Chapter 7: Internal Tables

Highlights

An internal table is a temporary table that contains the records of an ABAP program while it is being executed. Consequently, an internal table exists only during the runtime of an SAP program. In addition, internal tables are used to process large volumes of data by using the ABAP language. You must declare an internal table in an ABAP program when you need to retrieve data from database tables.

Similar to any other table, data in an internal table is stored in rows and columns, where each row is called a line and each column is called a field. In an internal table, all the records have the same structure and key. The individual records of an internal table are accessed with an index or a key. Because an internal table exists till the associated program is being executed, the records of the internal table are discarded when the execution of the program is terminated.

In this chapter, you learn about internal tables, types of internal tables, and the data types of internal tables. You also learn how to create internal tables and perform operations on them, such as moving, clearing, and refreshing internal tables. In addition, you learn how to insert, modify, and delete the data stored in internal tables. The chapter concludes with a description of control break processing statements, such as `AT FIRST ... AT LAST` and `AT NEW ... AT END OF`, which are used in loops to retrieve data from internal tables and prepare reports based on the retrieved data.

Overview of Internal Tables

As you learned previously, an internal table consists of one or more rows of the same structure. However, unlike a database table, an internal table does not hold data after the execution of the program has completed. Therefore, internal tables are used as temporary storage areas or temporary buffers where data can be modified as required. These tables occupy memory only at runtime and not at the time of their declaration.

Internal tables provide a means by which data from one or more database tables can be processed within a program. The size of an internal table or the number of lines it contains is not fixed. The size of an internal table changes according to the requirement of the program associated with the internal table.

Internal tables are used to perform calculations on the data stored in the fields of database tables and re-organize the data according to the requirements. For example, users can retrieve or read certain type of data from a database table and store the data in an internal table. Users can then perform operations on the data stored in the internal table, such as calculating the sum of the data or generating a ranked list from the table.

Internal tables are also used to reorganize the content of database tables according to the requirements of a program. For example, suppose the user wants to create a list of phone numbers of different customers from one or several large customer tables. To do this, the user first creates an internal table, selects the relevant data from the customer tables, and then places the data in the internal table. Now, users can access and use the internal table directly to retrieve the desired information, instead of writing a time-consuming database query to perform each operation during the runtime of the program.

Internal tables can also act as data types and data objects. A data type is the abstract description of a data object. Therefore, when an internal table acts as a data type, the corresponding data objects are also represented as internal tables that can be declared either in a program or in the ABAP Dictionary.

An internal table is accessed by using the concept of a work area. A work area is a temporary memory space that helps you in reading and modifying the data of an internal table, line by line. The work area must have the same structure as that of the associated internal table. The structure of an internal table consists of two basic parts; i.e., a body and a header line. A header line acts as an implicit work area of an internal table. Note that creating a header line for an internal table is optional, which means an internal table can be created with or without a header line. When an internal table is created with a header line, the SAP system automatically creates a work area with the same name as that of the name of the internal table, and of the same data type as that of the lines of the table. However, when an internal table is created without a header line, the SAP system does not create a work area for the table implicitly. In this case, the user has to create the work area explicitly.

You can create an internal table in the following two ways:

- Creating the internal table as a data type and then creating a data object that refers to that data type

- Creating the internal table data object directly

A data type is defined as an internal table by using the `TYPES` statement with the `OCCURS` clause. A data object is defined as an internal table either by using the `DATA` statement along with the `OCCURS` clause or by referring to another internal table by using the `TYPE` or `LIKE` clause. A data object with a data type defined as an internal table is the actual internal table with which the user works.

An internal table can have any number of rows or lines of the same data type, which can be elementary or structured. Some examples of elementary data types are `C`, `I`, and `N` (predefined data types). Here, `C` stands for characters, `I` for integers, and `N` for numeric characters. Examples of structured data types are field strings and internal tables themselves. A user can access a particular line or row of an internal table by specifying a single field or a combination of fields to identify the line.

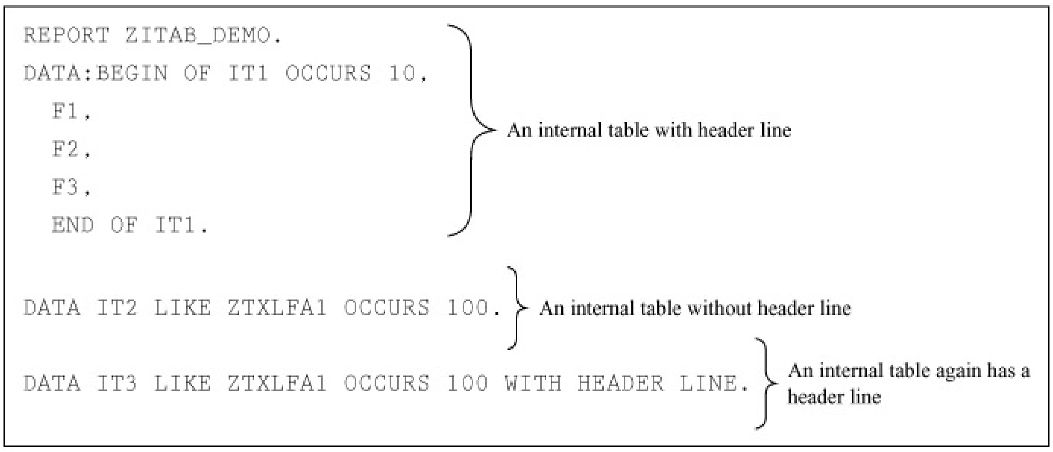
In an internal table based on the relational model of the SAP R/3 system, the required minimum number of fields to identify a row or line is known as a key and the fields that define the key are called key fields. The following categories of keys can be defined in the fields of an internal table:

- **Standard key—** Defines those fields of the internal table whose data types are not numeric, such as `Float`, `Integers`, and `Packed` (`F`, `I`, and `P`, respectively), and the internal table itself.
- **Self-defined or user-defined key—** Defined by a user by using the `READ` statement.

The definition of an internal table consists of a body and an optional header line. The body holds the rows (that are similar in structure) of the internal table. The header line is a field string with the same structure as a row of the body, but it can hold only a single row. Header lines are buffers used to hold a record before it is added to or retrieved from an internal table.

The `OCCURS` clause is used to define the body of an internal table by declaring the fields for the table. When the `OCCURS` clause is used, you can specify a numeric constant, *n*, to determine additional default memory if required. The default size of memory that is used by the `OCCUR 0` clause is 8 KB.

As stated earlier, an internal table can be created with or without using a header line. To create an internal table with a header line, use either the `BEGIN OF` clause before the `OCCURS` clause or the `WITH HEADER LINE` clause after the `OCCURS` clause in the definition of the internal table. However, to create an internal table without a header line, use the `OCCURS` clause without the `BEGIN OF` and `WITH HEADER LINE` clauses. Figure 7.1 shows the structure of an internal table with and without a header line:



```
REPORT ZITAB_DEMO.
DATA:BEGIN OF IT1 OCCURS 10,
  F1,
  F2,
  F3,
  END OF IT1.

DATA IT2 LIKE ZTXLFA1 OCCURS 100.

DATA IT3 LIKE ZTXLFA1 OCCURS 100 WITH HEADER LINE.
```

An internal table with header line
An internal table without header line
An internal table again has a header line

Figure 7.1: Creating an internal table with and without a header line

Note As a best practice, avoid creating an internal table with a header line inside nested structures or internal tables, as it may lead to confusion.

Data Types of an Internal Table

An internal table is one of the complex types in the ABAP language (the other being the structure). The data type of an internal table is fully defined by three parameters:

- **Line type or row type—** Defines the attributes of the individual fields of an internal table by specifying any ABAP data type. The data type can be an element, a structure, or an internal table itself.
- **Key type—** Identifies the rows of an internal table. As already stated, two kinds of keys can be defined for an internal table: standard keys and self-defined keys (also called user-defined keys). A key that is defined for an internal table can be either unique or non-unique. When a key in an internal table is defined as unique, the table cannot have duplicate entries; however, this is not so in the case of the non-unique key. If the row type of an internal table is a structure, the standard key is defined on the fields that have the character data type. However, if the row type is an element, the entire row or line of the internal table uses a key, which is also the default key of the table. However, an internal table whose row type is also an internal table has an empty default key. In the case of a user-defined key, the fields of the internal table can neither be internal tables themselves nor contain internal tables. Internal tables with user-defined keys are also called key tables.
- **Table type—** Determines how ABAP accesses individual table entries by using the assigned key or index number.

Types of Internal Tables

Internal tables can be categorized into three types: standard tables, sorted tables, and hashed tables. In addition, internal tables can be categorized on the basis of their definition, as a fully specified or generic type. A fully specified table type determines how the SAP system accesses the entries of the table based on performing key-based operations. This table type performs a linear search in standard tables, a binary search in sorted tables, and a hash algorithm search in hashed tables. However, in the generic table type, the tables are searched on the basis of indexes.

Figure 7.2 shows the hierarchy of the different types of internal tables:

Let's now describe these table types in detail.

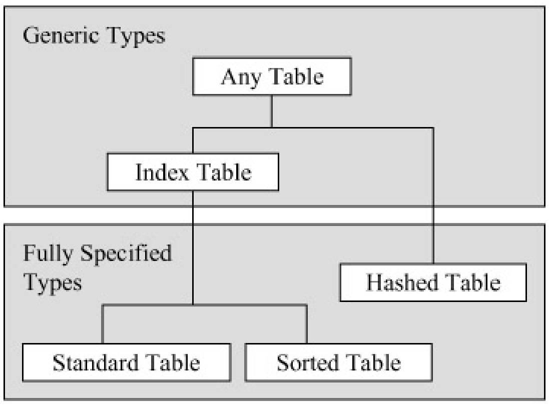


Figure 7.2: Hierarchy of table types

Standard Tables

Standard tables have a linear index, which manages the table as a treelike structure. The records of a standard table are accessed by using a table index or key. Standard tables always have a non-unique key. Therefore, the SAP system does not need to check the table's existing entries when new entries are added to the table. You can use standard tables when individual table entries need to be addressed on the basis of indexes, because this is the quickest way to access a table. A standard table can be populated by using the `APPEND` statement and its entries can be read, modified, and deleted by specifying the index number in the relevant ABAP statement.

Sorted Tables

Sorted tables also have internal indexes, similar to standard tables, and are sorted with a key. You can access the records of a sorted table by using a table index or key. In this case, a key also is defined as either unique or non-unique. Sorted and standard tables use indexes and are therefore also called index tables. In sorted tables, records entered are sorted automatically. These tables are populated by using the `INSERT` statement according to the sort sequence defined in the table key. As a result, whenever a user tries to add a nonsequential entry to the table, it is recognized easily. Sorted tables are used in particular situations, such as when you want to perform partial sequential processing of code. In case of partial sequential processing of code, the code is written in a loop and the beginning of the table key is specified in the `WHERE` clause.

Hashed Tables

These tables have no linear index and are accessed only by the keys. The SAP system accesses the entries of a hashed table by using a hash algorithm. While defining a hashed table, the key of the hashed table must be defined as unique. Hashed tables are selected when the user wants to perform operations based on the key and not the index. Hashed tables are useful when you want to create and use an internal table that resembles a database table, or if you want to process large volumes of data.

As stated earlier, the data type of an internal table is fully specified by its line type, key, and table type. However, it is not necessary to specify the data type of an internal table fully. Instead, you can specify a generic construction for the table by leaving the key or line type unspecified. Generic internal tables cannot be used to declare data objects.

Creating Internal Tables

As stated earlier, internal tables can be declared as data types or data objects in ABAP. When internal tables are declared as data types, either in programs or in the ABAP Dictionary, they are used to define data objects. The minimum size of an internal table is 256 bytes. Unlike other ABAP data objects, you do not need to specify the memory required for an internal table, because, at runtime, rows are inserted and deleted dynamically by using statements, such as `INSERT`, `MODIFY`, and `DELETE`.

Let's now explore how to create internal tables as data types and data objects.

Creating Internal Tables as Data Types

You can create an internal table as a local data type (a data type used only in the context of the current program) by using the `TYPES` statement. This statement uses the `TYPE` or `LIKE` clause to refer to an existing table. The syntax to create an internal table as a local data type is:

```
TYPES  <internal_tab>  TYPE|LIKE  <internal_tab_type>  OF
<line_type_itab> [WITH <key>]
       [INITIAL SIZE <size_number>].
```

In the preceding syntax, a data type has not been specified after the `TYPE` or `LIKE` clause. However, a type constructor has been specified after these clauses. The type constructor is shown in the following part of the preceding syntax:

```
<internal_tab_type> OF <line_type_itab> [WITH <key>]
```

The various expressions used with the `TYPES` statement are as follows:

- `<internal_tab_type>`—Specifies a table type for an internal table `<internal_tab>`. Table 7.1 describes the table types:
- `<line_type_itab>`—Specifies the type for a line of an internal table. In the `TYPES` statement, you can use the `TYPE` clause to specify the line type of an internal table as a data type and the `LIKE` clause to specify the line type as a data object. When you specify a line type in an internal table, the data type can be a predefined ABAP type, a local type declared in a program, or a data type from the ABAP Dictionary. If any of the generic element types, such as `C`, `N`, `P`, or `X`, is specified as a line type, the attributes (such as the length of the field and a range of numbers) that are left unspecified are populated automatically with default values.
- `<key>`—Specifies the type of key for the internal table `<t>`, as given in the following syntax:

  ```
  [UNIQUE|NON-UNIQUE] KEY col1 ... coln
  ```

Table 7.1: List of table types

Table Type	Description
Index Table	Creates a generic table type with index access
Any Table	Creates a fully generic table type
Standard Table or Table	Creates a standard table
Sorted Table	Creates a sorted table
Hashed Table	Creates a hashed table

In internal tables with a structured line type, `col1`, `col2`, `col3...coln` are key fields, which are not of an internal table data type or references to such data types.

Besides specifying the key fields individually, you can specify the entire line (including all the fields) of an internal table as a table key. The following syntax is used to define the entire line as a key of an internal table:

```
[UNIQUE|NON-UNIQUE] KEY table_line
```

An internal table must have any of the following elementary line types, `C`, `D`, `F`, `I`, `N`, `P`, `T`, and `X` (where `C` is character, `D` is decimal, `F` is float, `I` is integer, `N` is numeric characters, `P` is packed, `T` is time, and `X` is hexadecimal), to define the entire line as a key. However, if the user tries to define the entire line as a key in an internal table whose line type is itself an internal table, a syntax error occurs. In addition, if an internal table has a structured line type, the entire line can be specified as a key; however, it is often inappropriate because this affects the efficiency of the SAP system. You can also define a default key for an internal table by using the following syntax:

```
[UNIQUE|NON-UNIQUE] DEFAULT KEY
```

In the preceding syntax, the `DEFAULT KEY` clause is used to define the fields of an internal table as key fields.

If an internal table has a structured line type, the default key contains all nonnumeric columns, which are neither internal tables themselves nor the references of internal tables. If the table has an elementary line type, the entire line is the default key. The default key of an internal table, whose line type is an internal table, is empty.

Specifying a key for an internal table is optional and if the user does not specify a key, the SAP system defines a table type with an arbitrary key. The optional clause (`UNIQUE` or `NON-UNIQUE`) determines if the key is unique or non-unique; that is, whether the table can accept duplicate entries. If the user does not specify the `UNIQUE` or `NON-UNIQUE` clause with the key field of an internal table, the internal table is generic. However, if you specify the type of the table simultaneously, there are two limitations. First, you cannot use the `UNIQUE` clause for standard tables because the SAP system always generates the `NON-UNIQUE` clause automatically; second, you always must specify the `UNIQUE` clause while creating a hashed table.

- `INITIAL SIZE <size_number>`—Creates an internal table object by allocating an initial amount of memory to it. Use the following syntax to allocate initial memory to an internal table:

```
INITIAL SIZE <size_number>
```

In the preceding syntax, the `INITIAL SIZE` clause reserves a memory space for `size_number` table lines. Whenever an internal table object is declared, the size of the table does not belong to the data type of the table. Note that much less memory is consumed when an internal table is populated for the first time.

Examples of the TYPES Statement

This section provides examples of the `TYPES` statement along with their clauses and expressions, such as `TYPE`, `LIKE`, `<internal_tab_type>`, and `<line_type_itab>`.

Listing 7.1 shows how to declare a table type with the help of the `TYPES` statement:

Listing 7.1: Declaring a table type with the TYPES statement

```
REPORT ZINTERNAL_TABLE_DEMO.
*/declaring table type by using the TYPES statement

TYPES: BEGIN OF DataLine,
              S_ID TYPE C,

             S_Name(20) TYPE C,
             S_Salary TYPE I,
      END OF DataLine.
TYPES MyTable TYPE SORTED TABLE OF DataLine WITH UNIQUE
KEY S_ID.
WRITE:/'MyTable is an internal table. It is a sorted type
of table with a unique key defined on the S_ID field.'.
```

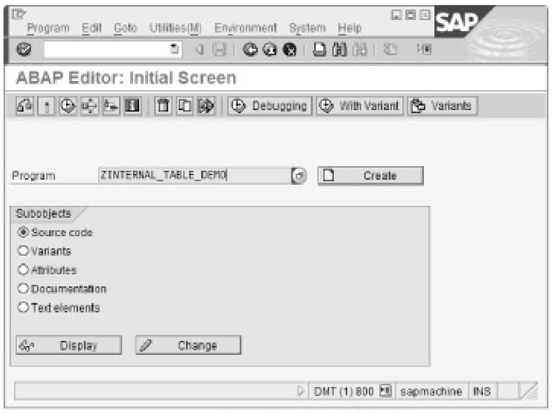
In Listing 7.1, MyTable is a sorted internal table with DataLine as its line type. This table is created with a unique key in the

S_ID field.

Note that in the mySAP ERP system, you write the source code of any ABAP program in ABAP Editor. Perform the following steps to write the code specified in Listing 7.1 in ABAP Editor:

1. Open the ABAP Editor by either navigating the SAP menu or executing the SE38 transaction code. The initial screen of ABAP Editor appears.
2. In the initial screen of ABAP Editor, enter a name for the program, select the Source code radio button in the Subobjects group box, and click the Create button to create a new program, as shown in Figure 7.3:

   The ABAP: Program Attributes dialog box appears.
3. In the ABAP: Program Attributes dialog box, enter a short description for the program in the Title field, select the Executable program option from the Type drop-down menu in the Attributes group box, as shown in Figure 7.4:
4. Now, click the Save (✓ Save) button in the ABAP: Program Attributes dialog box or press the ENTER key (see Figure 7.4). The Create Object Directory Entry dialog box appears.
5. In the Create Object Directory Entry dialog box, enter ZKOG_PCKG as the package name in the Package field and click the Save (💾) icon, as shown in Figure 7.5:

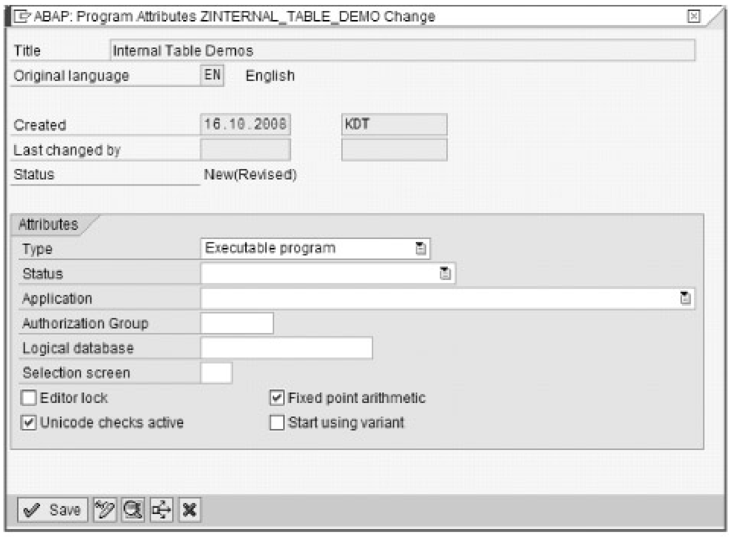

Program Edit Goto Utilities(M) Environment System Help
SAP
ABAP Editor: Initial Screen
Debugging
With Variant
Variants
Create
Display
Change
DMT (1) 800
sapmachine
INS

Program ZINTERNAL_TABLE_DEMO

Subobjects
- (●) Source code
- ( ) Variants
- ( ) Attributes
- ( ) Documentation
- ( ) Text elements

© SAP AG. All rights reserved.

Figure 7.3: Creating a new program



ABAP: Program Attributes ZINTERNAL_TABLE_DEMO Change

Title	Internal Table Demos	
Original language	EN	English
Created	16.10.2008	KDT
Last changed by		
Status	New(Revised)	

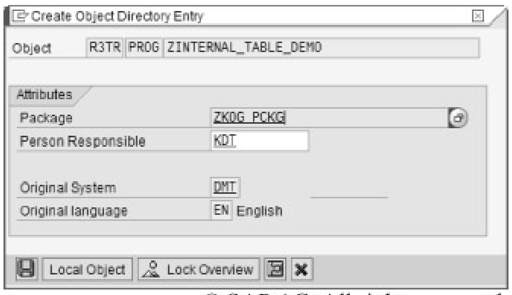
Attributes

Type	Executable program
Status	
Application	
Authorization Group	
Logical database	
Selection screen	

- [ ] Editor lock
- [✓] Fixed point arithmetic
- [✓] Unicode checks active
- [ ] Start using variant

© SAP AG. All rights reserved.

Figure 7.4: The ABAP—Program attributes dialog box



Create Object Directory Entry

Object R3TR PROG ZINTERNAL_TABLE_DEMO

Attributes

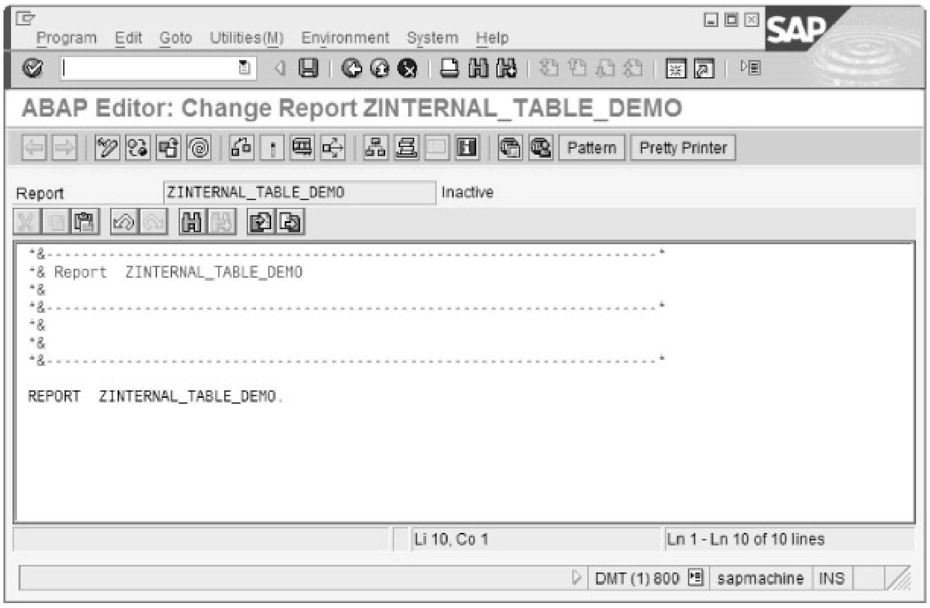
Package	ZKOG_PCKG
Person Responsible	KDT
Original System	DMT
Original language	EN English

Figure 7.5: Create object directory entry dialog box

The `ABAP Editor: Change Report` screen appears, as shown in Figure 7.6:

6. Write the code given in Listing 7.1 in the `ABAP Editor: Change Report` screen, as shown in Figure 7.7:
7. Click the `Save` () icon or press the CTRL + S key combination to save the changes in the program.
8. Next, click the `Check` () icon or press the CTRL + F2 key combination to check and remove syntax errors or warnings (if any) in the program.
9. Next, click the `Activate` () icon or press the CTRL + F3 key combination to activate the program, as shown in Figure 7.8:
10. Now, click the `Direct Processing` () icon or press the F8 key to execute the program.

    Figure 7.9 shows the output of Listing 7.1:



ABAP Editor: Change Report ZINTERNAL_TABLE_DEMO

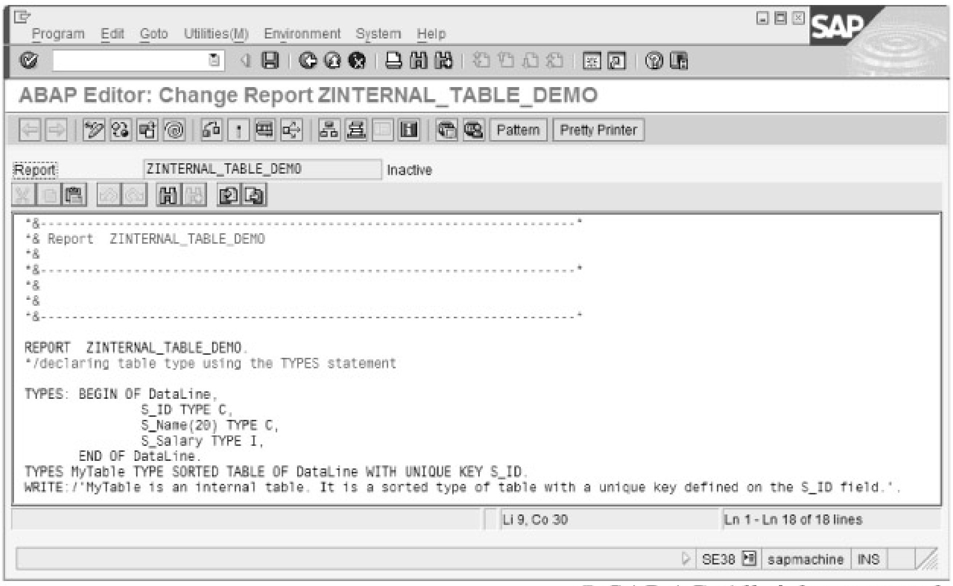
Report ZINTERNAL_TABLE_DEMO Inactive

```
*&---------------------------------------------------------------------*
*& Report  ZINTERNAL_TABLE_DEMO
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  ZINTERNAL_TABLE_DEMO.
```

Li 10, Co 1 Ln 1 - Ln 10 of 10 lines

DMT (1) 800 sapmachine INS

Figure 7.6: ABAP editor—change report screen



```
*&---------------------------------------------------------------------*
*& Report  ZINTERNAL_TABLE_DEMO
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  ZINTERNAL_TABLE_DEMO.
*/declaring table type using the TYPES statement

TYPES: BEGIN OF DataLine,
              S_ID TYPE C,
              S_Name(20) TYPE C,
              S_Salary TYPE I,
       END OF DataLine.
TYPES MyTable TYPE SORTED TABLE OF DataLine WITH UNIQUE KEY S_ID.
WRITE:/'MyTable is an internal table. It is a sorted type of table with a unique key defined on the S_ID field.'.
```

Figure 7.7: Adding code in the ABAP editor—change report screen

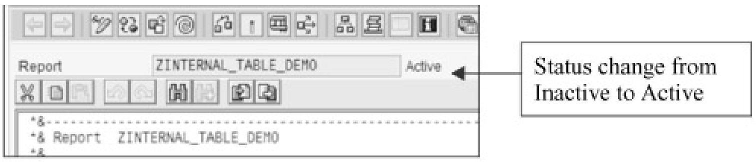
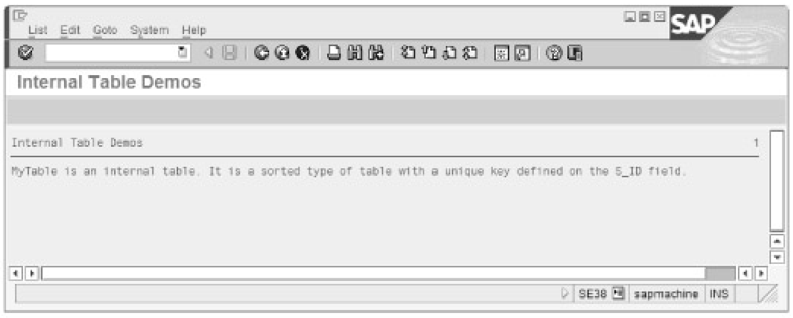

Report
ZINTERNAL_TABLE_DEMO
Active
Status change from Inactive to Active
*& Report ZINTERNAL_TABLE_DEMO

Figure 7.8: Activating a program



```
Internal Table Demos                                                    1

MyTable is an internal table. It is a sorted type of table with a unique key defined on the S_ID field.
```

Figure 7.9: Output of the ZINTERNAL_TABLE_DEMO program

Note The preceding steps need to be performed to display the output for all the programs created in this chapter.

Listing 7.2 is another example of how to declare the table type for an internal table by using the TYPES statement:

Listing 7.2: Creating three internal tables by using the TYPES statement

```
Report ZTYPES_DEMO

*/declaring a hashed table type by using the TYPES
statement

TYPES VectorTab TYPE HASHED TABLE OF I WITH UNIQUE KEY
table_line.

TYPES: BEGIN OF LINE,
                 S_ID TYPE C,
                 S_Name(20) TYPE C,
                 S_Salary TYPE I,
           END OF LINE.
*/declaring a sorted table type by using the TYPES
statement
TYPES MyTable TYPE SORTED TABLE OF LINE WITH UNIQUE KEY
S_ID.
TYPES: BEGIN OF line2,
                 Myfield TYPE c,
                 tableX TYPE VectorTab,
                 tableY TYPE MyTable,
           END OF line2.
TYPES MyTable2 TYPE STANDARD TABLE OF line2
WITH DEFAULT KEY.
```

In Listing 7.2, the TYPES statement is used to create three internal tables: VectorTab, MyTable, and MyTable2. The VectorTab table is a hashed table, which is defined by using Type I as its line type and table_line as its unique key. MyTable is a sorted table with LINE as its line type and the S_ID field as its unique key. MyTable2 is a standard table, having a default key, with line2 as its line type. Note that the key in MyTable2 is non-unique, as the table is a standard table.

Now, let's learn how to create an internal table as a data object.

Creating Internal Tables as Data Objects

You can create an internal table as a data object by using the DATA statement. In this section, you first learn how to create a new internal table object and then to create a table object from an existing table object.

Creating a New Internal Table Object

The DATA statement is used to construct a new internal table, as shown in the following syntax:

```
DATA <internal_tab> TYPE|LIKE <internal_tab_type> OF <line_
type_itab> WITH <key>
           [INITIAL SIZE <size_number>]
           [WITH HEADER LINE].
```

In the preceding syntax, the TYPE clause is used to define the <internal_tab_ type> table type, the <line_type_itab> line type, and the <key> keyfield of the <internal_tab> internal table. You can only create fully specified table types but not generic table types. In addition, a key must be specified as unique or non-unique. Listing 7.3 shows how to create a new internal table object:

Listing 7.3: Creating a new internal table

```
Report ZINTERNAL_TABLE_DEMO
DATA MyTable TYPE HASHED TABLE OF MARA WITH UNIQUE KEY
MANDT MATNR.
```

In Listing 7.3, MyTable is a hashed table with MARA as its line type that corresponds to the MARA table in the ABAP Dictionary. In addition, MyTable has a unique key with the MANDT and MATNR fields. MyTable can be regarded as an internal template for the MARA database table.

The WITH HEADER LINE clause is used to create the header line of an internal table. This clause, which is optional, is used to declare an extra data object with the same name and line type as that of the internal table. The extra data object declared by the WITH HEADER LINE clause is known as the header line of the internal table and is used as a work area while working with an internal table.

Creating an Internal Table Object from an Existing Table Object

Internal tables are created either as new tables or from existing tables. The following is the syntax of the DATA statement used to create an internal table from an existing table type:

```
DATA  <internal_tab> TYPE <internal_tab_type>|LIKE <obj>
[WITH HEADER LINE].
```

In the preceding syntax, the `DATA` statement creates an internal table either from an existing internal table object or from an existing internal table type by using the `TYPE` clause. The `WITH HEADER LINE` clause, which is optional, is used to create the header line of the internal table. While using internal tables with header lines, it is mandatory for the header line and the body of the table to have the same name. If a user wants to specify the complete body of an internal table with a header line, brackets must be placed after the table name `(<internal_tab>[]);` otherwise, ABAP interprets the name as the name of the header line and not the body. Listing 7.4 shows how to represent the body of an internal table:

Listing 7.4: Using square brackets to represent the body of an internal table

```
REPORT ZINTERNAL_TABLE_DEMO.
TYPES VectorTab TYPE SORTED TABLE OF I WITH UNIQUE KEY
TABLE LINE.

DATA: TABLEX TYPE VectorTab,
      TABLEY LIKE TABLEX WITH HEADER LINE.

*/Table without body
MOVE TABLEX TO TABLEY. "1: will produce syntax error!

*/Table with body
MOVE TABLEX TO TABLEY[]. "2: error free line
```

In Listing 7.4, VectorTab is a sorted table type, which has `I` as its line type and `TABLE LINE` as its unique key. The `TABLEX` object is created by referring to the VectorTab table type. The `TABLEY` object has the same data type as the `TABLEX` object. In addition, the `TABLEY` object has a header line.

The `MOVE` statement is used two times (marked 1 and 2), as shown in Listing 7.4. The line marked 1 generates a syntax error since square brackets [] are not used with the `TABLEX` internal table. However, the line marked 2 does not generate a syntax error because square brackets [] are used to represent the body of the TABLEX table.

Note To learn more about the `MOVE` statement, refer to the "Moving and Assigning Internal Tables" section of this chapter.

You can resolve the error generated at the line marked 1 in Listing 7.4 by using internal tables without header lines, as shown in the following code snippet:

```
REPORT ZINTERNAL_TABLE_DEMO.
...
DATA: TABLEX TYPE VectorTab,
       TableY LIKE TABLEX. "1: will not produce syntax
       error
MOVE TABLEX TO TABLEY.
...
```

Performing Operations on an Entire Internal Table

You can access an internal table as a single data object (including all its lines) or its individual lines to perform operations such as insertion, deletion, and sorting. In this section, we perform the following operations by treating the body of an internal table as a single data object:

- Moving and assigning internal tables
- Clearing internal tables
- Refreshing internal tables
- Releasing memory of internal tables
- Comparing internal tables
- Performing the sort operation in internal tables
- Determining the attributes of internal tables

Moving and Assigning Internal Tables

Similar to other data objects, internal tables are also used as operands in the MOVE statement, as shown in the following syntax:

```
MOVE <internal_tab1> TO <internal_tab2>.
```

The preceding syntax is equivalent to the following syntax, in which one internal table is assigned to another:

```
<internal_tab2> = <internal_tab1>.
```

In this syntax, both operands, internal_tab1 and internal_tab2, must be either compatible or convertible. In both cases, the entire content of the <internal_tab1> internal table is assigned to the <internal_tab2> internal table. In this process, the original content of the target table (<internal_ tab2>) is overwritten.

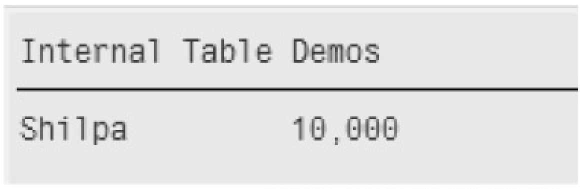
When internal tables are created with header lines, the names of the header line and the body of the internal table are the same. When the MOVE statement is used to assign one internal table to another, square brackets ([]) are used after the table name to represent the entire body of the internal table. Listing 7.5 shows how to assign and move the content of one internal table into another internal table:

Listing 7.5: Assigning and moving the content of one internal table to another

```
REPORT ZINTERNAL_TABLE_DEMO.
*/Creating two internal tables-Tab1 and Tab2
DATA: BEGIN OF line OCCURS 0,
                 Name(10) TYPE c,
                 Salary TYPE I,
          END OF line.

DATA: Tab1 LIKE TABLE OF line,
       Tab2 LIKE TABLE OF line.

*/Appending a line into Tab1

line-Name = 'Shilpa'.
line-Salary = 10000.
APPEND line TO Tab1.

*/Moving a line from Tab1 to Tab2

MOVE Tab1[] TO Tab2.

LOOP AT Tab2 INTO line.
     WRITE: / line-Name, line-Salary.
ENDLOOP.
```

In Listing 7.5, two standard tables, Tab1 and Tab2, are created with the line type of the LINE structure. Note that Tab1 contains a header line. This table is first populated by using the APPEND statement, line by line, and then its entire content is moved to Tab2 by using the MOVE statement. Figure 7.10 shows the output of Listing 7.5:

Listing 7.6 assigns the content of a hashed table to a sorted table.



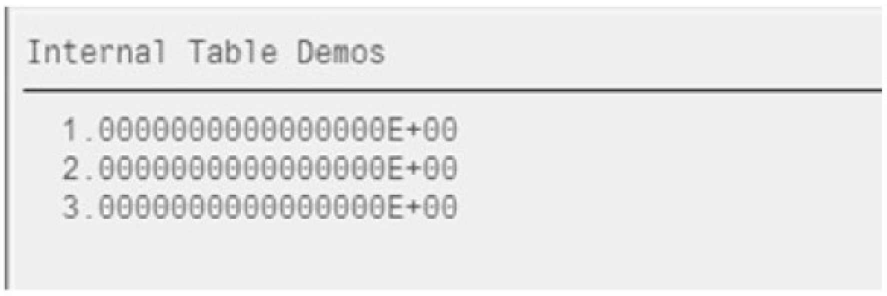
```
Internal Table Demos
Shilpa          10,000
```

Figure 7.10: Moving the content between two standard tables

Listing 7.6: Assigning the content of one table to another

```
REPORT ZINTERNAL_TABLE_DEMO.
DATA: TABLEF TYPE SORTED TABLE OF F
          WITH NON-UNIQUE KEY table_line,
      TABLEI TYPE HASHED TABLE OF I
          WITH UNIQUE KEY table_line,
      Var1 TYPE F.

*/Inserting three lines into TABLEI

DO 3 TIMES.
   INSERT sy-index INTO TABLE TABLEI.
ENDDO.
*/Assigning the data of TABLEI into TABLEF

TABLEF = TABLEI.
LOOP AT TABLEF INTO Var1.
   WRITE: / Var1.
ENDLOOP.
```

Listing 7.6 contains two tables, `TABLEF` and `TABLEI`. `TABLEF` is a sorted table with a non-unique key and is of line type `F`, while `TABLEI` is a hashed table with a unique key and is of line type `I`. When you execute Listing 7.6, the content of `TABLEF` is overwritten by the content of `TABLEI`, as shown in Figure 7.11:



```
Internal Table Demos

  1.0000000000000000E+00
  2.0000000000000000E+00
  3.0000000000000000E+00
```

© SAP AG. All rights reserved.

Figure 7.11: Displaying the overwritten data

Note that when a user assigns an unsorted table (`TABLEI`) to a sorted table (`TABLEF`), the entire content of the unsorted table is sorted automatically by the key of the sorted table.

Initializing Internal Tables

Similar to all data objects, an internal table can be initialized by using the `CLEAR` statement. The syntax to initialize an internal table is:

```
CLEAR <internal_tab>.
```

This syntax restores the original state of the `<internal_tab>` internal table. The original state is the state immediately after the creation of the internal table; i.e., when the internal table contains no lines. However, the memory allocated to the internal table remains unchanged until the memory is released manually by the user. The following syntax is used to clear the entire body of an internal table:

```
CLEAR <internal_tab>[].
```

Refreshing Internal Tables

The `REFRESH` statement is used to ensure that an internal table is initialized. The following syntax shows the use of the `REFRESH` statement:

```
REFRESH <internal_tab>.
```

This syntax always applies to the entire body of the internal table. Note that the memory space being used by an internal table is not released.

Releasing the Memory of Internal Tables

The FREE statement is used to initialize an internal table and release the memory allocated to it. The following syntax is used to release the memory allocated to an internal table:

```
FREE <internal_tab>.
```

Similar to the REFRESH statement, the FREE statement works only on the body of the table and not on its work area. The name of an internal table can be referenced even after using the FREE statement because even after the memory allocated to the internal table is released, the internal table continues to occupy the memory required for its header. However, when the internal table is populated again, the SAP system allocates new memory to the table.

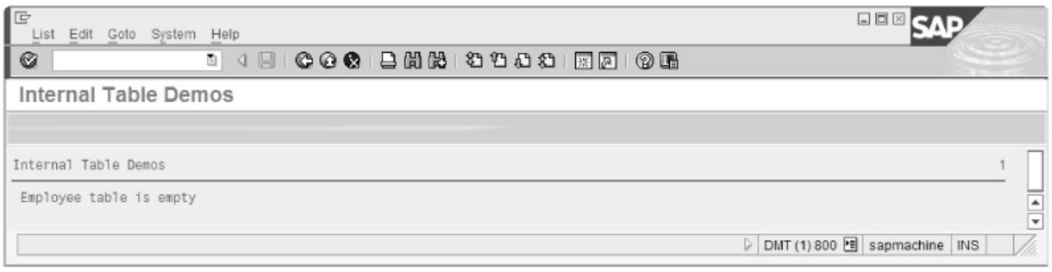
Note The memory space initially allocated to an internal table is 256 bytes.

Listing 7.7 shows how the memory of the EmployeeDataTable internal table is released by using the REFRESH and FREE statements:

Listing 7.7: Using the REFRESH and FREE statements with internal tables

```
REPORT ZINTERNAL_TABLE_DEMO.

DATA: BEGIN OF LINE,
COL1(10) Type C,
COL2(20) Type C,
END OF LINE.

DATA EmployeeDataTable LIKE TABLE OF LINE.
LINE-COL1 = 'Emp_ID'.
LINE-COL2 = 'Emp_Name'.
*/Appending a line into EmployeeDataTable table

APPEND LINE TO EmployeeDataTable.

*/Refreshing the data of EmployeeDataTable table

REFRESH EmployeeDataTable.

*/Checking whether EmployeeDataTable table is empty or not,
if it is empty then it is released by the FREE statement

IF EmployeeDataTable IS INITIAL.
WRITE 'Employee table is empty'.
FREE EmployeeDataTable.
ENDIF.
```

In Listing 7.7, EmployeeDataTable is an internal table that is populated and then initialized with the REFRESH statement. The IF statement is used to check whether EmployeeDataTable is empty. If the table is empty, its memory is released by using the FREE statement and a message is displayed stating that the table is empty, as shown in Figure 7.12:



Internal Table Demos

```
Internal Table Demos                                              1

 Employee table is empty
```

© SAP AG. All rights reserved.

Figure 7.12: Displaying the message "Employee table is empty"

Comparing Internal Tables

Internal tables are also used as operands in a logical expression in ABAP. Consider the following syntax:

```
.... <internal_tab1> <operator> <internal_tab2> ...
```

In this syntax, the `<operator>` expression stands for an operator from among the comparison operators `EQ`, `=`, `NE`, `<>`, `><`, `GE`, `>=`, `LE`, `<=`, `GT`, `>`, `LT`, and `<`.

Note that two internal tables are compared based on the number of lines they contain. If two internal tables contain the same number of lines, they are compared line by line and component by component. If the components of an internal table line are internal tables themselves, they are compared recursively. Listing 7.8 shows the comparison of two tables on the basis of the number of lines they contain:

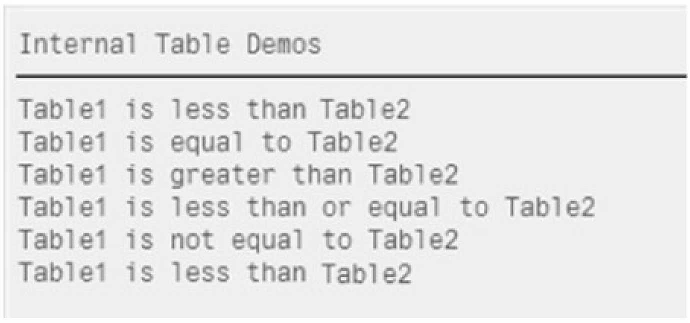
Listing 7.8: Comparing two tables on the basis of the number of lines

```
REPORT ZCOMPARE_TABLES_DEMO.

DATA: BEGIN OF LINE,
NumCol1 TYPE I,
NumCol2 TYPE I,
END OF LINE.

*/creating two tables Table1 and Table2

DATA: Table1 LIKE TABLE OF LINE,
Table2 LIKE TABLE OF LINE.

*/appending four lines in Table1

DO 4 TIMES.
LINE-NumCol1 = SY-INDEX.
LINE-NumCol2 = SY-INDEX ** 2.
  APPEND LINE TO Table1.
ENDDO.

*/Moving the data of Table1 into Table2

MOVE Table1 TO Table2.

*/appending a new line into Table2
LINE-NumCOL1 = 100.
LINE-NumCOL2 = 30.
APPEND LINE TO Table2.
IF Table1 LT Table2.
WRITE / 'Table1 is less than Table2'.
ENDIF.

*/appending the same line into Table1

APPEND LINE TO Table1.
IF Table1 EQ Table2.
WRITE / 'Table1 is equal to Table2'.
ENDIF.
*/appending a new line into Table1

LINE-NumCOL1 = 33.
LINE-NumCOL2 = 33.
APPEND LINE TO Table1.
IF Table1 GT Table2.
WRITE / 'Table1 is greater than Table2'.
ENDIF.

*/appending the same line two times into Table2

APPEND LINE TO Table2.
APPEND LINE TO Table2.
IF Table1 LE Table2.
```

```
WRITE / 'Table1 is less than or equal to Table2'.
ENDIF.

*/appending a new line into Table1

LINE-NumCOL1 = 20. LINE-NumCOL2 = 80.
APPEND LINE TO Table1.

IF Table1 NE Table2.
WRITE / 'Table1 is not equal to Table2'.
ENDIF.

*/appending a new line into Table2

LINE-NumCOL1 = 70. LINE-NumCOL2 = 30.
APPEND LINE TO Table2.
IF Table1 LT Table2.
WRITE / 'Table1 is less than Table2'.
ENDIF.
```

In Listing 7.8, we create two internal tables of the standard type, Table1 and Table2. Table1 is populated initially with four lines and then its content is moved to Table2. Next, Table1 and Table2 are appended by one or two lines simultaneously and both are compared by using comparison operators, such as `GT`, `EQ`, and `LE`. Figure 7.13 shows the output of Listing 7.8:

In Figure 7.13, the first message shows that Table1 is less than Table2. Because Table1 originally had four lines and its data was moved to Table2, Table2 also contains four lines. Now, one line is appended to Table2, making a total of five lines in Table2. Therefore, when the comparison operator `LT` compares the number of lines in Table1 with Table2, it returns the first message.



```
Internal Table Demos

Table1 is less than Table2
Table1 is equal to Table2
Table1 is greater than Table2
Table1 is less than or equal to Table2
Table1 is not equal to Table2
Table1 is less than Table2
```

© SAP AG. All rights reserved.

Figure 7.13: Result of comparing two tables

Now, we append the same line to Table1 and the second comparison operator `EQ` checks whether the number of lines in both tables is equal. Another line is appended to Table1 and the third comparison operator, `GT`, checks whether the number of lines in Table1 is greater than those in Table2. The same line is now appended twice in Table2 and the fourth comparison operator, `LE`, checks whether the number of lines in Table1 is less than or equal to those in Table2. After appending one more line to Table1, the next operator, `NE`, checks to verify if the number of lines in Table1 is not equal to Table2. Finally, one more line is appended to Table2 and the last operator, `LT`, checks whether the number of lines in Table1 is less than the number of lines in Table2.

Performing the Sort Operation in Internal Tables

The `SORT` statement is used to perform the sort operation in an internal table. However, for this to work, the internal table must be of the standard or hashed type. The syntax of the `SORT` statement used to sort a standard or hashed table by using its key is:

```
SORT <internal_tab> [ASCENDING|DESCENDING] [AS TEXT]
[STABLE].
```

In this syntax, the `SORT` statement sorts the `<internal_tab>` internal table in ascending order by using the key of the table. The SORT statement does not apply to the header line of the internal table. The sort order depends on the sequence of fields in the standard key of an internal table. In this case, the default key consists of nonnumeric fields of the lines of the table in the sequence they occur. The ASCENDING or DESCENDING clause is used in the SORT statement to specify the direction of sort operation, which can be either in ascending or descending order. However, the default direction in the SORT statement is ascending.

The larger the sort key is, the greater the time taken by the SAP system to sort the table. If the sort key contains an internal table, the sorting process may slow down considerably.

You cannot sort a sorted table by using the SORT statement. The SAP R/3 system always maintains these tables automatically by their sort order. If an internal table is recognizable statically as a sorted table, using the SORT statement in the table generates a syntax error. However, if the table is a generic sorted table, the SORT statement generates a runtime error under the following three conditions:

- The specified key is not the same as the table key of the sorted table.
- Sorting is done in descending order or in the direction opposite to that of the sorted table.
- Sorting is done by using the AS TEXT clause.

In other words, the SORT statement is allowed only for generic internal tables, if it does not violate the internal sort order.

An internal table with a structured line type can also be sorted by specifying a key in the SORT statement that is different from the default key of the table. The following syntax of the SORT statement shows how to sort an internal table by specifying a different key for the table:

```
SORT  <internal_tab>  [ASCENDING|DESCENDING]  [AS TEXT]
[STABLE]
   BY <internal_tab_field 1> [ASCENDING|DESCENDING] [AS
   TEXT]
   ... <internal_tab_field n> [ASCENDING|DESCENDING] [AS
   TEXT].
```

In this syntax, the <internal_tab> internal table is sorted by the key fields specified in the <internal_tab_field 1>...<internal_tab_field n> expressions instead of by the table key. The number of key fields for the sort operation is limited to 250, and the sort order depends on the sequence of the <internal_tab_field 1>, <internal_tab_field 2>, and <internal_tab_field 3>...<internal_tab_field n> key fields. If the sort order is specified before the BY clause, the sort order applies to all the fields of an internal table. However, if the sort order is specified after the name of the field of the table, the sort order applies only to that field. The default sort order is ascending.

You can specify a key field for the sort operation dynamically by specifying the <internal_tab_field> field instead of using a collection of the <internal_tab_field 1>, <internal_tab_field 2>, <internal_ tab_field 3>...<internal_tab_field n> fields. In such a case, the <internal_tab_field> field is treated as the key field of the table for the sort operation. Note that if the sort operation is applied to an empty field, the sort operation for this field is ignored. Moreover, if the <internal_tab_field> field contains an invalid field name, a runtime error occurs.

Besides using the ASCENDING or DESCENDING clause in the SORT statement, the user can also specify either the entire sort field or each sort field alphabetically by using the AS TEXT clause, as shown in the following syntax:

```
SORT <internal_tab> ... AS TEXT ...
```

If you do not use the AS TEXT clause, strings are sorted according to the sequence specified by the SAP system. If you use the AS TEXT clause, the SAP system sorts the character fields alphabetically. This saves the time that otherwise would have been consumed in converting strings in to a format that can be sorted. Such a conversion is necessary only if you either want to sort an internal table alphabetically and then use it in a binary search or want to construct an alphabetical index for database tables in the user program.

Note Using the AS TEXT clause in the SORT statement only affects the sort fields of the data type C. However, if the AS TEXT clause is applied to a single sort field, then the field must be of the C type.

The STABLE clause is used along with the SORT statement to perform a stable sort. The stable sort is a sort in which the relative sequence of lines, which generally is changed by a sort, remains unchanged. The following syntax shows how to use the STABLE clause with the SORT statement:

```
SORT <internal_tab> ... STABLE.
```

Note that a stable sort takes longer to execute than an unstable sort.

Now, let's consider some examples of how the sort operation is performed in internal tables.

Examples of the Sort Operation in Internal Tables

In this section, we consider different examples of performing the sort operation in internal tables by using the SORT statement along with specific clauses, such as AS TEXT, STABLE, and DESCENDING.

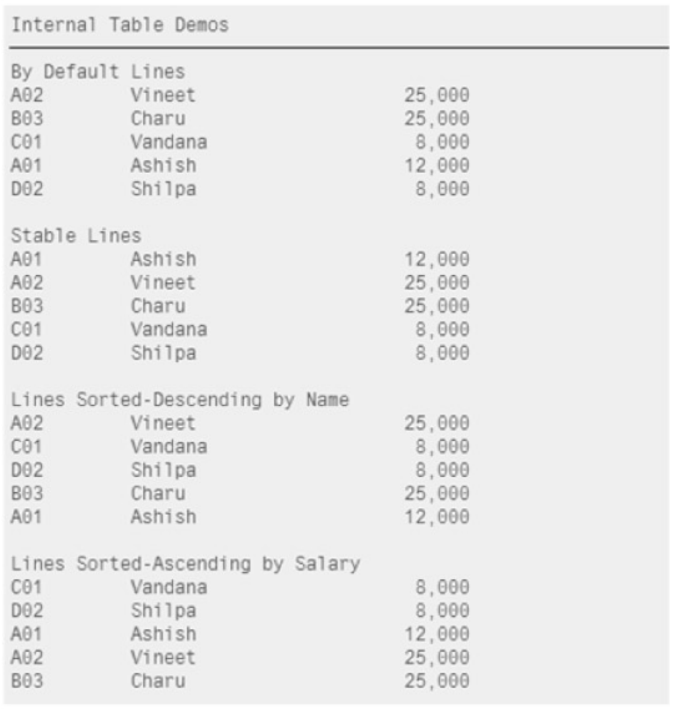
Listing 7.9 shows how to sort an internal table by using the SORT statement with the AS TEXT, STABLE, and DESCENDING clauses:

Listing 7.9: Using the AS TEXT, STABLE, and DESCENDING clauses with the SORT statement

```
REPORT ZINTERNAL_TABLE_DEMO.
DATA: BEGIN OF LINE,
ID(10) Type C,
Name(20) Type C,
Salary Type I,
END OF LINE.
DATA EmployeeDataTable LIKE STANDARD TABLE OF LINE WITH
NON-UNIQUE KEY ID.

*/Appending the lines into EmployeeDataTable

LINE-ID = 'A02'. LINE-NAME = 'Vineet'.
LINE-Salary = 25000.
APPEND LINE TO EmployeeDataTable.
LINE-ID = 'B03'. LINE-NAME = 'Charu'.
LINE-Salary = 25000.
APPEND LINE TO EmployeeDataTable.
LINE-ID = 'C01'. LINE-NAME = 'Vandana'.
LINE-Salary = 8000.
APPEND LINE TO EmployeeDataTable.
LINE-ID = 'A01'. LINE-NAME = 'Ashish'.
LINE-Salary = 12000.
APPEND LINE TO EmployeeDataTable.
LINE-ID = 'D02'. LINE-NAME = 'Shilpa'.
LINE-Salary = 8000.
APPEND LINE TO EmployeeDataTable.

*/Showing the default order of lines

WRITE: / 'By Default Lines'.
PERFORM LOOP_AT_EmployeeDataTable.

*/Showing the stable order of lines

WRITE: / 'Stable Lines'.
SORT EmployeeDataTable STABLE.
PERFORM LOOP_AT_EmployeeDataTable.

*/Showing the lines sorted in descending order of
employee name

WRITE: / 'Lines Sorted-Descending by Name'.
SORT EmployeeDataTable DESCENDING BY Name.
PERFORM LOOP_AT_EmployeeDataTable.

*/Showing the lines sorted in ascending order of employee
salary

WRITE: / 'Lines Sorted-Ascending by Salary'.
SORT EmployeeDataTable ASCENDING BY SALARY.
```

```
PERFORM LOOP_AT_EmployeeDataTable.

FORM LOOP_AT_EmployeeDataTable.
  LOOP AT EmployeeDataTable INTO LINE.
    WRITE: / LINE-ID, LINE-NAME, LINE-Salary.
  ENDLOOP.
  SKIP.
ENDFORM.
```

In Listing 7.9, EmployeeDataTable is an internal standard table with five lines. This table is sorted four times: by the default order, by using the `STABLE` statement, in descending order of the `Name` field, and finally in ascending order of the `Salary` field. The result of these sorts is shown in Figure 7.14:



```
Internal Table Demos

By Default Lines
A02        Vineet                   25,000
B03        Charu                    25,000
C01        Vandana                   8,000
A01        Ashish                   12,000
D02        Shilpa                    8,000

Stable Lines
A01        Ashish                   12,000
A02        Vineet                   25,000
B03        Charu                    25,000
C01        Vandana                   8,000
D02        Shilpa                    8,000

Lines Sorted-Descending by Name
A02        Vineet                   25,000
C01        Vandana                   8,000
D02        Shilpa                    8,000
B03        Charu                    25,000
A01        Ashish                   12,000

Lines Sorted-Ascending by Salary
C01        Vandana                   8,000
D02        Shilpa                    8,000
A01        Ashish                   12,000
A02        Vineet                   25,000
B03        Charu                    25,000
```

© SAP AG. All rights reserved.

Figure 7.14: Sorting of an internal table in different ways

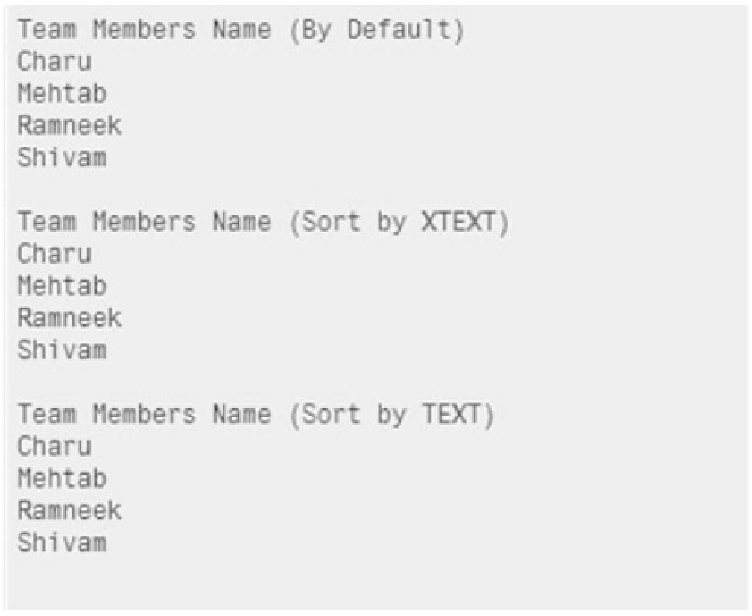
Let's consider another example of sorting, in which an internal table is sorted in the default order, by specifying a key field, and by specifying the `AS TEXT` clause. Listing 7.10 shows an example of sorting by using the key field and the `AS TEXT` clause:

Listing 7.10: An alphabetical sorting by using the key field and the AS TEXT clause

```
REPORT ZINTERNAL_TABLE_DEMO.
DATA: BEGIN OF LINE,
       TEXT(10),
       XTEXT(160) TYPE X,
END OF LINE.

DATA NameTable LIKE HASHED TABLE OF LINE WITH UNIQUE KEY
TEXT.
*/Inserting lines into NameTable

LINE-TEXT = 'Shivam'.

CONVERT TEXT LINE-TEXT INTO SORTABLE CODE
LINE-XTEXT.
INSERT LINE INTO TABLE NameTable.

LINE-TEXT = 'Mehtab'.
```

```
CONVERT TEXT LINE-TEXT INTO SORTABLE CODE
LINE-XTEXT.
INSERT LINE INTO TABLE NameTable.

LINE-TEXT = 'Ramneek'.
CONVERT TEXT LINE-TEXT INTO SORTABLE CODE
LINE-XTEXT.
INSERT LINE INTO TABLE NameTable.

LINE-TEXT = 'Charu'.
CONVERT TEXT LINE-TEXT INTO SORTABLE CODE
LINE-XTEXT.
INSERT LINE INTO TABLE NameTable.

LINE-TEXT = 'Mehtab'.
CONVERT TEXT LINE-TEXT INTO SORTABLE CODE
LINE-XTEXT.
INSERT LINE INTO TABLE NameTable.

*/Used to show the default order of lines

WRITE: 'Team Members Name (By Default)'.
SORT NameTable.
PERFORM LOOP_AT_NameTable.

*/Used to show the lines sorted by XTEXT

WRITE: /'Team Members Name (Sort by XTEXT)'.
SORT NameTable BY XTEXT.
PERFORM LOOP_AT_NameTable.

*/Used to show the lines sorted by TEXT

WRITE: 'Team Members Name (Sort by TEXT)'.
SORT NameTable AS TEXT.
PERFORM LOOP_AT_NameTable.

FORM LOOP_AT_NameTable.
  LOOP AT NameTable INTO LINE.
   WRITE / LINE-TEXT.
  ENDLOOP.
  SKIP.
ENDFORM.
```

In Listing 7.10, NameTable is an internal table that contains a column with character data and another column with the corresponding binary data, which can be sorted alphabetically. NameTable is sorted three times: in the default order, in hexadecimal order, and in alphabetical text order. The default order of the sort is specified by the `BY DEFAULT` clause, hexadecimal order of sort is specified by the `XTEXT` field, and the alphabetical sort is specified by the `TEXT` field. Figure 7.15 shows the result of these three types of sorting:



```
Team Members Name (By Default)
Charu
Mehtab
Ramneek
Shivam

Team Members Name (Sort by XTEXT)
Charu
Mehtab
Ramneek
Shivam

Team Members Name (Sort by TEXT)
Charu
Mehtab
Ramneek
Shivam
```

Figure 7.15: Alphabetical sorting of an internal table

In Figure 7.15, you can see that the team members' names are sorted alphabetically three times: in the default order, in hexadecimal order, and, finally, by using the AS TEXT clause.

Determining the Attributes of Internal Tables

The DESCRIBE TABLE statement is used to determine attributes of an internal table that are not available statically at runtime, such as the initial size, current size, and table type. The following syntax shows the use of the DESCRIBE TABLE statement:

```
DESCRIBE TABLE <internal_tab> [LINES <l>] [OCCURS <n>]
[KIND <k>].
```

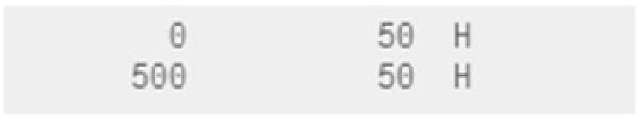
In this syntax, the LINES clause is used to specify the number of populated lines in the <l> expression, the OCCURS clause is used to specify the value of the INITIAL SIZE clause of the internal table in the <n> expression, and the KIND clause is used to specify the table type of the internal table in the <k> expression. Note that the table type specified by T is a standard table. If it is specified by S, it is a sorted table, and if it is specified by H, it is a hashed table. Listing 7.11 shows how to determine the attributes of an internal table by using the DESCRIBE TABLE statement.

Listing 7.11: Using the DESCRIBE TABLE statement

```
REPORT ZDESCRIBE_TABLE_DEMO.

DATA: BEGIN OF LINE,
         COL_1_A TYPE I,
         COL_1_B TYPE I,
      END OF LINE.
DATA Table1 LIKE HASHED TABLE OF LINE WITH UNIQUE KEY
COL_1_A
                         INITIAL SIZE 50.
DATA: Col_2_A TYPE I,
      Col_2_B TYPE I,
      Col_2_C TYPE C.

*/ The DESCRIBE TABLE statement is used to show its initial
size and current size of Table1 before filling the lines
in it.

DESCRIBE TABLE Table1 LINES Col_2_B OCCURS Col_2_A KIND
Col_2_C.
WRITE: / Col_2_B, Col_2_A, Col_2_C.
```

```
DO 500 TIMES.
  LINE-Col_1_A = SY-INDEX.
  LINE-Col_1_B = SY-INDEX ** 3.
INSERT LINE INTO TABLE Table1.
ENDDO.

*/ The DESCRIBE TABLE statement is used to show its initial
size and current size of Table1 after filling the lines in
it.

DESCRIBE TABLE Table1 LINES Col_2_B OCCURS Col_2_A KIND
Col_2_C.
WRITE: / Col_2_B, Col_2_A, Col_2_C.
```

In Listing 7.11, we create Table1 as a hashed table with `LINE` as its line type and `COL_1_A` as its unique key. The initial size of Table1 is declared 50; however, the table is populated with 500 records or lines. The `DESCRIBE TABLE` statement is used to show the number of records and the size of Table1 before and after filling records in it. Figure 7.16 shows the output of Listing 7.11.



```
        0            50  H
      500            50  H
```

© SAP AG. All rights reserved.

Figure 7.16: Displaying the initial and current size of an internal table

Note The number of lines in an internal table can be changed by filling lines within it. However, the initial size of the table remains the same.

Now, let's discuss the operations that can be performed on the individual lines of an internal table, such as reading, modifying, and deleting the lines of the table.

Operations on Individual Lines

As already discussed, various operations can be performed on internal tables (where an internal table is treated as a single data object), such as moving or refreshing data, comparing the records of two or more tables, and determining the attributes of a table. You can also perform DML operations on single or individual lines of an internal table. All these operations involve internal tables as a whole. However, you can perform operations on single or individual lines of an internal table as well. These operations include insertion, modification, and deletion of records, and reading and searching of records within the internal tables.

However, to perform these operations on the lines of an internal table, the user must be familiar with the following two ways of accessing a single table entry:

- **Using the work area—** Acts as a data object for an internal table so that table entries written in the internal table can be sent to the user's program and vice versa. In other words, a work area is an interface between the entries of a table and a user's program. When a user reads or fetches the data from an internal table, the data overwrites the current content of the work area, after which the new data can be used in the user's program. In addition, the data can be written back into an internal table by moving the data from a program to a work area. The header line of an internal table is also used as a work area. Moreover, the work area must be convertible to the line type of the internal table, which means the data type of a work area must be compatible with the line type of a table entry.
- **Using a field symbol—** Acts as a variable that contains an entire line of an internal table. If you access an internal table by using a field symbol, you do not need to copy the data into a work area. When a line is assigned to a field symbol, the changes made in the field symbol are also reflected in the line assigned to it. Moreover, the field symbol must have the same type as the line type of the internal table.

Figure 7.17 shows how to access the data of an internal table by using a work area or a field symbol:

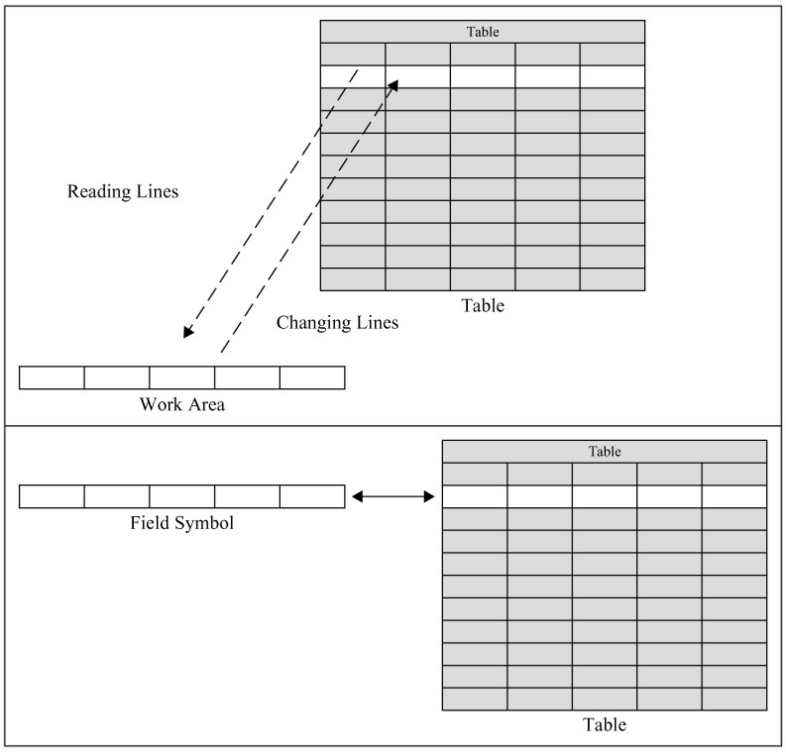


Figure 7.17: Accessing data by using a work area and a field symbol

Various operations that can be performed on individual lines of an internal table include the following:

- Inserting lines into tables
- Inserting summarized lines
- Appending lines
- Reading lines of tables
- Changing lines of tables
- Deleting lines
- Searching table entries or lines
- Maintaining internal tables

Now, let's perform each operation, one by one.

Inserting Lines in Internal Tables

The `INSERT` statement is used to insert a single line or a group of lines into an internal table. The following is the syntax to add a single line to an internal table:

```
INSERT [work_area_itab INTO|INITIAL LINE INTO] internal_tab
[INDEX index_number].
```

In this syntax, the `INSERT` statement inserts a new line in the `internal_tab` internal table. A new line can be inserted by using either the `work_area_itab INTO` expression or the `INITIAL LINE INTO` clause before the internal_tab parameter. When the `work_area_itab INTO` expression is used, the new line is taken from the `work_area_itab` work area and inserted into the internal_tab table. The `INITIAL LINE INTO` clause is used to insert an initial line in the internal_tab table. However, when neither the `work_area_itab INTO` expression nor the `INITIAL LINE INTO` clause is used to insert a line, the line is taken from the header line of the `internal_tab` table.

When a new line is inserted in an internal table by using the `INDEX` clause, the index number of the lines after the inserted line is incremented by 1. If an internal table contains `<index_number> - 1` lines, the new line is added at the end of the table. When the SAP system successfully adds a line to an internal table, the `SY-SUBRC` variable is set to 0. However, if an internal table has less than `<index_number> - 1` lines, the new line cannot be inserted and the `SY-SUBRC` variable is set to 4. Note that without the `INDEX` clause, the `INSERT` statement can be used only in a loop and the new line can be inserted only before the current line.

Standard or sorted internal tables have a non-unique key, which means that these tables can have duplicate entries. However, a runtime error occurs if the user attempts to add a duplicate entry to a sorted table with a unique key. Similarly, a runtime error occurs if the user violates the sort order of a sorted table by appending lines to the table.

The following is the syntax to add multiple lines to an internal table:

```
INSERT LINES OF <internal_tab1> [FROM <n1>] [TO <n 2>] INTO
TABLE <internal_tab2>.
```

In this syntax, `<internal_tab1>` and `<internal_tab2>` are internal tables with a compatible line type. The SAP system inserts the lines of the `<internal_tab1>` table, one-by-one, into the `<internal_tab2>` table by following the same rules as when inserting single lines in an internal table. If the `<internal_tab1>` internal table is an index table, the user can append the first and last lines of the table by specifying them in the `<n1>` and the `<n2>` expressions, respectively.

The following is the syntax to insert multiple lines from one internal table to another by using a line index:

```
INSERT LINES OF <internal_tab1> INTO <internal_tab2> [INDEX
<index_number>].
```

In this syntax, the lines of the `<internal_tab1>` internal table are inserted into the `<internal_tab2>` internal table, one-by-one, by following the same rules as when inserting a single line into an internal table. The `<internal_tab1>` internal table can be any type of internal table. In addition, the line type of the `<internal_tab1>` table must be compatible and convertible with the line type of the `<internal_tab2>` table.

The following is the syntax to insert the lines of an index table into another index table:

```
INSERT LINES OF <internal_tab1> [FROM <n1>] [TO <n2>] INTO
<internal_tab2>
[INDEX <index_number>].
```

In this syntax, the `<n1>` and the `<n2>` expressions specify the indexes of the first and last lines, respectively, of the `<internal_tab1>` internal table that the user wants to insert into the `<internal_tab2>` internal table.

Now, let's consider some examples of how to insert lines to an internal table.

Examples of Line Insertion

In this section, we discuss some examples of how lines are inserted to an internal table by using the `INSERT` statement.

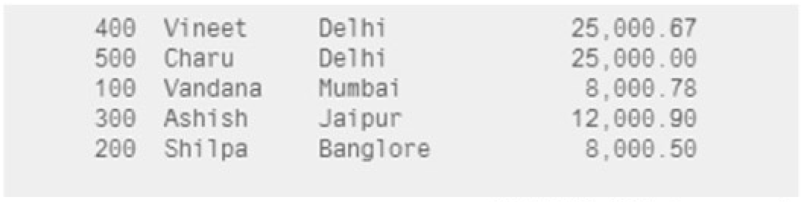
Listing 7.12 shows how to insert lines in a sorted internal table:

Listing 7.12: Inserting lines in a sorted table

```
Report ZINTERNAL_TABLE_DEMO

*/Inserting lines into a standard table EmployeeDataTable
by using INSERT statement

DATA: BEGIN OF LINE,
ID Type I,
Name(10) Type C,
City(10) Type C,
Salary Type P DECIMALS 2,
END OF LINE.
DATA EmployeeDataTable LIKE STANDARD TABLE OF LINE WITH
NON-UNIQUE KEY ID.

LINE-ID = 400. LINE-NAME = 'Vineet'. LINE-City = 'Delhi'.
LINE-Salary = '25000.67'.
INSERT LINE INTO TABLE EmployeeDataTable.
```

```
LINE-ID = 500. LINE-NAME = 'Charu'. LINE-City = 'Delhi'.
LINE-Salary = '25000.00'.
INSERT LINE INTO TABLE EmployeeDataTable.
LINE-ID = 100. LINE-NAME = 'Vandana'. LINE-City =
'Mumbai'. LINE-Salary = '8000.78'.
INSERT LINE INTO TABLE EmployeeDataTable.
LINE-ID = 300. LINE-NAME = 'Ashish'. LINE-City =
'Jaipur'. LINE-Salary = '12000.90'.
INSERT LINE INTO TABLE EmployeeDataTable.

LINE-ID = 200. LINE-NAME = 'Shilpa'. LINE-City =
'Banglore'. LINE-Salary = '8000.50'.
INSERT LINE INTO TABLE EmployeeDataTable.
LOOP AT EmployeeDataTable INTO LINE.
   WRITE: / LINE-ID, LINE-Name, LINE-City,
   LINE-Salary.
ENDLOOP.
```

In Listing 7.12, `EmployeeDataTable` is a standard internal table with `LINE` as its line type and a non-unique key defined on its ID field. Figure 7.18 shows the entries in `EmployeeDataTable`:



```
400  Vineet     Delhi              25,000.67
500  Charu      Delhi              25,000.00
100  Vandana    Mumbai              8,000.78
300  Ashish     Jaipur             12,000.90
200  Shilpa     Banglore            8,000.50
```

© SAP AG. All rights reserved.

Figure 7.18: Displaying the entries of EmployeeDataTable

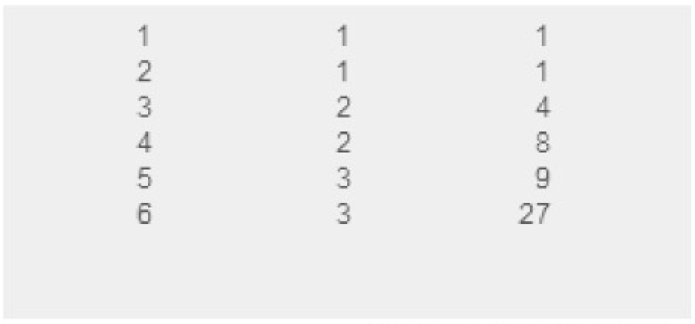
Listing 7.13 shows the insertion of lines from a standard table to a sorted table:

Listing 7.13: Inserting lines of a standard table in a sorted table

```
REPORT ZINTERNAL_TABLE_DEMO.

DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.

*/Creating two internal tables-Table1 and Table2
DATA: Table1 LIKE STANDARD TABLE OF LINE,
       Table2 LIKE SORTED TABLE OF LINE
            WITH NON-UNIQUE KEY COLA COLB.

*/Table1 and Table2 are populated with three lines

*/Table1 contains square of SY-INDEX system variable

DO 3 TIMES.
  LINE-COLA = SY-INDEX. LINE-COLB = SY-INDEX ** 2.
  APPEND LINE TO Table1.
  LINE-COLA = SY-INDEX. LINE-COLB = SY-INDEX ** 3.
  APPEND LINE TO Table2.
ENDDO.

*/Inserting the lines of Table1 into Table2

INSERT LINES OF Table1 INTO TABLE Table2.

LOOP AT Table2 INTO LINE.
  WRITE: / SY-TABIX, LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.13, two internal tables, Table1 and Table2, are created with the same line type but with different table types. The table type of Table1 is standard and the table type of Table2 is sorted. Each table has two columns, `COLA and COLB`, and is populated with three lines. Table1 stores the values of the `SY-INDEX` variable in `COLA` (1, 2, 3) and the square of the values of the `SY-INDEX` variable in `COLB` (1, 4, 9). Table2 stores the values of the `SY-INDEX` variable in `COLA` (1, 2, 3) and the cube of the values of the `SY-INDEX` variable in `COLB` (1, 8, 27).

Now the lines in Table1 are inserted in Table2, as shown in Figure 7.19:



```
1    1    1
2    1    1
3    2    4
4    2    8
5    3    9
6    3   27
```

© SAP AG. All rights reserved.

Figure 7.19: Inserting the lines of one internal table into another

Figure 7.19 displays the square and cube of the values of the `SY-INDEX` variable.

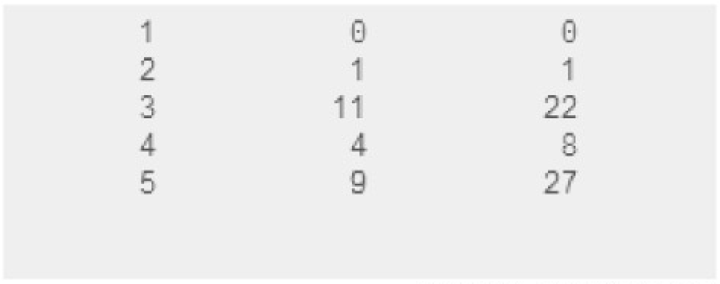
Listing 7.14 shows how to insert the lines of an index table into another index table by using a line index:

Listing 7.14: Using a line index to insert the lines of an index table into another index table

```
REPORT ZINTERNAL_TABLE_DEMO.
DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
DATA: Table1 LIKE STANDARD TABLE OF LINE.
*/Appending three lines into the table
DO 3 TIMES.
  LINE-COLA = SY-INDEX ** 2.LINE-COLB = SY-INDEX ** 3.
  APPEND LINE TO Table1.
  ENDDO.
*/Inserting a line into Table1 at the index number 2
LINE-COLA = 11. LINE-COLB = 22.
INSERT LINE INTO Table1 INDEX 2.
*/Inserting the first line into Table1 at the index
number 1
INSERT INITIAL LINE INTO Table1 INDEX 1.

LOOP AT Table1 INTO LINE.
  WRITE: / SY-TABIX, LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.14, Table1 is a standard internal table, initially populated with three lines. The `COLA` field of Table1 contains the square of the values of the `SY-INDEX` variable (1, 4, 9) and the `COLB` field of Table1 contains the cube of the values of the `SY-INDEX` variable (1, 8, 27). A new line is inserted at index number 2, so that the original line is shifted to index number 3, and so on. Now, we insert an initial line at index 1, so that the rest of the lines are shifted once again to their next places.

Figure 7.20 shows the output of Listing 7.14:



```
         1              0              0
         2              1              1
         3             11             22
         4              4              8
         5              9             27
```

© SAP AG. All rights reserved.

Figure 7.20: Inserting lines by using the line index

Listing 7.15 shows how to insert the lines of one internal table into another internal table by using line index in a loop construction:

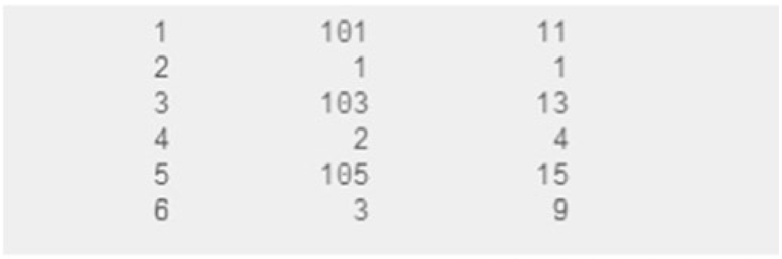
Listing 7.15: Using the line index in a loop construction

```
REPORT ZINTERNAL_TABLE_DEMO.

DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
DATA: Table1 LIKE STANDARD TABLE OF LINE.
DO 3 TIMES.
  LINE-COLA = SY-INDEX. LINE-COLB = SY-INDEX ** 2.
  APPEND LINE TO Table1.
  ENDDO.
*/Using a line index in a table
LOOP AT Table1 INTO LINE.
  LINE-COLA = 100 + SY-TABIX. LINE-COLB = 10 + SY-TABIX.
  INSERT LINE INTO Table1.
ENDLOOP.

LOOP AT Table1 INTO LINE.
  WRITE: / SY-TABIX, LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.15, Table1 is a standard internal table that contains the `COLA` and the `COLB` fields. Initially, Table1 is populated with three lines, in which the `COLA` field contains the value of the `SY-INDEX` variable (1, 2, 3) and the `COLB` field contains the square of the values of the `SY-INDEX` variable (1, 4, 9). Now, we insert a new line before each of the previously stored lines in Table1. This means Table1 has a total of six lines. The new lines store the addition of 100 to the corresponding values of the `SY-TABIX` variable in the `COLA` field and the addition of 10 to the corresponding values of the `SY-TABIX` variable in the `COLB` field.

Therefore, the `COLA` field of Table1 has the values 101, 1, 103, 2, 105, and 3, and the `COLB` field of Table1 has the values 11, 1, 13, 4, 15, and 9.

Figure 7.21 shows the output of Listing 7.15:



```
         1            101             11
         2              1              1
         3            103             13
         4              2              4
         5            105             15
         6              3              9
```

© SAP AG. All rights reserved.

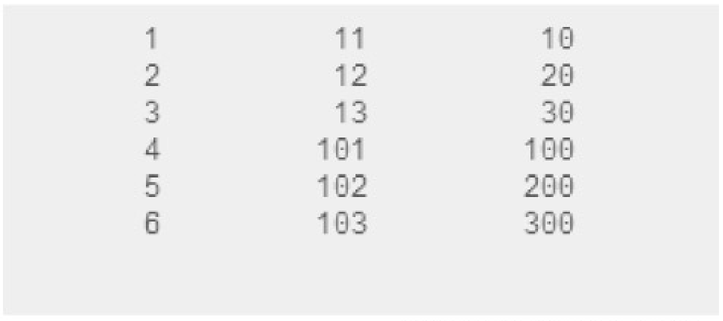
Figure 7.21: Inserting lines by using a line index in a loop construction

Listing 7.16 shows how to insert the lines of one standard table before the lines of another standard table by using an index number:

Listing 7.16: Inserting the lines of one table before the lines of another table

```
REPORT ZINTERNAL_TABLE_DEMO.
DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
*/Creating two internal tables-Table1 and Table2
DATA: Table1 LIKE STANDARD TABLE OF LINE, Table2 LIKE
Table1.
*/Each table is populated with three lines
DO 3 TIMES.

  LINE-COLA = SY-INDEX + 10. LINE-COLB = SY-INDEX * 10.
  APPEND LINE TO Table1.
  LINE-COLA = SY-INDEX + 100. LINE-COLB = SY-INDEX * 100.
  APPEND LINE TO Table2.
  ENDDO.
*/Inserting the data of Table1 before the data of Table2
INSERT LINES OF Table1 INTO Table2 INDEX 1.
LOOP AT Table2 INTO LINE.
  WRITE: / SY-TABIX, LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.16, there are two internal tables, Table1 and Table2. Each table has two columns, `COLA` and `COLB`, and is populated with three lines. Table1 stores the addition of 10 to the values of the `SY-INDEX` variable in `COLA` (11, 12, 13) and the multiplication of 10 with the values of the `SY-INDEX` variable in `COLB` (10, 20, 30). Table2 stores the addition of 100 to the values of the `SY-INDEX` variable in `COLA` (101, 102, 103) and the multiplication of 100 with the values of the `SY-INDEX` variable in `COLB` (100, 200, 300). Finally, Table1 is inserted completely before the first line of Table2, as shown in Figure 7.22:



```
1          11         10
2          12         20
3          13         30
4         101        100
5         102        200
6         103        300
```

Figure 7.22: Inserting data of a table before the first line of another table

In Figure 7.22, the first three lines show the addition to and the multiplication of 10 with the corresponding values of the `SY-INDEX` variable and the last three lines show the addition and multiplication of 100 with the corresponding values of the `SY-INDEX` variable.

Inserting Summarized Lines in Internal Tables

In ABAP, the `COLLECT` statement is used to summarize the data of internal tables. The syntax to summarize the entries of an internal table is as follows:

```
COLLECT <work_area_itab> INTO <internal_tab>.
```

In this syntax, the `<internal_tab>` expression is an internal table and the `<work_area_itab>` expression is a work area. All the fields of an internal table that are not part of the key of the table should be of numeric type. In other words, the fields that are not part of the table key should be of either `F`, `I`, or `P` type. The user specifies the line to be added in a work area that is compatible with the line type.

When we use the `COLLECT` statement to add a new line or record in an internal table, the SAP system checks whether the line with the same key value already exists in the table. If no such line exists, the `COLLECT` statement inserts the line in the internal table. However, if a line having the same key value is found in the table, the `COLLECT` statement adds the content of numeric fields in the work area with the content of numeric fields in the existing entry of the table.

Note The user should use the `COLLECT` statement only to create summarized tables. Using other statements to insert table entries may result in duplicate entries in an internal table.

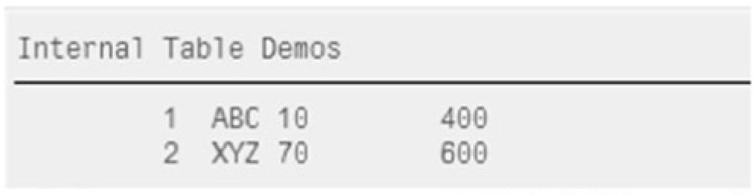
Listing 7.17 shows the use of the `COLLECT` statement.

Listing 7.17: Using of the COLLECT statement

```
REPORT ZINTERNAL_TABLE_DEMO.

*/Creating an internal table with three fields, where Field1
and Field2 would be act as non-unique key

DATA: BEGIN OF LINE,
        Field1(3) TYPE C,
        Field2(2) TYPE N,
        Field3 TYPE I,
       END OF LINE.
DATA MyTable LIKE SORTED TABLE OF LINE
WITH NON-UNIQUE KEY Field1 Field2.

*/ The COLLECT statement used to summarize the values of
table fields, on the basis of key fields

LINE-Field1 = 'ABC'. LINE-Field2 = '10'.
LINE-Field3 = 100.
COLLECT LINE INTO MyTable.

LINE-Field1 = 'XYZ'. LINE-Field2 = '70'.
LINE-Field3 = 600.
COLLECT LINE INTO MyTable.

LINE-Field1 = 'ABC'. LINE-Field2 = '10'.
LINE-Field3 = 300.
COLLECT LINE INTO MyTable.

LOOP AT MyTable INTO LINE.
  WRITE: / SY-TABIX, LINE-Field1, LINE-Field2,
  LINE-Field3.
ENDLOOP.
```

In Listing 7.17, MyTable is a sorted internal table containing three fields: `Field1`, `Field2`, and `Field3`. The MyTable table has `LINE` as its line type and a combination of the `Field1` and `Field2` fields as its unique key. The `COLLECT` statement is used three times to populate MyTable. The first two `COLLECT` statements are used to populate the ABC and XYZ values in `Field1`, the 10 and 70 values in `Field2`, and the 100 and 600 numbers in `Field3`. The third or last `COLLECT` statement is used to add 300 in `Field3`, where the values in `Field1` and `Field2` are ABC and 10, respectively. Figure 7.23 shows the output of the MyTable table:

In Figure 7.23, `Field3` shows the number 400 after adding the numbers 100 and 300, where the value of `Field1` is ABC and the value of `Field2` is 10.



```
Internal Table Demos

         1  ABC 10          400
         2  XYZ 70          600
```

© SAP AG. All rights reserved.

Figure 7.23: Result of the COLLECT statement

Appending Lines to Internal Tables

The `APPEND` statement is used to add a single row or line to an existing internal table. This statement copies a single line from a work area and inserts it after the last existing line in an internal table. The work area can be either a header line or any other field string with the same structure as a line of an internal table. The following is the syntax of the `APPEND` statement used to append a single line in an internal table:

```
APPEND <record_for_itab> TO <internal_tab>.
```

In this syntax, the `<record_for_itab>` expression can be represented by the `<work_area_itab>` work area, which is convertible to a line type or by the `INITIAL LINE` clause. If the user uses a `<work_area_itab>` work area, the SAP system adds a new line to the `<internal_tab>` internal table and populates it with the content of the work area. The `INITIAL LINE` clause appends a blank line that contains the initial value for each field of the table structure. After each `APPEND` statement, the `SY-TABIX` variable contains the index number of the appended line.

Appending lines to standard and sorted tables with a non-unique key works regardless of whether the lines with the same key already exist in the table. In other words, duplicate entries may occur. However, a runtime error occurs if the user attempts to add a duplicate entry to a sorted table with a unique key or if the user violates the sort order of a sorted table by appending the lines to it.

The user can also append the entries of an internal table to another internal table by using the following syntax of the `APPEND` statement:

```
APPEND LINES OF <internal_tab1> TO <internal_tab2>.
```

In this syntax, the `<internal_tab1>` and `<internal_tab2>` expressions are two internal tables. The lines of the `<internal_tab1>` table are appended to the `<internal_tab2>` table. The `<internal_tab1>` table can be of any type, but its line type must be convertible to the line type of the `<internal_tab2>` table.

The following syntax is used to append the lines of one index table to another index table:

```
APPEND LINES OF <internal_tab1> [FROM <n1>] [TO <n2>] TO
<internal_tab2>.
```

In this syntax, the `<n1>` and `<n2>` expressions are the indexes of the first and last lines, respectively, of the `<internal_tab1>` table, which the user wants to append to the `<internal_tab2>` table.

When the user appends several lines to a sorted table, the unique key (if defined) must not violate the sort order; otherwise, a runtime error will occur.

The `APPEND` statement is also used to create ranked lists in standard tables. To do this, create an empty table and then use the following syntax of the `APPEND` statement:

```
APPEND <work_area_itab> TO <internal_tab> SORTED BY
<internal_tab_field>.
```

In this syntax, the new line added by using the `APPEND` statement cannot be appended at the end of the `<internal_tab>` table. Instead, the table is sorted by the `<internal_tab_field>` field in descending order. The `<work_area_itab>` work area must be compatible with the line type of the `<internal_tab>` table.

Note that if you select the table type as a sorted table, you cannot use the `SORTED BY` clause.

Using the *`APPEND`* Statement

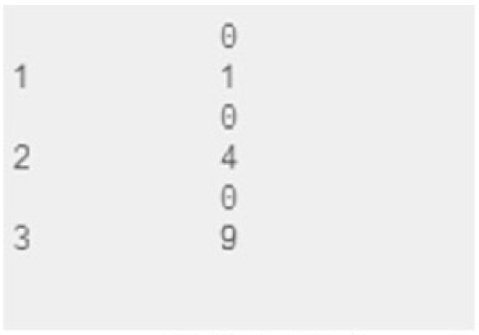
Listing 7.18 shows the use of the `APPEND` statement in the `DO` loop:

Listing 7.18: Using the APPEND statement in the DO loop

```
REPORT ZINTERNAL_TABLE_DEMO.
*/Creating an internal table
DATA: BEGIN OF ROW,
          COLA TYPE C,
          COLB TYPE I,
        END OF ROW.
```

```
DATA MyTable LIKE TABLE OF ROW.
*/Using APPEND statement in the DO loop
DO 3 TIMES.
  APPEND INITIAL LINE TO MyTable.
  ROW-COLA = SY-INDEX. ROW-COLB = SY-INDEX ** 2.
  APPEND ROW TO MyTable.
ENDDO.

LOOP AT MyTable INTO ROW.
  WRITE: / ROW-COLA, ROW-COLB.
ENDLOOP.
```

In Listing 7.18, we create the MyTable internal table with `ROW` as its line type, and `COLA` and `COLB` as its two columns. MyTable is populated with six lines by using the `DO` loop. Each time the loop is processed, an initialized line is appended to the MyTable table and the work area of the table is populated with the values of the `SY-INDEX` variable (in the `COLA` field) and the square of the values of the `SY-INDEX` variable (in the `COLB` field). Figure 7.24 shows the output of Listing 7.18:



```
             0
1            1
             0
2            4
             0
3            9
```

© SAP AG. All rights reserved.

Figure 7.24: The result of the APPEND statement

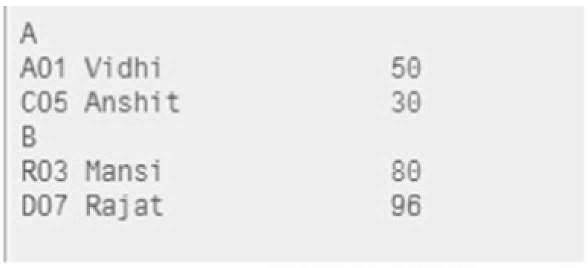
Listing 7.19 shows how to append the lines of one table with that of another table.

Listing 7.19: Appending the lines of one table with another table

```
REPORT ZAPPEND_TABLE_DEMO.
*/Appending one table by another table
DATA: BEGIN OF Record1,
         ID(3) TYPE C,
         Name(10) TYPE C,
         Marks TYPE I,
       END OF Record1,
       Stud1 LIKE TABLE OF Record1.

DATA: BEGIN OF Record2,
         ColumnA(1) TYPE C,
         ColumnB LIKE Stud1,
       END OF Record2,
       Stud2 LIKE TABLE OF Record2.

Record1-ID = 'AO1'. Record1-Name = 'Vidhi'.
Record1-Marks = 50.
APPEND Record1 TO Stud1.

Record1-ID = 'CO5'. Record1-Name = 'Anshit'.
Record1-Marks = 30.
APPEND Record1 TO Stud1.
Record2-ColumnA = 'A'. Record2-ColumnB = Stud1.
APPEND Record2 TO Stud2.

REFRESH Stud1.

Record1-ID = 'RO3'. Record1-Name = 'Mansi'.
Record1-Marks = 80.
```

```
APPEND Record1 TO Stud1.
Record1-ID = 'DO7'. Record1-Name = 'Rajat'.
Record1-Marks = 96.
APPEND Record1 TO Stud1.

Record2-ColumnA = 'B'. Record2-ColumnB = Stud1.
APPEND Record2 TO Stud2.

LOOP AT Stud2 INTO Record2.
  WRITE: / Record2-ColumnA.
  LOOP AT Record2-ColumnB INTO Record1.
  WRITE: / Record1-ID, Record1-Name, Record1-Marks.
  ENDLOOP.
ENDLOOP.
```

In Listing 7.19, two internal tables, Stud1 and Stud2, are created. The line types of the Stud1 and Stud2 tables are `Record1` and `Record2`, respectively. The `Record1` line type contains the `ID`, `Name`, and `Marks` fields, while the `Record2` line type contains the `ColumnA` and `ColumnB` fields. The data types of the `ID`, `Name`, `Marks`, and `ColumnA` fields are elementary, while the data type of the `ColumnB` field is itself an internal table—that is, Stud1. The `APPEND` statement is used to populate the Stud1 and Stud2 tables by using their respective line types. Furthermore, the `REFRESH` statement is used to refresh the data of the Stud1 table. Figure 7.25 shows the lines stored in the Stud2 table by using the table in a loop:



```
A
A01 Vidhi              50
C05 Anshit             30
B
R03 Mansi              80
D07 Rajat              96
```

© SAP AG. All rights reserved.

Figure 7.25: The result of appending the lines of one table to another table

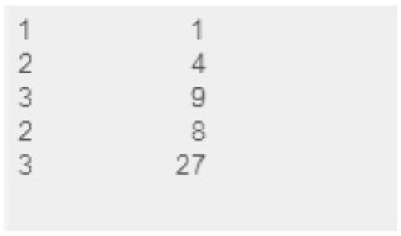
Listing 7.20 shows how to append specific lines of one table to another:

Listing 7.20: Appending specific lines of one index table to another table

```
REPORT ZAPPEND_TABLE_DEMO.
*/Creating two internal tables
DATA: BEGIN OF LINE,
COLA TYPE C,
COLB TYPE I,
END OF LINE.

DATA: Table1 LIKE TABLE OF LINE, Table2 LIKE Table1.
*/Inserting lines into the internal tables

DO 3 TIMES.
  LINE-COLA = SY-INDEX. LINE-COLB = SY-INDEX ** 2.
  APPEND LINE TO Table1.
  LINE-COLA = SY-INDEX. LINE-COLB = SY-INDEX ** 3.
  APPEND LINE TO Table2.
  ENDDO.
*/Appending the lines of Table2 to Table1
APPEND LINES OF Table2 FROM 2 TO 3 TO Table1.
*/Displaying the lines of Table1
LOOP AT Table1 INTO LINE.
  WRITE: / LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.20, two internal tables, Table1 and Table2, are created with the `LINE` line type. Each table has two columns, `COLA` and `COLB`, and is populated with three lines. Table1 stores the values of the `SY-INDEX` variable in `COLA` (1, 2, 3) and the square of the values of the `SY-INDEX` variable in `COLB` (1, 4, 9). Table2 stores the values of the `SY-INDEX` variable in `COLA` (1, 2, 3) and the cube of the values of the `SY-INDEX` variable in `COLB` (1, 8, 27). The last two lines of Table2 are then appended to Table1, as shown in Figure 7.26:

Listing 7.21 shows how to use the `APPEND` statement with the `SORTED BY` clause.



```
1          1
2          4
3          9
2          8
3         27
```

Figure 7.26: Appending the last two lines from a table to another table

Listing 7.21: Using the APPEND statement with the SORTED BY clause

```
REPORT ZAPPEND_TABLE_DEMO.

*/Creating an internal table with the initial size 2

DATA: BEGIN OF ROW,
         COL1 TYPE I,
         COL2 TYPE I,
         COL3 TYPE I,
       END OF ROW.
DATA MyTable LIKE TABLE OF ROW INITIAL SIZE 2.
*/Appending the lines in the table, by the SORTED BY clause,
so that the table is sorted by COL2 field in descending
order.

ROW-COL1 = 1. ROW-COL2 = 10. ROW-COL3 = 100.
APPEND ROW TO MyTable SORTED BY COL2.

ROW-COL1 = 2. ROW-COL2 = 20. ROW-COL3 = 200.
APPEND ROW TO MyTable SORTED BY COL2.

ROW-COL1 = 3. ROW-COL2 = 30. ROW-COL3 = 300.
APPEND ROW TO MyTable SORTED BY COL2.

ROW-COL1 = 4. ROW-COL2 = 40. ROW-COL3 = 400.
APPEND ROW TO MyTable SORTED BY COL2.

ROW-COL1 = 5. ROW-COL2 = 50. ROW-COL3 = 500.
APPEND ROW TO MyTable SORTED BY COL2.

LOOP AT MyTable INTO ROW.
  WRITE: / ROW-COL1, ROW-COL2, ROW-COL3.
ENDLOOP.
```

In Listing 7.21, MyTable is an internal table with three columns: `COL1`, `COL2`, and `COL3`. In this table, we append five lines by using the `APPEND` statement and sort the lines in descending order by using the `SORTED BY` clause with the `COL2` field. However, the last three lines of MyTable are skipped because we have specified the initial size of the table as 2 by using the `INITIAL SIZE` clause. As a result, only two lines with the largest values in the `COL2` field of MyTable are displayed, as shown in Figure 7.27:

```
5          50          500
4          40          400
```

Figure 7.27: Using the APPEND statement with the SORTED BY clause

Reading the Lines of Internal Tables

In ABAP, you can read the lines of a table by using the following syntax of the `READ TABLE` statement:

```
READ TABLE <internal_tab> <key> <result>.
```

In this syntax, you have to specify a key for the table. Remember that you must specify the line by using the key, not the index. The key is specified in the `<key>` expression and the `<result>` expression is used to specify an additional processing option for the line to be retrieved.

Now, let's learn how to specify a key in the `<key>` expression and a processing option in the `<result>` expression.

Specifying a Key in the <key> Expression

In the `<key>` expression, you can specify either a table key or a different key. A table key is required to search an internal table and is specified by using the `READ TABLE` statement. The syntax to use the `READ TABLE` statement is as follows:

```
READ TABLE <internal_tab> FROM <work_area_itab>.
```

or

```
READ TABLE <internal_tab> WITH TABLE KEY <k1> = <f1> ...
<kn> = <fn>.
```

In the first syntax, the `<work_area_itab>` expression represents a work area that is compatible with the line type of the `<internal_tab>` table. The values in the key fields of this table are taken from the corresponding components of the work area.

In the second syntax, you need to provide the values of each key field explicitly. If the name of any key field is not known until the execution of the program, you can specify it as the content of the `<ni>` field by using the `(<ni>) = <fi>` form. If the data types of the `<f1>`, `<f2>...<fn>` fields are not compatible with the corresponding key fields, the SAP system converts them to a compatible type automatically.

You can specify a search key, but not a table key, within the `READ` statement by using the `WITH KEY` clause, as shown in the following syntax:

```
READ TABLE <internal_tab> WITH KEY = <internal_tab_
field>.
```

or

```
READ TABLE <internal_tab> WITH KEY <internal_tab_key1> =
<internal_tab_field1> ...
<internal_tab_key n> = <internal_tab_field n>.
```

In the first syntax, the entire line of the internal table is used as a search key. The content of the entire line of the table is compared with the content of the `<internal_tab_field>` field. If the values of the `<internal_tab_field>` field are not compatible with the line type of the table, these values are converted according to the line type of the table. The search key allows you to find entries in internal tables that do not have a structured line type; that is, where the line is a single field or an internal table type.

In the second syntax, the search key is specified by using the `<internal_tab_key 1>...<internal_tab_key` *n*`>` table fields. If the data types of the `<internal_tab_field 1>...<internal_tab_field` *n*`>` fields are not compatible with the components in the internal table, the SAP system first converts them into compatible forms and then performs the read operation.

Specifying a Processing Option in the <result> Expression

You can specify a work area or a field symbol in the `<result>` expression. The following syntax of the `READ` statement is used to specify a work area or field symbol by using the `COMPARING` or `TRANSPORTING` clause:

```
READ TABLE <internal_tab> <key> INTO <work_area_itab>
[COMPARING <f1> <f2> ...
                        |ALL FIELDS]
            [TRANSPORTING <f1> <f2> ...
                            |ALL FIELDS
                            |NO FIELDS].
```

If the `COMPARING` or `TRANSPORTING` clause is not used, the content of the table line must be convertible into the data type of the `<work_area_itab>` work area, and if the clause is specified, the line type and work area must be compatible. You must always use a work area that is compatible with the line type of the relevant internal table.

When the `COMPARING` clause is used, the specified table fields `<f1>`, `<f2>....<fn>` of the structured line type are compared with the corresponding fields of the work area before being transported. If the `ALL FIELDS` clause is specified, the SAP system compares all the components. When the SAP system finds an entry on the basis of a key, the value of the `SY-SUBRC` variable is set to 0. In addition, the value of the `SY-SUBRC` variable is set to 2 or 4 if the content of the compared fields is not the same or if the SAP system cannot find an entry. However, the SAP system copies the entry into the target work area whenever it finds an entry, regardless of the result of the comparison.

The `TRANSPORTING` clause is used to specify the table fields of the structured line type that need to be transported into the work area. If the `ALL FIELDS` clause is specified without using the `TRANSPORTING` clause, the content of all the fields is transported. However, the `NO FIELDS` clause is specified, no content is transported.

In both clauses (`COMPARING` and `TRANSPORTING`), you can dynamically specify a field `<fi>` as the content of the `<ni>` field in the `<ni>` form. When the `READ TABLE` statement is executed, the `<ni>` field is ignored if it is empty.

You can assign a table line or table entry to a field symbol, taken from an internal table, by using the following syntax:

```
READ TABLE <internal_tab> <key> ASSIGNING <internal_tab_
fieldsymbol>.
```

In this syntax, a table entry from the `<internal_tab>` table is assigned to the `<internal_tab_fieldsymbol>` field symbol by using the `ASSIGNING` clause in the `READ TABLE` statement. The data type of the field symbol must be compatible with the line type of the table.

Examples of Reading Lines from Tables

In this section, we consider some examples to read one of more lines from internal tables by using the `READ` statement.

Listing 7.22 shows how to read lines by using the `READ TABLE` statement and the `COMPARING` clause:

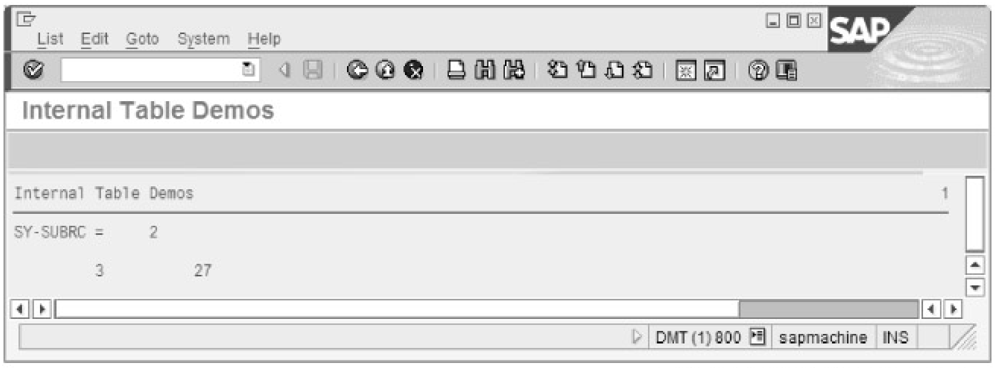
Listing 7.22: Using the COMPARING clause in the READ TABLE statement

```
REPORT ZREADLINES_DEMO.
*/Creating an internal table
DATA: BEGIN OF RECORD,
COLA TYPE I,
COLB TYPE I,
END OF RECORD.
DATA MyTable LIKE HASHED TABLE OF RECORD WITH UNIQUE KEY
COLA.
*/Filling the internal table with four lines
DO 4 TIMES.
  RECORD-COLA = SY-INDEX.
  RECORD-COLB = SY-INDEX ** 3.
INSERT RECORD INTO TABLE MyTable.
ENDDO.
*/Reading the lines of the internal table, on the basis COLA
but not on COLB, as COLA is unique key of the table

RECORD-COLA = 3. RECORD-COLB = 64.

READ TABLE MyTable FROM RECORD INTO RECORD COMPARING
COLB.

WRITE: 'SY-SUBRC =', SY-SUBRC.
```

```
SKIP.
WRITE: / RECORD-COLA, RECORD-COLB.
```

In Listing 7.22, MyTable is an internal table of the hashed table type, with RECORD as the work area and COLA as the unique key. Initially, MyTable is populated with four lines, where the COLA field contains the values of the SY-INDEX variable and the COLB field contains the cube of the values of the SY-INDEX variable. The RECORD work area is populated with 3 and 64 as values for the COLA and COLB fields, respectively. The READ statement reads the line of the table after comparing the value of the COLA key field with the value in the RECORD work area by using the COMPARING clause, and then copies the content of the read line in the work area. Figure 7.28 shows the content of the RECORD work area:

In Figure 7.28, the value of the SY-SUBRC variable is displayed as 2 because when the value in the COLA field is 3, the value in the COLB field is not 64, but 27.



```
Internal Table Demos                                1
SY-SUBRC =      2

        3          27
```

© SAP AG. All rights reserved.

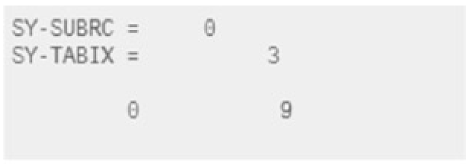
Figure 7.28: The result of using the COMPARING clause

Listing 7.23 shows how to read lines from a sorted table by using the TRANSPORTING clause:

Listing 7.23: Reading lines by using the TRANSPORTING clause

```
REPORT ZREADLINES_DEMO.
DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
DATA MyTable LIKE Sorted TABLE OF LINE WITH UNIQUE KEY
COLA.

DO 4 TIMES.
  LINE-COLA = SY-INDEX.
  LINE-COLB = SY-INDEX ** 2.
INSERT LINE INTO TABLE MyTable.
ENDDO.

CLEAR LINE.
*/ Using the TRANSPORTING clause to transport the table
fields into the work area
READ TABLE MyTable WITH TABLE KEY COLA = 3
                 INTO LINE TRANSPORTING COLB.
WRITE: 'SY-SUBRC =', SY-SUBRC,
        / 'SY-TABIX =', SY-TABIX.
SKIP.
WRITE: / LINE-COLA, LINE-COLB.
```

In Listing 7.23, MyTable is a sorted table with the LINE line type, initially populated with four lines. In MyTable, the COLA field contains the values of the SY-INDEX variable and the COLB field contains the square of the values of the SY-INDEX variable. The READ statement reads the line of the table in which the COLA field has the same value as that in the work area and copies the line to the work area. Only the content of the COLB field is copied to the LINE work area. The value of the SY-SUBRC variable is 0 and that of the SY-TABIX variable is 3, because MyTable is an index table. Figure 7.29 shows the output of Listing 7.23:



```
SY-SUBRC =     0
SY-TABIX =         3

        0          9
```

© SAP AG. All rights reserved.

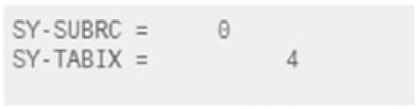
Figure 7.29: The result of using the TRANSPORTING clause

Listing 7.24 shows how to read lines by using a search key other than a table key:

Listing 7.24: Reading lines of tables by using a search key different from a table key

```
REPORT ZREADLINES_DEMO.
DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
DATA MyTable LIKE Sorted TABLE OF LINE WITH UNIQUE KEY
COLA.
DO 4 TIMES.
  LINE-COLA = SY-INDEX.
  LINE-COLB = SY-INDEX ** 2.
INSERT LINE INTO TABLE MyTable.
ENDDO.
*/ Using the TRANSPORTING NO FIELDS clause, so that no
fields are transported
READ TABLE MyTable WITH KEY COLB = 16 TRANSPORTING NO
FIELDS.

WRITE: 'SY-SUBRC =', SY-SUBRC,
/ 'SY-TABIX =', SY-TABIX.
```

In Listing 7.24, MyTable is a sorted table with the LINE line type, initially populated with four lines. In MyTable, the COLA field contains the values of the SY-INDEX variable and the COLB field contains the square of the values of the SY-INDEX variable. The COLA field is the table key of the MyTable table. However, the READ statement reads the line of the table on the basis of another key, which is defined in the COLB field. The result of the operation performed by the READ statement is represented by the SY-SUBRC and SY-TABIX variables. In this case, the READ statement successfully finds the line where the COLB field is 16 and shows the values of the SY-SUBRC and SY-TABIX variables as 0 and 4, respectively, as shown in Figure 7.30:



```
SY-SUBRC =     0
SY-TABIX =         4
```

© SAP AG. All rights reserved.

Figure 7.30: Reading table lines by using a different key

Listing 7.25 shows how to read the lines of tables by using field symbols:

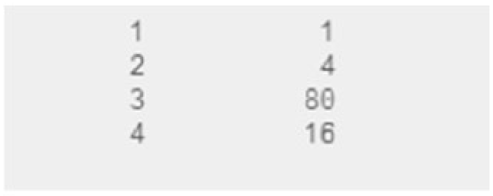
Listing 7.25: Reading lines of tables by using field symbols

```
REPORT ZREADLINES_DEMO.
DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
DATA MyTable LIKE Hashed TABLE OF LINE WITH UNIQUE KEY
COLA.
*/Declaring a field symbol
```

```
FIELD-SYMBOLS <MyField> LIKE LINE OF MyTable.
DO 4 TIMES.
LINE-COLA = SY-INDEX.
LINE-COLB = SY-INDEX ** 2.
INSERT LINE INTO TABLE MyTable.
ENDDO.
*/Reading a table with field symbol

READ TABLE MyTable WITH TABLE KEY COLA = 3 ASSIGNING
<MyField>.

<MyField>-COLB = 80.
LOOP AT MyTable INTO LINE.
  WRITE: / LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.25, MyTable is a hashed table with the `LINE` line type and the `COLA` and `COLB` fields. Initially, MyTable is populated with four lines, where the `COLA` field contains the values 1, 2, 3, and 4, while the `COLB` field contains the values 1, 4, 9, and 16. The `READ` statement reads the line of the table in which the value of the `COLA` field is 3 and assigns it to the `<MyField>` field symbol. In Listing 7.25, the value 80 is assigned to the `COL2` field of the `<MyField>` field symbol, which changes the value of the corresponding table field. Figure 7.31 shows the changed value of the `COLB` field when the value of the `COLA` field is 3:

In Figure 7.31, the value of the `COLB` field in the third line changes from 9 to 80.



```
1        1
2        4
3       80
4       16
```

© SAP AG. All rights reserved.

Figure 7.31: Assigning a value by using a field symbol

Modifying the Lines of Internal Tables

In ABAP, the `MODIFY` statement is used to modify the content of one or more records stored in an internal table. To modify the content of a single record of an internal table, you must specify a table key in the `MODIFY` statement. Similarly, to modify the content of multiple rows, you must specify conditions within the `MODIFY` statement.

Besides the `MODIFY` statement, the `WRITE TO` statement is used to change the content of the lines of an internal table.

Modifying the lines of an internal table involves the following operations:

- Modify a line by using a table key
- Modify lines by using a condition

Now, let's discuss each in detail.

Modifying a Line by Using a Table Key

The `MODIFY` statement modifies a single line of an internal table. The following syntax shows the use of the `MODIFY` statement:

```
MODIFY TABLE <internal_tab> FROM <work_area_itab>
[TRANSPORTING <f1> <f2> ...].
```

In this syntax, `<internal_tab>` represents an internal table, `<work_area_itab>` represents the work area of the `<internal_tab>` table, and `<f1> <f2>...<fn>` represent non-key fields of the `<work_area_itab>` work area. The `MODIFY` statement is used to modify the content of a line of the `<internal_tab>` table with the help of the `<work_area_itab>` work area. The `<work_area_itab>` work area must be compatible with the line type of the

<internal_tab> table. It is used for two purposes: first, to find the line that the user wants to change, and second, to insert a new line in the table.

The TRANSPORTING clause is used to specify the non-key fields that can be assigned to the table line. The SAP system searches the internal table for the line whose table key corresponds to the key fields in the <work_ area_itab> work area. If a line is retrieved successfully in the specified internal table, the content of the non-key fields of the work area is copied to the corresponding fields of the line and the value of the SY-SUBRC variable is set to 0; otherwise, the value of the SY-SUBRC variable is set to 4. If the table has a non-unique key and the SAP system finds duplicate entries, it modifies the first entry.

Use the following syntax of the MODIFY statement to change lines in tables by using their indexes is as follows:

```
MODIFY <internal_tab> FROM <work_area_itab> [INDEX <index_
number>] [TRANSPORTING <f1> <f2> ... ].
```

In this syntax, the work area <work_area_itab> specified in the FROM clause replaces the existing line in the <internal_tab> internal table. The work area must be convertible into the line type of the internal table.

If the INDEX option is used, the content of the work area overwrites the content of the line with the <index_number> index; if the operation is successful, the value of the SY-SUBRC variable is set to 0. However, if the internal table contains fewer lines than the <index_number> index, no line is changed and the value of the SY-SUBRC variable is set to 4.

The MODIFY statement can be used within a loop construction, only when the INDEX clause is not used.

Modifying Multiple Lines by Using a Condition

The MODIFY statement can also modify one or more lines by specifying a condition in the WHERE clause. The following syntax is used to specify a condition in the WHERE clause of the MODIFY statement:

```
MODIFY <internal_tab> FROM <work_area_itab> TRANSPORTING
<f1> <f2> ... WHERE <cond>.
```

In this syntax, the MODIFY statement processes all the lines that meet the logical condition <cond>. The logical condition can be a combination of one or more comparisons. The <work_area_itab> work area, which is specified in the FROM clause, replaces a line in the <internal_tab> internal table. If any line is modified in an internal table, the SAP system sets the value of the SY-SUBRC variable to 0; otherwise the value is set to 4.

Now, let's consider some examples of how lines are modified in an internal table.

Examples of Changing Lines

In this section, we consider some examples that show how to change one or more lines of an internal table by using the MODIFY statement.

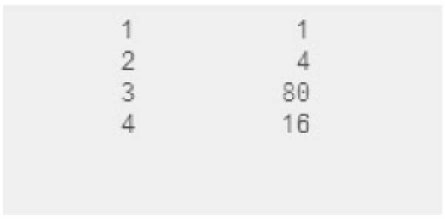
Listing 7.26 shows how to modify lines by specifying a unique key:

Listing 7.26: Modifying the lines of a table by specifying a unique key

```
REPORT ZMODIFYLINES_DEMO.

*/Creating an internal table

DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
DATA MyTable LIKE Hashed TABLE OF LINE WITH UNIQUE
KEY COLA.

*/Filling the table with four lines

DO 4 TIMES.
  LINE-COLA = SY-INDEX.
  LINE-COLB = SY-INDEX ** 2.
INSERT LINE INTO TABLE MyTable.
ENDDO.
```

```
*/Modifying a particular line of the table

LINE-COLA = 3. LINE-COLB = 80.
MODIFY TABLE MyTable FROM LINE.

LOOP AT MyTable INTO LINE.
  WRITE: / LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.26, MyTable is a hashed table with the `LINE` line type and the `COLA` and `COLB` fields. Initially, MyTable is populated with four lines, where the `COLA` field contains the values 1, 2, 3, and 4, and the `COLB` field contains the values 1, 4, 9, and 16. The `MODIFY` statement is used to change the value of the `COLB` field to 80 when the value of the `COLA` key field is 3, as shown in Figure 7.32:



```
         1              1
         2              4
         3             80
         4             16
```

© SAP AG. All rights reserved.

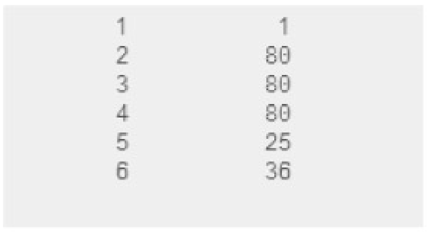
Figure 7.32: Modifying table lines by using a unique key

Listing 7.27 shows how to modify lines by specifying a condition:

Listing 7.27: Modifying the lines of a table by specifying a condition

```
REPORT ZMODIFYLINES_DEMO.
*/Creating an internal table

DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
DATA MyTable LIKE Hashed TABLE OF LINE WITH UNIQUE KEY
COLA.
*/Filling the table with six lines

DO 6 TIMES.
  LINE-COLA = SY-INDEX.
  LINE-COLB = SY-INDEX ** 2.
INSERT LINE INTO TABLE MyTable.
ENDDO.
*/Modifying a particular line of the table, on the basis
of a condition

LINE-COLB = 80.
MODIFY MyTable FROM LINE TRANSPORTING COLB
WHERE (COLB > 2) AND (COLA < 5).

LOOP AT MyTable INTO LINE.
  WRITE: / LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.27, MyTable is a hashed table with the `LINE` line type and the `COLA` and `COLB` fields. Initially, MyTable is populated with six lines, where the `COLA` contains the values 1, 2, 3, 4, 5, and 6, and the `COLB` contains the values 1, 4, 9, 16, 25, and 36. The `MODIFY` statement is used to change the lines of the table where the value of `COLB` is greater than 2 and less than 5. Figure 7.33 shows the output of Listing 7.27:



```
1          1
2         80
3         80
4         80
5         25
6         36
```

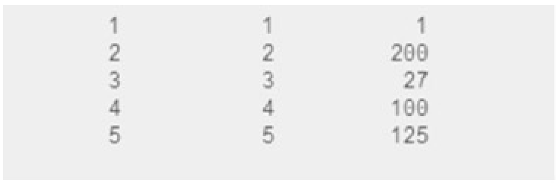
Figure 7.33: Modifying table lines by using a condition

Listing 7.28 shows how to modify the lines of a sorted table by using the `TRANSPORTING` clause:

Listing 7.28: Modifying the lines of a sorted table by using the TRANSPORTING clause

```
REPORT ZMODIFYLINES_DEMO.
*/Creating an internal table with five lines
DATA NAME(4) VALUE 'COLB'.

DATA: BEGIN OF LINE,
          COLA TYPE I,
          COLB TYPE I,
       END OF LINE.

DATA MyTable LIKE SORTED TABLE OF LINE WITH UNIQUE KEY
COLA.

DO 5 TIMES.
  LINE-COLA = SY-INDEX.
  LINE-COLB = SY-INDEX ** 3.
  APPEND LINE TO MyTable.
ENDDO.
*/Modifying the value of the line, at the index 4, by using
TRANSPORTING clause

LINE-COLB = 100.
MODIFY MyTable FROM LINE INDEX 4 TRANSPORTING (NAME).

*/Modifying the value of the line, at the index 2, without
using TRANSPORTING clause

LINE-COLA = 2.
LINE-COLB = 200.
MODIFY MyTable FROM LINE INDEX 2.

LOOP AT MyTable INTO LINE.
  WRITE: / SY-TABIX, LINE-COLA, LINE-COLB.
ENDLOOP.
```

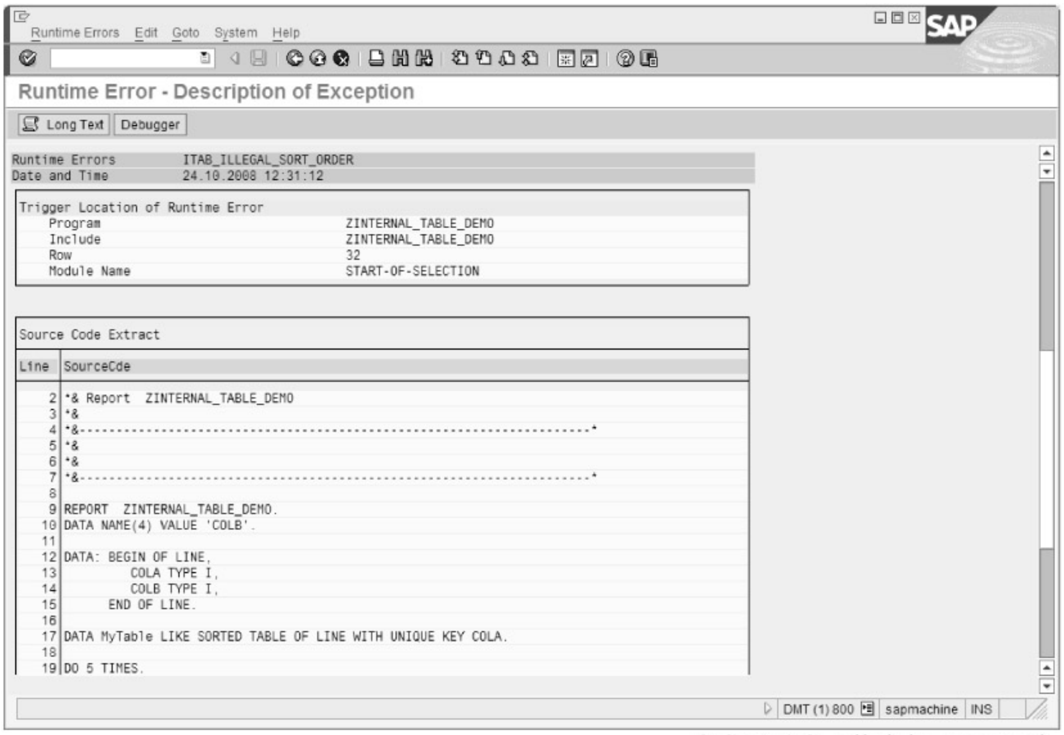
In Listing 7.28, MyTable is a sorted table with `LINE` as its line type and the `COLA` and `COLB` fields. Initially, MyTable is populated with five lines, where the `COLA` field contains the values 1, 2, 3, 4, and 5, and the `COLB` field contains the values 1, 8, 27, 64, and 125. The `MODIFY` statement is used to change the value of the `COLB` field at the specified `INDEX` numbers 2 and 4. The value of the `COLB` field at `INDEX` 2 is changed from 8 to 200, whereas at `INDEX` 4, the value is changed from 64 to 100. Figure 7.34 shows the modified values in the `COLB` field of MyTable:



```
1          1          1
2          2        200
3          3         27
4          4        100
5          5        125
```

Figure 7.34: Modified values of a sorted table

A runtime error occurs if the `INDEX` number and the `LINE-COLA` value do not match because the key fields of sorted tables cannot be changed. Figure 7.35 shows the runtime error screen if the value of the `LINE-COLA` variable is 2 while that of `INDEX` is 3:



```
Runtime Errors         ITAB_ILLEGAL_SORT_ORDER
Date and Time          24.10.2008 12:31:12

Trigger Location of Runtime Error
    Program                                ZINTERNAL_TABLE_DEMO
    Include                                ZINTERNAL_TABLE_DEMO
    Row                                    32
    Module Name                            START-OF-SELECTION

Source Code Extract
Line  SourceCde
*& Report  ZINTERNAL_TABLE_DEMO
*&
*&---------------------------------------------------------------------*
*&
*&
*&---------------------------------------------------------------------*

REPORT  ZINTERNAL_TABLE_DEMO.
DATA NAME(4) VALUE 'COLB'.

DATA: BEGIN OF LINE,
          COLA TYPE I,
          COLB TYPE I,
       END OF LINE.

DATA MyTable LIKE SORTED TABLE OF LINE WITH UNIQUE KEY COLA.

DO 5 TIMES.
```

Figure 7.35: Displaying a runtime error when the INDEX and LINE-COLA do not match

In Figure 7.35, you can see the runtime error screen, which shows the details of the generated exception, such as the name of the program, date, time, and line.

Using the *`WRITE TO`* Statement

Besides the `MODIFY` statement, you can use the `WRITE TO` statement to modify the lines of an internal table. The syntax to use the `WRITE TO` statement to modify table lines is:

```
WRITE <f> TO <internal_tab> INDEX <index_number>.
```

In this syntax, the `WRITE TO` statement converts the content of the `<f>` field to the `C` data type and transfers the resulting character string to the line with the <index_number> index. If the operation is successful, the value of the `SY-SUBRC` variable is set to 0; otherwise, the value of the `SY-SUBRC` variable is set to 4. Note that the data type of the `<f>` field must be able to be converted into a character field; if not, a syntax or runtime error occurs.

Listing 7.29 shows how to modify the lines of an internal table by using the `WRITE TO` statement:

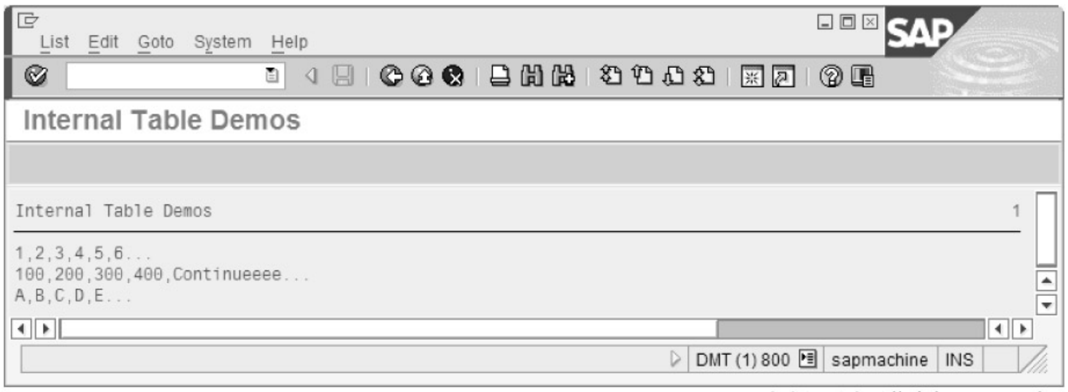
Listing 7.29: Using the WRITE TO statement

```
REPORT ZWRITELINE_DEMO.
*/Creating an internal table with three lines
DATA series(100).

DATA SeriesStore LIKE TABLE OF series.

series = '1,2,3,4,5,6...'.

APPEND series TO SeriesStore.
series = '100,200,300,400,500,600...'.

APPEND series TO SeriesStore.
```

```
series = 'A,B,C,D,E...'.

APPEND series TO SeriesStore.
*/Changing the second line of the table with WRITE
statement

WRITE 'Continueeee...' TO SeriesStore+16 INDEX 2.

LOOP AT SeriesStore INTO series.

  WRITE / series.

ENDLOOP.
```

In Listing 7.29, SeriesStore is an internal table created with the elementary type `C` field of 100 characters. SeriesStore is populated with three series of lines, in which the first series is 1, 2, 3, 4, 5, 6…, the second series is 100, 200, 300, 400, 500, 600…, and the third series is A, B, C, D, E…. After populating the SeriesStore table with these lines, we change the second line by using the `WRITE TO` statement. As a result, the numbers after 400 are replaced by the `Continueeee...` string in the second series, as shown in Figure 7.36:

In Figure 7.36, notice that the `Continueeee...` string has replaced the values 500 and 600 from the second series.



```
Internal Table Demos                                                      1

1,2,3,4,5,6...
100,200,300,400,Continueeee...
A,B,C,D,E...
```

© SAP AG. All rights reserved.

Figure 7.36: Modifying a series by using the WRITE TO statement

Deleting Lines

The `DELETE` statement is used to delete one or more records from an internal table. The records of an internal table are deleted either by specifying a table key or condition or by finding duplicate entries. If an internal table has a non-unique key and contains duplicate entries, the first entry from the table is deleted.

Various operations that can be performed using the `DELETE` statement include:

- Deleting a line by specifying a table key
- Deleting a line by using the `INDEX` clause
- Deleting multiple lines by specifying a condition
- Deleting multiple lines by using an index or condition
- Deleting duplicate entries

Now, let's discuss each operation in detail.

Deleting a Line by Specifying a Table Key

The syntax to use the `DELETE` statement to delete a record or line from an internal table is as follows:

```
DELETE TABLE <internal_tab> FROM <work_area_itab>.
```

In this syntax, the `<work_area_itab>` expression is a work area, which must be compatible with the line type of the `<internal_tab>` internal table. The delete operation is performed on the basis of a default key, which is taken from the components of the work area.

You can also specify a table key explicitly in the `DELETE TABLE` statement by using the following syntax:

```
DELETE TABLE <internal_tab> WITH TABLE KEY <k1> = <f1> ...
<kn> = <fn>.
```

In this syntax, `<f1>`, `<f2>....<fn>` are the fields of an internal table and `<k1>`, `<k2>....<kn>` are the key fields of the table. The `DELETE` statement is used to delete the records or lines of the `<internal_tab>` table based on the expressions `<k1> = <f1>`, `<k2> = <f2>...<kn> = <fn>`. Moreover, if the data types of the `<f1>`, `<f2>....<fn>` fields are not compatible with the `<k1>`, `<k2>...<kn>` key fields, the SAP system automatically converts them into the compatible format.

Deleting a Line by Using the INDEX Clause

Use the following syntax to delete the records of an internal table based on its index:

```
DELETE <internal_tab> [INDEX <index_number>].
```

In this syntax, the `DELETE` statement is used to delete a line from the `<internal_tab>` table by specifying the `<index_number>` index with the `INDEX` clause. If the line is deleted successfully, the index of the subsequent lines in the tables is reduced by 1 and the value of the `SY-SUBRC` variable is set to 0. However, if the delete operation is unsuccessful, the value of the `SY-SUBRC` variable is set to 4. In other words, when the table does not contain the line of the given `<index_number>` index number, the value of the `SY-SUBRC` variable is set to 4.

Note Without the `INDEX` clause, you can use the `DELETE` statement only in a loop construction. In this case, the user deletes the current loop line and the `<index_number>` index is set implicitly to the value of the `SY-TABIX` variable.

Deleting Multiple Lines by Specifying a Condition

The `DELETE` statement is also used to delete one or more lines by specifying a condition in the `WHERE` clause. The syntax of the `DELETE` statement used to delete multiple lines based on specific conditions is:

```
DELETE <internal_tab> WHERE <cond>.
```

In this syntax, all the lines that satisfy the `<cond>` logical condition are deleted from the `<internal_tab>` table. The logical condition can consist of more than one comparison. In each comparison, the first operand must be a component of the line structure. If the lines of the table are not structured, the first operand can be the expression `TABLE LINE`. The comparison then applies to the entire line. If at least one line is deleted, the SAP system sets the value of the `SY-SUBRC` variable to 0; otherwise, the value is set to 4.

Deleting Multiple Lines by Using an Index or Condition

To delete one or more lines from an internal table, you can specify a condition in the `WHERE` clause or an index within the `DELETE` statement. The following syntax of the `DELETE` statement is used to delete multiple lines from an internal table, on the basis of either their line index or by the conditions specified in the `WHERE` clause:

```
DELETE <internal_tab> [FROM <n1>] [TO <n2>] [WHERE
<condition>].
```

In this syntax, the use of the `FROM`, `TO`, and `WHERE` clauses is optional in the `DELETE` statement. Apart from using the `WHERE` clause in the `DELETE` statement, the user can specify the lines that have to be deleted by specifying the index within the `FROM` and `TO` clauses. The SAP system deletes all the lines of the `<internal_tab>` internal table whose indexes lie between the `<n1>` and `<n2>` expressions. The task of deleting lines from an internal table by specifying an index between the `FROM` and `TO` clauses is similar to the task of deleting lines by specifying a condition in the `WHERE` clause.

Note that if the `FROM` clause is not specified, the SAP system deletes lines from the first line onwards; and if the `TO` clause is not specified, the SAP system deletes lines up to the last line (including the last line).

Deleting Duplicate Entries

The syntax to use the DELETE statement to delete adjacent duplicate lines or entries of an internal table is:

```
DELETE ADJACENT DUPLICATE ENTRIES FROM <internal_tab>
                [COMPARING <f1> <f2> ...
                             |ALL FIELDS].
```

In this syntax, the SAP system deletes all adjacent duplicate entries from the <internal_tab> internal table.

An internal table can have duplicate entries if it fulfills any of the following criteria:

- The content of the key fields of the table is identical in one or more lines when the COMPARING clause is not used.
- The content of the specified <f1> <f2>... fields of the table is identical in one or more lines when the COMPARING clause is used.
- The content of all fields of the table is identical in one or more lines when the COMPARING ALL FIELDS clause is used.

Examples Showing Deletion of Lines

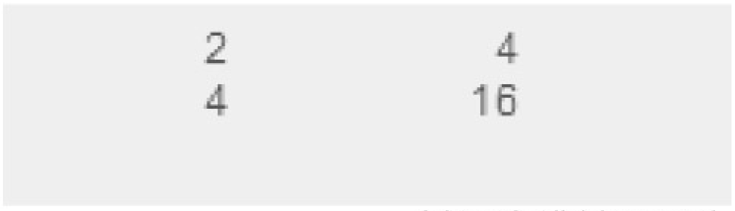
Listing 7.30 shows how to delete a line by specifying a table key:

Listing 7.30: Deleting a line by a table key

```
REPORT ZDELETELINES_DEMO.

*/Creating an internal table with four lines

DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.

DATA MyTable LIKE HASHED TABLE OF LINE WITH UNIQUE KEY
COLA.

DO 4 TIMES.
LINE-COLA = SY-INDEX.
LINE-COLB = SY-INDEX ** 2.
INSERT LINE INTO TABLE MyTable.
ENDDO.

*/Deleting the first and third lines from the internal
table

LINE-COLA = 1.

DELETE TABLE Mytable: FROM LINE,
WITH TABLE KEY COLA = 3.

LOOP AT MyTable INTO LINE.
WRITE: / LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.30, MyTable is a hashed table with two fields, COLA and COLB. Initially, MyTable is populated with four lines, where the COLA contains the values 1, 2, 3, and 4, while the COLB contains the values 1, 4, 9, and 16. The DELETE statement is used to delete the lines from MyTable where the value of the COLA key field is 1 or 3.

After deletion, the COLA field of MyTable contains the values 2 and 4, as shown in Figure 7.37:



```
         2                    4
         4                   16
```

Figure 7.37: Deleting lines by using a table key

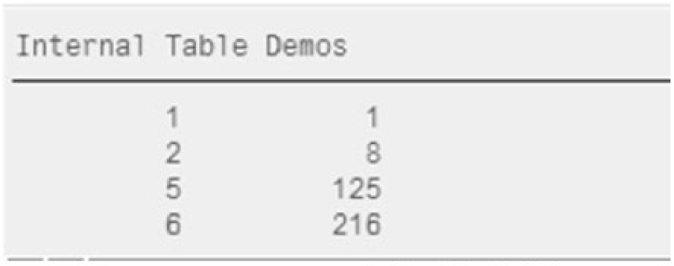
Listing 7.31 shows how to delete lines from an internal table by specifying a condition that uses the > and < comparison operators in the `DELETE` statement:

Listing 7.31: Deleting lines by specifying a condition in the DELETE statement

```
REPORT ZDELETELINES_DEMO.

*/Creating an internal table with six lines

DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.

DATA MyTable LIKE HASHED TABLE OF LINE WITH UNIQUE KEY
COLA.

DO 6 TIMES.
LINE-COLA = SY-INDEX.
LINE-COLB = SY-INDEX ** 3.
INSERT LINE INTO TABLE MyTable.
ENDDO.

*/Deleting the lines of the internal table, on the basis of
a condition by using comparison operators

DELETE MyTable WHERE (COLB > 8) AND (COLA < 5).
LOOP AT MyTable INTO LINE.
WRITE: / LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.31, MyTable is a hashed table with two fields, `COLA` and `COLB`. Initially, MyTable is populated with six lines, where the `COLA` field contains the values 1, 2, 3, 4, 5, and 6, and the `COLB` field contains the values 1, 8, 27, 64, 125, and 216. The `DELETE` statement deletes the lines of MyTable in which the values of the `COLB` field are greater than 8 and the `COLA` field values are less than 5. Figure 7.38 shows the remaining lines of MyTable, after the deletion:

Listing 7.32 shows how to delete duplicate lines of a table.



```
Internal Table Demos
--------------------------------------
          1            1
          2            8
          5          125
          6          216
```

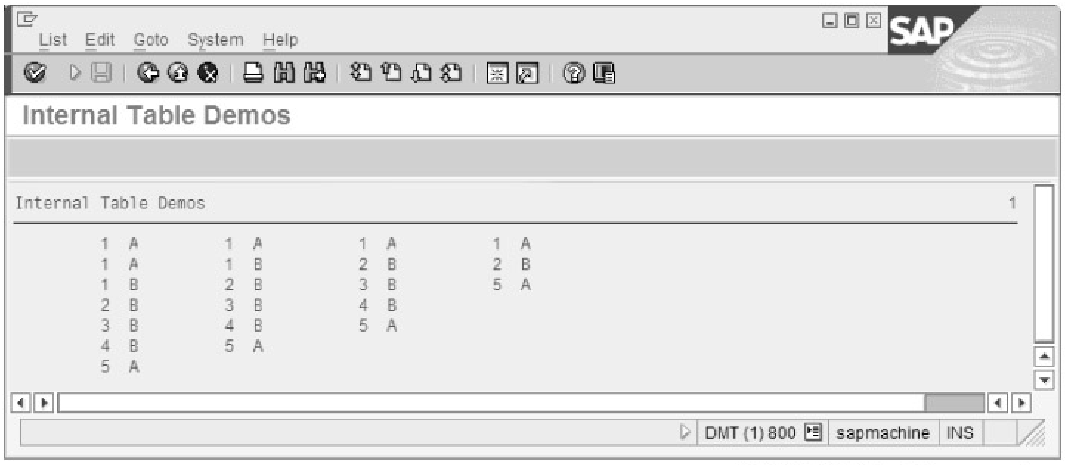
Figure 7.38: Deleting lines by specifying a condition

Listing 7.32: Deleting duplicate lines of a table

```
REPORT ZDELETELINES_DEMO.
```

```
*/Creating an internal table with seven lines
DATA OFF TYPE I.

DATA: BEGIN OF LINE,

COL1 TYPE I,
COL2 TYPE C,
END OF LINE.

DATA INTERNAL_TAB LIKE STANDARD TABLE OF LINE
       WITH NON-UNIQUE KEY COL2.

LINE-COL1   =  1.  LINE-COL2   =  'A'.  APPEND  LINE  TO
INTERNAL_TAB.
LINE-COL1   =  1.  LINE-COL2   =  'A'.  APPEND  LINE  TO
INTERNAL_TAB.
LINE-COL1   =  1.  LINE-COL2   =  'B'.  APPEND  LINE  TO
INTERNAL_TAB.
LINE-COL1   =  2.  LINE-COL2   =  'B'.  APPEND  LINE  TO
INTERNAL_TAB.
LINE-COL1   =  3.  LINE-COL2   =  'B'.  APPEND  LINE  TO
INTERNAL_TAB.
LINE-COL1   =  4.  LINE-COL2   =  'B'.  APPEND  LINE  TO
INTERNAL_TAB.
LINE-COL1   =  5.  LINE-COL2   =  'A'.  APPEND  LINE  TO
INTERNAL_TAB.

*/Deleting the adjacent duplicate entries on the basis of
all fields

OFF = 0. PERFORM LIST.

DELETE ADJACENT DUPLICATES FROM INTERNAL_TAB COMPARING ALL
FIELDS.

*/Deleting the adjacent duplicate entries on the basis of
COL1 field

OFF = 14. PERFORM LIST.

DELETE ADJACENT DUPLICATES FROM INTERNAL_TAB COMPARING
COL1.

*/Deleting the adjacent duplicate entries on the basis of
any individual field

OFF = 28. PERFORM LIST.

DELETE ADJACENT DUPLICATES FROM INTERNAL_TAB.
OFF = 42. PERFORM LIST.

FORM LIST.
  SKIP TO LINE 3.
  LOOP AT INTERNAL_TAB INTO LINE.
    WRITE: AT /OFF LINE-COL1, LINE-COL2.
  ENDLOOP.
ENDFORM.
```

In Listing 7.32, a standard table is created and populated with data. The first `DELETE` statement deletes the second line from the `INTERNAL_TAB` internal table because this line has the same content as the first line. The second `DELETE` statement deletes the second line from the remaining lines stored in the table because the content of the `COL1` field is the same as that of the first line. The third `DELETE` statement deletes the third and fourth lines from the remaining lines of the table because the content of the `COL2` default key field is the same as that of the second line. Although the content of the default key is the same for the first and the fifth lines, the fifth line is not deleted because it is not adjacent to the first line. Figure 7.39 shows the output of Listing 7.32:



```
Internal Table Demos                                         1

        1  A        1  A        1  A        1  A
        1  A        1  B        2  B        2  B
        1  B        2  B        3  B        5  A
        2  B        3  B        4  B
        3  B        4  B        5  A
        4  B        5  A
        5  A
```

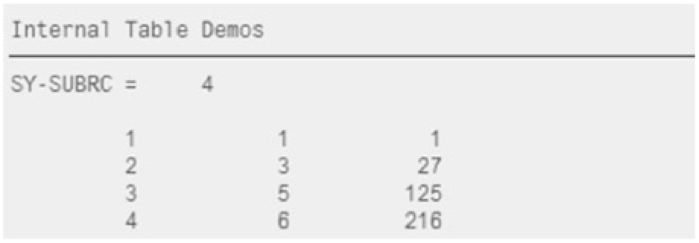
Figure 7.39: Deleting duplicate lines

Listing 7.33 shows how to delete the lines of an internal table by using the `INDEX` clause in the `DELETE` statement:

Listing 7.33: Deleting lines by using the INDEX clause

```
REPORT ZDELETELINES_DEMO.
*/Creating an internal table with six lines
DATA: BEGIN OF LINE,
        COLA TYPE I,
        COLB TYPE I,
       END OF LINE.
DATA MyTable LIKE SORTED TABLE OF LINE WITH UNIQUE KEY
COLA.
DO 6 TIMES.
  LINE-COLA = SY-INDEX.
  LINE-COLB = SY-INDEX ** 3.
  APPEND LINE TO MyTable.
ENDDO.
*/Deleting the lines specified at index number 2, 3, and
5
DELETE Mytable INDEX: 2, 3, 5.
WRITE: 'SY-SUBRC =', SY-SUBRC.

SKIP.
LOOP AT MyTable INTO LINE.
  WRITE: / SY-TABIX, LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.33, MyTable is a sorted table with the `LINE` line type, initially populated with six lines. The `COLA` field of MyTable contains the values of the SY-INDEX variable and the `COLB` field of MyTable contains the cube of the values of the `SY-INDEX` variable. The `DELETE` statement is used to delete the lines of this table at the specified `INDEX` numbers, i.e., 2, 3, and 5. The value of the `SY-SUBRC` variable is set to 4 as the third delete operation, which deletes the line at the index number 5, fails. This is because `INDEX 2` deletes the second line, `INDEX 3` deletes the fourth line (because the fourth line shifts to the third place when the second line is deleted), but `INDEX 5` has no line to delete. In other words, the third delete operation fails because the table now has only four lines. Figure 7.40 shows the output of Listing 7.33:



```
Internal Table Demos

SY-SUBRC =      4

          1           1           1
          2           3          27
          3           5         125
          4           6         216
```

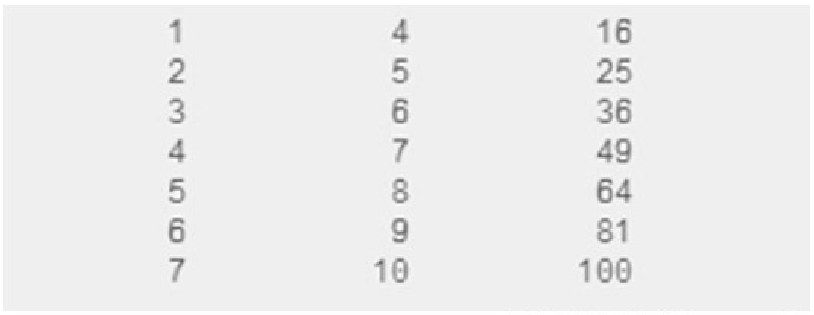
Figure 7.40: Deleting lines by using the INDEX clause

Listing 7.34 shows how to delete the lines from an internal table by using the `DELETE` statement in a loop construction:

Listing 7.34: Deleting lines by using the DELETE statement in a loop construction

```
REPORT ZDELETELINES_DEMO.
*/Creating an internal table with ten lines
DATA: BEGIN OF LINE,
          COLA TYPE I,
          COLB TYPE I,
      END OF LINE.

DATA MyTable LIKE TABLE OF LINE.
DO 10 TIMES.
  LINE-COLA = SY-INDEX.
  LINE-COLB = SY-INDEX ** 2.
  APPEND LINE TO MyTable.
ENDDO.
*/Deleting the lines of the table, in which the value of
COLA is less than 4
LOOP AT MyTable INTO LINE.
  IF LINE-COLA < 4.
    DELETE MyTable.
  ENDIF.
ENDLOOP.
LOOP AT MyTable INTO LINE.
  WRITE: / SY-TABIX, LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.34, MyTable is an internal table with the `LINE` line type, initially populated with 10 lines. The `COLA` field of MyTable contains the values of the `SY-INDEX` variable, and the `COLB` field of MyTable contains the square of the values of the `SY-INDEX` variable. The `DELETE` statement is used in a loop construction to delete the lines from MyTable where the value of the `COLA` field is less than 4. Figure 7.41 shows the lines of MyTable after deleting the lines of the table by using the `DELETE` statement in a loop construction:



```
         1          4         16
         2          5         25
         3          6         36
         4          7         49
         5          8         64
         6          9         81
         7         10        100
```

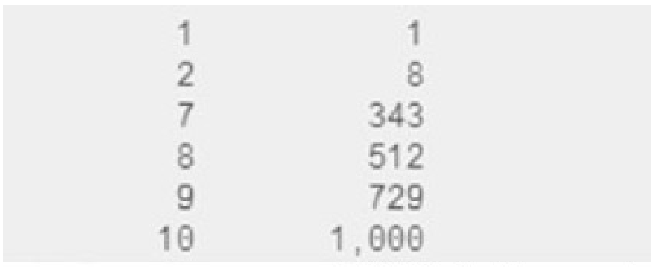
Figure 7.41: Deleting lines by using a loop construction

Listing 7.35 shows how to delete lines from an internal table based on an index and a condition:

Listing 7.35: Deleting lines by specifying an index with a condition in the DELETE statement

```
REPORT ZDELETELINES_DEMO.
*/Creating an internal table with ten lines
DATA: BEGIN OF LINE,
COLA TYPE I,
COLB TYPE I,
END OF LINE.
DATA MyTable LIKE TABLE OF LINE.
DO 10 TIMES.
  LINE-COLA = SY-INDEX.
  LINE-COLB = SY-INDEX ** 3.
  APPEND LINE TO MyTable.
ENDDO.
*/Deleting the lines of the table on the basis of a range
and a condition
DELETE MyTable FROM 3 TO 6 WHERE COLB > 8.
LOOP AT MyTable INTO LINE.
  WRITE: / LINE-COLA, LINE-COLB.
ENDLOOP.
```

In Listing 7.35, MyTable is an internal table with the `LINE` line type, initially populated with 10 lines. The `COLA` field of MyTable contains the values of the `SY-INDEX` variable, and the `COLB` field of MyTable contains the cube of the values of the `SY-INDEX` variable. The `DELETE` statement is used to delete a range of lines from the third line to the sixth line. In addition, the value of the `COLB` field is greater than 8. Figure 7.42 shows the output of Listing 7.35:



```
         1            1
         2            8
         7          343
         8          512
         9          729
        10        1,000
```

© SAP AG. All rights reserved.

Figure 7.42: Deleting lines by specifying an index and a condition

In Figure 7.42, you see that the records corresponding to the values 3 to 6 in the `COLA` field are deleted.

Searching Table Entries

Besides operations such as inserting, modifying, and deleting lines, you can perform a search operation in internal tables. The `SEARCH...FOR` statement is used to search for character strings in index tables. The `BINARY SEARCH` clause is used with the `READ` statement to perform a binary search in standard tables.

Now, let's learn how to search character strings and perform binary searches in index tables.

Finding Character Strings in Index Tables

The `SEARCH...FOR` statement is used to find a character string in the records of an index table. The syntax of the `SEARCH...FOR` statement is:

```
SEARCH <internal_tab> FOR <text_string> <options>.
```

In this syntax, the `<internal_tab>` expression represents an indexed internal table and the `<text_string>` expression represents a character string. The `<text_string>` string is searched within the records of the `<internal_tab>` table. If the SAP system finds the `<text_string>` string, the value of the `SY-SUBRC` system variable is set to 0, and the value of the `SY-TABIX` system variable is set to the index of the record in the table in which the string is found. On the contrary, if the string is not found in any record of the table, the value of the `SY-SUBRC` variable is set to 4.

In the `<options>` expression, you can use any of the following clauses to search entries in internal tables:

- `ABBREVIATED`—Searches the `<internal_tab>` table for an abbreviated word in the `<text_string>` expression. In the `<internal_tab>` table, the abbreviated characters can be separated by using other characters, such as, `and;`. Note that the first character of the word stored in the `<text_string>` expression and the first character of a record of the internal table must be the same.
- `STARTING AT <internal_tab_line>`—Searches the `<text_string>` string that starts from the `<internal_tab_line>` line in the `<internal_tab>` table. However, the `<internal_tab_ line>` expression can also be a variable.
- `ENDING AT <internal_tab_line>`—Searches the `<text_string>` string up to the `<internal_tab_line>`

line of the <internal_tab> table. The <internal_tab_line> expression can also be a variable.

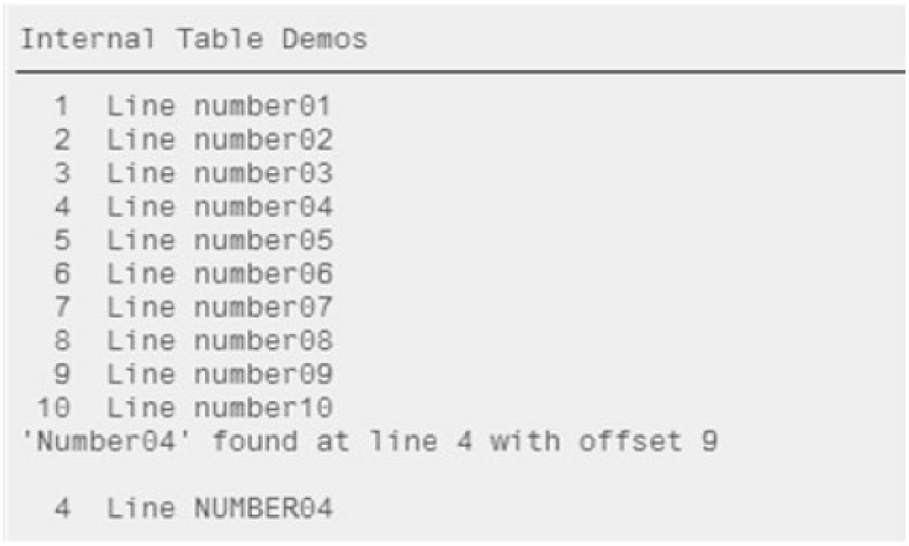
- AND MARK—Searches the <text_string> string in the <internal_tab> table. If the specified string is found in the table, all the characters in the search string (and those characters when the ABBREVIATED clause is used) are converted to uppercase.

Listing 7.36 shows how to search for a string in an index table by using the AND MARK clause in the SEARCH…FOR statement:

Listing 7.36: Using the SEARCH…FOR statement to find a character string in an index table

```
REPORT ZSEARCHLINES_DEMO.

*/Creating an internal table with ten lines

DATA: BEGIN OF line,
         ID(4) TYPE c,
         Name(20) TYPE c,
       END OF line.

DATA MyTable LIKE SORTED TABLE OF line WITH UNIQUE KEY
ID.
DATA num(2) TYPE n.
DO 10 TIMES.
   line-ID = sy-index.
   num = sy-index.
   CONCATENATE 'Line number ' num INTO line-Name.
   APPEND line TO MyTable.
ENDDO.
LOOP AT MyTable INTO LINE.

    WRITE: / LINE-ID, LINE-Name.
ENDLOOP.
*/Searching a particular line within the table

SEARCH MyTable FOR 'Number04' AND MARK.

WRITE: /'''Number04'' found at line', (1) sy-tabix,
          'with offset', (1) sy-fdpos.
SKIP.

READ TABLE MyTable INTO line INDEX sy-tabix.
WRITE: / line-ID, line-Name.
```

In Listing 7.36, MyTable is a sorted table containing 10 lines. The SEARCH statement is used to find the line that contains the string text Number04. The value of the SY-FDPOS variable is set to 9, because it contains the offset position of the string in the table line. When the search string is found, all the characters in the search string are converted to uppercase because of the AND MARK clause. Figure 7.43 shows the output of Listing 7.36:



```
Internal Table Demos

   1  Line number01
   2  Line number02
   3  Line number03
   4  Line number04
   5  Line number05
   6  Line number06
   7  Line number07
   8  Line number08
   9  Line number09
  10  Line number10
'Number04' found at line 4 with offset 9

   4  Line NUMBER04
```

Figure 7.43: Searching with the AND MARK option

In Figure 7.43, notice that the string Number04 is found at the fourth line.

Binary Search in Standard Tables

The `BINARY SEARCH` clause is used in the `READ` statement to search for the records of a standard table based on a key other than the default key.

The following syntax shows the use of the `BINARY SEARCH` clause in the `READ` statement:

```
READ TABLE <internal_tab> WITH KEY <k1> = <f1>...<kn> =
<fn> <result>
BINARY SEARCH.
```

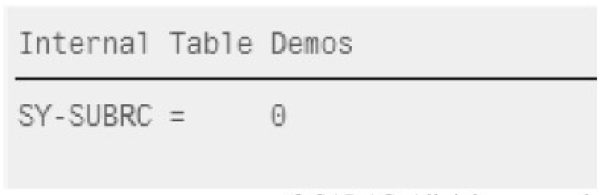
When the `BINARY SEARCH` clause is used to search the records of internal tables, the tables must be of the standard type and must be sorted in ascending order by specifying a search key. Listing 7.37 shows a binary search in a standard table:

Listing 7.37: Performing a binary search in a standard table

```
REPORT ZBINARYSEARCH_DEMO.

*/Creating an internal table with three lines
DATA: BEGIN OF LINE,
         COLA TYPE I,
         COLB TYPE I,
       END OF LINE.
DATA MYTABLE LIKE STANDARD TABLE OF LINE.

DO 3 TIMES.
 LINE-COLA = SY-INDEX.
 LINE-COLB = SY-INDEX ** 2.
 APPEND LINE TO MYTABLE.
ENDDO.

SORT MYTABLE BY COLB.

*/Performing binary search on the basis of a key value

READ TABLE MYTABLE WITH KEY COLB = 4 INTO LINE BINARY
SEARCH.

WRITE: 'SY-SUBRC =', SY-SUBRC.
```

In Listing 7.37, MYTABLE is a standard internal table, initially populated with three lines. The `COLA` field of MYTABLE contains the values of the `SY-INDEX` variable (1, 2, 3), and the `COLB` field of MYTABLE contains the square of the values of the SY-INDEX variable (1, 4, 9). The SORT statement is used to sort the contents of MYTABLE according to the values in the COLB field. The READ statement uses a binary search to find the line in the table where the value of COLB is 4. Figure 7.44 displays the output of Listing 7.37:

In Figure 7.44, the value of the SY-SUBRC variable is shown as 0, which means that a line has been searched successfully based on a binary search, where the value of the COLB field in MYTABLE is 4.



```
Internal Table Demos

SY-SUBRC =      0
```

© SAP AG. All rights reserved.

Figure 7.44: The result of a binary search

Maintaining Internal Tables

ABAP provides various statements to maintain internal tables. Table 7.2 shows a list of ABAP statements used to maintain internal tables with and without header lines:

Table 7.2: Operations for all table types (Standard, Sorted, and Hashed)

Statement for Internal Tables Without Header Lines	Statement for Internal Tables With Header Lines
`INSERT <work_area_itab> INTO TABLE <internal_tab>.`	`INSERT TABLE <internal_tab>.`
`COLLECT <work_area_itab> INTO<internal_tab>.`	`COLLECT <internal_tab>.`
`READ TABLE <internal_tab>... INTO <work_area_itab>.`	`READ TABLE <internal_tab>...`
`MODIFY TABLE <internal_tab> FROM <work_area_itab>...`	`MODIFY TABLE <internal_tab>...`
`MODIFY <internal tab> FROM <work_area_tab>...WHERE...`	`MODIFY <internal_tab>... WHERE...`
`DELETE TABLE <internal_tab> FROM <work_area_itab>.`	`DELETE TABLE <internal_tab>.`
`LOOP AT <internal_tab> INTO <work_area_itab> ...`	`LOOP AT <internal_tab> ...`

Table 7.3 lists ABAP statements in which the header lines and work areas are the same/different for indexed internal tables:

Table 7.3: Operations for index tables

Statement for Internal Tables Without Header Lines	Statement for Internal Tables With Header Lines
`APPEND <work_area_itab> TO <internal_tab>.`	`APPEND <internal_tab>.`
`INSERT <work_area_itab> INTO <internal_tab> ...`	`INSERT <internal_tab> ...`
`MODIFY <internal_tab> FROM <work_area_itab> ...`	`MODIFY <internal_tab> ...`

Note The fact that a table and its header line have the same name can cause confusion in operations involving entire internal tables. To avoid such confusion, the user should use internal tables with differently named work areas.

Now, let's discuss control break processing statements, such as AT FIRST, AT LAST, AT NEW, AT END OF, and ON CHANGE OF, which are used in loops and help retrieve data from internal tables and prepare a report from the retrieved data.

Control Break Processing

Sometimes a user may want to display summarized data, such as, the sum of data, at the top or bottom of the report or the subtotals of data in the body of the report. In such cases, users can read the data from internal tables by using control break statements in the loops. Control break statements are used within the body of a loop construction and are executed when a specific condition within the loop is met. Control break statements include the following:

- The AT FIRST and AT LAST statements

- The AT NEW and AT END OF statements
- The SUM statement
- The ON CHANGE OF statement

Now, let's discuss each statement in detail.

The *AT FIRST* and *AT LAST* Statements

The AT FIRST...ENDAT and AT LAST...ENDAT statements are used to control code execution when the loop construction containing the code is processed for the first and last time, respectively. The following syntax shows the use of the AT FIRST and AT LAST statements:

```
LOOP AT <internal_tab>.
.....
    AT FIRST.
.......
    ENDAT.
AT LAST.
.......

    ENDAT.
.......
ENDLOOP.
```

In the preceding syntax, <internal_tab> is the name of an internal table. The dotted lines (...) represent any number of lines of code (even zero).

The AT FIRST and AT LAST statements are used only in the loop, not in the SELECT statement. These statements can be used in any sequence, meaning that it is not necessary that the AT FIRST statement should be written before the AT LAST statement. There is no limitation on the number of times the statements appear inside the loop. In other words, the user can use any number of AT FIRST and AT LAST statements inside the loop. However, remember that nesting of these statements is not allowed; that is, the user cannot nest the AT FIRST statement inside the AT LAST...ENDAT pair. As soon as the processing of a loop starts, the execution of the AT FIRST...ENDAT statement also starts, and when the loop is processed for the last time, the AT LAST...ENDAT statement is executed. Moreover, if there are more than one occurrence of the AT FIRST...ENDAT statement, all the occurrences are executed at the time the loop starts processing. Similarly, in the case of the AT LAST...ENDAT statement, all the occurrences of the statement are executed when the loop is processed for the last time.

The AT FIRST...ENDAT statement is used to initialize a loop for processing, displaying the total sum of the data at the top of a report and information in the heading section of a report.

The AT LAST...ENDAT statement is used to terminate the processing of a loop, displaying the total sum of data at the bottom of a report and information in the footer section of a report.

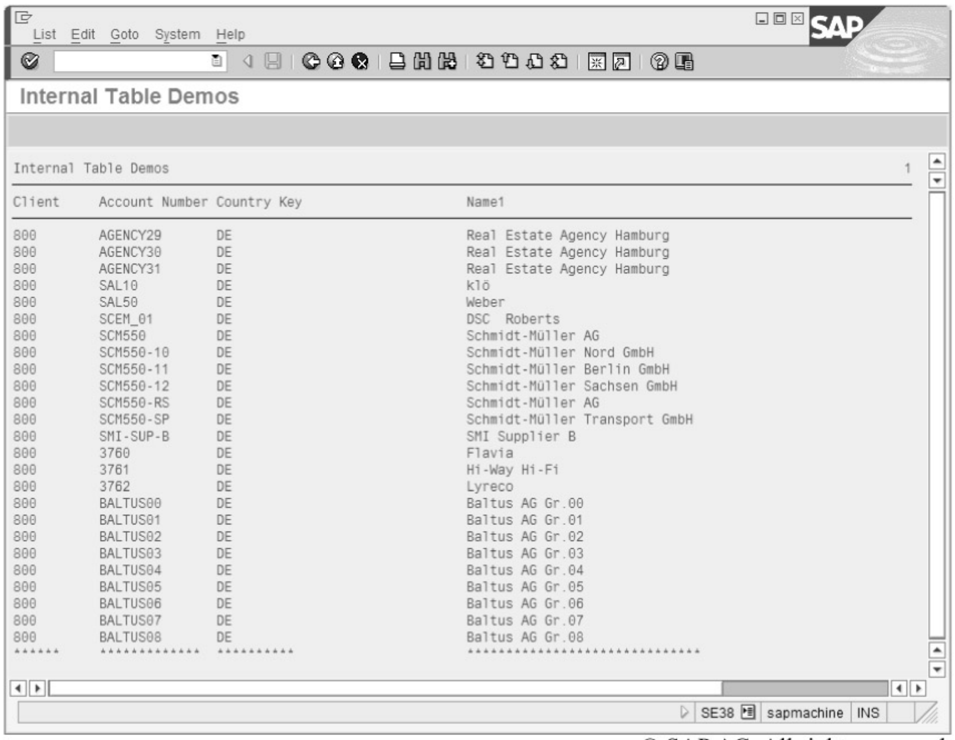
Listing 7.38 shows the working of the AT FIRST and AT LAST statements:

Listing 7.38: Using the AT FIRST and AT LAST statements

```
REPORT ZCONTROL_BREAK_PROS_DEMO.
TABLES: LFA1.
DATA IT1 LIKE LFA1 OCCURS 25 WITH HEADER LINE.
SELECT * FROM LFA1 UP TO 25 ROWS INTO TABLE IT1 WHERE
LAND1 = 'DE'.
LOOP AT IT1.

*/Using AT FIRST statement
AT FIRST.
        WRITE: / 'Client',
12 'Account Number',
27 'Country Key',
59 'Name1'.
ULINE.
```

```
ENDAT.

WRITE:/ IT1-MANDT,
       12 IT1-LIFNR,
       27 IT1-LAND1,
       59 IT1-NAME1.
*/Using AT LAST statement
       AT LAST.

        WRITE:/ '******',
             12 '*************',
             27 '**********',
             59 '******************************'.
ENDAT.
ENDLOOP.
```

In Listing 7.38, the TABLES statement is used to create a work area having the same structure as that of the LFA1 table. The DATA statement is used to define the IT1 internal table with a structure similar to that of the LFA1 database table. The SELECT statement is used to fill the IT1 table with 25 records of the LFA1 table, where the LAND1 field of the LFA1 table is DE. The AT FIRST statement is used to show the title of the fields of the IT1 table and the AT LAST statement is used to show the asterisk (*) character at the end, after displaying all the records. Figure 7.45 shows the output of Listing 7.38:



```
Internal Table Demos                                                            1

Client      Account Number Country Key                   Name1

800         AGENCY29       DE                            Real Estate Agency Hamburg
800         AGENCY30       DE                            Real Estate Agency Hamburg
800         AGENCY31       DE                            Real Estate Agency Hamburg
800         SAL10          DE                            klö
800         SAL50          DE                            Weber
800         SCEM_01        DE                            DSC  Roberts
800         SCM550         DE                            Schmidt-Müller AG
800         SCM550-10      DE                            Schmidt-Müller Nord GmbH
800         SCM550-11      DE                            Schmidt-Müller Berlin GmbH
800         SCM550-12      DE                            Schmidt-Müller Sachsen GmbH
800         SCM550-RS      DE                            Schmidt-Müller AG
800         SCM550-SP      DE                            Schmidt-Müller Transport GmbH
800         SMI-SUP-B      DE                            SMI Supplier B
800         3760           DE                            Flavia
800         3761           DE                            Hi-Way Hi-Fi
800         3762           DE                            Lyreco
800         BALTUS00       DE                            Baltus AG Gr.00
800         BALTUS01       DE                            Baltus AG Gr.01
800         BALTUS02       DE                            Baltus AG Gr.02
800         BALTUS03       DE                            Baltus AG Gr.03
800         BALTUS04       DE                            Baltus AG Gr.04
800         BALTUS05       DE                            Baltus AG Gr.05
800         BALTUS06       DE                            Baltus AG Gr.06
800         BALTUS07       DE                            Baltus AG Gr.07
800         BALTUS08       DE                            Baltus AG Gr.08
******      *************  **********                    ******************************
```

© SAP AG. All rights reserved.

Figure 7.45: Output showing the use of control break statements

In Figure 7.45, you can see that the column titles are shown at the top of the records and the asterisk (*) character is used at the bottom of the records.

The *AT NEW* and *AT END OF* Statements

The AT NEW...ENDAT and AT END OF...ENDAT statements are used to detect any change when one loop passes to the next value in a field of an internal table. These statements help execute the code at the beginning and end of a group of lines of an internal table. The following syntax shows how to use the AT NEW... ENDAT and AT END OF...ENDAT statements:

```
SORT BY Col_name.
   LOOP AT <internal_tab>.
```

```
.....
   AT NEW Col_name.
      .......
     ENDAT.
   AT END OF Col_name.
      .......
     ENDAT.
.......
ENDLOOP.
```

In this syntax, `<internal_tab>` is the name of the internal table. The dotted lines (….) denote the number of lines of code. The `Col_name` expression is a column or field name of the `AT NEW` statement, called control level. The `AT NEW...ENDAT` and `AT END OF...ENDAT` statements can only be used within a loop construction. They cannot be used within the `SELECT` statement. These statements can also appear in any order, which means it is not necessary for the `AT NEW...ENDAT` statement to always appear before the `AT END OF...ENDAT` statement. In a loop, these statements can be used multiple times for execution. These statements also cannot be nested within each other.

Whenever the value in the `Col_name` control level component changes, the lines of code in the `AT NEW...ENDAT` statement are executed. This block (lines of code) is also executed during passing the first loop or if any fields to the left of the `Col_name` control level component change. Between the `AT NEW` and the `ENDAT` statements, the numeric fields to the right of the `Col_name` control level component are set to zero, while the non-numeric fields are populated with the asterisk (`*`) character.

Listing 7.39 shows how to use the `AT NEW...ENDAT` statements:

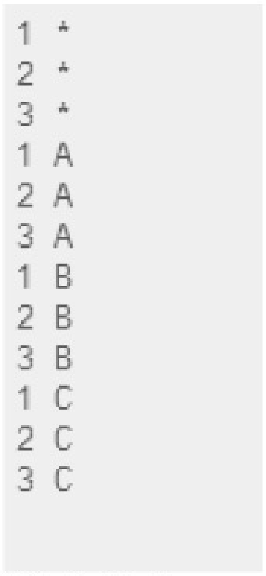
Listing 7.39: Using the AT NEW…ENDAT statements

```
REPORT ZCONTROL_BREAK_PROS_DEMO.
*/Using AT NEW and ENDAT statements
DATA : BEGIN OF ROW OCCURS 0,
FIELD1,
FIELD2,
END OF ROW.
ROW = '1A'.

APPEND ROW.
ROW = '1B'.
APPEND ROW.

ROW = '2B'.
APPEND ROW.

ROW = '3A'.
APPEND ROW.

ROW = '2A'.
APPEND ROW.

ROW = '1C'.
APPEND ROW.

ROW = '3B'.
APPEND ROW.

ROW = '3C'.
APPEND ROW.

ROW = '2C'.
APPEND ROW.

ROW = '2A'.
APPEND ROW.

ROW = '1C'.
APPEND ROW.
```

```
SORT ROW BY FIELD1.
LOOP AT ROW.
  AT NEW FIELD1.
      WRITE: / ROW-FIELD1, ROW-FIELD2.
ENDAT.

ENDLOOP.
SORT ROW BY FIELD2.
LOOP AT ROW.
AT NEW FIELD2.
    WRITE: / ROW-FIELD1, ROW-FIELD2.
ENDAT.
ENDLOOP.
```

In Listing 7.39, the `Field1` field is sorted. The `AT FIRST` statement is triggered for the first time through a loop and each time, there is a change in the `Field1` field. Now, the internal table is sorted again by the `Field2` field. The `AT NEW` statement is triggered each time there is a change in the value of the `Field1` or `Field2` field, because a control level is triggered whenever the value of the `Field2` field or of a field before `Field2` changes. Figure 7.46 shows the output of Listing 7.39:

In Figure 7.46, the first five records are shown in numerical order and the last five records are shown in alphabetical order.



```
1 *
2 *
3 *
1 A
2 A
3 A
1 B
2 B
3 B
1 C
2 C
3 C
```

Figure 7.46: Result of the AT NEW statement

The lines of code between the `AT END OF...ENDAT` pair of statements is executed under the following conditions:

- When there is a change in the value of the `Col_name` control level
- When there is a change in a field before the `Col_name` control level
- When the SAP system finds the last row of the table

Listing 7.40 shows the use of the `AT END OF` statement:

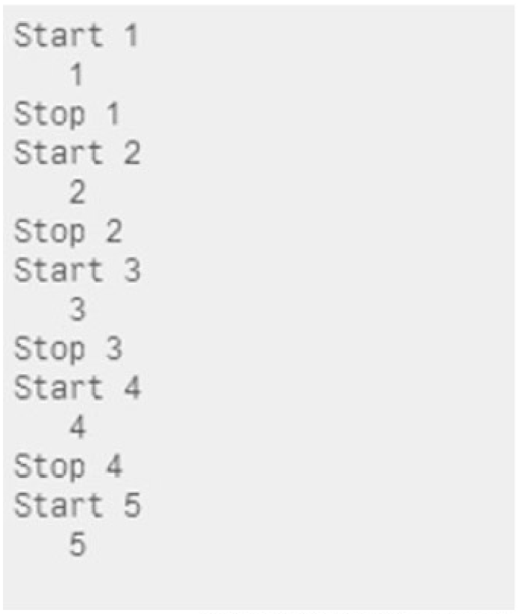
Listing 7.40: Using the AT END OF statement

```
REPORT ZCONTROL_BREAK_PROS_DEMO.
*/Using at end and ENDAT statements
DATA : BEGIN OF ROW OCCURS 0,
FIELD1,
FIELD2,
END OF ROW.
ROW = '1A'.
```

```
APPEND ROW.

ROW = '4B'.
APPEND ROW.

ROW = '2C'.
APPEND ROW.

ROW = '5D'.
APPEND ROW.

ROW = '3E'.
APPEND ROW.

SORT ROW BY FIELD1.
LOOP AT ROW.
  AT NEW FIELD1.
  WRITE: /'Start', ROW-FIELD1.

  ENDAT.
  WRITE: /4 ROW-FIELD1.
AT END OF FIELD1.
WRITE:/ 'Stop', ROW-FIELD1.
ENDAT.
ENDLOOP.
```

In Listing 7.40, `Field1` is a control level and is sorted twice. The `AT NEW` statement is triggered each time the value in the `Field1` control level changes. The `AT END OF` statement executes when there is a change in the value of the `Field1` control level and reads the next or last row. Figure 7.47 shows the output of Listing 7.40:

In Figure 7.47, the loop shows the message `Start` five times, along with the field number. However, the message `Stop` is displayed only four times because it appears only after the execution of the `AT END OF` statement. The last execution of this statement breaks the loop without displaying the `STOP` message.



```
Start 1
   1
Stop 1
Start 2
   2
Stop 2
Start 3
   3
Stop 3
Start 4
   4
Stop 4
Start 5
   5
```

© SAP AG. All rights reserved.

Figure 7.47: Sorting by using the AT NEW and AT END statements

The *SUM* Statement

The `SUM` statement is used to calculate the total of the values stored in the rows of a control level. The syntax to use the `SUM` statement is as follows:

```
AT FIRST/LAST/NEW/END OF.
```

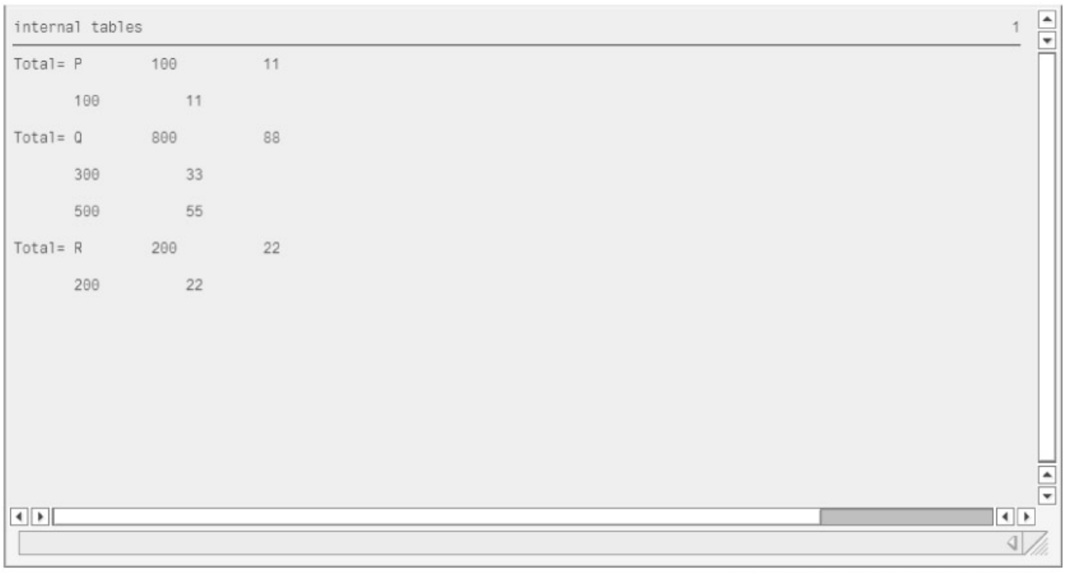
```
....
SUM.
...
ENDAT.
```

In this syntax, the SUM statement is used to calculate the total of the values stored in a control-level component. Listing 7.41 illustrates the working of the SUM statement.

Listing 7.41: Using the SUM statement

```
REPORT ZCONTROL_BREAK_PROS_DEMO.
DATA: BEGIN OF ROW OCCURS 0,
           FIELD1,
           FIELD2 TYPE I,
           FIELD3 TYPE I,
       END OF ROW.

ROW-FIELD1 = 'P'.
ROW-FIELD2 = 100.
ROW-FIELD3 = 11.
APPEND ROW.

ROW-FIELD1 = 'Q'.
ROW-FIELD2 = 300.
ROW-FIELD3 = 33.
APPEND ROW.

ROW-FIELD1 = 'R'.
ROW-FIELD2 = 200.
ROW-FIELD3 = 22.
APPEND ROW.

ROW-FIELD1 = 'Q'.
ROW-FIELD2 = 500.
ROW-FIELD3 = 55.
APPEND ROW.

SORT ROW BY FIELD1.
LOOP AT ROW.
*/Using the At new...Sum...Endat statement

AT NEW FIELD1.
SUM.
WRITE:/ 'Total=', ROW-FIELD1, ROW-FIELD2, ROW-FIELD3.
ENDAT.
WRITE:/ ROW-FIELD2, ROW-FIELD3.
ENDLOOP.
```

In Listing 7.41, the SUM statement is used to get the total of the values of the Field2 and Field3 fields based on the changes in the values of Field1. Figure 7.48 shows the output of Listing 7.41:



```
internal tables                                                    1
Total= P        100          11

        100          11

Total= Q        800          88

        300          33

        500          55

Total= R        200          22

        200          22
```

© SAP AG. All rights reserved.

Figure 7.48: Displaying the Sum of the values at the top

In Figure 7.48, you see that the total value is shown at the top of each record where the values of the `Field1` field are P, Q, and R.

The *ON CHANGE OF* Statement

The functionality of the `ON CHANGE OF...ENDON` statement is similar to that of the `AT NEW` statement. The syntax to use the `ON CHANGE OF...ENDON` statement is as follows:

```
ON CHANGE OF V1 [OR V2...]
- - - - -
[ELSE.
- - - ]
ENDON.
```

In this syntax, `V1` and `V2` are variable or field string names. The dotted line (…) represents the conditions specified on the variables or field string names, and the dashed lines (- - -) denote any number of lines of code, or the lines of code between the `ON CHANGE OF...ENDON` pair of statements are executed if the value of any of the variables (`V1`, `V2`, and so on) changes. However, if no change is detected in the values of these variables and the `ELSE` clause is specified, the lines of code following the `ELSE` clause are executed.

The `ON CHANGE OF` statement can be executed by a change in one or more fields named after the `OF` clause and separated by the `OR` clause. These fields can be elementary fields or field strings. When the `ON CHANGE OF` statement is used within a loop, a change in a field to the left of the control level does not execute a control break statement. When the `ON CHANGE OF` statement is used in a loop, the values of the fields to the right side of the statement still contain their original values. The `ON CHANGE OF` statement can be used along with the `SUM` statement. The `SUM` statement finds the sum of all numeric fields except the fields that appear after the `OF` clause.

The use of the `ON CHANGE OF` statement is similar to that of the `AT NEW` statement, but there are few differences between the two as well. Table 7.4 lists the differences between the `AT NEW...ENDAT` and `ON CHANGE OF...ENDON` statements:

Table 7.4: Comparing the AT NEW…ENDAT and ON CHANGE OF…ENDON Statements

Parameter	`ON CHANGE OF` Statement	`AT NEW` Statement
Loop construction	Can be used in any kind of loop construction, beside `AT LOOP...ENDLOOP`, such as `SELECT...ENDSELECT`, `CASE...ENDCASE, DO...ENDDO,` and `WHILE...ENDWHILE`, and even inside `GET` events. In addition, the `ON CHANGE OF` statement can be used outside a loop construction.	Can only be used inside the `AT LOOP...ENDLOOP` statements. Note that the `AT NEW` statement is used with internal tables only.
`WHERE`	Can be used in a loop along with the `<internal_tab>WHERE...`	Cannot be used in the loop along with

Expression	expression.	the `<internal_tab> WHERE...` expression.
ELSE Clause	Can be used between the ON CHANGE OF and ENDON statements.	Cannot be used between the AT NEW and ENDAT statements.

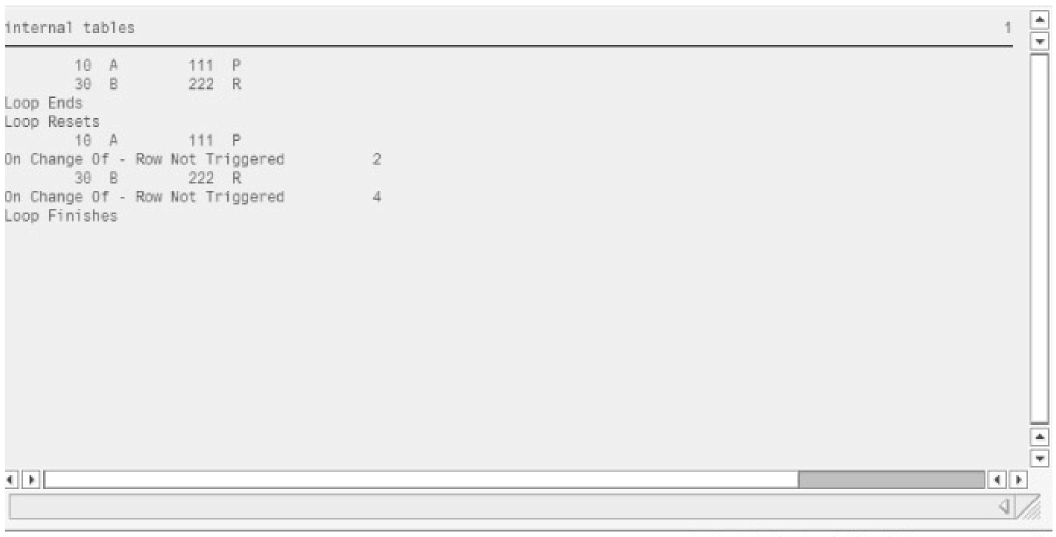
Now, let's consider some examples to show the use of the ON CHANGE OF statement and the difference between the ON CHANGE OF and AT NEW statements.

Listing 7.42 shows the working of the ON CHANGE OF statement:]

Listing 7.42: Using the ON CHANGE OF statement

```
REPORT ZCONTROL_BREAK_PROS_DEMO.
DATA: BEGIN OF ROW OCCURS 8,
FIELD1 TYPE I,
FIELD2,
FIELD3 TYPE I,
FIELD4,
END OF ROW.
ROW-FIELD1 = 10.
ROW-FIELD2 = 'A'.

ROW-FIELD3 = 111.
ROW-FIELD4 = 'P'.
APPEND ROW.

ROW-FIELD1 = 40.
ROW-FIELD2 = 'A'.
ROW-FIELD3 = 333.
ROW-FIELD4 = 'Q'.
APPEND ROW.

ROW-FIELD1 = 30.
ROW-FIELD2 = 'B'.
ROW-FIELD3 = 222.
ROW-FIELD4 = 'R'.
APPEND ROW.

ROW-FIELD1 = 40.
ROW-FIELD2 = 'B'.
ROW-FIELD3 = 444.
ROW-FIELD4 = 'S'.
APPEND ROW.

*/Using the On change of statements
LOOP AT ROW.
ON CHANGE OF ROW-FIELD2.
       WRITE: / ROW-FIELD1, ROW-FIELD2, ROW-FIELD3,
       ROW-FIELD4.
       ENDON.
ENDLOOP.
WRITE: /'Loop Ends'.

LOOP AT ROW.
AT FIRST.
WRITE: / 'Loop Resets'.
ENDAT.
ON CHANGE OF ROW-FIELD2.

   WRITE: / ROW-FIELD1, ROW-FIELD2, ROW-FIELD3,
   ROW-FIELD4.
ELSE.
WRITE: / 'On Change Of - Row Not Triggered',
SY-TABIX.
ENDON.
ENDLOOP.
WRITE: /'Loop Finishes'.
```

In Listing 7.42, ROW is an internal table and `FIELD1`, `FIELD2`, `FIELD3`, and `FIELD4` are its fields. This table is populated initially with four lines, where `FIELD1` stores 10, 140, 30, and 40; `FIELD2` stores A, A, B, and B; `FIELD3` stores 111, 333, 222, and 444; and `FIELD4` stores P, Q, R, and S. When the first loop is processed, it displays the values of all the fields only when there is a change in the values of `FIELD2`. The `WRITE` statement is then used to display the message Loop Ends. Finally, another loop starts along with displaying the message Loop Resets. It is processed by showing the values of all the fields only when there is a change in the values of `FIELD2`; otherwise, the message "`On Change Of - Row Not Triggered`" appears. Finally, this loop displays the message Loop Finishes. Figure 7.49 shows the output of Listing 7.42:



```
internal tables                                                          1

        10  A        111  P
        30  B        222  R
Loop Ends
Loop Resets
        10  A        111  P
On Change Of - Row Not Triggered          2
        30  B        222  R
On Change Of - Row Not Triggered          4
Loop Finishes
```

© SAP AG. All rights reserved.

Figure 7.49: Result of the ON CHANGE OF statement

Listing 7.43 shows the difference between the `AT NEW...ENDAT` and `ON CHANGE OF...ENDON` statements:

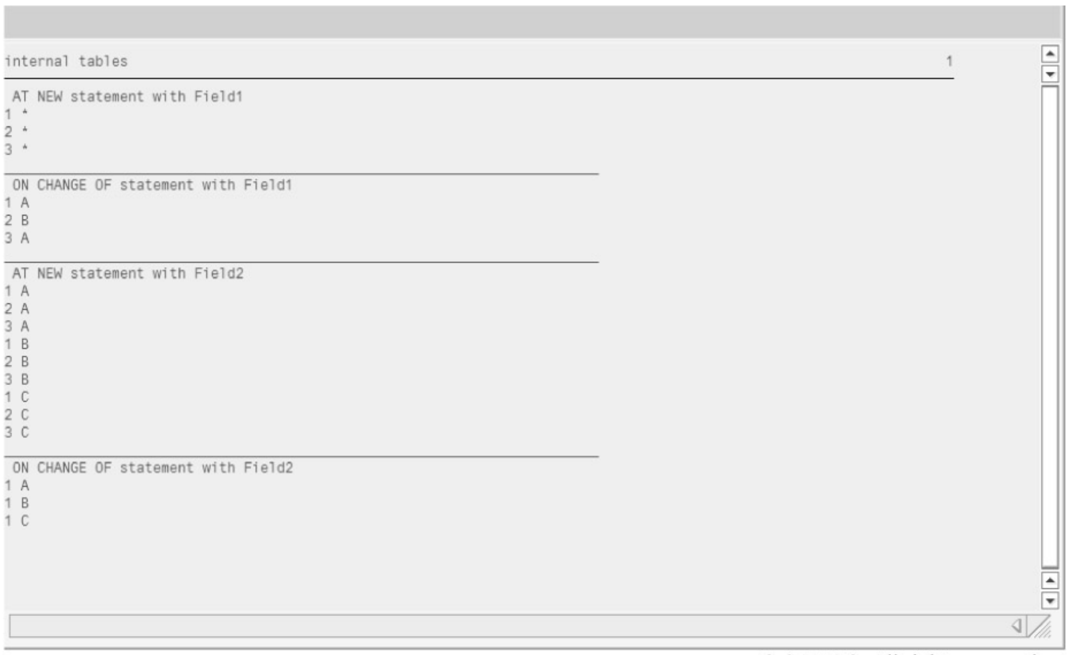
Listing 7.43: Showing the difference between the AT NEW…ENDAT and ON CHANGE OF…ENDON statements

```
REPORT ZCONTROL_BREAK_PROS_DEMO.

* Difference between the AT NEW...ENDAT and ON CHANGE OF...
ENDON statements
DATA : BEGIN OF ROW OCCURS 0,
FIELD1,
FIELD2,
END OF ROW.
ROW = '1A'.

APPEND ROW.
ROW = '1B'.

APPEND ROW.

ROW = '2B'.
APPEND ROW.

ROW = '3A'.

APPEND ROW.
ROW = '2A'.

APPEND ROW.
ROW = '1C'.

APPEND ROW.
ROW = '3B'.
```

```
APPEND ROW.
ROW = '3C'.

APPEND ROW.

ROW = '2C'.

APPEND ROW.
ROW = '2A'.

APPEND ROW.
ROW = '1C'.

APPEND ROW.
SORT ROW BY FIELD1.
WRITE:' AT NEW statement with Field1'.

LOOP AT ROW.
*the AT NEW statement with Field1
AT NEW FIELD1.
WRITE: / ROW-FIELD1, ROW-FIELD2.
ENDAT.

ENDLOOP.

WRITE:/'__________________________________________'.
WRITE:/' ON CHANGE OF statement with Field1'.

LOOP AT ROW.
*the ON CHANGE OF statement with Field1
ON CHANGE OF ROW-FIELD1.
    WRITE: / ROW-FIELD1, ROW-FIELD2.
ENDON.

ENDLOOP.
SORT ROW BY FIELD2.
WRITE:/'__________________________________________'.
WRITE:/' AT NEW statement with Field2'.

LOOP AT ROW.
*the AT NEW statement with Field2

AT NEW FIELD2.

WRITE: / ROW-FIELD1, ROW-FIELD2.
ENDAT.
ENDLOOP.
WRITE:/'__________________________________________'.

WRITE:/' ON CHANGE OF statement with Field2'.

LOOP AT ROW.
*the ON CHANGE OF statement with Field2

ON CHANGE OF ROW-FIELD2.
WRITE: / ROW-FIELD1, ROW-FIELD2.
ENDON.
ENDLOOP.
```

In Listing 7.43, ROW is an internal table and FIELD1 and FIELD2 are its fields. These fields are populated by various records, where FIELD1 stores numbers while FIELD2 stores characters. FIELD1 is sorted by the SORT statement and then used with the AT NEW statement in the AT LOOP...ENDAT statement. In addition, sorted FIELD1 is used with the ON CHANGE OF statement. Next, FIELD2 is sorted by the SORT statement and then used with the AT NEW and ON CHANGE OF statements.

In the output of Listing 7.43, you see that there is not much difference between the AT NEW and ON CHANGE OF statements when these statements are used with FIELD1, except that in case of AT NEW statement, characters are shown in the form of the asterisk (`*`) symbol. However, the values of the `AT NEW` and `ON CHANGE OF` statements are different if these statements are used with `FIELD2`. The difference is that the `AT NEW` statement shows the values when any change occurs in the `FIELD2` or `FIELD1` field. However, the `ON CHANGE OF` statement shows the values any change that occured only in the `FIELD2` field. Figure 7.50 shows the output of Listing 7.43:



```
internal tables                                                  1
----------------------------------------------------------------
 AT NEW statement with Field1
1 *
2 *
3 *
----------------------------------------------------------------
 ON CHANGE OF statement with Field1
1 A
2 B
3 A
----------------------------------------------------------------
 AT NEW statement with Field2
1 A
2 A
3 A
1 B
2 B
3 B
1 C
2 C
3 C
----------------------------------------------------------------
 ON CHANGE OF statement with Field2
1 A
1 B
1 C
```

Figure 7.50: Showing the difference between ON CHANGE OF and AT NEW statements

Summary

In this chapter, you have learned about internal tables, which are temporary tables created at runtime. In addition, you have learned about the data types of an internal table and the various types of internal tables, such as generic tables (any tables and index tables) and fully specified tables (standard tables, sorted tables, and hashed tables). You have also learned how to create internal tables by using the `TYPES` and `DATA` statements and perform functions on them, such as assigning, clearing, refreshing, and sorting internal tables, releasing the memory of internal tables, and comparing two tables. You have also learned about various operations, such as insertion, modification, and deletion of lines in an internal table, by using the `INSERT`, `READ`, `MODIFY`, and `DELETE` statements. Finally, you have learned about control break processing statements, such as `AT FIRST`, `AT NEW`, and `AT LAST`.

In the next chapter, you learn how ABAP programs access data in the SAP system through Open SQL statements, such as `INSERT`, `DELETE`, `UPDATE`, and `MODIFY`.