weights:   F: 4.4   G: 7.3   A:  .0  Bb: 1.6   C: 2.7   D: 1.0   E:  .0
Sum: 17.08   RANF: .1789   Scaled value:  3.05  Selected degree:  F

Queue:  G  F  C  Bb  D  A  E  F
Degrees and weights:   F:  .0   G:12.0   A: 1.0  Bb: 2.7   C: 4.4   D: 1.6   E:  .0
Sum: 21.78   RANF: .9108   Scaled value: 19.83  Selected degree:  C

Queue:  F  C  Bb  D  A  E  F  C
Degrees and weights:   F:  .0   G:12.0   A: 1.6  Bb: 4.4   C:  .0   D: 2.7   E: 1.0
Sum: 21.78   RANF: .7963   Scaled value: 17.34  Selected degree: Bb
```

Table 10-1

Fig. 10-4

of the seven scale degrees as DEGCHN composes a chain of degrees. The musical result appears in Figure 10-4.

-- Programming example 10-6: program DEGCHN --

-- Table 10-1 --

Figure 10-4: 8th-order Markov chain of degrees - The results of the 8th-order Markov chain traced in Table 10-1, transcribed into musical notation.

10.2.2 Stacks

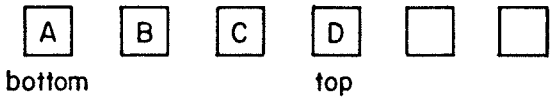
Stacks are useful for deferring portions of tasks which must be resumed at a later time. They are sequential lists of information processed on a "last-in-first-out" basis. Illustrative of stacks are the spring-loaded dish dispensers we often encounter in cafeterias. Each time a busboy inserts a clean plate into the dispenser, the spring contracts so that the topmost position remains at a constant level. When a patron removes a plate, the spring lifts another into position.

10.2.2.1 Implementation - A programmer implements a stack in FORTRAN by declaring an array (or perhaps several parallel arrays if several elements of information pertain to each item) large enough to accomodate all the items the computer could conceivably need to store at once. The element holding the earliest item is called the "bottom" of the stack, while the element holding the most recently inserted item is the "top". Among computer programmers it is conventional jargon to "push" information onto a stack and "pop" information from a stack. As with the implementation of the queue, wholesale shifting of items may be avoided by using a pointer to keep track of the top (the bottom remains fixed at location 1). Array INDX in program COMBN2 serves as a stack whose top (the variable LEVEL) is manipulated explicitly by the main program. Figure 10-5 illustrates the mechanics of stack operations.

Figure 10-5 Implementation of stacks - Boxed letters depict filled positions in the stack, blank boxes depict empty positions. Insertions draw letters from the "input" stream and place them in the stack; extractions draw letters from the stack and place them in the "output" stream.

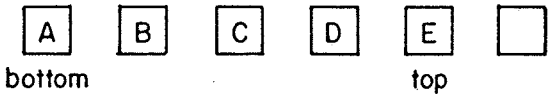
In many cases it is useful to implement general

Initial state:



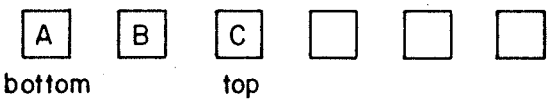
Input: E, F, G, H, I
Output:

Insert one item:



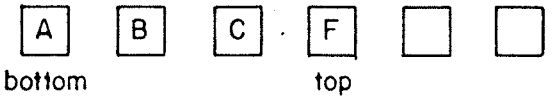
Input: F, G, H, I
Output:

Extract two items:



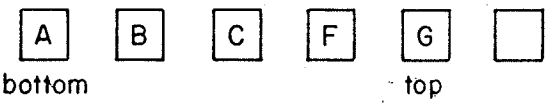
Input: F, G, H, I
Output: E, D

Insert one item:



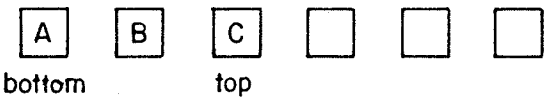
Input: G, H, I
Output: E, D

Insert one item:



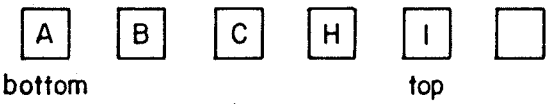
Input: H, I
Output: E, D

Extract two items:



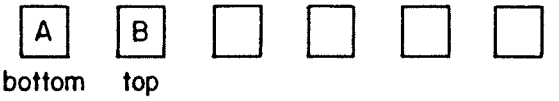
Input: H, I
Output: E, D, G, F

Insert two items:



Input:
Output: E, D, G, F

Extract three items:



Input:
Output: E, D, G, F, I, H, C

Fig. 10-5

stack-handling utilities such as the library subroutines PUSH and POP, given below: PUSH places a single value onto a stack while its counterpart, POP, extracts such values. Like GET and PUT, these subroutines serve in conjunction with one another. They require the following arguments:

1. ELEMNT - Value to be either placed on stack by PUSH or extracted from stack by POP. ELEMNT may be either a real or an integer.

2. TOP - Pointer to top of stack. TOP must be an integer.

3. STACK - stack. STACK must be an array of dimension LENGTH whose type matches that of ELEMNT.

4. LENGTH (required by subroutine PUSH only) - Maximum number of positions in stack.

When TOP=0, the stack is empty. When TOP=LENGTH, the stack is full.

-- Programming example 10-7: subroutines PUSH and POP --

Ex 10-7

```
1           subroutine PUSH(ELEMNT,TOP,STACK,LENGTH)
2           dimension STACK(1)
3           integer TOP
4           if (TOP.ge.LENGTH) stop 'Stack overflow in PUSH.'
5           TOP = TOP + 1
6           STACK(TOP) = ELEMNT
7           return
8           end
```

```
1           subroutine POP(ELEMNT,TOP,STACK)
2           dimension STACK(1)
3           integer TOP
4           if (TOP.le.0) stop 'Stack underflow in POP.'
5           ELEMNT = STACK(TOP)
6           TOP = TOP - 1
7           return
8           end
```

10.2.2.2 Application: Partition/Exchange Sorting - One of the more significant applications of stack-oriented recursion is the partition/exchange sort or quicksort. This algorithm for sorting was developed by C.A.R. Hoare (1962), drawing on work by R. Sedgewick; Knuth describes it at length in Sorting and Searching (1973). While less intuitive than the insertion sort discussed in Chapter 9 (Heading 9.2), partition/exchange sorting has great advantages for sorting large collections of items whose keys are highly disordered. Given a well disordered collection with N records, the average time required by a partition/exchange sort for a disordered collection increases with

$$N \log_e N,$$

while the average time required by an insertion sort increases with

$$N^2.$$

However, insertion sorts ~~retain an advantage~~ excel over partition/exchange sorts when collections are small or when the order of items is close to sequential.

Figure 10-6 illustrates a partition/exchange sort in action.

The basic procedure involves taking a reference item with key K (in this case, the leftmost item) and partitioning the collection of items into two sub-collections: one containing only items with keys not larger than K, the other containing only items with keys not smaller than K. Once the collection has been partitioned, it then becomes necessary to apply this procedure recursively to its own results: each of the two sub-collections derived before is regarded as a full-fledged collection in its own right and itself partitioned into sub-collections. Continuing in this manner yields sub-collections with progressively fewer items, until finally no sub-collection contains more than one item and the sort is complete.

Figure 10-6: Mechanics of partition/exchange sorting - Each section of this figure (partition numbers 1-10) illustrates acts of partitioning as applied to an array of 15 numbers and to various sub-arrays. The two rows of elements depict the state of the array before and after partitioning. Double-headed arrows indicate the sequence of exchanges. Bold squares indicate the reference keys. Numbers designated in stack operations refer to the 15 array positions indicated at the top of the figure.

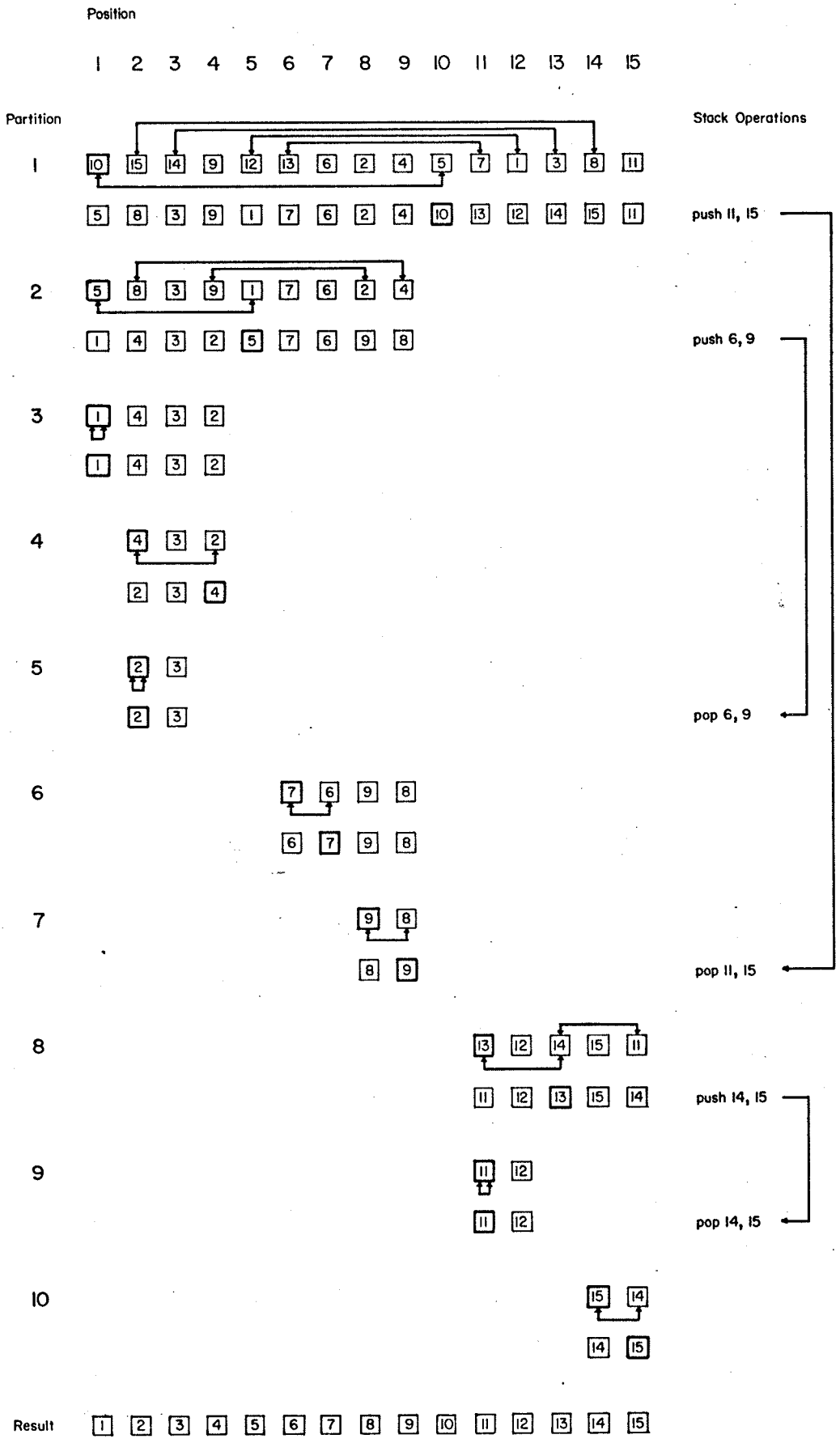

Position
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Partition
Stock Operations
1
10 15 14 9 12 13 6 2 4 5 7 1 3 8 11
5 8 3 9 1 7 6 2 4 10 13 12 14 15 11
push 11, 15
2
5 8 3 9 1 7 6 2 4
1 4 3 2 5 7 6 9 8
push 6, 9
3
1 4 3 2
1 4 3 2
4
4 3 2
2 3 4
5
2 3
2 3
pop 6, 9
6
7 6 9 8
6 7 9 8
7
9 8
8 9
pop 11, 15
8
13 12 14 15 11
11 12 13 15 14
push 14, 15
9
11 12
11 12
pop 14, 15
10
15 14
14 15
Result
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Fig 10-6

The act of partitioning a collection into sub-collections involves setting up two pointers which migrate from the right and left extremes inward. Whenever the right pointer encounters an item belonging in the left sub-collection while the left pointer encounters items belonging in the right sub-collection, such items are exchanged. The meeting place of the two pointers provides the proper position of the reference item.

The library subroutine ISORTQ implements a partition/exchange sort given three arguments:

1. SCHED - Schedule of items. SCHED must be an integer array of dimension NUM containing pointers to each item.

2. KEY - Array of keys, one for each item in the collection. KEY must be an integer array of dimension NUM.

3. GRAIN - Size of the smallest sub-collection to be partitioned. GRAIN must be an integer.

4. STACK - Workspace for a stack, to be used temporarily by ISORTQ. STACK must be an integer array of dimension MSTK in the calling program.

5. MSTK - Dimension of array STACK. MSTK must be an integer.

6. NUM - Number of items in the collection.

A rule of thumb for determining MSTK is:

$$MSTK = 7 + 3\log_e N$$

Where N is the largest number of items contained in any collection to be sorted, that is, the largest possible value of NUM.

Setting GRAIN to 2 insures a complete sort. However, because the partition/exchange sort is relatively inefficient with small sub-collections, it pays to partition sub-collections no smaller than, say, 7 items and to use a straight insertion sort to clean up the last stages. Knuth recommends this practice in his discussion of the algorithm, where he also suggests alternate ways selecting the reference item in order to obtain certain advantages over using the leftmost item by rote.

-- Programming example 10-8: subroutine ISORTQ --

Ex 10.8

```
1           subroutine ISORTQ(SCHED,KEY,STACK,MSTK,GRAIN,NUM)
2           integer SCHED(1),KEY(1),STACK(1),MSTK,GRAIN,NUM,
3          :        K,LEFT,L,RIGHT,R,TOP
4       C
5           if (NUM.lt.GRAIN) return
6           TOP = 0
7           LEFT = 1
8           RIGHT = NUM
9           do
10      C     Partition items LEFT through RIGHT relative to item LEFT
11            K = KEY(SCHED(LEFT))
12            L = LEFT + 1
13            R = RIGHT
14            do
15      C       Decrement R to next item belonging left of item LEFT
16              do
17                if (KEY(SCHED(R)).lt.K) exit
18                R = R - 1
19                if (R.lt.L) go to 47
20              repeat
21      C       Increment L to next item belonging right of item LEFT
22              do
23                if (KEY(SCHED(L)).gt.K) exit
24                L = L + 1
25                if (R.le.L) go to 47
26              repeat
27      C       Exchange items L and R
28              I = SCHED(L)
29              SCHED(L) = SCHED(R)
30              SCHED(R) = I
31            repeat
32      C     Exchange items LEFT and R
33         47 I = SCHED(LEFT)
34            SCHED(LEFT) = SCHED(R)
35            SCHED(R) = I
36      C     Select next subcollection for partitioning
37            if (R-LEFT.ge.GRAIN) then
38              if (RIGHT-R.ge.GRAIN) then
39                call PUSH(R+1,TOP,STACK,MSTK)
40                call PUSH(RIGHT,TOP,STACK,MSTK)
41              end if
42              RIGHT = R - 1
43            else
44              if (RIGHT-R.ge.GRAIN) then
45                LEFT = R + 1
46              else
47                if (TOP.le.0) return
48                call POP(RIGHT,TOP,STACK)
49                call POP(LEFT,TOP,STACK)
50              end if
51            end if
52          repeat
53          end
```

10.2.3 Deques

A deque generalizes a stack and a queue in that with a deque, items of information may be both inserted at either end and extracted from either end. Such structures are implemented as arrays using two pointers, wrapping around in the manner of the circular queue. However, a deque requires utilities analogous to PUSH and POP for each of these two pointers.

10.3 NOTES

1. Savants often speak loosely of the "depth" of recursion in this context, rather than "remoteness"; however, when the number of levels reaches up into the hundreds or thousands, the analogy to physical "depth" ceases to have much meaning.