```
Natchez, Mississippi
New Orleans, Louisiana
Gulf of Mexico

DELETE Vicksburg from between Memphis and Natchez
==================================================

Cities passed going upstream ...
Gulf of Mexico
New Orleans, Louisiana
Natchez, Mississippi
Memphis, Tennesee
Osceola, Arkansas
Wickliffe, Kentucky
St. Louis, Missouri
Hannibal, Missouri
Rock Island, Illinois
Dubuque, Iowa
La Crosse, Wisconsin
Minneapolis, Minnesota
Lake Itaska, Minnesota

DEALLOCATE the linked list
==========================

Cities deallocated going downstream ...
Lake Itaska, Minnesota
Minneapolis, Minnesota
La Crosse, Wisconsin
Dubuque, Iowa
Rock Island, Illinois
Hannibal, Missouri
St. Louis, Missouri
Wickliffe, Kentucky
Osceola, Arkansas
Memphis, Tennesee
Natchez, Mississippi
New Orleans, Louisiana
Gulf of Mexico
```

```
La Crosse, Wisconsin
Davenport, Iowa
Rock Island, Illinois
St. Louis, Missouri
Wickliffe, Kentucky
Osceola, Arkansas
Memphis, Tennesee
Vicksburg, Mississippi
Natchez, Mississippi
New Orleans, Louisiana
Gulf of Mexico

REPLACE Davenport with Dubuque
==============================

Cities passed going upstream ...
Gulf of Mexico
New Orleans, Louisiana
Natchez, Mississippi
Vicksburg, Mississippi
Memphis, Tennesee
Osceola, Arkansas
Wickliffe, Kentucky
St. Louis, Missouri
Rock Island, Illinois
Dubuque, Iowa
La Crosse, Wisconsin
Minneapolis, Minnesota
Lake Itaska, Minnesota

INSERT Hannibal between Rock Island and St. Louis
================================================

Cities passed going downstream ...
Lake Itaska, Minnesota
Minneapolis, Minnesota
La Crosse, Wisconsin
Dubuque, Iowa
Rock Island, Illinois
Hannibal, Missouri
St. Louis, Missouri
Wickliffe, Kentucky
Osceola, Arkansas
Memphis, Tennesee
Vicksburg, Mississippi
```

```
!
!
!                                   Show the upstream route.
subroutine uroute
  use city_list
  print *, " "
  print *, " Cities passed going upstream ..."
  travel => finish
  do
    print *, " ", travel%city
    if ( .not. associated ( travel%upstream ) ) exit
    travel => travel%upstream
  end do
end subroutine uroute


OSCA$ f90 tour.f90


OSCA$ a.out
Lake Itaska, Minnesota
Minneapolis, Minnesota
La Crosse, Wisconsin
Davenport, Iowa
Rock Island, Illinois
St. Louis, Missouri
Wickliffe, Kentucky
Osceola, Arkansas
Memphis, Tennesee
Vicksburg, Mississippi
Natchez, Mississippi
New Orleans, Louisiana
Gulf of Mexico


 Enter cities ...

 All city names have been read ...

 LIST the original tour
 ======================

 Cities passed going downstream ...
 Lake Itaska, Minnesota
 Minneapolis, Minnesota
```

```
!
!                          Clear the linked list allocation.
  print *, " "
  print *, "DEALLOCATE the linked list"
  print *, "=========================="
  print *, " "
  print *, " Cities deallocated going downstream ..."
  travel => start
  do
    print *, " ", travel%city
    temp => travel%downstream
    deallocate ( travel )
    if ( .not. associated ( temp%downstream ) ) exit
    travel => temp
  end do
  nullify ( finish )
  nullify ( hannibal )
  nullify ( memphis )
  nullify ( natchez )
  nullify ( rock_island )
  nullify ( st_louis )
  nullify ( start )
  nullify ( temp )
  nullify ( vicksburg )
!
!                        End of program.
  print *, " "
!

end program tour
!
!
!
!                          Show the downstream route.
subroutine droute
  use city_list
  print *, " "
  print *, " Cities passed going downstream ..."
  travel => start
  do
    print *, " ", travel%city
    travel => travel%downstream
    if ( .not. associated ( travel%downstream ) ) exit
  end do
end subroutine droute
```

```
  call uroute
!
! Insert Hannibal below Rock Island and above St. Louis.
  print *, " "
  print *, "INSERT Hannibal between Rock Island and St. Louis"
  print *, "================================================"
  travel => start
  do
    if ( travel%city(1:4) .eq. 'Rock' ) then
      rock_island => travel
      st_louis => rock_island%downstream
      allocate ( hannibal )
      hannibal%upstream => rock_island
      hannibal%city = 'Hannibal, Missouri'
      hannibal%downstream => st_louis
      rock_island%downstream => hannibal
      st_louis%upstream => hannibal
      exit
    else
      travel => travel%downstream
      if ( .not. associated ( travel%downstream ) ) exit
    end if
  end do
  call droute
!
! Delete Vicksburg (below Memphis and above Natchez).
  print *, " "
  print *, "DELETE Vicksburg from between Memphis and Natchez"
  print *, "================================================"
  travel => start
  do
    if ( travel%city(1:4) .eq. 'Vick' ) then
      vicksburg => travel
      memphis => vicksburg%upstream
      natchez => vicksburg%downstream
      memphis%downstream => natchez
      natchez%upstream => memphis
      deallocate ( vicksburg )
      exit
    else
      travel => travel%downstream
      if ( .not. associated ( travel%downstream ) ) exit
    end if
  end do
  call uroute
```

```
!                                  Record the current city.
    travel%city = city_input
!
!                            Allocate memory for the next city and
!                                  point to that storage.
    allocate ( travel%downstream )
    travel => travel%downstream
!
!                         Link the previous city as "upstream" from
!                                  the current city.
    travel%upstream => temp
  end do
200 continue
!
!                                  Keep track of the last city.
  finish => temp
!
!                                  Clear the end-of-list.
  nullify ( travel%downstream )
!
! Advise the user that all data have been read.
!
  print *, " "
  print *, "All city names have been read ..."
!
!                                  Show the cities going downstream.
  print *, " "
  print *, "LIST the original tour"
  print *, "======================"
  call droute
!
!                                  Replace Davenport with Dubuque.
  print *, " "
  print *, "REPLACE Davenport with Dubuque"
  print *, "============================="
  travel => start
  do
    if ( travel%city(1:4) .eq. 'Dave' ) then
      travel%city = 'Dubuque, Iowa'
      exit
    else
      travel => travel%downstream
      if ( .not. associated ( travel%downstream ) ) exit
    end if
  end do
```

```
module city_list
    type river
      type (river), pointer :: upstream
      character (len=25) :: city
      type (river), pointer :: downstream
    end type river
    type ( river ), pointer :: finish
    type ( river ), pointer :: start
    type ( river ), pointer :: travel
end module city_list

program tour
!
! Use a doubly linked list to travel downstream and
! upstream along a river. The sample input file
! lists the start and end points and selected cities
! on the Mississippi.
!
  use city_list
  character (len=25) :: city_input
  type ( river ), pointer :: hannibal
  type ( river ), pointer :: memphis
  type ( river ), pointer :: natchez
  type ( river ), pointer :: rock_island
  type ( river ), pointer :: st_louis
  type ( river ), pointer :: temp
  type ( river ), pointer :: vicksburg
!
!                                 Initialize the first data element.
  allocate ( start )
  nullify ( start%upstream )
  start%city = ' '
  nullify ( start%downstream )
  travel => start
!
!                                 Read the cities along the way.
  print *, " "
  print *, "Enter cities ..."
  do
    read ( *,100,end=200 ) city_input
100 format ( a25 )
!
!                                 Keep track of the previous city.
    temp => travel
!
```

```
subroutine factor( matrix, rank, factored )
!
  integer :: rank
  real, dimension(:,:), allocatable :: across
  real, dimension(:,:), allocatable :: down
  real, dimension(:,:), allocatable :: extract
  real, dimension(rank,rank) :: factored
  real, dimension(rank,rank) :: matrix
  real :: minor
  integer :: rankm1
!
  rankm1 = rank - 1
  allocate( across(rankm1,rankm1) )
  allocate( down(rank,rank) )
!
column: &
  do j = 1, rank, 1
    if( j .ne. 1 ) &
      down( :, 1:j-1 ) = matrix( :, 1:j-1 )
    if( j .ne. rank ) &
      down( :, j:rankm1 ) = matrix( :, j+1:rank )
row: &
    do i = 1, rank, 1
      if( i .ne. 1 ) &
        across( 1:i-1, : ) = down( 1:i-1, : )
      if( i .ne. rank ) &
        across( i:rankm1, : ) = down( i+1:rank, : )
      call determine( transpose(across), rankm1, minor )
      if( mod(i+j,2) .eq. 1 ) minor = - minor
      factored(i,j) = minor
    end do row
  end do column
  deallocate( down )

deallocate( across )
end subroutine factor
```

```
subroutine invert( matrix, rank, determinant, inverse )
!
  real, dimension(:,:), allocatable :: adjoint
  real, dimension(:,:), allocatable :: cofactor
  real :: determinant
  real, dimension(:,:), allocatable :: identity
  integer :: rank
  real, dimension(rank,rank) :: inverse
  real, dimension(rank,rank) :: matrix
!
  if( rank .lt. 1 .or. rank .gt. 3 ) then
    print *, "ERROR! matrix rank not 1, 2, or 3 (INVERT)"
    goto 200
  endif
!
  if( rank .eq. 1 ) then
    if( matrix(1,1) .ne. 0.0 ) then
      determinant = 0.0
      inverse(1,1) = 1.0 / matrix(1,1)
    else
      print *, "ERROR! 1x1 matrix has no inverse"
    endif
    goto 200
  endif
!
  allocate( adjoint(rank,rank) )
  call factor( transpose( matrix ), rank, adjoint )
  call determine( matrix, rank, determinant )
  if( determinant .ne. 0.0 ) then
    inverse = ( 1.0 / determinant ) * adjoint
    allocate( identity(rank,rank) )
    identity = matmul( matrix, inverse )
    if( abs( float( rank ) &
           - sum( identity, mask=identity.gt.0.0 ) ) &
             .gt. 0.05 ) &
               print *, "ERROR! sum identity diagonal ", &
                 "differs from rank (INVERT)."
    deallocate( identity )
  else
    print *, "ERROR! determinant zero ... no inverse."
  endif
  deallocate( adjoint )
200 continue
end subroutine invert
```

```
program part2
!
  integer, parameter :: m = 2
  integer, parameter :: n = 3
  real :: determinant
  real, dimension (n,n) :: identity
  real, dimension (n,n), target :: inverse
  real, dimension (n,n), target :: matrix
  real, dimension (m,m) :: t_nw
  real, dimension (:,:), pointer :: i_nw, i_ne, i_sw, i_se
  real, dimension (:,:), pointer :: nw, ne, sw, se
  data matrix / 1, 2, 3, 1, 0, 7, 1, 6, 1 /
!
  nw => matrix( 1 :m, 1 :m )
  sw => matrix( m+1:n, 1 :m )
  ne => matrix( 1 :m, m+1:n )
  se => matrix( m+1:n, m+1:n )
!
  i_nw => inverse( 1 :m, 1 :m )
  i_sw => inverse( m+1:n, 1 :m )
  i_ne => inverse( 1 :m, m+1:n )
  i_se => inverse( m+1:n, m+1:n )
!
  call invert( se, n-m, determinant, i_se )
  t_nw = nw - matmul( matmul( ne, i_se ), sw )
  call invert( t_nw, m, determinant, i_nw )
  i_sw = - matmul( matmul( i_se, sw ), i_nw )
  i_ne = - matmul( matmul( i_nw, ne ), i_se )
  i_se = i_se - matmul( matmul( i_se, sw ), i_ne )
!
  call echo ( "MATRIX", matrix, n )
  call echo ( "INVERSE", inverse, n )
  identity = matmul( matrix, inverse )
  call echo ( "IDENTITY", identity, n )
!
end program part2
```

```
subroutine determine( matrix, rank, determinant )
!
  integer :: rank
  real :: determinant
  real, dimension(rank,rank) :: matrix
!
  select case( rank )
    case ( :0 )
      determinant = 0.0
    case ( 1 )
      determinant = matrix( 1,1 )
    case ( 2 )
      determinant  =    ( matrix(1,1)*matrix(2,2) ) &
                   -    ( matrix(1,2)*matrix(2,1) )
    case ( 3 )
      determinant  = (  (matrix(1,1)*matrix(2,2)*matrix(3,3)) + &
                        (matrix(1,2)*matrix(2,3)*matrix(3,1)) + &
                        (matrix(1,3)*matrix(2,1)*matrix(3,2)) ) &
                   - (  (matrix(1,3)*matrix(2,2)*matrix(3,1)) + &
                        (matrix(2,3)*matrix(3,2)*matrix(1,1)) + &
                        (matrix(3,3)*matrix(1,2)*matrix(2,1)) )
    case( 4: )
      determinant = 0.0
  end select
end subroutine determine
```

```
  MATRIX row 1: 1.0000 1.0000 1.0000
  MATRIX row 2: 2.0000 0.0000 6.0000
  MATRIX row 3: 3.0000 7.0000 1.0000

  INVERSE row 1: 3.5000 -0.5000 -0.5000
  INVERSE row 2: -1.3333 0.1667 0.3333
  INVERSE row 3: -1.1667 0.3333 0.1667

  IDENTITY row 1: 1.0000 0.0000 0.0000
  IDENTITY row 2: 0.0000 1.0000 0.0000
  IDENTITY row 3: 0.0000 0.0000 1.0000
```