Skip-Lists with |findmin| extensions

Aleksandar Donev

February 2000

Contents

1 Module Skip_Lists .................................................. 1
   1.1 Key Data Types .................................................. 1
   1.2 Node Data Types ................................................. 3

2 Allocation Management for Skip Lists .......................... 8
   2.1 Stack-Based Allocation System .............................. 9
       2.1.1 Node Allocation ........................................ 10
       2.1.2 Node Deallocation ...................................... 15
       2.1.3 Initialization and Termination ...................... 16

3 Searching, Insertion and Deletion (and Printing) Operations 19
   3.0.1 Searching .................................................. 21
   3.0.2 Insertion .................................................... 22
   3.0.3 Deletion ..................................................... 23
   3.0.4 Printing ..................................................... 25

4 Test Program ...................................................... 26

5 Formatting rules for HPF/F90 files ............................. 29
1 Module `Skip_Lists`

This module is an implementation of a probabilistic search data structure called a skip list. Skip lists are sorted linked lists (ordered lists) in which search takes a logarithmic time. The implementation is not optimized in many respects, and is not finished either, but it has some points worth taking.

The primary goal of the implementation is to provide the tools needed to manipulate a small collection (set) of skip lists which have size of the same order (and this size should be a large number, since the implementation has large overheads per list), under the operations of merging, concatenation, splitting, deletion, insertion and of course to provide efficient search operations. At present the splitting, merging and concatenation operations have not yet been implemented, but this can be done easily using the structures and macros developed here. The primary operations are `InsertInSkipLists`, `DeleteFromSkipList` and `SearchInSkipList`. All of these are generic procedures overloaded for different types of keys.

One of the major optimizations performed here is the internal handling of memory allocation. Using the run-time system to allocate and deallocate the nodes one by one has severe overheads. Here I implement a paging system for node allocation, which is not quite optimized, but is rather efficient and a good example of what one needs to do in serious implementations.

"WEAVE.f90" 1.0.1 ≡

⟨Key_Types 1.1.2⟩
⟨Skip_Lists 1.2.12⟩
⟨TestProgram 4.1⟩

1.1 Key Data Types

The primary component of a node is its key, which can be of different types here, and is boxed inside the derived type `key_type`. Only single integer keys are used here, but the macros below allow one to implement other types easily. Another component of a node is the data associated with the node. This can be anything. If it is a pointer to an object, the list is exogenous and in Fortran 200x one can make these pointers generic and thus implement generic skip lists (but the keys are fixed and are overloaded with generic interfaces here, which makes the whole implementation fast). If the data is an actual object, then the list is (but generic lists can not be implemented this way with this module).

I decided to box the key and the data together into one type, and box the key itself into another type (to avoid naming troubles). All routines here have names after the type of key and are then overloaded generically:
"WEAVE.f90" 1.1.1

@m _DeclareSingleKeyType( _key_type, _type, _kind, initial_value, Data,...
   _type(KIND = _kind), PARAMETER, PUBLIC :: huge&_key_type = HUGE(huge&_key_type)
   PUBLIC :: _key_type
   _type(KIND = _kind): my_value
ENDTYPE
   PUBLIC :: Data&_key_type   // A node has a key and maybe additional data
   TYPE Data&_key_type
   _type(_key_type): my_key   // The search key
   #!Data(.): // If we need additional values
ENDTYPE

(KeyTypes 1.1.2) ≡

MODULE KeyTypes
   USE Precision
   IMPLICIT NONE
   PRIVATE
   _DeclareSingleKeyType(Key_i_wp, INTEGER, i_wp, huge_Key_i_wp, Dummy)
   _DeclareSingleKeyType(Key_i_wp,_wp, INTEGER, i_wp, huge_Key_i_wp, _Declare_i_wp, my_cost)
END MODULE KeyTypes

This code is used in section 1.0.1.
1.2 Node Data Types

To make a node, other than its datum (key+other data), we need its next-in-the-list pointers. For a skip list a node has not one, but several next pointers, arranged in levels, and this type does not need to be stored explicitly (the algorithm never references a level higher then what was allocated during insertion of the node into the list). However, here I preallocate the nodes and their next pointers, so each node has a fixed largest possible level \( my._{\text{max_level}} \) (the actual node level can be capped to a number smaller then this. This way each page can essentially support an infinite skip list, since the maximum level is not capped at all). The preallocated nodes will be organized in pages, whose allocation will be managed internally. So each node needs to store it’s page number \( my._{\text{page}} \).

A typical declaration of a next field for a skip list would be:

```
_TYPE (NodePointer), DIMENSION (:), DYNAMIC :: my._{\text{next}} = _NULL
```

where \( \text{DYNAMIC} \) is either \( \text{POINTER} \) or \( \text{ALLOCATABLE} \) (in F9x), preferably the second one. This declaration makes coding very easy, but it is extremely memory inefficient, since along with the actual node pointers a whole array descriptor needs to be stored as well, which is more memory then usually needed for the pointers themselves. Therefore here I preallocate pages of node pointers and then each node gets a piece of this array starting at the index \( my._{\text{next}} \). If Fortran had an equivalent to C’s pointers, then one could make a declaration such as:

```
_TYPE (NodePointer), DIMENSION (*), POINTER :: my._{\text{next}} = _NULL
```

and then associate \( my._{\text{next}} \Rightarrow \text{page of pointers} \cdot my._{\text{next}} + my._{\text{max_level}} \) and this would completely separate the allocation system from the skip-list implementation. However, Fortran does not offer such contiguous pointers and so here I use macros to make the implementation of the search routines independent of the allocation system.

---

"WEAVE.f90" 1.2.1 ≡

@m \_DYNAMIC POINTER
@m \_NOT\_NULL ASSOCIATED

"WEAVE.f90" 1.2.2 ≡

@m \_DeclareNodeType (_key_type) PUBLIC :: Node _& key_type

_TYPE Node _& key_type

_PRIVATE

_TYPE (Data _& key_type) :: my._{\text{data}} // The key and data of the node

INTEGER (KIND = i_{\text{byte}}) :: my._{\text{max_level}} = 1, my._{\text{page}} = 1

// The allocated level and page number

INTEGER (KIND = i_{\text{up}}) :: my._{\text{next}} = 0 // Zero-index into the block of nexts

ENDTYPE

PUBLIC :: NodePointer _& key_type

_TYPE NodePointer _& key_type

_PRIVATE

_TYPE (Node _& key_type), POINTER :: my._{\text{node}} = _NULL

ENDTYPE
I will not explain the memory system in any detail. The main type MemoryMapOfNodes associates some memory to a given type of preallocated nodes (meaning nodes of a given type and with a given probability \( p \) for the next pointers). This memory can then be shared by several skip-lists, and the number of skip-lists should be kept small if efficiency is desired. The memory is organized in pages, which contain preallocated blocks of nodes and next pointers, and pages will later be allocated and deallocated as needed:

"WEAVE.f90" 1.2.4 =
@m DeclareMemoryType(_key_type_

TYPE BlockOfNodes _key_type_
  _TYPE (Node _key_type_), DIMENSION (:), DYNAMIC :: my_nodes = NULL
  _TYPE (NodePointer _key_type_), DIMENSION (:), DYNAMIC :: my_pointers = NULL
  INTEGER (KIND = i_up) :: my_n_nodes = 0   // Counter
ENDTYPE
PUBLIC :: PageOfNodes _key_type_    // For now each page will only have one block

_PRIVATE
  _TYPE (BlockOfNodes _key_type_) :: my_block
  _TYPE (NodePointer _key_type_) :: my_free_node   // Each page has its stack of free nodes
  INTEGER (KIND = i_byte) :: my_next = 0   // Pages are linked in a stack
ENDTYPE
PUBLIC :: MemoryMapOfNodes _key_type_  // The user needs access to these

TYPE MemoryMapOfNodes _key_type_    // A collection of memory pages
  _TYPE (PageOfNodes _key_type_), DIMENSION (:), DYNAMIC :: my_pages = NULL
  REAL (KIND = r_double) :: my_p = 0.25, my_log_p = -1.386294, my_log_p  // The probability \( p \)
  INTEGER (KIND = i_up) :: my_page_size = 100   // How many nodes per page
  INTEGER (KIND = i_byte) :: my_max_lists = 0, my_max_lists = 10
  // How many lists will use this memory
INTERGER (KIND = i_byte) :: my_max_pages = 10, my_n_pages = 0
  // Estimated number of needed pages
  INTEGER (KIND = i_byte) :: my_free_page = 0    // Top of stack of free pages
ENDTYPE

Each skip list has some counters and pointers associated with it, which are packaged in the data-type SkipList. This includes the virtual memory system associated with the skip list, the dummy (sentinel) head and tail nodes, etc. Some of these variables are overhead, but since usually the skip lists will contain many nodes, this is not too bad:
To make my life (read: typing) easier and to separate the exact organization of the data types from the rest of the algorithm, I declare a bunch of macros here whose names start with an underscore:
A further macro layer are the macros that take a finger pointer and associate it or with it some node pointer (for a given layer), and macros that find and associate the next pointer for a node at a given node. All algorithms will be written in terms of these four macros, thus making changing some of the above data types easy:
"WEAVE.f90" 1.2.10 ≡
@m  FindFinger(search_finger, node, level)
    node ⇒ search_finger_level % my_node
@m  AssociateFinger(search_finger, node, level)
    search_finger_level % my_node ⇒ node
@m  FindNext(node, next_node, level)
    next_node ⇒ nexts_block(page(node))(next(node)+level % my_node
@m  AssociateNext(node, next_node, level)
    nexts_block(page(node))(next(node)+level % my_node ⇒ next_node
@m  FindLevel(node, level)
    level = 1     // The actual level is not stored!

And finally, here is the organization of the Skip_Lists module:

(Skip_Lists 1.2.12) ≡

MODULE Skip_Lists
    USE Precision
    USE Error_Handling
    USE System_Monitors
    USE Random_Numbers
    USE Key_Types
    IMPLICIT NONE
    PUBLIC:: InitializeSkipList, DestroySkipList, SearchInSkipList, InsertInSkipList, DeleteFromSkipList, PrintSkipList
    PRIVATE
        DeclareSkipList(Key_i_wp)
        DeclareSkipList(Key_i_wp_r_wp)
        ⟨GenericInterfaces 2.1.2.2⟩
    CONTAINS
        ⟨MemoryManagement 2.1.2.1⟩
        ⟨InitializeSkipList 2.1.3.1⟩
        ⟨DestroySkipList 2.1.3.4⟩
        ⟨SearchInSkipList 3.0.1.1⟩
        ⟨InsertInSkipList 3.0.2.1⟩
        ⟨DeleteFromSkipList 3.0.3.1⟩
        ⟨PrintSkipList 3.0.4.3⟩

END MODULE Skip_Lists

This code is used in section 1.0.1.
And two more auxilliary macros I will use, along with local variable declarations that will be in common to all the routines in this module:

"WEAVE.f90" 1.2.14  
@m _GenerateProcedureBody( Procedure, ...)  
  ## Procedure( Key_i_wp, _CompareSingleKeys, _TestEqualitySingleKeys, #.)  
  ## Procedure( Key_i_wp, i_wp, _CompareSingleKeys, _TestEqualitySingleKeys, #.)

"WEAVE.f90" 1.2.15  
@m _GenerateGenericInterface( Procedure)  
  INTERFACE Procedure  
    MODULE PROCEDURE Procedure@&_Key_i_wp  
    MODULE PROCEDURE Procedure@&_Key_i_wp_i_wp  
  END INTERFACE  
@m _DeclareAuxiliaryVariables( key_type)  
  _TYPE ( Node_&key_type), POINTER :: node, next_node, search_node  
  INTEGER ( KIND = i_byte) :: level, node_level  
  LOGICAL :: comparison

2  Allocation Menagement for Skip Lists

Each node in the skip list has a random number of pointers to nodes following it in the skip list. The maximum node level needed to manipulate n nodes in logarithmic time is denoted here with L(n). There are several ways to generate a random node level (from a binomial distribution), some better when the node level is large (this involves an expensive logarithm), others better for small levels (several random number generations, but no logarithm). I use the one suitable for large lists, but it is possible to switch between the two:
"WEAVE.f90" 2.0.0.1 ≡
@m \texttt{\_}(n, \texttt{log}_{p}) \ \text{MAX}(\texttt{INT}(\text{\textsc{-log}}(\text{REAL}(n, \text{\texttt{KIND}} = \text{r}..\text{wp}))/\text{\texttt{log}}_{p}), 1)
@m \texttt{\_RandomLevelLarge}\left(\texttt{level}, \texttt{log}_{p}\right)
\begin{align*}
&\text{CALL RandomUniform}\left(\texttt{dice}\right) \\
&\texttt{level} = \text{MAX}(1, \text{\texttt{byte}}, \text{CEILING}(\text{\texttt{LOG}}(\texttt{dice})/\text{\texttt{log}}_{p}, \text{\texttt{KIND}} = \texttt{i}..\texttt{byte}))
\end{align*}
@m \texttt{\_RandomLevelSmall}\left(\texttt{level}, p\right)
\begin{align*}
\texttt{level} &= 1 \\
\text{DO} & \\
&\text{IF (\texttt{dice} \geq p)} \text{EXIT} \\
&\texttt{level} = \texttt{level} + 1 \\
\text{END DO}
\end{align*}
@m \texttt{\_GenerateNodeLevel}\left(\texttt{skip\_list}, \texttt{n\_levels}\right)
\begin{align*}
\texttt{RandomLevelLarge(n\_levels, \texttt{log}_{p})} \\
&\texttt{\_} \text{Generate a random level}
\end{align*}

2.1 Stack-Based Allocation System

The allocator developed here uses a stack-based system to minimize memory overhead and memory fragmentation. Namely, when a node is deleted, it is placed on a stack of free nodes (there is one stack for each page in the virtual memory). Since we already have next pointers, this does not cost us any additional memory. Then when a new node needs to be allocated, the first thing to check is whether there are any previously deallocated nodes on the stack. If not, there may be still untouched nodes in the page, and if not, then a new page may need to be allocated. Here pages that have open spots in them are also placed on a stack, so that a page is filled completely before taken off the stack. This will tend to minimize memory fragmentation when data-locality or locality-of-reference is present, but other schemes, such as keeping a heap of pages sorted on the number of free spots.

The macros below are the ones that perform the stack operations. I can not fully explain the system here, but the general principles should be evident from looking at the code and comments:
"WEAVE.f90" 2.1.0.1 ≡
  @m _PushOnNodeStack(node, stack)
    _AssociateNext(node, stack, 1)
    stack ⇒ node
  @m _PopOffNodeStack(node, stack)
    _FindNext(stack, next_node, 1)
    node ⇒ stack
    stack ⇒ next_node
  @m _PushOnPageStack(page, stack)
    _pagesstack, stack
    % my_next = stack
    stack = page
  @m _PopOffPageStack(page, stack, next_page)
    _next_page = _pagesstack, stack
    % my_next
    page = stack
    stack = next_page

2.1.1 Node Allocation

The routine that allocates nodes is the most complicated. It has to first look for a page that has empty nodes, and if there is no such page, allocate one. All pages may be full, in which case these need to be reallocated. Empty nodes are popped from a stack of empty nodes maintained on each page. Head and tail nodes must be handled separately, because their node levels are not random but rather fixed. So here the sentinel page number 0 is used to store heads and tails and is managed by a separate piece of code. The routine is somewhat clumsy and can be optimized further, but I think the basic ideas are promising:
"WEAVE.f90"  2.1.1.1 ==
@m AllocateNode(key_type, ...)

SUBROUTINE AllocateNode(&key_type(list, new_node)
  _TYPE (SkipList&key_type), POINTER :: skip_list
  _TYPE (Node&key_type), POINTER :: new_node
  _DeclareAuxiliaryVariables(key_type)
    // We may need some reallocation buffers for this procedure
    _TYPE (PageOfNodes&key_type), DIMENSION (:), ALLOCATABLE :: temp_pages
    _TYPE (Node&key_type), DIMENSION (:), ALLOCATABLE :: temp_nodes
    _TYPE (NodePointer&key_type), DIMENSION (:), ALLOCATABLE :: temp_pointers
    REAL :: dice          // For random level generation
    INTEGER (KIND = i_byte) :: page, temp_page
    INTEGER (KIND = i_w) :: n_nodes, n_pointers       // Counters
    INTEGER :: alloc_status
    LOGICAL :: find_free_node, get_next_node, get_next_page, new_page, reallocate_pages
    // Action selectors—to avoid complex nested IF constructs

HeadAndTail: IF (n_elements ≤ 1) THEN         // This is still the head and tail node
  IF (n_elements = 0) THEN
    n_lists = n_lists + 1
    IF (_trace_allocation) WRITE (*, *) "Allocating head for list ", n_lists
  END IF
  _AllocateHeadAndTail
  RETURN
END IF HeadAndTail

find_free_node = F          // Do we need to look for a free node or is there one on the stacks
page = free_page
IF (page ≠ 0) THEN          // There are no pages on the stack
  IF (NOT ASSOCIATED(free_node(page))) find_free_node = T
ELSE
  find_free_node = T
END IF

FindFreeNode: IF (find_free_node)
  reallocate_pages = F
  get_next_node = F
  get_next_page = F
  new_page = F

FindFreePage: DO           // Look for a page that has empty nodes
  IF (page ≠ 0) THEN       // All allocated pages have been filled
    n_nodes = HUGE(i_w)   // Search for page with least active nodes
  END IF

CheckAllPages: DO temp_page = 1, n_pages
  CheckForOpenings: IF (NOT NULL(nodes(block(temp_page)))) THEN
    // Look if some of the full pages have opened spots
    IF (nodes(temp_page) < size(nodes(block(temp_page), i_w)) THEN
      // Empty spots
      IF (nodes(temp_page) < n_nodes) page = temp_page
    END IF
  ELSE                    // Now look for previously deallocated pages
    CHECK FREE NODES
  END IF
IF (page \equiv 0) \text{ THEN} \quad \text{// Still haven't found openings}
\quad \text{page} = \text{temp} \_ \text{page} \quad \text{// A deallocated page}
\text{END IF}
\text{END IF CheckForOpenings}
\text{END DO CheckAllPages}
IF (page \neq 0) \text{ THEN}
\quad _\text{PushOnPageStack}(\text{page}, \_\text{free}_{\_}\text{page})
\quad \text{EXIT FindFreePage} \quad \text{// Found a partially empty or deallocated page}
\text{END IF}
new \_ \text{page} = \top \quad \text{// All allocated pages are full}
\quad \text{get}\_\text{next}_{\_}\text{node} = \top
\quad get\_\text{next}\_\text{page} = \top \quad \text{// Take the next unallocated page \_n\_pages + 1}
\text{IF (\_n\_pages \geq \text{SIZE(\_nodes\_block)}) \_reallocate\_pages = \top} \quad \text{// All pages are full}
\text{EXIT FindFreePage}
\text{END IF}

\textit{FullPage:} \text{IF (NOT\_NULL(\_nodes\_block(\text{page}))) \text{ THEN} \quad \text{// Check if this page is full}}
\text{IF (\_nodes(\text{page}) \equiv \text{SIZE(\_nodes\_block(\text{page}), i_up)) \text{ THEN} \quad \text{// Page is full}}
\quad \text{PopOffPageStack}(\text{page}, \_\text{free}_{\_}\text{page}, \text{temp} \_ \text{page})
\quad \text{page} = \_\text{free}_{\_}\text{page} \quad \text{// Take the next free page off the stack}
\quad \text{CYCLE FindFreePage} \quad \text{// Look at the next page}
\text{ELSE}
\quad \text{get}\_\text{next}_{\_}\text{node} = \top \quad \text{// There are free nodes left over}
\quad \text{EXIT FindFreePage}
\text{END IF}
\text{ELSE} \quad \text{// This page has not been allocated yet or it was deallocated previously}
\quad new\_\text{page} = \top
\quad \text{get}\_\text{next}_{\_}\text{node} = \top
\quad \text{EXIT FindFreePage}
\text{END IF}
\text{END DO FindFreePage}

\text{IF (\_reallocate\_pages) \text{ THEN} \quad \text{// Reallocate \_pages to have more pages}}
\text{IF (\_trace\_allocation) \text{ THEN} \quad \text{// Record this event}}
\quad \text{CALL Warning(message = "Reallocation of pages in virtual memory in skip list", caller = "AllocateNode")}
\text{END IF}
\quad _\text{ReallocateArray}(\_\text{pages}, \_\text{temp}_{\_}\text{pages}, 0, \_n\_pages + 1)
\text{END IF}

\text{IF (get\_\text{next}\_\text{page}) \text{ THEN}}
\quad \_\text{n}_{\_}\text{pages} = \_n\_pages + 1
\quad \text{page} = \_n\_pages
\quad _\text{PushOnPageStack}(\text{page}, \_\text{free}_{\_}\text{page})
\text{END IF}

\textit{NewPage:} \text{IF (new\_\text{page}) \text{ THEN} \quad \text{// Allocate a new page of nodes and pointers}}
\text{IF (\_trace\_allocation) \text{ THEN}}
\quad \text{WRITE(message, print\_unit, *) "Allocating page of nodes and links ", page, " of size ", \_page\_size, ", n\_pages=" , \_n\_pages, ", n\_elements=" , \_n\_elements}
\text{END IF}
\quad \text{ALLOCATE(\_nodes\_block(\text{page}), \_page\_size)}
\quad \_\text{n}_{\_}\text{pointers} = 0
\textit{CreateNodes:} \text{DO \_n\_nodes = 1, \_page\_size}
node \Rightarrow \text{nodes\_block(page)}_{n\_nodes}
\text{GenerateNodeLevel}(\text{skip\_list, node\_level})
\text{max\_level}(\text{node}) = \text{node\_level}
\text{page}(\text{node}) = \text{page}
\text{next}(\text{node}) = n\_pointers
n\_pointers = n\_pointers + \text{node\_level}
\text{END DO CreateNodes}
\text{ALLOCATE(\text{next\_block(page)}_{n\_pointers})} \quad \text{// Allocate fields for the link pointers}
\text{END IF NewPage}
\text{IF (get\_next\_node) THEN}
\quad \text{node} \Rightarrow \text{nodes\_block(page)}_{n\_nodes(page)+1}
\quad \text{PushOnNodeStack(\text{node, free\_node(page)})} \quad \text{// The next free node}
\text{END IF}
\text{END IF FindFreeNode}
\text{PopOffNodeStack(new\_node, free\_node(page))} \quad \text{// Take the next available node}
\text{n\_nodes(page)} = n\_nodes(page) + 1
\text{END SUBROUTINE} \quad \text{// AllocateNode}

Heads and tails are allocated on page 0 for each skip list that uses the given memory system:
"WEAVE.f90" 2.1.1.3 ≡ 

@m .AllocateHeadAndTail
  IF (.NOT._NULL(.nodes_block(0))) THEN
    n_nodes = 2 * _max_lists
    n_pointers = n_nodes * 1._max_pages * _page_size, _log_p)
  IF (.trace_allocation) THEN
    WRITE (message_print_unit, *) "Allocating page 0 of heads and tails ", ", " for ", 
      n_nodes, " sentinel nodes and ", n_pointers, " next pointers"
  END IF
  ALLOCATE (.nodes_block(0):n_nodes)
  ALLOCATE (.nexts_block(0):n_pointers)
  node ⇒ _nodes_block(0):n_pointers
      // Yet another sentinel node
      _page(node) = 0
      _max_level(node) = 0
      _next(node) = 0
  END IF

  n_nodes = .nodes(0)
  node ⇒ _nodes_block(0):n_nodes
  n_pointers = .next(node) + _max_level(node)
  IF ((n_nodes + 1) > SIZE(_nodes_block(0))) THEN  // Allocate more sentinel nodes
      IF (.trace_allocation) THEN
        WRITE (message_print_unit, *) "Reallocation nodes block of page 0 of heads and 
          tails!"
      END IF
      _ReallocateArray(_nodes_block(0), temp_nodes, 1, n_nodes + 1)
  END IF
  IF ((n_pointers + _max_levels) > SIZE(_nexts_block(0))) THEN  // More next fields
      IF (.trace_allocation) THEN
        WRITE (message_print_unit, *) "Reallocation nexts block of page 0 of heads and 
          tails!"
      END IF
      _ReallocateArray(_nexts_block(0), temp_pointers, 1, n_pointers + _max_levels)
  END IF

  n_nodes(page) = .nodes(0) + 1  // Add this head or tail
  new_node ⇒ _nodes_block(0):n_nodes(page)
      _page(new_node) = 0
      _max_level(new_node) = _max_levels
      _next(new_node) = n_pointers

For some reason reallocation as given here does not work for reallocating the pages, at least with LF95, but it seems to me that this is a compiler bug, not a programming error. With LF95 just be careful to allocate enough pages each time, until the bug is killed! The following macro reallocates a page, or any array for that matter to a new index range:
"WEAVE.f90" 2.1.1.5

```fortran
@intrinsic _ReallocateArray(array, buffer, new_lb, new_ub)
   ALLOCATE (buffer(new_lb : new_ub))
   buffer(new_lb : new_lb+SIZE(array)-1) = array
   DEALLOCATE (array) // Reallocation
   ALLOCATE (array(new_lb : new_ub))
   array = buffer // Copy the buffer
   DEALLOCATE (buffer)
```

### 2.1.2 Node Deallocation

Deallocating nodes is easier—push the empty nodes onto the empty node and check if this was the last used node in the page. If so, the block of nodes in the page is deallocated. At this point we are not sure where in the stack this page is, so we just leave it as it is, although we should really take it off the stack if it is on it (this requires doubly linked lists for $O(1)$ implementations).

In this whole scheme, the part I am not yet pleased with is what happens to full pages. At first I tried to have them put at the bottom of the page stack, since later openings might be created. But this creates endless loops of popping off the stack and pushing at the bottom of the stack. The best solution is probably to have a separate stack for either the deallocated pages or the full (at some point in time) pages. I just haven’t implemented this yet so the above allocation routine scans through all pages when the page stack is empty.
"WEAVE.f90" 2.1.2.1

```fortran
@m DeallocationNode(_key_type,...)

SUBROUTINE DeallocationNode(@_key_type(skip_list, free_node)
  _TYPE (SkipList @_key_type), POINTER :: skip_list
  _TYPE (Node @_key_type), POINTER :: free_node
  _DeclareAuxiliaryVariables(_key_type)
  INTEGER (KIND = ibyte) :: page, temp_page
  page = _page(free_node)
  PushOnNodeStack(free_node, free_node(page))
  _n_nodes(page) = _n_nodes(page) - 1
  IF (_n_nodes(page) == 0) THEN   // Free the page
    IF (_trace_allocation) THEN
      WRITE (message_print_unit, *) "Deallocating page of nodes and links ", page, ",
      _n_pages="", _n_pages, ", _n_elements="", _n_elements
    END IF
    NULLIFY (_free_node(page))
    DEALLOCATE (_nodes_block(page))
    DEALLOCATE (_nexts_block(page))
  END IF

END SUBROUTINE   // DeallocationNode
```

(Generics 2.1.2.1) ≡

```fortran
_GenericProcedureBody(AllocateNode)
_GenericProcedureBody(DeallocationNode)
```

This code is used in section 2.1.2.2.

(Generics 2.1.2.2) ≡

```fortran
_GenericGenericInterface(AllocateNode)
_GenericGenericInterface(DeallocationNode)
```

See also sections 2.1.3.2, 2.1.3.5, 3.0.1.2, 3.0.2.2, 3.0.3.2, and 3.0.4.4.

This code is used in section 2.1.2.2.

### 2.1.3 Initialization and Termination

The routines `InitializeSkipList` and `DestroySkipList` are the creator/destructor routines for the skip list data type. `InitializeSkipList` will create a legal skip list from either an empty pointer or a pointer that is already allocated by the caller (this way some of the parameters used by initialization, such as the maximum allowed level in the skip list, can be set by the caller before calling the initialization routine. Please note that no routine should be changes to the list data should be made after the initialization so as to avoid messing something up. I could have boxed the types a couple more levels to make some things private and some public, but care if all that is needed.
"WEAVE.f90" 2.1.3.1 ==
@m _InitializeSkipList(_key_type,...)

SUBROUTINE _InitializeSkipList(_& key_type (skip_list)
   _TYPE (SkipList(_& key_type), POINTER :: skip_list

   DeclareAuxiliaryVariables(_key_type)
   INTEGER (KIND = i_byte) :: page

   IF (.NOT. ASSOCIATED(skip_list)) ALLOCATE (skip_list)  // Create a fresh skip list

   IF (.NOT. ASSOCIATED(_memory)) THEN  // Create a virtual memory just for this list
      ALLOCATE (_memory)
      _page_size = CEILING (REAL (_max_elements) / REAL (_max_pages))
      // Estimated page size (plus head and tail)
   END IF

   IF (.NOT. NULL (_pages)) ALLOCATE (_pages(0 : _max_pages))
   _log_p = LOG(_p)

   IF (_max_levels <= 0) _max_levels = INT (_max_elements, _log_p)

   _n_elements = 0
   CALL AllocateNode(skip_list, _head)
   _n_elements = 1
   CALL AllocateNode(skip_list, _head)
   _n_elements = 2
   _key(_head) = huge(_& key_type)  // The smallest possible key—a sentinel
   _key(_tail) = huge(_& key_type)  // The largest possible key—a sentinel
   ALLOCATE (_search_finger(_max_levels))

   DO level = 1, _max_levels  // Point to the sentinel node by default
      _AssociateNext(_head, _tail, level)
      _AssociateNext(_tail, NULL(), level)
      _AssociateFinger(_search_finger, _head, level)
   END DO

END SUBROUTINE  // InitializeSkipList

<InitializeSkipList 2.1.3.1> ==
   GenerateProcedureBody(InitializeSkipList)

This code is used in section 1.2.12.

<GenericInterfaces 2.1.2.2> +=
   GenerateGenericInterface(InitializeSkipList)
The routine \texttt{DestroySkipList} will deallocate all memory (note that memory leaks have not been fully investigated yet) associated with a skip list initialized with \texttt{InitializeSkipList}. Please note that this will deallocate the skip list pointer even if this was allocated outside. In the current version I explicitly delete all present nodes before a possible memory deallocation. The memory does not have to be deallocated, as indicated by \texttt{deallocate\_memory}, since other skip lists might share the same virtual memory.

\begin{verbatim}
"WEAVE.f90" 2.1.3.4 \equiv
  @m \textbf{DestroySkipList(\_key\_type, \ldots)}
  \texttt{SUBROUTINE DestroySkipList(\_key\_type \_type (SkipList \_key\_type), POINTER :: skip\_list}
  \texttt{LOGICAL, INTENT (IN), OPTIONAL :: deallocate\_memory}
  \texttt{DeclareAuxiliaryVariables(\_key\_type)}
  \texttt{INTEGER :: alloc\_status}
  \texttt{LOGICAL :: free\_memory}
  \texttt{free\_memory = \_F}
  \texttt{IF (PRESENT(deallocate\_memory)) free\_memory = deallocate\_memory}
  \texttt{DEALLOCATE (\_search\_finger)}
    \texttt{node \_=} \_head
  \texttt{DeleteNodes: DO} \; // Delete all left-over nodes from the list
    \texttt{FindNext(node, next\_node, 1)}
    \texttt{CALL DeallocateNode(skip\_list, node)}
    \texttt{IF (\_NOT ASSOCIATED(next\_node)) EXIT DeleteNodes} \; // Reached the end
    \texttt{node \_=} next\_node
  \texttt{END DO DeleteNodes}
  \texttt{IF (free\_memory)} \texttt{THEN}
    \texttt{DEALLOCATE (\_pages)}
    \texttt{DEALLOCATE (\_memory, STAT = alloc\_status)}
    \texttt{IF (alloc\_status \_= \_0) NULLIFY (\_memory)}
  \texttt{END IF}
  \texttt{DEALLOCATE (skip\_list, STAT = alloc\_status)}
  \texttt{IF (alloc\_status \_= \_0) NULLIFY (skip\_list)}
\texttt{END SUBROUTINE \; // DestroySkipList}
\end{verbatim}

\langle \texttt{DestroySkipList 2.1.3.4} \rangle \equiv
\texttt{\_GenerateProcedureBody(DestroySkipList)}

This code is used in section 1.2.12.

\langle \texttt{GenericInterfaces 2.1.22} \rangle \equiv
\texttt{\_GenerateGenericInterface(DestroySkipList)}
3 Searching, Insertion and Deletion (and Printing) Operations

This section implements the free basic elementary skip list operations. They are all implemented in a similar fashion to that prescribed in the papers by W. Pugh, and are fully coded with the macros defined previously for full ease of maintainance. The central piece of the code is the macro `SearchSkipList` which searches for a given key and returns a search finger pointing to the forward-most nodes that have a key smaller than the given search node and are of the corresponding level. A search finger is used to start the search if requested via `use_search_finger`, so that when data locality or locality-of-reference is present the operations are much faster:
WEAVE.f90  3.0.0.1 =
@m _SearchSkipList(ID, _CompareKeys, skip_list, search_key, search_finger)
  IF (use_search_finger) THEN  // Start as close as possible to the search finger
    level = 1
    _FindFinger(search_finger, search_node, level)
    !_CompareKeys(comparison, search_key, key(search_node))
  IF (~comparison) THEN  // Move search finger as forward as possible
    PointAhead.@&ID:DO
      level = level + 1
      IF (level > n_levels) EXIT PointAhead.@&ID
      _FindNext(node, next_node, level)
      !_CompareKeys(comparison, search_key, key(next_node))
    IF (comparison) THEN  // Jumped too far forward
      EXIT PointAhead.@&ID
    ELSE  // Save this node
      search_node => next_node  // Jump ahead
    END IF
  END DO PointAhead.@&ID
ELSE  // Move back from search finger
  level = 1
  PointBack.@&ID:DO
    level = level + 1
    IF (level > n_levels) THEN  // We need to restart the finger to the header
      search_node => _head
      EXIT PointBack.@&ID
    END IF
    _FindFinger(search_finger, node, level)
    !_CompareKeys(comparison, search_key, key(node))
    IF (~comparison) THEN  // Found the nearest suitable node further to front
      search_node => node  // Jump ahead
      EXIT PointBack.@&ID  // Jumped back far enough
    END IF
  END DO PointBack.@&ID
END IF
ELSE  // Start from the beginning of the list
  level = n_levels + 1
  search_node => _head
END IF
SEARCHAhead.@&ID:DO
  level = level - 1
  IF (level < 1) EXIT SEARCHAhead.@&ID
  _AssociateFinger(search_finger, search_node, level)
SKIPEAhead.@&ID:DO
  _FindFinger(search_finger, node, level)
  _FindNext(node, next_node, level)
  !_CompareKeys(comparison, search_key, key(next_node))
  IF (~comparison) THEN  // OK to jump ahead
    _AssociateFinger(search_finger, next_node, level)
  ELSE  // Go to next level
3.1 Searching

The routine SearchInSkipList is basically a wrapper around the search macro and it returns the datum of the node that matched the given key, if the key was actually present in the list, as indicated by the return value found:

"WEAVE.F90" 3.0.1.1 ==
@m _SearchInSkipList(_key_type, _CompareKeys, _TestEqualityOfKeys, ...)

SUBROUTINE SearchInSkipList(@&_key_type, skip_list, key, data, found)
  _TYPE (SkipList@&_key_type), POINTER :: skip_list
  _TYPE (_key_type), INTENT (IN) :: key
  _TYPE (Data@&_key_type), INTENT (OUT), OPTIONAL :: data
  LOGICAL, INTENT (OUT), OPTIONAL :: found
  _DeclareAuxiliaryVariables(_key_type)
  _SearchSkipList(Search, !_CompareKeys, skip_list, key % my_value, _search_finger)
  _FindFinger(_search_finger, search_node, 1) // Find the finger
  _FindNext(search_node, node, 1) // Look at the next node
  !_TestEqualityOfKeys(comparison, _key(node), key % my_value)
  IF (PRESENT(found)) found = comparison
  IF (PRESENT(data) .AND. comparison) data = _data(node) // Return the info about the node
END SUBROUTINE // SearchInSkipList

(SearchInSkipList 3.0.1.1) ==
  _GenerateProcedureBody(SearchInSkipList)

This code is used in section 1.2.12.

(GenericInterfaces 2.1.2) +=
  _GenerateGenericInterface(SearchInSkipList)
3.2 Insertion

When inserting a node, we simply perform a search to find the location of insertion and then just break some of the next-node links to accommodate the new node. The node is explicitly allocated by `InsertInSkipList` here before insertion (and deallocated upon deletion). Since I do the memory management myself, this is not a problem (the node will most likely not really be deallocated, but just popped onto the empty-node stack):

```
"WEAVE.f90" 3.0.2.1 ≡
@m  InsertInSkipList(_key_type, _CompareKeys, _TestEqualityOfKeys, ...)
SUBROUTINE InsertInSkipList(_key_type, skip_list, data)
  _TYPE (SkipList@_key_type), POINTER :: skip_list
  _TYPE (Data@_key_type), INTENT (IN) :: data
DeclareAuxiliaryVariables(_key_type)
  _TYPE (Node@_key_type), POINTER :: new_node
LOGICAL :: duplicate
CALL SearchInSkipList(skip_list = skip_list, key = data % my_key, found = duplicate)
IF ((~keep_duplicates) \& duplicate) RETURN // A duplicate key
CALL AllocateNode(skip_list, new_node)
  n_elements = n_elements + 1
new_node % my_data = data
node_level = MIN(_max_levels, max_level(new_node)) // Do not cross over _max_levels
  _n_levels = MAX(_n_levels, node_level) // The new list level
DO level = 1, node_level
  _FindFinger(search_finger, search_node, level) // search_node ⇒ search_finger_level
  _FindNext(search_node, next_node, level) // next_node ⇒ Next(search_finger, level)
  _AssociateNext(new_node, next_node, level) // Next(new_node, level) ⇒ next_node
  _AssociateNext(search_node, new_node, level) // Next(search_finger, level) ⇒ new_node
END DO
END SUBROUTINE // InsertInSkipList

@<InsertInSkipList 3.0.2.1> ≡
  _GenerateProcedureBody(InsertInSkipList)

This code is used in section 1.2.12.

@<GenericInterfaces 2.1.2.2> ≡
  _GenerateGenericInterface(InsertInSkipList)
3.3 Deletion

Deletion is just like insertion—search for the key to see if it is present, and then remove the node from the list by bypassing it underneath the pointers that ended or started from it, and finally, deallocating the node. Here I check to see if this was the node of highest level, so as to reduce \texttt{.n levels} and possibly speed operation (this almost never happens in moderately sized lists):
"WEAVE.f90" 3.0.3.1 ≡
@m _DeleteFromSkipList(_key_type , _CompareKeys , _TestEqualityOfKeys , ...)

SUBROUTINE DeleteFromSkipList(_key_type (skip_list , key , data , deleted)
  _TYPE (SkipList& _key_type ) , POINTER :: skip_list
  _TYPE (_key_type ) , INTENT (IN) :: key
  _TYPE (Data& _key_type ) , INTENT (OUT) , OPTIONAL :: data
  LOGICAL , INTENT (OUT) , OPTIONAL :: deleted
  DeclareAuxiliaryVariables(_key_type )
  _TYPE (Node& _key_type ) , POINTER :: old_node
  LOGICAL :: found
  */ First perform the standard search for the node: */
  IF (~PRESENT(data)) THEN
    CALL SearchInSkipList(skip_list = skip_list , key = key , found = found)
  ELSE
    CALL SearchInSkipList(skip_list = skip_list , key = key , data = data , found = found)
  END IF
  IF (PRESENT(deleted)) deleted = found  // If node was found it will be deleted
  IF (~found) RETURN  // Node was not found

BypassNode: DO level = 1 , _n_levels
  _FindFinger(_search_finger , search_node , level)
  _FindNext(search_node , next_node , level)
  IF (level ≡ 1) old_node ⇒ next_node  // This is the node to delete
  IF (~ASSOCIATED(old_node , next_node)) EXIT BypassNode
    // We exceeded the level of node—no need to store it explicitly
  _FindNext(old_node , next_node , level)
  _AssociateNext(search_node , next_node , level)  // Bypass the deleted node
END DO BypassNode

ReduceLevel: DO  // If the largest-level node was deleted, reduce _n_levels
  _FindNext(_head , next_node , _n_levels)
  IF (~ASSOCIATED(next_node , _tail))( _n_levels ≤ 1) EXIT ReduceLevel
  _n_levels = _n_levels - 1
END DO ReduceLevel

CALL DeallocateNode(skip_list , old_node)  // Deallocate the deleted node
  _n_elements = _n_elements - 1
END SUBROUTINE  // DeleteFromSkipList

<DeleteFromSkipList 3.0.3.1> ≡
  _GenerateProcedureBody(DeleteFromSkipList)

This code is used in section 1.2.12.

<GenericInterfaces 2.1.22> ≡
  _GenerateGenericInterface(DeleteFromSkipList)
3.4 Printing

This is just one utility routine for printing the contents of a SkipList. For a descent user-friendly implementation of a linked list, I should really provide a routine for accessing all the keys in a list (so one can see what is in the list), and a modification of PrintSkipList could do something like this. I just couldn't decide on the interface of such a "peek in skip list" routine, so I left it out for now.

"WEAVE.f90" 3.0.4.1
@m _key_format "(G10.3)"

"WEAVE.f90" 3.0.4.2
@m _PrintSkipList(_key_type,...)

SUBROUTINE PrintSkipList(_key_type,skip_list)
 _TYPE (SkipList,_key_type), POINTER :: skip_list
 _DeclareAuxiliaryVariables(_key_type)
 WRITE(*,*) _FindNext(_head, node, 1) // Do not print the head
 PrintList: DO
 _FindNext(node, next_node, 1)
 IF (ASSOCIATED(next_node)) THEN
 WRITE (UNIT = *, FMT = _key_format, ADVANCE = "NO") _key(node)
 WRITE (UNIT = *, FMT = "(A)", ADVANCE = "NO") "--- " _FindLevel(node, node_level)
 DO level = 1, node_level
 _FindNext(node, search_node, level)
 WRITE (UNIT = *, FMT = _key_format, ADVANCE = "NO") _key(search_node)
 END DO
 WRITE(*,*) // New line
 node => next_node
 ELSE
 EXIT PrintList
 END IF
 END DO PrintList
 WRITE(*,*)
END SUBROUTINE // PrintSkipList

(PrintSkipList 3.0.4.3) ≡
 _GenerateProcedureBody(PrintSkipList)

This code is used in section 1.2.12.
\langle \text{GenericInterfaces 2.1.2}\rangle \equiv \\
\text{GenerateGenericInterface} (\text{PrintSkipList})

4 Test Program

The following is a simple test program which tests the module \textit{Skip Lists} for integer keys that are partially ordered already (to test performance under the presence of temporal and spatial data locality). It performs insertion, deletion, mixed insertion/deletion stress-test rounds, as well as just pure search, and times these for comparison:
PROGRAM Test_SkipLists
USE Precision
USE Error_Handling
USE System_Monitors
USE Initialization_Termination
USE Random_Numbers
USE Sorting_Ranking
USE Key_Types
USE Skip_Lists
IMPLICIT NONE

INTEGER (KIND = i_wp) :: key
INTEGER, DIMENSION (:, :), ALLOCATABLE :: permutation
TYPE (Data_Key_i_wp) :: data
TYPE (SkipList_Key_i_wp), POINTER :: skip_list = NULL

DeclareAuxiliaryVariables(Key_i_wp)
INTEGER :: element, n_elements, min_elements = HUGE(0), max_elements = 0, n_reps, reps
REAL :: variance, disorder, random, elapsed_time
LOGICAL :: test, found

CALL StartProgram()

ALLOCATE (skip_list)
ALLOCATE (memory)

WRITE (*, *) "Enter max_elements, page_size, max_pages, and p"
READ (*, *) max_elements, page_size, max_pages, p
WRITE (*, *) "Enter size variance, key disorder and use_search_finger"
READ (*, *) variance, disorder, use_search_finger

max_lists = 1   // Only 1 list for now
keep_duplicates = FALSE   // No duplicate keys for this timing test
trace_allocation = FALSE   // This is a timing test

CALL InitializeSkipList(skip_list)

CALL RandomUniform (random, range = (1.0 - variance, 1.0 + variance / max_elements))
n_elements = INT(random * REAL(max_elements))
   // In real life n_elements may be unknown
WRITE (*, *) "The maximum size of the list will be", n_elements
ALLOCATE (permutation(n_elements))
CALL DisorderPermutation(disorder = disorder, permutation = permutation, disorder_distribution = 'U')

WRITE (*, *) "Starting insertion of nodes!"
\_TimeCode (CALL Insertion (), 1, 1, elapsed_time)
WRITE (*, *) "Insertion took", elapsed_time, " seconds."

WRITE (*, *) "Starting search for disordered keys!"
\_TimeCode (CALL Search(), 2, 1, elapsed_time)
WRITE (*, *) "Search took", elapsed_time, " seconds."

WRITE (*, *) "Starting allocation stress test of mixed insertion/deletion!"
\_TimeCode (CALL InsertionDeletion(), 3, 1, elapsed_time)
WRITE (*, *) "Mixed insertion/deletion took", elapsed_time, " seconds."
WRITE (*, *) "Minimum and maximum number of nodes was", min_elements, max_elements

WRITE (*, *) "Inserting all nodes back and then deleting them!"
CALL Insertion()  // Insert all elements again
_TimeCode( CALL Deletion(), 4, 1, elapsed_time )
WRITE(*, *) "Deletion took:, elapsed_time, " seconds."

DEALLOCATE(permutation)
CALL DestroySkipList(skip_list, deallocate_memory = T)
CALL EndProgram()

CONTAINS

SUBROUTINE Insertion()
  DO element = 1, n_elements
    data % my_key % my_value = permutation_element
    CALL InsertInSkipList(skip_list, data)
  END DO
END SUBROUTINE Insertion

SUBROUTINE Deletion()
  DO element = 1, n_elements
    CALL DeleteFromSkipList(skip_list = skip_list, key = Key_i_wp(permutation_element),
                             data = data, deleted = found)
  END DO
END SUBROUTINE Deletion

SUBROUTINE InsertionDeletion()
  DO element = 1, n_elements
    CALL RandomUniform( key, range =/(1, n_elements/) )
    key = permutation_element  // A random choice
    CALL DeleteFromSkipList(skip_list = skip_list, key = Key_i_wp(key), deleted = found)
    IF (.NOT.found) THEN
      data % my_key = Key_i_wp(key)
      CALL InsertInSkipList(skip_list = skip_list, data = data)
    END IF
    min_elements = MIN(min_elements, n_elements)
    max_elements = MAX(max_elements, n_elements)
  END DO
END SUBROUTINE InsertionDeletion

SUBROUTINE Search()
  DO element = 1, n_elements
    CALL SearchInSkipList(skip_list = skip_list, key = Key_i_wp(permutation_element),
                           data = data, found = found)
  END DO
END SUBROUTINE Search

END PROGRAM Test_Skip_Lists

This code is used in section 1.0.1.
5 Formatting rules for HPF/F90 files

These are just some auxiliary formatting rules and useful macros I use from time to time.

\begin{verbatim}
@m  _TYPE TYPE
@m  _NULL > NULL(
@m  PRIVATE PRIVATE
@m  _SIZE(array, _kind, .)
\@IFELSE (#0, 0, INT(SIZE(array), KIND=_kind), INT(SIZE(array,_.), KIND=_kind))
@m  _MAXLOC(array, _kind, .)
\@IFELSE (#0, 0, INT(MAXLOC(array), KIND=_kind), INT(MAXLOC(array,_.), KIND=_kind))
@m  _MINLOC(array, _kind, .)
\@IFELSE (#0, 0, INT(MINLOC(array), KIND=_kind), INT(MINLOC(array,_.), KIND=_kind))
@m  _BOUND(array, _kind, .) \@IFELSE (#0, 0, INT(LBOUND(array, DIM=1), KIND=_kind),
\@INT(LBOUND(array, _.), KIND=_kind))
@m  _UBOUND(array, _kind, .) \@IFELSE (#0, 0, INT(UBOUND(array, DIM=1), KIND=_kind),
\@INT(UBOUND(array, _.), KIND=_kind))
@m  _GenericInterface(generic_name, .)
\@INTERFACE generic_name
\@MODULE PROCEDURE #.
\@END INTERFACE generic_name
@m  _Declare_i_word(.)
\@INTEGER :: #.
@m  _Declare_i_wp(.)
\@INTEGER (KIND = i_wp) :: #.
@m  _Declare_r_wp(.)
\@REAL (KIND = r_wp) :: #.
@m  _Declare_r_sp(.)
\@REAL (KIND = r_sp) :: #.
@m  _Declare_r_dp(.)
\@REAL (KIND = r_dp) :: #.
@m  _FullExtent(_rank) :: DO (DIM, 2, _rank) { : ; }
@m  _VarSequence(_variable, _start, _end)
\@var\# _start \@DO (DIM, \@EVAL (_start + 1), \@end) { , _variable@DIM } \@DO (_DIM, _rank, 1, -1) { DO _variable@DIM = _BOUND(_array, _kind, DIM = _DIM),
\@UBOUND(_array, _kind, DIM = _DIM) }
@m  _NestedLoopEnd(_rank) \@DO (_DIM, 1, _rank) { END DO }
@m  _Dummy(.)
@m  _DisplayArray(message, array)
\@IF (SIZE(array) \leq 20) THEN
\@WRITE(message_print_unit, "(A)") message
\@WRITE(message_print_unit, "(2005.2)") array
\@ENDIF
\end{verbatim}