A Portable Ada Implementation of Index Sequential Input-Output

-Part 1-

Karl Kurbel and Wolfram Pietsch*

Editor's note: Part 2 will appear in the next issue of Ada Letters. The appendix is included in this part.

Keywords

Index sequential input-output, file management, B-tree, total index, data encapsulation, packages

1 Summary

Efficient and easy access to file elements is of crucial importance for many applications in business and industrial data processing. Ada's features for file organization and access methods are rather poor, however. Index sequential input-output, for example, is not supported. Thus user defined keys cannot be employed to access file elements. In order to improve Ada's facilities, a completely portable generic package for index sequential input-output was developed. We describe design considerations for the package interface which was specified in close analogy to the predefined package *direct_io*. The package implementation based on a B-tree structure is outlined. A brief assessment of the implementation strategy is given. With regard to performance aspects, the results of runtime measurements are presented. Finally, a method is provided with which the user may influence access efficiency. The specification of the package *index_sequential_io* is listed in the appendix.

* Authors' addresses:

Karl Kurbel, Universität Dortmund, Fachbereich Wirtschafts- -und Sozialwissenschaften, Lehrstuhl für Betriebsinformatik Postfach 50 05 00, D-4600 Dortmund 50

Wolfram Pietsch, Universität Bielefeld, Fakultät für Wirtschaftswissenschaften, Postfach 86 40, D-4800 Bielefeld 1

2 Ada File Management and User Requirements

For commercial applications, file organization and access methods are of crucial importance. Therefore any adaequate language has to provide convenient features for file management as do Cobol and Pl/1, for example. Considering Ada's power in other fields, the language support of file management, however, is relatively poor. The designers seem to have treated input-output and file management with subordinate priority and left the implementation of convenient language features to the user.

The current Ada standard of 1983 defines two kinds of files: sequential files and direct files [2]. The corresponding file types and access operations are provided by the generic packages *sequential_io* and *direct_io*. A sequential file is regarded as a sequence of values which are accessed in the order of their appearance. A direct file is considered as a set of elements in consecutive file positions which are numbered 1,2,3,... . Access to individual elements may be achieved in any order by means of the position number (index). Direct files in Ada are comparable to Cobol files with *organization relative* which may be accessed through record numbers 1,2,3,... or in sequential order [1].

Identifying file elements by natural numbers is a very efficient but not at all user-friendly way of file access. User defined keys cannot immediately be employed as file indices unless they form a set of contiguous natural numbers 1,2,3,..., too. This is rarely the case, however. Consider, for example, a master file of 1000 parts with the part number serving as the record key. Part numbers are often not represented by numbers 1,2,...,1000. Instead, they may contain alphanumeric characters or have a hierarchical structure which reveals additional information. Since such a key cannot be employed in Ada to identify file elements directly, the user would have to administer and supervise the association of record keys and file indices himself.

None of the common programming languages in business data processing forces the user to perform file management at such a low level of abstraction. Instead, the index sequential file organiis provided. This organization is often referred to as ISAM (index sequential access method).

From the user's view the index sequential organization can be characterized as a form of file organzation which permits efficient sequential access as well as direct access to file elements by means of user defined keys. ANS Cobol and most implementations of Pl/1, for example, offer index sequential files. In Cobol, any elementary item of a data record can be declared as the record key. For sequential access, records are transferred in ascending order of the record keys. For direct access, records are identified by the key item and may be transferred in any order. Direct access includes reading, rewriting, insertion, and deletion of a record.

In consideration of the practical importance of index sequential files, the authors designed and implemented a generic package *index_sequential_io*. The Ada package concept proved to be a very powerful feature for language enhancements. Since hardware independence and portability seemed highly desirable the implementation is completely based on standard Ada elements. In particular, all input-output operations involving external files are implemented by means of the generic package *direct_io*.

Because of Ada's powerful concepts our ISAM implementation is very flexible as compared to Cobol or Pl/1. This will be demonstrated below in detail. In the following chapters, the user interface and the implementation within the package body are described. We will discuss design considerations, efficiency aspects, and implementation decisions which had to be made. Finally, a suggestion is made as to how the proposed data structure could be used to implement a database management system.

3 User Interface for Index Sequential Input-Output

3.1 Design Considerations

The ANSI Ada standard represents a language definition which is much more comprehensive and stringent than definitions of other programming languages. It comprises features even like low level input-output and representation of the hardware interfaces. From this underlying idea, neither compiler subsets nor extensions are permitted by the trademark holders of Ada®.

Index sequential input-output should in our opinion be part of the Ada language. As the standard does not include it, however, a user-developed package remains the only appropriate way of language enhancement. For reasons of uniformity and user-friendliness, such a package should conform as closely as possible to the structure of the standardized input-output packages, *sequential_io* and *direct_io*.

Upward compatibility can be observed among those two packages; i.e. the package interface of *direct_io* includes the interface of *sequential_io* as a subset. Thus *direct_io* could be used in exactly the same way as *sequential_io* as long as sequential access to file elements is required. The goal of upward compatibility was pursued as much as possible in the design of the new io-package in the sense that *index_sequential_io* should be applicable with the same effects as *direct_io* if natural numbers 1,2,3... are used as key elements.

An index sequential file organization has to allow both random access to file elements by means of key elements and sequential access in the order of the keys. A unique association of key elements to data elements and a greater-smaller relation which specifies the order of the key elements are necessary for this purpose. The index sequential structure of a file is thus determined by three kinds of information: the type of the data elements, the type of the keys associated with the data elements, and the order of the keys.

Translated into Ada notation, three generic actual parameters have to be supplied for a particular instantiation of the package *index_sequential_io*: the *element_type*, the *key_type*, and the operator "<" for the comparison of keys. In this way the user has complete freedom to specify key types as complex as he desires, for example hierarchically structured keys. Considering the history of advanced access methods like HISAM, HIDAM etc. [8], the need for hierarchically structured keys becomes immediately obvious. Even variant records mey serve as key types. The user is not restricted (e.g. to string types as in other languages) as long as he can supply a comparison function "<" for the *key_type* and ensure transitivity of the key values. Hence the minimal information in the generic formal part of an index sequential input-output package has to include the following:

```
GENERIC
   TYPE element _type IS PRIVATE;
   TYPE key_type       IS PRIVATE;
   WITH FUNCTION "<" (left, right : IN key_type) RETURN boolean;
PACKAGE index_sequential_io IS
```


At first sight this declaration seems to conform to the goal of upward compatibility with respect to *direct_io*. If an integer type is chosen as *key_type* and the predefined operator "<" for integer comparison is supplied with it, an instance of *index_sequential_io* can be utilized with the same calls and effects as an instance of *direct_io* for the same *element_type*.

There is one restriction, however. Whereas the indices of a direct file form a set of contiguous natural numbers, the keys of an index sequential file are generally noncontiguous. An instance of *direct_io* allows the user to output data elements in sequential order or to add data elements at the end of a file without explicitly specifying the file index (sequential output). In order to preserve this property, an extension of the generic part by two more parameter declarations would have been necessary -- by a successor function and by the smallest possible value of *key_type*:

```
GENERIC
   TYPE element_type IS PRIVATE;
   TYPE key_type     IS PRIVATE;
   WITH FUNCTION "<" (left, right : IN key_type) RETURN boolean;
   WITH FUNCTION succ (key : IN key_type) RETURN key_type;
   smallest_key :    IN key_type;
PACKAGE index_sequential_io IS
```

An instantiation of *index_sequential_io* with *smallest_key => 1* and *succ => count'SUCC* would now yield a package which is completely upward compatible with an instance of *direct_io*.

We decided not to pursue upward compatibility to this extent. Modification of our package with respect to the additional parameters would have been easy. However, this extension would impose a substantial problem upon the user, requiring him to define a sequential order for his key type; i.e. he would have to supply an implementation of the successor function. This could be extremely difficult for noncontiguous key types and would result in a considerable loss of user-friendliness. It should be mentioned, however, that the package can of course be extended for a particular environment if the user is able to provide a successor function.

None of the other languages with index sequential facilities permit the user to output file elements without reference to a record key, although the reference may be implicit. This means that a value of the record key has to exist when a write statement is executed but there is not necessarily an explicit reference to it. We decided to follow this custom and require the user to specify the key element whenever output to a file is to be performed. Of course, sequential input is permitted as in any other index sequential implementation.

In addition to the parameters *element_type*, *key_type*, and the smaller function, three more parameters were included in the generic formal part of *index_sequential_io*. They will be discussed later on in detail.

3.2 Specification of Package *index_sequential_io*

As upward compatibility was desired the specification part of the package *index_sequential_io* is built up in the same way as the specification of *direct_io*. The packages export:

(1) types
(2) exceptions
(3) operations for file management
(4) input-output operations

There are very few differences between the two packages. We will only discuss those items of *index_sequential_io* which differ from the specification of *direct_io* as described in the ANSI Ada reference manual [2].Most variations relate to input-output operations. A complete listing of the specification part is given in the appendix.

ad (1): The types *file_type* and *file_mode* are left unchanged. The integer types *count* and *positive_count* used for file indices in *direct_io* are omitted because the *key_type* for index sequential organization is supplied by instantiation.

ad (2): All exceptions of *direct_io* are also available. One additional exception, *key_error*, is defined. It is raised if the key to be used in a read or delete operation does not exist in the file.

ad (3): Operations for file management - *create, open, close, delete, reset, mode, name, form,* and *is_open* - are exactly the same as defined by the Ada standard.

ad (4): Input-output operations have been modelled in close analogy to the *direct_io* package. There are some modifications, however, which are primarily because access to data elements is achieved by keys and not by simple file indices. Parameter and function types therefore refer to *key_type* instead of *count* or *positive_count*.

One major difference has already been mentioned: There is no sequential write operation. The reasons have been discussed in the preceding chapter. Another difference relates to the *current key* associated with an index sequential file. The notion of a current key is defined analogous to the "current index" of *direct_io* [2]:

An open index sequential file has a current key which is the key that will be used by the next sequential read operation. When an existing nonempty index sequential file is opened, the current key is set to the smallest key value of the file.

The current key is used and changed explicitly or implicitly by input-output operations as will be shown below. However, it is not automatically advanced to the next value after a direct read or write operation as is the current index after the respective operations of *direct_io*.

Considering these modifications the input-output operations are specified as follows:

```
PROCEDURE read (file : IN file_type;
                item : OUT element_type;
                from : IN key_type);

PROCEDURE read (file : IN file_type;
                item : OUT element_type);
```

In the first form, the current key is set to the value of parameter *from*; then the data element associated with this value

is returned in the parameter *item*. In the second form, the data element associated with the current key is returned in *item*; then the current key is advanced to the next value of *key_type* in accordance with the specified smaller function.

The exception *key_error* is raised if no data element with key value of *from* exists in the file; *mode_error* and *data_error* are raised under the same conditions as in *direct_io*.

```
PROCEDURE write (file : IN file_type;
                 item : IN element_type;
                 to   : IN key_type);
```

If the value of parameter *to* already exists as a key of the file the current key is set to this value. If it does not exist a new entry is created for this value. Then, in both cases, the value of parameter *item* is written to the file and connected with the key value of *to*. The exceptions *mode_error* and *use_error* are raised as in *direct_io*.


```
PROCEDURE set_key (file : IN   file_type;
                   to   : IN   key_type;
                   ok   : OUT  boolean);
```

If a key with the value of parameter *to* exists in the file the current key is set to this value; in the parameter *ok*, the value *true* is returned. If no key with the value of *to* exists *false* is returned in *ok*. The procedure operates on a file with any mode.

```
FUNCTION key (file : IN file_type) RETURN key_type;
```

If the file is not empty the current key is returned; otherwise the exception *end_error* is raised. The file may be of any mode.

```
FUNCTION size (file : IN file_type) RETURN integer;
```

The current size of the external file associated with the given file object is returned. The file may be of any mode.

```
FUNCTION end_of_file (file : IN file_type) RETURN boolean;
```

The function returns *true* if no more data elements can be read sequentially; otherwise it returns *false*. The exception *end_error* is raised as in *direct_io*.

All of the above input-output operations have corresponding counterparts in the package *direct_io*. The only additional operation is deletion of an existing key together with the associated data element:

```
PROCEDURE delete (file : IN OUT file_type;
                  key  : IN     key_type);
```

After deletion the current key is advanced to the next value of *key_type* unless the deleted element was the last element of the file. In this case, the current key is set to the value of the preceding key element.

The exception *key_error* is raised if the value of parameter *key* does not exist in the file; *mode_error* is raised if the file mode is not *inout_file* or *out_file*.

3.3 An Example

For illustration of the package interface, a master part file is considered which is to be kept as an index sequential file. A key element consists of a part identification and a variant number. The file is to be organized in ascending order of part id's and - within the same part id - in descending order of variant numbers if variants of a part exist. This small example of a variation of the order relations demonstrates that the user is free to define any order of keys if he supplies an appropriate smaller function.

```
WITH index_sequential_io; USE index_sequential_io;

PROCEDURE user_program IS

--  key type  --

    TYPE part_key IS RECORD
                        part_id    : string (1..7);
                        variant_no : integer := 0;
                     END RECORD;

--  element type  --

    TYPE part_info  IS RECORD
                         part_name : string (1..12);
                         inventory : float := 0.0;
                         orders    : float := 0.0;
                       END RECORD;

--  user-defined order relation  --

    FUNCTION smaller (left, right : IN part_key) RETURN boolean IS
    BEGIN
       IF left.part_id < right.part_id THEN
          RETURN true;
       ELSIF left.part_id = right.part_id
             AND left.variant_no > right.variant_no THEN
          RETURN true;
       ELSE
          RETURN false;
       END IF;
    END smaller;

--  package instantiation  --

    PACKAGE part_io IS NEW index_sequential_io
                          (element_type => part_info,
                           key_type     => part_key,
                           "<"          => smaller);
     .
     .
     .

    part_file       : part_io.file_type;
    ok              : boolean;
    current_part    : part_key;
    current_element : part_info;

--  creation and output to master part file  --

    create (file => part_file,
            mode => inout_file,
            name => "all_my_parts");

    write (file => part_file,
           item => ("screwdriver ", 0.0, 0.0),
           to   => ("abc4711", 0));
    .
    .
    .

--  processing of parts with part_id "abc4711" to "abx2111"  --

    set_key (file => part_file,
             to   => ("abc4711", 99),
             ok   => ok);

    IF ok THEN
       WHILE key (file => part_file).part_id <= "abx2111" LOOP
          current_part := key (file => part_file);
          read (file => part_file,
                item => current_element);
          .
          .
          .
       END LOOP;
    END IF;

END user_program;
```

```
-----------------------------------------------------------------------
-----------------------------------------------------------------------
--                                                                   --
-- Package Specification  >> INDEX _ SEQUENTIAL _ IO <<              --
--                                                                   --
-- Author :   Wolfram Pietsch                                        --
--                                                                   --
-- Date   :   February 28th, 1985                                    --
--                                                                   --
-----------------------------------------------------------------------
-----------------------------------------------------------------------

   WITH system, io_exceptions, direct_io;

GENERIC

   TYPE key_type IS PRIVATE;
   TYPE element_type IS PRIVATE;
      -- an io-element consists of key and corresponding element

   WITH FUNCTION  "<" (left, right : IN key_type) RETURN boolean;
      -- function to provide a user-defined order of the elements

   traverse_stack_size : IN natural := 4;
   max_files           : IN natural := 2;
      -- constants for storage and access optimization

PACKAGE index_sequential_io IS

   index_page_size : natural;

   TYPE file_type IS LIMITED PRIVATE;
      -- type of objects which may be associated with an external
      -- file

   TYPE file_mode IS (in_file, inout_file, out_file);
      -- in_file    : only INPUT
      -- out_file   : only OUTPUT
      -- inout_file : INPUT or OUTPUT
```

```
-----------------------------------------------------------------------
-- File management ----------------------------------------------------
-----------------------------------------------------------------------

   PROCEDURE create (file : IN OUT file_type;
                     mode : IN file_mode := inout_file;
                     name : IN string := "";
                     form : IN string := "");
      -- creates an external file with the name "name" and associates
      -- the file object "file" with it.

   PROCEDURE open (file : IN OUT file_type;
                   mode : IN file_mode;
                   name : IN string;
                   form : IN string := "");
      -- associates "file" with the external file specified by "name".

   PROCEDURE close (file : IN OUT file_type);
      -- disconnects the current association of "file" with an external file

   PROCEDURE delete (file : IN OUT file_type);
      -- disconnects the current association of "file" with an external
      -- file; the external file ceases to exist.

   PROCEDURE reset (file : IN OUT file_type;
                    mode : IN file_mode);
      -- sets the current key to the smallest key of the associated
      -- external file and changes the file mode to "mode".

   PROCEDURE reset (file : IN OUT file_type);
      -- sets the current key to the smallest key of the associated
      -- external file.

   FUNCTION  mode (file : IN file_type) RETURN file_mode;
      -- returns the current file mode.

   FUNCTION  name (file : IN file_type) RETURN string;
      -- returns the name which was used to create the external
      -- file associated with "file".

   FUNCTION  form (file : IN file_type) RETURN string;
      -- returns the form used in the current association of "file"
      -- with an external file.

   FUNCTION  is_open (file : IN file_type) RETURN boolean;
      -- returns true if "file" is currently associated with an exter-
      -- nal file, false otherwise.
```

```
-----------------------------------------------------------------------
-- Input and output operations -----------------------------------------
-----------------------------------------------------------------------

   PROCEDURE read (file : IN file_type;
                   item : OUT element_type;
                   from : IN key_type);
      -- returns in parameter "item" the element with key "from".

   PROCEDURE read (file : IN file_type;
                   item : OUT element_type);
      -- returns in parameter "item" the element associated with the
      -- current key and advances the current key to the next key value.

   PROCEDURE write (file : IN file_type;
                    item : IN element_type;
                    to   : IN key_type);
      -- associates the value of "item" with the value of "to" if "to"
      -- already exists in the associated external file; else writes a
      -- key with the value of "to" and associates the value of "item"
      -- with it.

   PROCEDURE delete (file : IN file_type;  -- not yet implemented
                     key  : IN key_type);
      -- deletes the value of "KEY"  and its associatetd element from
      -- the "file".

-----------------------------------------------------------------------
-- Input-output control functions --------------------------------------
-----------------------------------------------------------------------

   PROCEDURE set_key (file : IN file_type;
                      to   : IN key_type;
                      ok   : OUT boolean);
      -- attempts to set the current key to the value of "to"; if
      -- successful, true is returned in "ok", otherwise false
      -- (i.e. no such key exists in the associated external file).

   FUNCTION  key (file : IN file_type) RETURN key_type;
      -- returns the current key.

   FUNCTION  size (file : IN file_type) RETURN integer;
      -- returns the current size of the external file associated
      -- with "file".

   FUNCTION  end_of_file (file : IN file_type) RETURN boolean;
      -- returns true if according to the provided "<" relation no key
      -- greater than the current key exists in the external file
      -- associated with "file".
```



```
-----------------------------------------------------------------------
-- Exceptions ----------------------------------------------------------
-----------------------------------------------------------------------

   key_error     : EXCEPTION;
      -- the value of the key argument is not valid.
   storage_error : EXCEPTION RENAMES standard.storage_error;
      -- not enough storage available to open a file.
   status_error  : EXCEPTION RENAMES io_exceptions.status_error;
      -- operation requested only allowed for opened or not opened file,
      -- respectively.
   mode_error    : EXCEPTION RENAMES io_exceptions.mode_error;
      -- invalid file_mode for requested operation.
   name_error    : EXCEPTION RENAMES io_exceptions.name_error;
      -- invalid "name" string specified.
   use_error     : EXCEPTION RENAMES io_exceptions.use_error;
      -- file capacity exceeded, not existing or already existing file,
      -- respectively.
   device_error  : EXCEPTION RENAMES io_exceptions.device_error;
   end_error     : EXCEPTION RENAMES io_exceptions.end_error;
      -- attempt to read sequentially after end_of_file.
   data_error    : EXCEPTION RENAMES io_exceptions.data_error;
      -- type or parameter used in current instantiation of
      -- index_sequential_io  does not match type or parameter in the
      -- instantiation used for the currently associated external file.

-----------------------------------------------------------------------
-- Invisible specification part ----------------------------------------
-----------------------------------------------------------------------

PRIVATE

   index_blk_factor : CONSTANT integer := 100;

   SUBTYPE natural IS integer RANGE 0 .. integer'last;

   PACKAGE index_page IS ---------------------------------------------

      TYPE index_item IS
         RECORD
            min_key   : key_type;
            data_pos  : natural;
            succ_keys : natural;
         END RECORD;

      SUBTYPE index_pos_range        IS integer RANGE 1 .. index_blk_factor;
      SUBTYPE index_pointer_range    IS integer RANGE 0 .. index_blk_factor;
      TYPE access_state              IS (empty, root, node, deleted);
```

```
TYPE index_page_type (entries   : index_pos_range := index_blk_factor;
                      allocated : access_state    := empty)
   IS PRIVATE;

--------------------------------------------------------------------
-- Operations on index_page_type -----------------------------------
--------------------------------------------------------------------

FUNCTION  pred_pos (page : IN index_page_type) RETURN natural;
   -- returns the file position number of the index page which
   -- preceeds "page" (with keys smaller than the keys of "page").

FUNCTION  pos (page : IN index_page_type) RETURN natural;
   -- returns the number of the file postion occupied by index page
   -- "page" in the underlying direct file.

FUNCTION  succ_pos (page : IN index_page_type) RETURN natural;
   -- returns the file position number of the index page which
   -- succedes "page" according to the current index position of
   -- "page" (with keys smaller than the current key of "page").

FUNCTION next_deleted (page : IN index_page_type) RETURN natural;
   -- returns the file position number of the next deleted index
   -- page in the linked list.

FUNCTION  data_pos (page : IN index_page_type) RETURN natural;
   -- returns the file position number of the element associated
   -- with the current key of "page".

PROCEDURE set (page : IN OUT index_page_type;
               pos  : IN index_pointer_range);
   -- sets the key in position "pos" of "page" to be current for
   -- index page "page".

PROCEDURE set_pos (page : IN OUT index_page_type;
                   pos  : IN natural);
   -- changes for the current key in "page" the reference to the
   -- associated element to the file position number "pos".

FUNCTION  pointer (page : IN index_page_type)
   RETURN index_pointer_range;
   -- returns the index position of the current key in "page".

FUNCTION  curr_key (page : IN index_page_type) RETURN key_type;
   -- returns the current key in "page".

FUNCTION  next_key (page : IN index_page_type) RETURN key_type;
   -- returns the next greater key than the current key in "page".
```

```
FUNCTION  entries (page : IN index_page_type)
   RETURN index_pointer_range;
   -- returns the current number of index item entries in "page".

FUNCTION  item (page : IN index_page_type;
                pos  : IN index_pos_range) RETURN index_item;
   -- returns the index item at position "pos" in "page".

FUNCTION  end_of_page (page : IN index_page_type) RETURN boolean;
   -- returns true if the current index position is equal to the
   -- number of entries in "page", false otherwise.

FUNCTION  page_full (page : IN index_page_type) RETURN boolean;
   -- returns true if the number of entries in "page" is the
   -- largest possible one, false otherwise.

FUNCTION  "<" (left  : IN key_type;
               right : IN index_page_type) RETURN boolean;
   -- returns true if all keys in "right" are greater
   -- than the value of "left", false otherwise.

FUNCTION  ">" (left  : IN key_type;
               right : IN index_page_type) RETURN boolean;
   -- returns true if all keys in "right" are smaller
   -- than the value of "left", false otherwise.

PROCEDURE search_page (page  : IN OUT index_page_type;
                       key   : IN key_type;
                       found : OUT boolean);
   -- searches "page" for the value of "key" and sets the
   -- current position to the positon of the entry whose key
   -- value is immediately greater than or equal to "key".

PROCEDURE add_item (page : IN OUT index_page_type;
                    item : IN index_item);
   -- enters the index item "item" into "page" maintaining the
   -- ascending order of keys.

PROCEDURE split_page (split      : IN index_page_type;
                      raise_item : IN index_item;
                      left       : OUT index_page_type;
                      new_item   : OUT index_item;
                      right      : OUT index_page_type);
   -- divides index page "split" at the middle position into two
   -- pages "left" and "right", returns in parameter "new_item"
   -- the index item in the middle of the page, and inserts
   -- "raise_item" into the correct left or right page.
```

```
   FUNCTION  new_root (pred_pos : IN natural;
                       item     : IN index_item)
      RETURN index_page_type;
      -- returns a root index page containing one entry with the
      -- value of "item" and the file position number "pred_pos"
      -- of the preceeding index page.

   FUNCTION  deleted (page         : IN index_page_type;
                      next_deleted : IN natural)
      RETURN index_page_type;
      -- returns a deleted index page with the file position number
      -- of "page".

   FUNCTION  root_allocated (page : IN index_page_type)
      RETURN boolean;
      -- returns true if "page" is a root, false otherwise.

   FUNCTION  leaf_allocated (page : IN index_page_type)
      RETURN boolean;
      -- returns true if "page" is a leaf, false otherwise.

PRIVATE

   TYPE index_table IS ARRAY (index_pos_range RANGE <>) OF index_item;

   TYPE index_page_type (entries   : index_pos_range := index_blk_factor;
                         allocated : access_state := empty) IS
      RECORD
         CASE allocated IS
            WHEN root | node =>
               page_pos  : natural;
               pred_keys : natural;
               pointer   : index_pointer_range;
               item      : index_table (1 .. entries);
            WHEN deleted =>
               pos : natural;
               next_deleted : natural:= 0;
            WHEN OTHERS =>
               NULL;
         END CASE;
      END RECORD;

   PRAGMA pack (index_page_type);

END index_page;
```



```
PACKAGE relative_io IS -----------------------------------------------

   io_buffer_size    : CONSTANT natural := 2000;
   TYPE data_item (present : boolean := true) IS
      RECORD
         CASE present IS
              WHEN true    => elem  : element_type;
              WHEN others  => next_deleted : natural := 0;
         END CASE;
      END RECORD;

   PACKAGE index_io IS NEW direct_io (
                          element_type => index_page.index_page_type);
      -- IO-package for the nodes of the B-tree.

   PACKAGE data_io  IS NEW direct_io (
                          element_type => data_item);
      -- IO-package for the data elements associated with the
      -- entries of the B-tree.

   ------------------------------------------------------------------
   -- EXTERNAL-IO PROCEDURES ----------------------------------------
   ------------------------------------------------------------------

   -- File management -----------------------------------------------

   PROCEDURE create (file : IN OUT file_type ;
                     name : IN string);
      -- creates one index and one data file.

   PROCEDURE open (file : IN OUT file_type; name : IN string);
      -- opens the index and the data file.

   PROCEDURE close (file : IN OUT file_type);
      -- closes the index and the data file.

   PROCEDURE delete (file : IN OUT file_type);
      -- deletes the data and the index file.

   FUNCTION  name  (file : IN file_type) RETURN string;
      -- returns the name used to create the external file
      -- associated with "file".
```

```
-- IO Procedures ------------------------------------------------

PROCEDURE get (  f   : IN OUT file_type;
              page : OUT index_page.index_page_type;
               pos : natural);
   -- reads an index page from position "pos" of the index file.

PROCEDURE get_root (f : IN OUT file_type;
                    r : OUT index_page.index_page_type);
   -- reads the root page from the index file.

PROCEDURE get ( f   : IN OUT file_type;
             data : OUT data_item;
              pos : natural);
   -- reads the data element "data" from position "pos" of the
   -- data file.

PROCEDURE put   (f    : IN OUT file_type;
                page : IN OUT index_page.index_page_type;
                pos  : OUT natural);
   -- writes "page" to the next free position in the index file
   -- (linked list) and returns that position in "pos".

PROCEDURE put_root (f : IN OUT file_type;
                    r : IN index_page.index_page_type);
   -- writes the root "r" to the index file.

PROCEDURE put  (f    : IN OUT file_type;
               data : IN  data_item;
               pos  : OUT natural);
   -- writes the data element "data" to the next free position
   -- in the data file (linked list) and returns the position
   -- in "pos".

PROCEDURE change (f  : IN OUT file_type;
               page : IN OUT index_page.index_page_type;
                pos : IN natural);
   -- rewrites the index page "page" to the position "pos" in the
   -- index file.

PROCEDURE change_root (f : IN OUT file_type;
                       r : IN index_page.index_page_type);
   -- rewrites "r" as the root page to the index file.

PROCEDURE change (f    : IN OUT file_type;
                 data : IN data_item;
                 pos  : IN natural);
   -- rewrites data item "data" to position "pos" of the
   -- data file.

PROCEDURE delete (f    : IN OUT file_type;
                 page : IN OUT index_page.index_page_type);
   -- deletes the index page "page" virtually from the index file
   -- by rewriting index page "page" with page.access_state :=
   -- deleted and adding the file position number to the linked
   -- list of deleted index pages for further usage.

PROCEDURE delete (f    : IN OUT file_type;
                 data : IN OUT data_item;
                  pos : IN natural);
   -- deletes the data element "data" virtually from the data
   -- file by rewriting the data element "data" with data.deleted
   -- := yes and adding the file position number to the linked
   -- list of deleted data items for further usage.

PRIVATE
   index_file_prefix : CONSTANT string := "IND.";
   index_file_form   : CONSTANT string := "IV";
   data_file_prefix  : CONSTANT string := "DAT.";
   data_file_form    : CONSTANT string := "IV";
   root_pos          : CONSTANT integer:= 1;
END relative_io;

PACKAGE b_tree IS ----------------------------------------------------

   FUNCTION empty (f : IN file_type) RETURN boolean;
      -- returns true if the file associated with "f" is empty,
      -- false otherwise.

   PROCEDURE allocate_root (f : IN OUT file_type);
      -- retrieves the root index page from the index file and
      -- initializes the traverse stack for traversal of the B-tree.

   FUNCTION page (f : IN file_type) RETURN index_page.index_page_type;
      -- returns the current page of the B-tree.

   FUNCTION key (f: IN file_type) RETURN key_type;
      -- returns the current key of the current page of the B-tree.

   PROCEDURE create_root (f   : IN OUT file_type;
                         key : IN key_type;
                         pos : IN natural);
      -- initializes the B-tree with a new root containing one entry.
```

```
   PROCEDURE locate_key (f     : IN OUT file_type;
                         key   : IN key_type;
                         found : OUT boolean);
      -- searches the B-tree for "key", allocates the current page
      -- according to the value of "key", and returns in parameter
      -- "found" the value true if the "key" was found in the B-tree,
      -- false otherwise.

   PROCEDURE add_key (f        : IN OUT file_type;
                      key      : IN key_type;
                      data_pos : IN natural);
      -- adds the requested "key" with pointer to the associated
      -- data element to the B-tree and invokes "split_page" if
      -- necessary.

   PROCEDURE set_first (f : IN OUT file_type);
      -- traverses the B-tree so that afterwards the current key is
      -- the smallest one (according to the "<" relation).

   PROCEDURE set_succ (f : IN OUT file_type);
      -- traverses the B-tree and advances the current key to the
      -- next greater key (according to the "<" relation).

   PROCEDURE delete_key (f   : IN OUT file_type; -- not yet
                         key : IN     key_type); -- implemented
      -- deletes the requested "key" in the B-tree
END b_tree;

PACKAGE traverse_stack IS ------------------------------------------

   SUBTYPE elem_type IS index_page.index_page_type;
   TYPE stack_type   IS PRIVATE;
   stack_overflow, stack_underflow : EXCEPTION;

   PROCEDURE open (stack : IN OUT stack_type);
   FUNCTION empty (stack : IN stack_type) RETURN boolean;
   PROCEDURE push (stack : IN OUT stack_type;
                   elem  : IN elem_type);
   PROCEDURE pop  (stack : IN OUT stack_type;
                   elem  : OUT elem_type);
   PROCEDURE swap (stack : IN OUT stack_type;
                   elem  : IN elem_type);
   FUNCTION bottom (stack : IN stack_type) RETURN elem_type;
   FUNCTION top (stack : IN stack_type) RETURN elem_type;
   PROCEDURE mark (stack : IN OUT stack_type);
   PROCEDURE reset_to_mark (stack : IN OUT stack_type);

PRIVATE
   TYPE elem_block IS
      RECORD
         elem : elem_type;
      END RECORD;
   TYPE elem_ptr IS ACCESS elem_block;
   TYPE stack_table IS ARRAY (1 .. traverse_stack_size) OF
      elem_ptr;
   TYPE stack_type IS
      RECORD
         empty      : boolean := true;
         marked     : boolean := false;
         mark, this : natural;
         pos        : stack_table;
      END RECORD;
   FOR elem_ptr'storage_size USE max_files *
       traverse_stack_size *
       ((elem_type'SIZE / system.storage_unit) + 1);
      -- allocates memory for "max_files" traverse stacks
      -- with sizes "traverse_stack_size".
END traverse_stack;

TYPE file_form IS (reallocate, dispose);
TYPE file_state IS (empty, open, closed);

TYPE file_control_block IS
   RECORD
      state       : file_state := empty;
      mode        : file_mode := inout_file;
      form        : file_form := dispose;
      eof         : boolean := true;
      index_file  : relative_io.index_io.file_type;
      index_size  : integer;
      data_file   : relative_io.data_io.file_type;
      data_size   : integer;
      index_stack : traverse_stack.stack_type;
   END RECORD;

   TYPE file_type IS ACCESS file_control_block;

FOR file_type'storage_size USE max_files *
   ((file_control_block'SIZE/system.storage_unit) + 1);
   -- allocates memory for "max_files" file control blocks.

-----------------------------------------------------------------------
END index_sequential_io;
-----------------------------------------------------------------------
-----------------------------------------------------------------------
```

A Portable Ada Implementation of Index Sequential Input-Output

-Part 2-

Karl Kurbel and Wolfram Pietsch*

Editor's note: Part 1, including the appendix, appeared in the previous issue of Ada Letters.

Keywords

Index sequential input-output, file management, B-tree, total index, data encapsulation, packages

1 Summary

Efficient and easy access to file elements is of crucial importance for many applications in business and industrial data processing. Ada's features for file organization and access methods are rather poor, however. Index sequential input-output, for example, is not supported. Thus user defined keys cannot be employed to access file elements. In order to improve Ada's facilities, a completely portable generic package for index sequential input-output was developed. We describe design considerations for the package interface which was specified in close analogy to the predefined package *direct_io*. The package implementation based on a B-tree structure is outlined. A brief assessment of the implementation strategy is given. With regard to performance aspects, the results of runtime measurements are presented. Finally, a method is provided with which the user may influence access efficiency. The specification of the package *index_sequential_io* is listed in the appendix.

* Authors' addresses:

Karl Kurbel, Universität Dortmund, Fachbereich Wirtschafts- -und Sozialwissenschaften, Lehrstuhl für Betriebsinformatik Postfach 50 05 00, D-4600 Dortmund 50

Wolfram Pietsch, Universität Bielefeld, Fakultät für Wirtschaftswissenschaften, Postfach 86 40, D-4800 Bielefeld 1

4 Implementation of Index Sequential Input-Output

4.1 The Underlying Data Structure

Implementation of the *index_sequential_io* package is completely based on standard Ada elements. For input-output to files, resources of the package *direct_io* are used. The data structure implementing the index sequential file organization can thus be viewed as a mapping of virtual index sequential files onto direct files.With respect to implementation methods, four were taken into consideration: a hashing technique, conventional ISAM implementation, binary trees, and B-trees.

(1) Hashing

If the distribution of the key values is known in advance hashing can be a very efficient technique for random access. The user has to supply a hashing function $h : k \rightarrow i$ which associates an index i of a direct file with each key value k. Sequential access is not part of the method. It has to be implemented separately, e.g. by establishing a linked list of the data elements. Hashing can be very inefficient if the actual distribution of keys differs considerably from the expected one. Furthermore, specification of a hashing function might be an extremely difficult problem if complex key types are permitted as in our implementation. Due to its lack of flexibility and ease of use, hashing was rejected.

(2) Conventional ISAM

The conventional ISAM technique as initially developed for IBM computer systems is defined with close reference to hardware characteristics [5]. Because of the very low abstraction level, the ISAM technique is very efficient. On the other hand, an implementation using *direct_io* should naturally apply abstract resources and not refer to hardware features. Going back to those would be contrary to common sense and detrimental to the goal of portability.

(3) Binary Trees

An index sequential organization based on a binary tree can be completely hardware independent but may pose severe efficiency problems. If the key sequence is unfavorable when the file is written (e.g. ascending or descending) the tree degenerates to a linear list where searching is performed sequentially. Frequent reorganizations are necessary. If a balanced tree is chosen the structure is self-reorganizing but the effort as such remains. Updates can be extremely time consuming if an adverse key requires complete rebalancing of the tree[7].

(4) B-Trees

A more efficient, and to a large degree hardware independent structure is the B-tree [3]. Efficiency of this structure is restricted by the size of the input-output buffers. It can be tuned to a particular environment by setting user parameters without loss of portability. The B-tree structure can be used to implement an index sequential file organization in two ways, either as a VSAM-like structure or as a pure file index structure.

In the VSAM implementation of IBM [9] the B-tree serves as a decision tree. Only the leaves of the tree contain data elements, together with their keys. The leaves are called control intervals and are usually blocks of maximum input-output buffer size. The B-tree is used to find the right control interval as fast as possible. The data elements are part of the tree. This kind of organization is particularly advantageous for sequential access because the data elements within a control interval are already arranged in the desired sequence.

On the other hand, the B-tree may be employed to store only the key elements whereas the data elements are kept in a separate file. This sort of a B-tree is called a total index. The nodes of the tree contain keys and references to the separately stored data elements.

In figure 1 an example of a total index is depicted where integers are used as keys. Every node contains a set of keys which did appear in the order 9, 20, 55, 1, 8, 7, 6, 5, 10, 21, 19, 30, 40 when the tree was constructed. Associated with each key are not only references to the respective left and right sons but also a reference d_i to the corresponding data element in the data file. The maximum number of keys per node is restricted to 3 in this example.

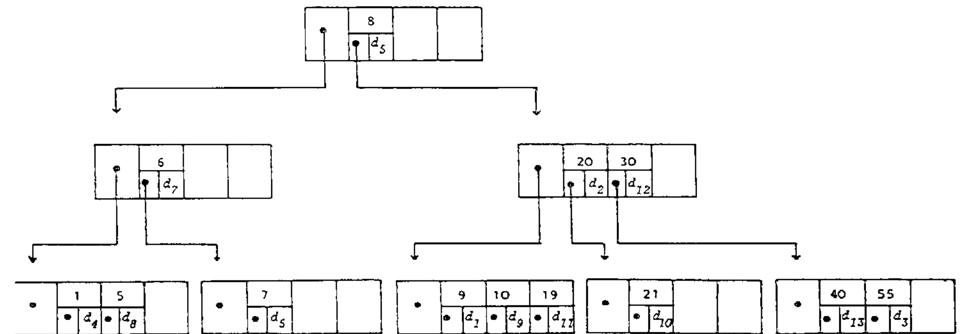


Figure 1: B-tree as a total index

If the B-tree is used as a total index the data elements cannot be blocked since individual access is required. Only the keys in the B-tree can be grouped into blocks the size of which is restricted by the io-buffer size. With this form of organization sequential access is not as efficient as with the VSAM structure because it requires tree traversal and an input-output operation for every node. However, the total index offers much more flexibility than the VSAM B-tree as far as modifications and extensions of the *index_sequential_io* package are concerned. With regard to an implementation based on *direct_io*, the total index is a more natural structure than a VSAM B-tree.

From the discussion above, the total index represents the best compromise among flexibility, efficiency, and portability for a high level implementation of index sequential input-output. Therefore it was chosen as the fundamental data structure.

4.2 Implementation Design

Implementation of index sequential input-output leads to a package of considerable size. A systematic modularization is necessary in order to reduce the complexity of the package.

4.2.1 Identification of Modules

As data structures play an outstanding part in the implementation, data abstraction and refinement of data structures are primarily used as modularization principles. Instead of proceeding top-down, however, basic data structures and operations are identified and implemented first. Then afterwards the operations specified in the package interface of *index_sequential_io* are implemented, i.e. adjusted to the interfaces of the internal data structures and operations.

This procedure reflects an underlying philosophy: As far as implementation is concerned, the abstract structures (e.g. the B-tree) are regarded with highest priority; the index sequential file organization represents just a particular application.

Decoupling of fundamental structures from the application is also pursued with respect to error handling. This means that application errors have to be checked and treated by the particular application module and not by modules suited for general use.

The central and most important data structure is the B-tree. Although it is only used here to implement the index sequential file organization,the B-tree represents a general data structure which could also be employed for many other applications.

Every node of the B-tree contains a set of keys of the virtual index sequential file which can be transferred by a single input-output operation. We will refer to such a set of keys as an *index page* [3].In order to minimize access to index pages the depth of the B-tree should be as small as possible. This means that an index page should hold a maximum number of keys which is only restricted by the size of the input-output buffers.

An index page is the object with which one has to deal when operations involving the B-tree are performed. It is considered as an abstract data structure and therefore encapsulated with its access operations in an internal package *index_page*.

Traversal of the B-tree is carried out in a nonrecursive manner. For our application, it is necessary to transfer control to the calling program during traversal, to process nodes there, and to return to a specific tree position afterwards. Furthermore, a recursive algorithm, although more elegant, would result in a critical overhead, considering the sizes of the local objects of each generation. For these reasons, recursion is resolved by means of a generalized data structure with stack properties which is implemented in the internal package *traverse_stack*.

The *direct_io* package does not provide a delete operation for individual file elements. Therefore logical deletion of a key has to be managed by the superstructure *index_sequential_io*. For this purpose, a linked list of logically deleted index pages and data elements is maintained within the external representation of the virtual index sequential file.

Although the implementation design of index sequential input-output was based on the standard package *direct_io* , we felt that other implementation approaches should remain possible, too (e.g. if efficiency is particularly critical). Therefore all input-output related declarations of types, objects, and operations are collected within the package *relative_io*. This package makes use of *direct_io* but could equally refer to other input-output facilities.

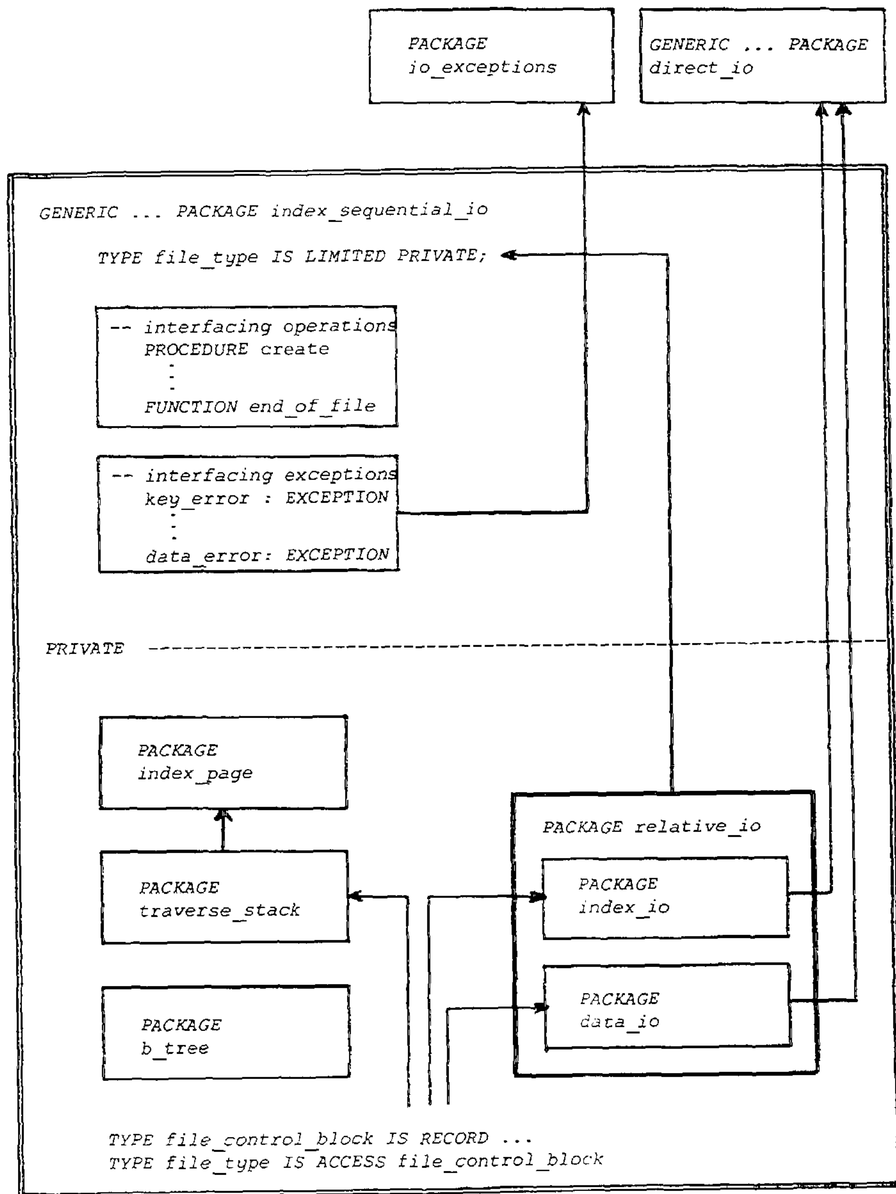


Figure 2: Interrelations among declarations of the specification part of *index_sequential_io*

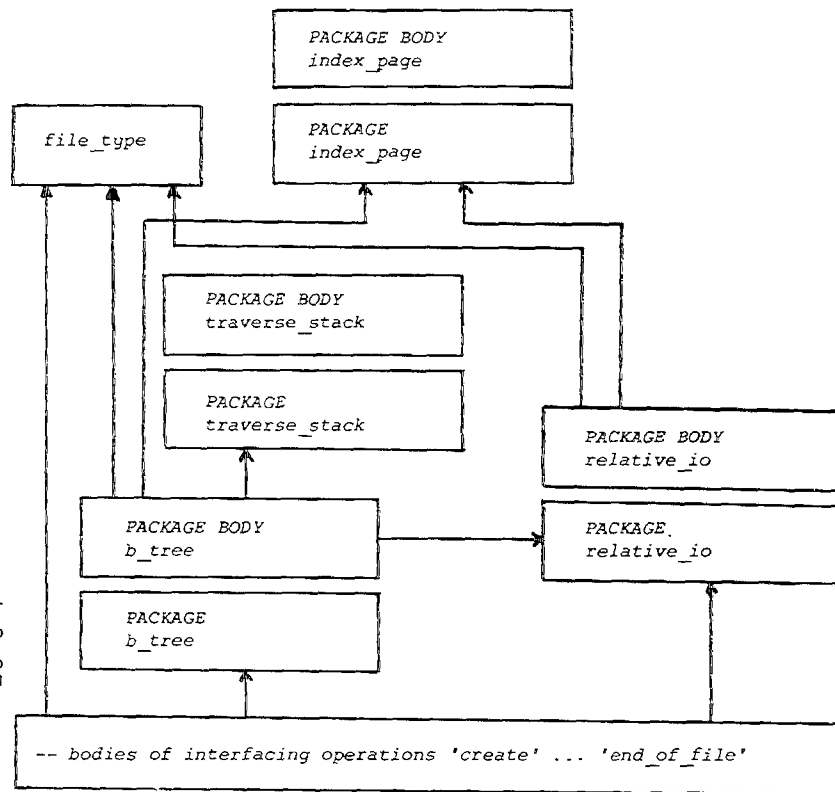
All information concerning a virtual index sequential file is collected within the private type *file_type*. According to the Ada standard, an object of a *file_type* represents an internal association with an external file. The internal file object has to provide all information necessary to describe the state of the associated external file. Since this information has to be available in many places the *file_type* is declared in the outermost declarative region of the package *index_sequential_io* and is thus global to the modules *b_tree*, *index_page*, *extended_stack*, and *relative_io*. The interrelations between the *file_type* and the packages used to implement *index_sequential_io* are illustrated in figures 2 and 3.

Figure 2 depicts the connections between the more important declarations of the specification part of the package *index_sequential_io*. An arrow a → b is employed to express that b is used for the declaration of a. Definition of the *file_type* is recursive. Since the type is limited private it is declared in the visible specification part and can thus be immediately used for further declarations. The complete type definition, however, is deferred to the private specification part. It makes use of types declared within packages which in turn use *file_type* for declaration of their interfacing operations.

Figure 3 demonstrates how the specification of *index_sequential_io* is implemented in the package body. It shows in particular which specifications are used by the bodies of the interfacing operations of *index_sequential_io* and by the bodies of the other packages. An arrow a → b expresses the relation "b is used to implement a".

4.2.2 Description of Modules

In the following paragraphs a brief description of the modules of *index_sequential_io* is given. We will point out some noteworthy characteristics but refrain from stating all details.



a → b : b is used to implement a

Figure 3: Interrelations of packages and *file_type* used to implement the bodies of *index_sequential_io* interfacing operations

The package specification is heavily commented and self-explanatory. A complete listing of the specification part is presented in the appendix.

(1) *b_tree*

The package *b_tree* comprises all operations necessary to build up, traverse, and reduce a B-tree. In particular, it provides operations *allocate_root* (initialization of the B-tree), *add_key* (addition of a new key to the B-tree), *locate_key* (search for an existing key), *delete_key* (deletion of an existing key), *set_first* and *set_succ* (traversal of the B-tree).

During execution of an *add_key* operation reorganization of the tree may become necessary. This is the case if the index page where the new key should be inserted is already full. The page has to be split, and the B-tree must therefore be reorganized. As an example, consider key 11 to be added to the B-tree of figure 1. Self-reorganization is carried out automatically within the *add_key* procedure. The resulting tree is shown in figure 4.

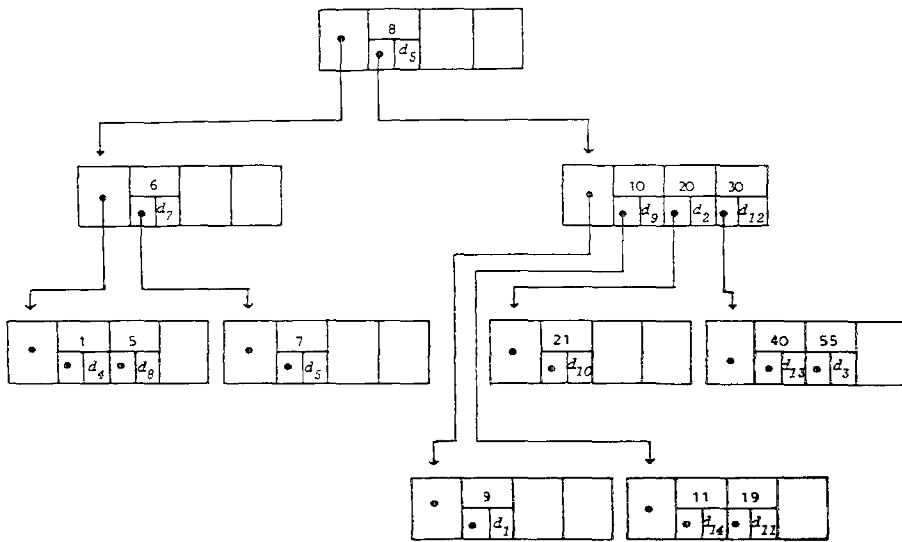


Figure 4: Reorganized B-tree after split operation

The opposite procedure has to be performed if deletion of a key would lead to an empty index page. In this case, two adjacent pages have to be concatenated which also requires reorganizaation of the tree. Figure 5 depicts the resulting structure if key 40 is deleted from the tree of figure 1.

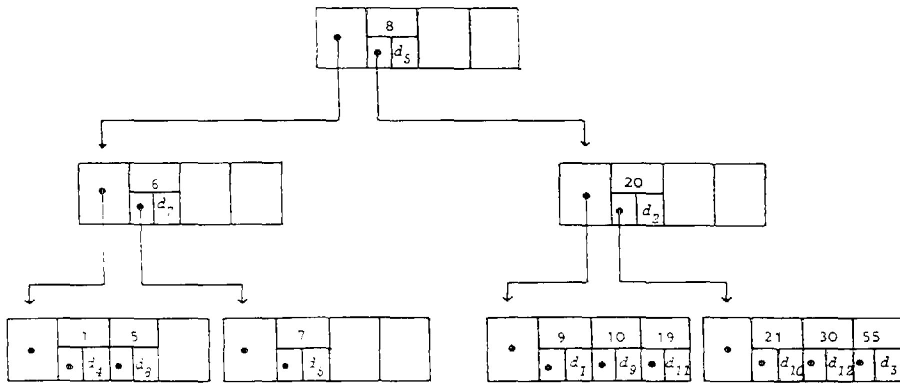


Figure 5: Reorganized B-tree after concatenation

n	k_1		k_2		k_3		k_4		...
p	d_1	s_1	d_2	s_2	d_3	s_3	d_4	s_4	...

n number of index page
p pointer to an index page with keys smaller than k_1
k_i key of i-th index item where $k_i < k_{i+1}$ (*min_key*)
d_i pointer to corresponding data element in data file (*data_pos*)
s_i pointer to an index page with keys k_j where $k_i < k_j < k_{i+1}$ for all j (*succ_keys*)

Figure 6: Logical view of an index page

(2) *index_page*

The nodes of a B-tree are index pages. The logical view of an index page is illustrated in figure 6. Items of an index page are keys k_i together with their respective pointers to the corresponding data element (d_i) and to the successor keys in a different index page (s_i):

```
TYPE index_item IS
   RECORD
      min_key  : key_type;
      data_pos : integer;
      succ_pos : integer;
   END RECORD;
```

The type of index page discriminates among empty pages, deleted pages, roots, and ordinary nodes:

```
TYPE access_state IS (empty, root, node, deleted);
```

A further distinction is made according to the number of entries in a page by means of an integer subtype *index_pos_range*. Hence the type definition of the abstract data type *index_page_type* is exported as follows:

```
TYPE index_page_type (entries : index_pos_range := max_index_pos;
                      state   : access_state    := empty)
     IS PRIVATE;
```

Operations defined for this type are listed in the specification part of the package *index_page* (see appendix). The purpose of the package is best characterized by the operations *search_page*, *add_item*, *split_page*, and *new_root*:

- *search_page* performs a binary search for a specified key element within an index page.
- *add_item* inserts a new index item into an index page maintaining the ascending order of keys.
- *split_page* divides an index page at the middle position into two pages, returns the index item in the middle (which has to be raised to an index page on the next higher level of the B-tree), and inserts the new item which caused the splitting into the correct left or right page.
- *new_root* creates a new index page to be used as root of a B-tree.

(3) *traverse_stack*

The well known stack data structure is augmented by several access operations to provide greater flexibility. For example, a *bottom* operation is added which returns the element on the bottom of the stack, and a *swap* operation which exchanges the top element by another one.

Traversal of the B-tree is facilitated if something like an "undo" operation is available. Procedures *mark* and *reset_to_mark* are defined for this purpose. The former one can be used to mark a particular state of the data structure; the latter one resets the data structure to the marked state in the sense that the effects of all push and pop operations between the two calls are repealed.

(4) *relative_io*

The package *relative_io* is used to decouple operations of higher abstraction levels from a particular input-output method. It provides a set of operations for file management and input-output.

The virtual index sequential file is represented by two physical files, a data file which contains the data elements and an index file which holds the index pages of the B-tree. Therefore most input-output operations are overloaded and supplied in duplicate for the respective types of file elements. Our implementation of the package is based on *direct_io*; i.e. two instances *data_io* and *index_io* of the generic package *direct_io* are used.

As mentioned before, deletion of a key and its associated data element cannot be realized by means of *direct_io*. Therefore we decided to implement the *delete* operation of *index_sequential_io* by utilizing linked lists which have to be maintained within *relative_io*.

(5) *file_type*

The data type *file_type* is declared as global to all modules within the package *index_sequential_io*. It is implemented as an access type pointing to a *file_control_block*. In order to make the structure more comprehensible we will give a brief explanation of the components of the *file_control_block* and point out the modules which make use of the respective information.

- *state* and *mode* are used by the operations of *index_sequential_io* to check the conditions under which the exceptions *status_error* and *mode_error* are raised.
- *form* is used by the *open* and *create* operations of *relative_io*. If the user supplies the value *reorganize* - through the parameter *form* of the procedure *index_sequential_io.open* - the linked lists of logically deleted elements will be maintained at the expense of access time. Otherwise, the respective file positions cannot be reused.

- *eof* is set when the B-tree is traversed. It is used by the function *index_sequential_io.end_of_file*.

- *index_file* and *data_file* are the file objects accessed by the operations of *relative_io* (with sizes *index_size* and *data_size*, resp.)

- *index_stack* is used by the module *b_tree* when tree traversal has to be performed.

4.3 Implementation Strategy

From experience gained in earlier software development projects the problems of using Ada as an implementation language were well known. Ada's powerful language features offer substantial flexibility to the programmer but are somewhat detrimental to the coding and testing activities. From our experience, the rate of errors encountered during program testing is much higher as compared to the error rates when less powerful languages are used for implementation.

Many errors are caused by incorrect application of sophisticated language features. When an error occurs, however, there is in most cases little indication whether the error is due to faulty design or to improper use of language elements. For example, the most frequent and least informative error message in Ada is the "constraint error". It may be prompted by a violation of rules for discriminants which are rather complicated. However, it may also result from a simple algorithmic mistake. The programmer is thus often hindered in two ways: The cause of the error may not be evident, and if the cause is a discriminant violation it is usually difficult to track down. Similar problems could be stated for other examples.

We consider the lack of an adequate number of predefined errors a deficiency of the Ada language. This might not be such a severe drawback if elaborate debugging facilities are available in the language environment. Our specific environment did not provide sufficiently convenient debugging support.

For all these reasons the implementation strategy described below was pursued. Utilization of sophisticated language elements seemed desirable in order to achieve maximum flexibility of the *index_sequential_io* package and we felt should not be renounced. We proceeded as follows.

The specifications of all packages were developed top down using Ada as specification language [4]. Advanced language features were exploited extensively, as desired from a conceptual point of view. Syntactic and semantic checks of the specification parts were carried out by means of the Ada compiler.

The package bodies were developed bottom up using Pascal as an intermediate implementation language first. The purpose of this procedure was to validate complicated algorithms (e.g. splitting of the B-tree) without being "disturbed" by errors due to highly sophisticated Ada semantics. Simplified versions of the data structures were expressed in Pascal so that the algorithmic parts of the modules *traverse_stack*, *index_page*, *relative_io*, and *b_tree* could be implemented in Pascal.

Only after the Pascal testing phase was successfully completed were the modules translated into Ada code and expanded to the full extent as required by the package specifications. In this way, the peculiarities of advanced Ada features (e.g. generics, discriminants etc.) could be isolated from purely algorithmic problems and tested separately.

From experience gained with this strategy, an assessment of Ada's virtues can be summarized in three points:

- The language elements offer good support for the specification of modules. Use of Ada as specification language is recommended.

- As an implementation language Ada turns out to be somewhat bulky. At least for large and sophisticated application problems, a preliminary implementation in a simpler language should be considered before the full richness of Ada's concepts can be exploited.

- Once the implementation barriers are surmounted the resulting product meets high quality standards. Flexibility, understandability, and reliability tend to be much higher as compared to systems implemented in other programming languages.

5 Performance and Tuning

Performance of the *index_sequential_io* package depends among other things on an efficient representation of the B-tree. A particularly critical factor is the number of input-output operations necessary for traversal of the tree. In the standard case (i.e. no splitting or concatenation) each access to an index page corresponds to one input-output operation. For a given set of key elements the depth of the B-tree and thus the total number of nodes are smaller the more keys are gathered within an index page.

The size of an index page is optimal if it has the same size as the input-output buffer, i.e. it is restricted by the buffer size. In a particular environment, the available *io_buffer_size* for *index_sequential_io* is a constant value which has to be set in the package interface of *relative_io*.

The maximum number of entries per index page depends both on the *io_buffer_size* and on the internal representation of values of the key type. The number of entries is called the *index_blk_factor*. It can be regarded as a measure of the relative efficiency of the access method.

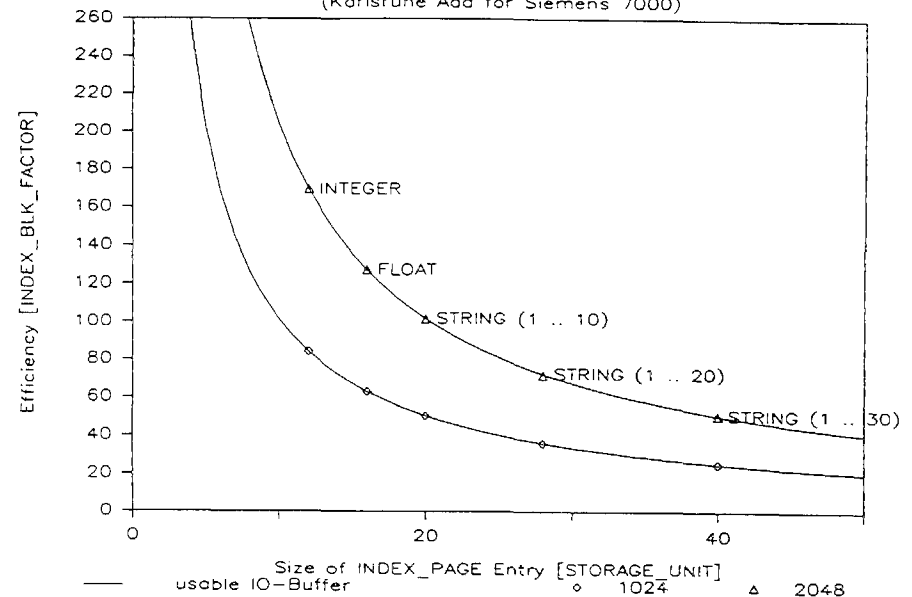


Figure 7: Influence of key and input-output buffer sizes on access efficiency

The relationship between efficiency (in terms of the number *index_blk_factor* of entries of an index page) and the key size (amount of storage units used to represent an object of *key_type* internally) is plotted in figure 7 for two specific input-output buffer sizes. It shows, for example, that the optimal *index_blk_factor* is 73 if strings of length 20 are used as keys and if the input-output buffer can hold 2048 storage units (bytes in our environment [6]).

According to the lower curve, a maximum of 36 entries per index page is feasible for the same key type if the *io_buffer_size* is

given as 1024 storage units. On the other hand, the diagram illustrates that for an *io_buffer_size* of 1024 the key size must not be greater than 14 if the same degree of access efficiency (i.e. the same *index_blk_factor)* is desired as for keys of type *string (1..20)* in an environment with *io_buffer_size* 2048.

Therefore, it would be highly desirable if the user could supply an appropriate *index_blk_factor* for a given key size as a generic parameter when the package is instantiated. The generic parameter would then be used as the upper bound of an array which holds the entries of an index page.

Because of type mapping problems this approach could not be used. The Ada reference manual does not define whether type mapping will be static or dynamic in the above case. In our environment, dynamic mapping is performed. This leads in most cases to an input-output buffer overflow because the maximum amount of memory is automatically allocated. Therefore a different approach had to be pursued.

For standard applications and applications not critical in terms of efficiency, a package version with a constant *index_blk_factor* (=100) is provided. The user may, of course, change the constant value to meet his needs. The maximum key size will from then on be restricted according to the given *io_buffer_size*.

For efficiency critical applications, another version of the package is provided. In this version, the user may supply a static discrete range type with lower bound zero and upper bound *index_blk_factor* for instantiation of the package. The type will then be used to ensure static mapping of the type *index_page*, i.e. the amount of physical storage will indeed be restricted by the *index_blk_factor*. In order to facilitate calculation of an appropriate *index_blk_factor* a generic function is provided with the package. Although this approach is less elegant than that outlined above it remained the only way of influencing access efficiency from outside.

The *index_blk_factor* is considered a property of the external file. This means that an index sequential file may only be accessed by means of instances of *index_sequential_io* which use the same *index_blk_factor* as the instance that created the file. Otherwise the exception *use_error* is raised.

In either version of *index_sequential_io*, two more generic parameters may be utilized to influence storage allocation when the package is instantiated. Description of the generic formal part of the package may now be completed:

```
GENERIC      --standard version--
   TYPE key_type IS PRIVATE;
   TYPE element_type IS PRIVATE;
   WITH FUNCTION "<" (left, right : key_type) RETURN boolean;
   traverse_stack_size : IN natural := 5;
   max_files           : IN natural := 4;
PACKAGE index_sequential_io IS ...
```

For the second version, the generic formal part becomes:

```
GENERIC      --tunable version--
   TYPE key_type IS PRIVATE;
   TYPE element_type IS PRIVATE;
   WITH FUNCTION "<" (left, right : key_type) RETURN boolean;
   traverse_stack_size : IN natural := 5;
   max_files           : IN natural := 4;
   TYPE index_pointer_range IS RANGE <>; -- for efficient mapping
PACKAGE index_sequential_io IS ...
```

where the parameter *index_pointer_range* has to be provided with lower bound and upper bound *index_blk_factor*.

The value of parameter *traverse_stack_size* is used to allocate memory for the stack structure employed for traversal of the B-tree. The size of the stack depends on the size of the index sequential file. The maximum number of elements that might be entered into the stack is equal to the depth of the B-tree. If *max* stands for the expected number of keys, the depth of the B-tree is given by

$$\Gamma (\log_{index_blk_factor/2} max)$$

assuming the worst utilization of index pages (50 %). The exception *storage_error* is raised if the allocated stack size is exceeded.

Finally, *max_files* specifies the maximum number of files that may be open at any one time when opened by the same instance of *index_sequential_io*. It is used to allocate memory for the collection *file_control_block* accessed by *file_type*. Any attempt to open more files will raise the exception *storage_error*.

The package *index_sequential_io* was designed as a superstructure based on the predefined package *direct_io*. Efficiency is thus determined to a large degree by the implementation of *direct_io*. Of course, performance is not the same since tree traversal and access to two external files are necessary for each index sequential read or write operation.

In order to obtain performance data, runtime tests were carried out for different instances of the package. The results achieved for a package instance as used in the example of section 3.3 are listed in figure 8. The table shows minimal, maximal, and average access times for single write and read operations. Ascending keys were chosen to enforce extensive application of the splitting algorithm when the B-tree is constructed.

ACCESS TIME [seconds]	setup of the entire file			sequential access to each element			random access to each element		
# records	min.	aver.	max.	min.	aver.	max.	min.	aver.	max.
10	0.036	0.126	0.148	0.019	0.021	0.027	0.036	0.051	0.059
100	0.036	0.182	0.273	0.019	0.021	0.062	0.033	0.077	0.155
1,000	0.054	0.256	0.540	0.019	0.022	0.088	0.039	0.152	0.398
10,000	0.056	0.345	15.174	0.019	0.023	0.204	0.101	0.383	1.396

Figure 8: Results of runtime tests

6 Conclusions

Flexibility of the package *index_sequential_io* was a major design goal. Some noteworthy effects of the flexibility goal will be pointed out as concluding comments.

The user is provided with a package that allows him to define keys of any type as long as an order relation for the key type is predefined or can be supplied by the user. Based on a B-tree, efficient direct access is ensured, and sequential access according to the order relation is also provided.

The B-tree is an excellent basis for generalization of the virtual index sequential file structure [10]. For example, the B-tree could be applied to create inverted lists on top of an already existing direct file. If multiple key elements are defined a B-tree structure can be generated for each key; it is superimposed on the existing direct file. The nodes of each B-tree point to data elements of the same file.

A further generalization is obtained if pointers to data elements refer to more than one file. Relations among several files could be created and evaluated in a controlled manner resulting in an extensive inverted file system. A complete data base system could be implemented easily and with little additional effort. This example illustrates clearly the flexibility of the implementation due to the encapsulation of the underlying data structure B-tree. Although our primary objective was to develop a package for index sequential input-output, extensions in many ways can easily be performed.

References

[1] American National Standard Programming Language COBOL. ANSI X3.23-1974. American National Standards Institute, New York, 1974.

[2] American National Standard Reference Manual for the Ada Programming Language. ANSI/MIL-STD 1815A-1983. American National Standards Institute, New York, 1983.

[3] Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indexes. Acta Informatica 1, 1972, 173 - 189.

[4] Booch, G. Software Engineering with Ada. The Benjamin/Cummings Publishing Company, 1983.

[5] Bradley, J. File & Data Base Techniques. CBS College Publishing, 1981.

[6] Dansmann, M., Jansohn, H.-S. et al. Karlsruhe Ada Environment. User Manual. Institut für Informatik II, Universität Karlsruhe, Bericht Nr.7/1983.

[7] Knuth, D.E. The Art of Computer Programming, Volume 3/ Sorting and Searching. Addison-Wesley, 1973.

[8] McGee, W.C. The Information Management System IMS/VS, Part II: Data Base Facilities. IBM Systems Journal 2, 1977, 96 - 122.

[9] Wagner, R.E. Indexing Design Considerations. IBM Systems Journal 4, 1973, 351 - 367.

[10] Wedekind, H. On the Selection of Access Paths in a Data Base System. Data Base Management. Edited by Klimbie, J.W. and Koferman, K.L., 1974, 385 - 396.