STATISTICS WITH R PROGRAMMING

UNIT-2

R Programming Structures :

Control Statements - Loops

Looping Over Nonvector Sets

If-Else

Arithmetic and Boolean Operators and values

Default Values for Argument

A. Return Values

B. Deciding Whether to explicitly call return

C. Returning Complex Objects

D. Functions are Objective,

No Pointers in R

Recursion –Example -1: A Quicksort Implementation

Example -2: ABinary Search Tree.

R PROGRAMMING STRUCTURES : CONTROL STATEMENTS

Control statements in R look very similar to C- Language.

A. LOOPS

Looping is the process of executing the statement(s) repeatedly until the condition is satisfy. Every looping statement has four features. They are

i. Initialization
ii. Condition
iii. Statement(s)
iv. Increment or Decrement

Every loop has group of iterations. Each iteration has condition, statement(s) and Increment and decrement.

R- Language provides three looping statements. They are

i. for statement
ii. while statement
iii. repeat statement

i. for statement

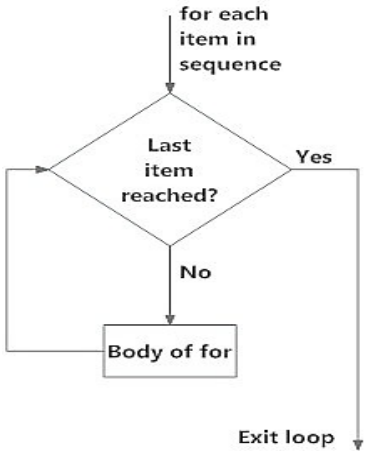
A for loop is used to iterate over a vector in R programming.

Syntax

```
for(valin sequence)
{
statement
}
```

Here, `sequence` is a vector and `val` takes on each of its value during the loop. In each iteration, `statement` is evaluated.

Flowchart of for loop



Example-1:

```
# printing numbers from 1 to 10
for( iin 1:10 ) print(i)
```

```
[1]  1   2   3   4   5   6   7   8   9   10
```

Example-2 :

```
# Finding squares for given vector
> x <- c(5,12,13)
>for (n in x) print(n^2)
[1] 25
[1] 144
[1] 169
```

ii. while statement

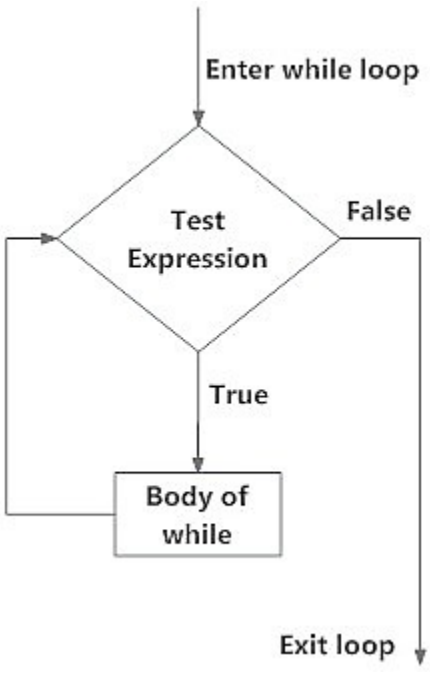
In R programming, while loops are used to loop until a specific condition is met.

Syntax:

```
Inititalization
while (condition) {

Statement(s)
Increment/ Decrement
}
```

Flowchart of while Loop



Example-1:

```
# printing numbers from 1 to 10
i<- 1
while(i<=10){
print(i)
i<-i+1
}

Output
[1] 1     2     3     4     5     6     7     8     9     10
```

Example-2 :

```
# Finding squares for given vector
x <- c(5,12,13)
i<-1
while(i<=length(x)){
     print(x[i]^2)
     i<-i+1
}


Output

[1] 25
[1] 144
[1] 169
```

iii. repeat statement

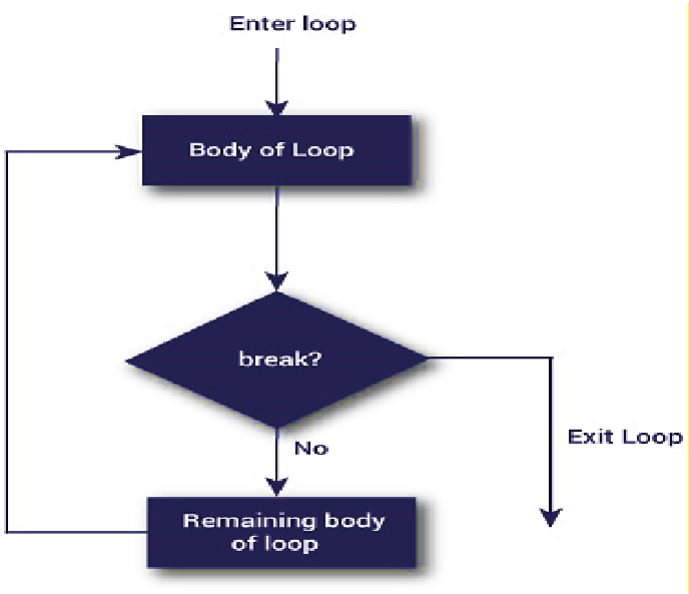
A repeat loop is used to iterate over a block of code multiple number of times. There is no condition check in repeat loop to exit the loop. We must ourselves put a condition explicitly inside the body of the loop and use the break statement to exit the loop. Failing to do so will result into an infinite loop.

Syntax:

Inititalization
repeat {

if(condition) break
Statement(s)
Increment/ Decrement
}

Flowchart of repeat loop



Example-1:

```
# printing numbers from 1 to 10
i<- 1
repeat{
if(i>10) break
print(i)
i<-i+1
}

Output
[1] 1     2     3     4     5     6     7     8     9     10
```

Example-2 :

```
# Finding squares for given vector
x<-c(5,12,13)
i<-1
repeat{
     if(i>length(x)) break
     print(x[i]^2)
     i<-i+1
}

Output

[1] 25
[1] 144
[1] 169
```

In addition to the Looping, R language provides two statement for looping. They are

i. break statement
ii. next statement

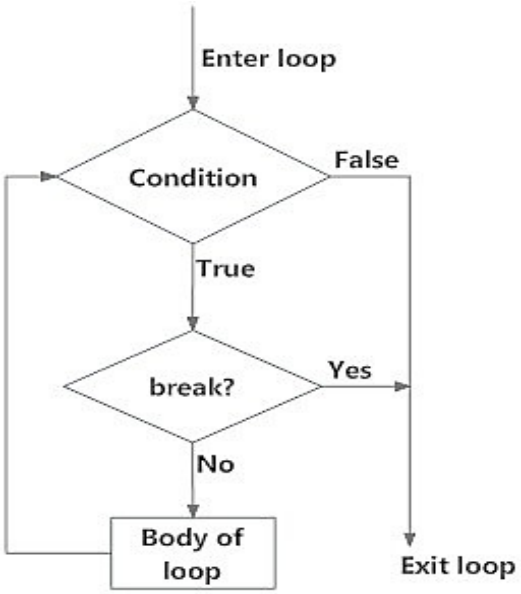
i. break statement

A `break` statement is used inside a loop (repeat, for, while) to stop the iterations and flow the control outside of the loop. In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

Syntax

```
break
```

Flowchart of break statement



Example:

```
x <-1:5
for(valin x){
if(val==3){
break
}
print(val)
}
Output
[1] 1
[2] 2
```

In this example, we iterate over the vector `x`, which has consecutive numbers from 1 to 5. Inside the for loop we have used a if condition to break if the current value is equal to 3. As we can see from the output, the loop terminates when it encounters the `break` statement.

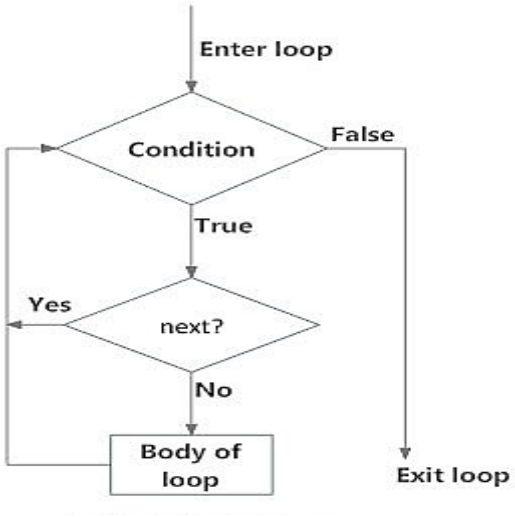
ii. next statement

A next statement is useful when we want to skip the current iteration of a loop without terminating it. On encountering `next`, the R parser skips further evaluation and starts next iteration of the loop.

Syntax

```
next
```

Flowchart of next statement



Example:

```
x <- 1:5
for (val in x) {
if (val == 3){
next
    }
print(val)
}
```

Output

```
[1] 1
[1] 2
[1] 4
[1] 5
```

In the above example, we use the `next` statement inside a condition to check if the value is equal to 3. If the value is equal to 3, the current evaluation stops (value is not printed) but the loop continues with the next iteration.

2.1.B: Looping Over Non-vector Sets : lapply() function

R does not directly support iteration over nonvector sets, but there are a couple of indirect easy ways to accomplish it:

• Use lapply(), assuming that the iterations of the loop are independent of each other, thus allowing them to be performed in any order.

Example:

```
a<-c(1:10)

b<-c(-5:4)

x<-list(a,b)

# compute the list mean for each list element

mean1<-lapply(x,mean)

print(mean1)
```

output

```
[[1]]
[1] 5.5

[[2]]
[1] -0.5
```

if-else Statement

An **if** statement can be followed by an optional **else** statement which executes when the boolean expression is false.

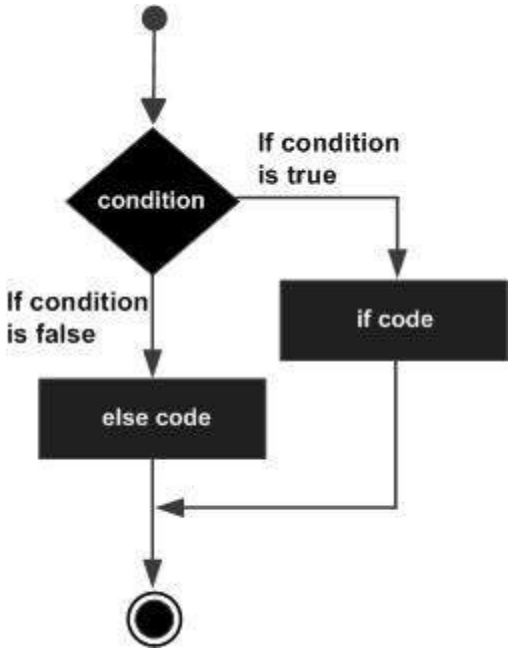
Syntax

The basic syntax for creating an **if...else** statement in R is −

```
if(boolean_expression) {
   // statement(s) will execute if the boolean expression is true.
} else {
   // statement(s) will execute if the boolean expression is false.
}
```

If the Boolean expression evaluates to be **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Flow Diagram



Example

```
x <-10

if(x<=10){
print(100)
}else{
print(200)
}
```

Output

```
100
```

Here x is equal to 10, the condition x<=10 is true, so if statement is printing 100.

```
x <- 10

if(x>10){
print(100)
}else{
print(200)
}
```

Output

```
200
```

Here x is equal to 10, the condition x>10 is false so else block will be executed and it prints 200 on the screen.

Arithmetic and Boolean Operators and Values

Operation	Description
x + y	Addition
x – y	Subtraction
x * y	Multiplication
x / y	Division
x ^ y	Exponentiation
x %% y	Modular arithmetic
x %/% y	Integer division
x == y	Test for equality
x <= y	Test for less than or equal to
x >= y	Test for greater than or equal to
x && y	Boolean AND for scalars
x || y	Boolean OR for scalars
x & y	Boolean AND for vectors (vector x,y,result)
x | y	Boolean OR for vectors (vector x,y,result)
!x	Boolean negation

Examples:

```
>x<-c(TRUE, FALSE,TRUE)
>x
[1] TRUE FALSE TRUE
>y<-c(TRUE, TRUE, FALSE)
>y
[1] TRUE TRUE FALSE
>x& y
[1] TRUE FALSE FALSE


>x[1] && y[1]
[1] TRUE
>x&& y # looks at just the first elements of each vector
[1] TRUE
>if (x[1] && y[1]) print("both TRUE")
[1] "both TRUE"
>if (x & y) print("both TRUE")
[1] "both TRUE"
Warning message:
In if (x & y) print("both TRUE") :
The condition has length > 1 and only the first element will
be used
```

Here &(AND) will be applied on “first element in x” and “first element in y”. For that it prints the warning message on the screen.

To avoid the warning message just use && on x and y

The Boolean values “TRUE can be represented as T” , similarly “FALSE can be represented as F” (both must be capitalized). These values change to 1 and 0 in arithmeticexpressions:

```
> T&T
[1] TRUE
> T&F
[1] FALSE
> F&F
[1] FALSE
> T*5
[1] 5
> F*5
[1] 0
```

We can apply the relational operator on numerical values.

```
>1 < 2
[1] TRUE
> (1 < 2) * (3 < 4)
[1] 1
```

Here the comparison 1 < 2 returns TRUE, and 3 < 4 yields TRUE as well. Both values are treated as 1 values, so the product is 1(1 * 1 = 1)

```
> (1 < 2) * (3 < 4) * (5 < 1)
[1] 0
> (1 < 2) == TRUE
[1] TRUE
> (1 < 2) == 1
[1] TRUE
```

Default Values for Arguments

We can define the value of the arguments in the function definition and **call the function without supplying any argument to get the default result**.

Example -1:

```
# Create a function with arguments.
new.function<- function(a = 3, b = 6) {
result<- a * b
print(result)
}

# Call the function without giving any argument.
new.function()
```

output

```
18
```

Example -2:

Here default values (a=3 and b=6) will be passed to new.function

```
# Call the function with giving new value of the argument.
new.function(9)
```
output

```
54
```
Here default value (b=6) will be passed to new.function because b is not specified.

Example -3:

```
# Call the function with giving new values of the argument.
new.function(9,5)
```

output

```
54
```
Here no default values will be passed to new.function because a and b are specified.

2.5. A. Return Values

Function is a block statements which performs specific task. Function can take the values through the formal arguments and may return the value if is required.

The return value of a function can be numeric, character, date, Boolean, vector, data frame, matrix, list or an array.

Function return the value through the return() function.

Syntax

return(value)

return() in R is optional. We can return the value by specifying the variable name.

Example:

```
Oddcount<- function(x) {
k <- 0 # assign 0 to k
for (n in x) {
if (n %% 2 == 1) k <- k+1 # %% is the modulo operator
}
return(k) # function returns the value k with return()
}
```

Output

```
>oddcount(1:10)
5
```

Here the function explicitly using the return() in oddcount.

The following R- Code return the value by specifying the variable name

```
Oddcount<- function(x) {
k <- 0 # assign 0 to k
for (n in x) {
if (n %% 2 == 1) k <- k+1  # %% is the modulo operator
}
k   # function returns the value k without return()
}

Output
>oddcount(1:10)
5
```

So using return() in the function is optional. Based on the program complexity we can use return() explicitly.

2.5.B. Deciding Whether to Explicitly Call return()

Using return() explicitly in the function is based on the program size and complexity. If the program size is short then no need use return() explicitly. For example.

```
Oddcount<- function(x) {
k <- 0 # assign 0 to k
for (n in x) {
if (n %% 2 == 1) k <- k+1  # %% is the modulo operator
}
k   # function returns the value k without return()
}

Output
>oddcount(1:10)
5
```

In the above example. It is simple to return the value. A call to return() wasn't necessary

For Good software design, it is better to use the return() explicitly for complex programs.

Returning Complex Objects

We know that R-function return any object like numerical, character, date, vector, list etc. A function can return the complex object like function.

Here is an example of a function being returned:

```
g<- function() {
t <- function(x) return(x^2) # Here t is a function like g
return(t)
}
```

Output

```
>g()
function(x) return(x^2)
<environment: 0x8aafbc0>
```

Here tis a function because t is created with function() t<-function(x)
After that the function g is returning t. That means g is returning functions.

Functions are Objects

R functions are *first-class objects* that means they can be used for the most part just like other objects. Consider the following example

```
 g <- function(x) {
return(x+1)
 }
```

Here, function() is a built-in R function whose job is to create functions.
Every function() has two arguments they are

1. The formal argument list- here it is x
2. The function body – here { return(x+1) }

We can access these two arguments can be accessed through formal() and body() functions

formals() function returns the formal arguments of given function.
Example:
>formals(g)
x

body() function returns the body of given function
Example:
>body(g)
{
return(x+1)
}

Function assignment

Since functions are objects, we can also assign them, use them as argumentsto other functions, and so on.

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)

> f <- f1  #f1 is assign to f
```

```
>f(3,2)
[1] 5

> f <- f2  #f2 is assign to f
>f(3,2)
[1] 1
```

Passing function as an argument

Since functions are objects, we can also pass them as an argument.
Example:

```
> g <- function(h,a,b) h(a,b)
>g(f1,3,2)
[1] 5
>g(f2,3,2)
[1] 1
```

Here g is a function which takes three arguments, they are

1. h – is a function which is taking two arguments as h(a,b)
2. a- is a numerical
3. b- is a numerical

When we pass f1 function to g, immediately f1(a,b) will be called and the result a+b (3+2=5) is printed on the screen

: No Pointers in R

R does not have variables corresponding to *pointers* or *references* like those of, say, the C language. This can make programming more difficult in some cases. (As of this writing, the current version of R has an experimental feature called *reference classes*)

Example-1:

```
> x <- c(13,5,12)
>sort(x)
[1] 5 12 13
>x
[1] 13 5 12
```

The function sort() does not change x. If we do want x to change in this R code, the solution is to reassign the arguments:

```
> x <- sort(x)
>x
[1] 5 12 13
```

Here R uses the reference classes to store the address of x and performing the sort on x and reassign values to x.

Example-2:

An example is the following function, which determines the indices ofodd and even numbers in a vector of integers:

```
Odds_evens<-function(v){
odds<- which(v %% 2 == 1)
```

```
evens<- which(v %% 2 == 1)
list(o=odds,e=evens)
}
```

Output

```
>x<-c(1:11)
>odds_evens(x)
$o
6
$e
5
```

```
Here odd numbers are 6 (1,3,5,7,9,11) and even numbers are 5
(2,4,6,8,10)
```

Recursion

A *recursive* function calls itself. To solve a problem of type X by writing a recursive function f():

1. Break the original problem of type X into one or more smaller problems of type X.
2. Within f(), call f() on each of the smaller problems.
3. Within f(), piece together the results of (b) to solve the original problem.

2.7.A : Example - A Quicksort Implementation

A classic example is Quickso1rt, an algorithm used to sort a vector of numbers from smallest to largest. For instance, suppose we wish to sort the vector(5,4,12,13,3,8,88).

We first compare everything to the first element, 5 (pivot element) ,to form two sub vectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us sub vectors (4,3) and (12,13,8,88). We then call the function on the sub vectors, returning (3,4) and (8,12,13,88). We string those together with the 5, yielding (3,4,5,8,12,13,88), as desired.

R's vector-filtering capability and its c() function make implementation of Quick sort quite easy.
This example is for the purpose of demonstrating recursion. R's own sort function, sort(), is much faster, as it is written in C.

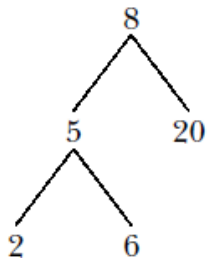
```
qs<- function(x) {
if (length(x) <= 1) return(x)
pivot<- x[1]
therest<- x[-1]
left<- therest[therest< pivot]
right<- therest[therest>= pivot]
left<- qs(left)
right<- qs(right)
return(c(left,pivot,right))
}
```

Output

```
>x<-c(5,4,12,13,3,8,88)
>qs(x)
[1] 3     4     5     8     12    13    88
```

2.7.B: Extended Example: A Binary Search Tree

Consider the following



R-code for Binary search tree is as follows

```
1 # routines to create trees and insert items into them are included
2 # below; a deletion routine is left to the reader as an exercise
3
4 # storage is in a matrix, say m, one row per node of the tree; if row
5 # i contains (u,v,w), then node i stores the value w, and has left and
6 # right links to rows u and v; null links have the value NA
7
8 # the tree is represented as a list (mat,nxt,inc), where mat is the
9 # matrix, nxt is the next empty row to be used, and inc is the number of
10 # rows of expansion to be allocated whenever the matrix becomes full
11
12 # print sorted tree via in-order traversal
13 printtree<- function(hdidx,tr) {
14 left <- tr$mat[hdidx,1]
15 if (!is.na(left)) printtree(left,tr)
16 print(tr$mat[hdidx,3]) # print root
17 right <- tr$mat[hdidx,2]
18 if (!is.na(right)) printtree(right,tr)
19 }
20
21 # initializes a storage matrix, with initial stored value firstval
22 newtree<- function(firstval,inc) {
23 m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3)
24 m[1,3] <- firstval
25 return(list(mat=m,nxt=2,inc=inc))
26 }
27
28 # inserts newval into the subtree of tr, with the subtree's root being
29 # at index hdidx; note that return value must be reassigned to tr by the
30 # caller (including ins() itself, due to recursion)
31 ins <- function(hdidx,tr,newval) {
32 # which direction will this new node go, left or right?
33 dir<- if (newval<= tr$mat[hdidx,3]) 1 else 2
34 # if null link in that direction, place the new node here, otherwise
35 # recurse
```

```
36 if (is.na(tr$mat[hdidx,dir])) {
37 newidx<- tr$nxt # where new node goes
38 # check for room to add a new element
39 if (tr$nxt == nrow(tr$mat) + 1) {
40 tr$mat<-
41 rbind(tr$mat, matrix(rep(NA,tr$inc*3),nrow=tr$inc,ncol=3))
42 }
43 # insert new tree node
44 tr$mat[newidx,3] <- newval
45 # link to the new node
46 tr$mat[hdidx,dir] <- newidx
47 tr$nxt<- tr$nxt + 1 # ready for next insert
48 return(tr)
49 } else tr<- ins(tr$mat[hdidx,dir],tr,newval)
50 }
```

Output

```
#tree building using its routines
> x <- newtree(8,3)
>x
$mat
[,1] [,2] [,3]
[1,] NA NA 8
[2,] NA NANA
[3,] NA NANA
$nxt
[1] 2
$inc
[1] 3

> x <- ins(1,x,5)
>x
$mat
 [,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA NA 5
[3,] NA NANA
$nxt
[1] 3
$inc
[1] 3

> x <- ins(1,x,6)
>x
$mat
[,1] [,2] [,3]
[1,] 2 NA 8
```

```
[2,] NA 3 5
[3,] NA NA 6
$nxt
[1] 4
$inc
[1] 3

> x <- ins(1,x,2)
>x
$mat
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NANA
[6,] NA NANA
$nxt
[1] 5
$inc
[1] 3

> x <- ins(1,x,20)
>x
$mat
[,1] [,2] [,3]
 [1,] 2 5 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NA 20
[6,] NA NANA
$nxt
[1] 6
$inc
[1] 3
```

Execution Explanation

First, the command containing our call newtree(8,3) creates a new tree, assigned to x, storing the number 8. The argument 3 specifies that we allocate storage room three rows at a time.

The result is that the matrix component of the list x is now as follows:

```
[,1] [,2] [,3]
[1,] NA NA 8
[2,] NA NANA
[3,] NA NANA
```

Three rows of storage are indeed allocated, and our data now consists just of the number 8. The two NA values in that first row indicate that this node of the tree currently has no children.

We then make the call ins(1,x,5) to insert a second value, 5, into thetree x. The argument 1 specifies the root. In other words, the call says,"Insert 5 in the subtree of x whose root is in row 1." Note that we need toreassign the return value of this call back to x. Again, this is due to the lackof pointer variables in R. The matrix now looks like this:

```
[,1] [,2] [,3]
[1,] 2 NA 8
[2,] NA NA 5
[3,] NA NANA
```

The element 2 means that the left link out of the node containing 8 ismeant to point to row 2, where our new element 5 is stored.
The session continues in this manner. Note that when our initial allotmentof three rows is full, ins() allocates three new rows, for a total of six. Inthe end, the matrix is as follows:

```
[,1] [,2] [,3]
[1,] 2 5 8
[2,] 4 3 5
[3,] NA NA 6
[4,] NA NA 2
[5,] NA NA 20
[6,] NA NANA
This represents the tree we graphed for this example.
```

Program Explanation

There is recursion in both printtree() and ins(). The former is definitelythe easier of the two, so let's look at that first. It prints out the tree, in sortedorder.

Recall our description of a recursive function f() that solves a problemof category X:
We have f() split the original X problem into one or moresmaller X problems, call f() on them, and combine the results.

In this case,our problem's category X is to print a tree, which could be a subtree of alarger one. The role of the function on line 13 is to print the given tree,which it does by calling itself in lines 15 and 18. There, it prints first the leftsubtree and then the right subtree, pausing in between to print the root.

This thinking—print the left subtree, then the root, then the rightsubtree—forms the intuition in writing the code, but again we must makesure to have a proper termination mechanism. This mechanism is seen inthe if() statements in lines 15 and 18.

When we come to a null link, we do not continue to recursion.The recursion in ins() follows the same principles but is considerablymore delicate. Here, our "category X" is an insertion of a value into a subtree.

We start at the root of a tree, determine whether our new value mustgo into the left or right subtree (line 33), and then call the function againon that subtree. Again, this is not hard in principle, but a number of detailsmust be attended to, including the expansion of the matrix if we run out ofroom (lines 40–41).

One difference between the recursive code in printtree() and ins() isthat the former includes two calls to itself, while the latter has only one. Thisimplies that it may not be difficult to write the latter in a nonrecursive form.