NAG Library Function Document

nag_chain_sort (m01cuc)

1 Purpose

nag_chain_sort (m01cuc) rearranges the links of a linked list into ascending or descending order of a specified data field of arbitrary type.

2 Specification

```
#include <nag.h>
#include <nagm01.h>
void nag_chain_sort (Pointer *base, ptrdiff_t offset,
     Integer  (*compare)(const Nag_Pointer a, const Nag_Pointer b),
     Nag_SortOrder order, NagError *fail)
```

3 Description

nag_chain_sort (m01cuc) uses a variant of list merging, as described in Knuth (1973). It uses a local stack to avoid the need for a flag bit in each pointer.

4 References

Knuth D E (1973) *The Art of Computer Programming (Volume 3)* (2nd Edition) Addison–Wesley

5 Arguments

1: **base** – Pointer * *Input/Output*

On entry: the pointer to the pointer to the first structure of the linked list.

On exit: the pointer to which **base** points is updated to point to the first element of the ordered list.

2: **offset** – ptrdiff_t *Input*

On entry: the offset within the structure of the pointer field which points to the next element of the linked list.

Note: this field in the last element of the linked list must have the value **NULL**.

3: **compare** – function, supplied by the user *External Function*

nag_chain_sort (m01cuc) compares the data fields of two elements of the list. Its arguments are pointers to the structure, therefore this function must allow for the offset of the data field in the structure (if it is not the first).

The function must return:

−1	if the first data field is less than the second,
0	if the first data field is equal to the second,
1	if the first data field is greater than the second.

The specification of **compare** is:

```
Integer  compare (const Nag_Pointer a, const Nag_Pointer b)
```

1: **a** – const Nag_Pointer *Input*

On entry: the first data field.

2: **b** – const Nag_Pointer *Input*

On entry: the second data field.

4: **order** – Nag_SortOrder *Input*

On entry: specifies whether the array will be sorted into ascending or descending order.

Constraint: **order** = Nag_Ascending or Nag_Descending.

5: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_BAD_PARAM

On entry, **order** had an illegal value.

NE_CH_LOOP

Too many elements in chain or chain in a loop. The linked list may have become corrupted.

7 Accuracy

Not applicable.

8 Parallelism and Performance

Not applicable.

9 Further Comments

The time taken by nag_chain_sort (m01cuc) is approximately proportional to $n\log(n)$.

10 Example

The example program declares a structure containing a data field, a pointer to the next record and an index field. It generates an array of such structures assigning random data to the data field. The index field is randomly assigned a unique value. The pointer to next record fields are assigned to create a linked list in the order of the index field. It sorts this linked list into ascending order according to the value of the data field.

10.1 Program Text

```
/* nag_chain_sort (m01cuc) Example Program.
 *
 * Copyright 1996 Numerical Algorithms Group.
 *
 * Mark 4, 1996.
 * Mark 5 revised, 1998.
 * Mark 7 revised, 2001.
 *
```

```
 * Mark 8 revised, 2004
 *
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nag_stddef.h>
#include <nagg05.h>
#include <nagm01.h>

struct recd
{
  double      data;
  struct recd *next;
  Integer     index;
};

#ifdef __cplusplus
extern "C" {
#endif
static Integer NAG_CALL compare(const Nag_Pointer a, const Nag_Pointer b);
#ifdef __cplusplus
}
#endif

int main(void)
{
  /* Integer scalar and array declarations */
  Integer     exit_status = 0;
  Integer     lstate;
  Integer     *state = 0, draw[1];

  /* Double scalars */
  double      p[1];

  /* NAG structures and types */
  NagError    fail;
  ptrdiff_t   ptroffset;
  size_t      i, j, l[20];
  struct recd *address, *base, *origbase, *vec = 0;

  /* Set the number of data fields */
  size_t      n = 10;

  /* Choose the base generator */
  Nag_BaseRNG genid = Nag_Basic;
  Integer     subid = 0;

  /* Set the seed */
  Integer     seed[] = { 1762543 };
  Integer     lseed = 1;

  /* Initialise the error structure */
  INIT_FAIL(fail);

  /* Get the length of the state array */
  lstate = -1;
  nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_rand_init_repeatable (g05kfc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  /* Allocate arrays */
  if (!(vec = NAG_ALLOC(20, struct recd)) ||
      !(state = NAG_ALLOC(lstate, Integer)))
    {
```

```
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

/* Initialise the generator to a repeatable sequence */
nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
if (fail.code != NE_NOERROR)
  {
    printf("Error from nag_rand_init_repeatable (g05kfc).\n%s\n",
            fail.message);
    exit_status = 1;
    goto END;
  }

ptroffset = (ptrdiff_t)(((char *) &(vec->next)) - ((char *) vec));

/* Set data field to random number between 0 and 5 */
for (i = 0; i < n; ++i)
  {
    /* Generate a random integer from a uniform distribution */
    nag_rand_discrete_uniform(1, (Integer) 0, (Integer) 5, state, draw,
                              &fail);
    if (fail.code != NE_NOERROR)
      {
        printf(
                "Error from nag_rand_discrete_uniform (g05tlc).\n%s\n",
                fail.message);
        exit_status = 1;
        goto END;
      }
    vec[i].data = (double) draw[0];
  }

/* Randomly set index fields from 0 to 9 */
for (i = 0; i < n; ++i)
  {
    /* Generate a random value from a uniform distribution */
    nag_rand_basic(1, state, p, &fail);
    if (fail.code != NE_NOERROR)
      {
        printf("Error from nag_rand_basic (g05sac).\n%s\n",
                fail.message);
        exit_status = 1;
        goto END;
      }
    j = (int)((i+1)*p[0]);

    /* j is less than or equal to i */
    (vec[i]).index = (vec[j]).index;
    (vec[j]).index = i;
  }

/* Set next pointers to make linked list in index field order */
for (i = 0; i < n; ++i)
  l[(vec[i]).index] = i;
for (i = 0; i < n-1; ++i)
  vec[l[i]].next = &vec[l[i+1]];
vec[l[n-1]].next = NULL;

/* Get pointers to the start of the linked list (base) and the start
   of the array (origbase) */
origbase = &vec[0];
base = &vec[l[0]];

/* Print Input Data */
printf("nag_chain_sort (m01cuc) Example Program Results\n");
printf("\nDATA\n\n");
printf("Matrix Order:\n");
printf("   Matrix Index      Linked List Index      Data\n");
for (i = 0; i < n; ++i)
```

```
    printf("%10u%20ld%20.6f\n", (unsigned long)i,
           vec[i].index, vec[i].data);
  printf("Linked List Order:\n");
  printf("   Matrix Index      Linked List Index      Data\n");
  for (address = base; address != NULL; address = (*address).next)
    printf("%10u%20ld%20.6f\n",
           (unsigned long)(address-origbase), (*address).index,
           (*address).data);

  /* Sort the linked list on the data field */
  /* nag_chain_sort (m01cuc).
   * Chain sort of linked list
   */
  nag_chain_sort((Pointer *) &base, ptroffset, compare, Nag_Ascending, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_chain_sort (m01cuc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }

  /* Output results */
  printf("\nRESULTS\n\n");

  /* The order in the input matrix is unchanged */
  printf("Matrix Order:\n");
  printf("   Matrix Index      Linked List Index      Data\n");
  for (i = 0; i < n; ++i)
    printf("%10u%20ld%20.6f\n", (unsigned long)i,
           vec[i].index, vec[i].data);

  /* But the linked list pointers have been changed to reflect
     the ascending sort on the data field */
  printf("Linked List Order:\n");
  printf("   Matrix Index      Linked List Index      Data\n");
  for (address = base; address != NULL; address = (*address).next)
    printf("%10u%20ld%20.6f\n",
           (unsigned long)(address-origbase), (*address).index,
           (*address).data);

 END:
  NAG_FREE(vec);
  NAG_FREE(state);

  return exit_status;
}

static Integer NAG_CALL compare(const Nag_Pointer a, const Nag_Pointer b)
{
  double x = ((struct recd *) a)->data;
  double y = ((struct recd *) b)->data;
  return(x < y?-1:(x == y?0:1));
}
```

10.2 Program Data

None.

10.3 Program Results

```
nag_chain_sort (m01cuc) Example Program Results

DATA

Matrix Order:
   Matrix Index      Linked List Index      Data
          0                   3           3.000000
          1                   6           0.000000
          2                   4           4.000000
          3                   1           4.000000
          4                   0           0.000000
          5                   9           2.000000
          6                   5           2.000000
          7                   7           4.000000
          8                   8           2.000000
          9                   2           4.000000
Linked List Order:
   Matrix Index      Linked List Index      Data
          4                   0           0.000000
          3                   1           4.000000
          9                   2           4.000000
          0                   3           3.000000
          2                   4           4.000000
          6                   5           2.000000
          1                   6           0.000000
          7                   7           4.000000
          8                   8           2.000000
          5                   9           2.000000

RESULTS

Matrix Order:
   Matrix Index      Linked List Index      Data
          0                   3           3.000000
          1                   6           0.000000
          2                   4           4.000000
          3                   1           4.000000
          4                   0           0.000000
          5                   9           2.000000
          6                   5           2.000000
          7                   7           4.000000
          8                   8           2.000000
          9                   2           4.000000
Linked List Order:
   Matrix Index      Linked List Index      Data
          4                   0           0.000000
          1                   6           0.000000
          6                   5           2.000000
          8                   8           2.000000
          5                   9           2.000000
          0                   3           3.000000
          3                   1           4.000000
          9                   2           4.000000
          2                   4           4.000000
          7                   7           4.000000
```