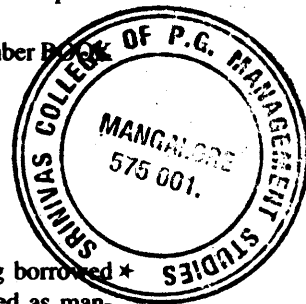
example, the set type *COLLECTION* is owned by the record type BRANCH; the member record type is BOOK_COPY and the membership criteria of this record type is manual insertion and optional retention. Therefore, an application program will have to insert an occurrence of the member record type in the appropriate set occurrence. The retention is optional, which means that if a BRANCH type record occurrence were to be deleted from the database, all member record occurrences in the set occurrence of the set type COLLECTION owned by the branch record occurrence will be removed from the set before the owner record occurrence is deleted. The member record occurrence continues to exist in the database.

```
set is COLLECTION
        owner is BRANCH
        member is BOOK_COPY manual optional
        end
```

Figure 8.24 shows the meaning of the combination of the two membership statuses for a member record type in a set type.

We can add the status information for insertion and retention of the member BOOK_DUE in the set borrowed as follows:

```
set is BORROWED
        owner is CLIENT
        member is BOOK_DUE automatic mandatory
        end
```

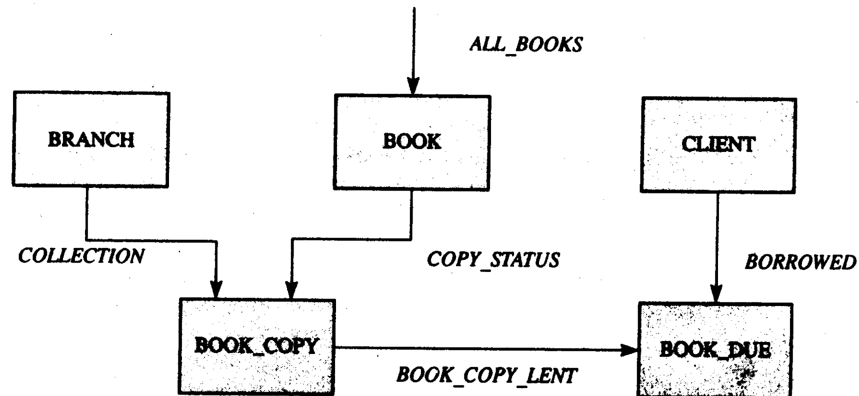


The insertion status is specified as automatic because a BOOK being borrowed must become the responsibility of a client. The retention status is specified as man-

Figure 8.24 Significance of membership status.

	FIXED	MANDATORY	OPTIONAL
AUTOMATIC	When a record is created, the DBMS places it in a set. The record stays there until it is deleted.	When a record is created, the DBMS places it in a set. The record can move from one occurrence of the set to another.	When a record is created, the DBMS places it in a set. The record can be moved to another occurrence of the set or removed and later reconnected.
MANUAL	The record has to be connected by appropriate data manipulation operations. Once it is connected it stays in the set occurrence until deleted.	The record has to be connected by appropriate data manipulation operations. The record can move from one set occurrence to another.	The record has to be connected by appropriate data manipulation operations. The record can be moved to another set occurrence or be removed and later reconnected.

Figure 8.25 Data structure diagram for the library schema.




```
type BOOK = record
        Author: string;
        Title; string;
        Call_No: integer;
        end
type BOOK_COPY = record
        Call_No: string;
        Copy_No: integer;
        Branch_Id: string;
        Current_Status: string;
        end
type CLIENT = record
        Client_No: string;
        Name: string;
        Address: string;
        end
type BOOK_DUE = record
        Call_No: string;
        Copy_No: integer;
        Client_No: string;
        Due_Date: string;
        end;
set is BORROWED
        owner is CLIENT
        member is BOOK_DUE automatic mandatory
        end
set is BOOK_COPY_LENT
        owner is BOOK_COPY
        member is BOOK_DUE automatic optional
        end
```

```
set is COPY_STATUS
        owner is BOOKS
        member is BOOK_COPY optional manual
        end
set is ALL_BOOKS
        owner is SYSTEM
        member is BOOK
        end
set is COLLECTION
        owner is BRANCH
        member is BOOK_COPY manual optional
        end
```

The subschema is a subset of the schema and corresponds to the ANSI/SPARC external schema. The subsetting of the schema is achieved by omitting from the subschema one or more data-items in a record, one or more record types, or one or more set types. In addition, aliases could be used for data-items, records, or sets. Furthermore, the data-items in the subschema may be given different data types from those defined for the corresponding data-items in the schema.

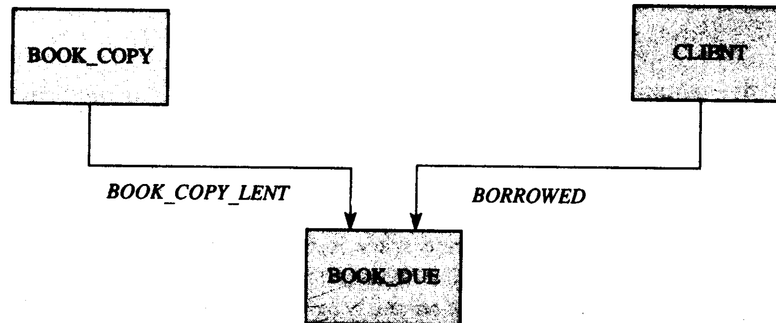
8.7 DBTG Data Manipulation Facility

The DBTG proposal included a data manipulation facility or language (DML). The facility included procedural statements, status and currency indicators, special registers, and conditions. The intent was to provide a number of operations or commands that could be embedded in a host language; the proposed host language was COBOL. For discussion here, we use a Pascal-like language as the host language. Before giving the details of the commands we consider some of the concepts used in the DBTG proposals to facilitate the understanding of the operations performed by the DBMS.

8.7.1 Run Unit

Run unit is a DBTG term that refers to each process or task (a program in execution is a process) that is running under the control of the DBMS. The process may be a user's application program containing DML commands or an interactive session with a user. Two or more users' processes may be concurrently executing the same application program or may be in an interactive session via on online terminal under the control of a teleprocessing monitor. The DBMS maintains separate records of the environment of each such run unit. An area of storage is set aside to provide an independent work space for each run unit. This work space is called the **user work area (UWA).** The UWA contains the processing environment of the run unit; the program being executed may be shared.

Figure 8.26 Data structure diagram for the subschema.



that portion of the database relevant to this user. We give the data structure diagram for this application in Figure 8.26 and the corresponding subschema below.

```
Subschema name is Circulation
        type BOOK_COPY = record
                Call_No: string;
                Copy_No: integer;
                Branch_Id: string;
                Current_Status: string;
                end
        type CLIENT = record
                Client_No: string;
                Name: string;
                Address: string;
                end
        type BOOK_DUE = record
                Call_No: string;
                Copy_No: integer;
                Client_No: string;
                Due_Date: string;
                end;
        set is BORROWED
                owner is CLIENT
                member is BOOK_DUE automatic mandatory
                end
        set is BOOK_COPY_LENT
                owner is BOOK_COPY
                member is BOOK_DUE automatic optional
                end
```

The DBTG proposal allows certain differences in the description of data between the schema and subschema, the DBMS performing the required transformation. For our purpose here, we used the same data descriptions as the schema.

Figure 8.27 Database contents.

234	Smith	Lynn
234	Klaf	Revere
236	Allard	Salem

CLIENT

1234	1	234	DEC 1
1237	1	234	DEC 1
1235	1	236	DEC 8
1236	1	236	DEC 11
1234	2	236	DEC 11

BOOK_COPY_LENT

1234	1	Lynn	LENT
1234	2	Revere	LENT
1235	1	Salem	LENT
1236	1	Salem	LENT
1237	2	Lynn	LENT
1237	1	Salem	LENT
1238	1	Lynn	IN
1238	2	Revere	IN
1238	3	Salem	IN

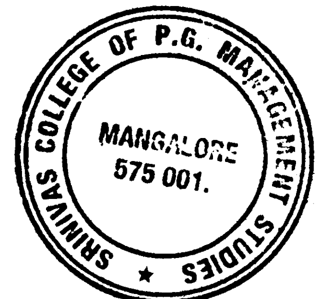
BOOK_COPY

The database contains the information for the records CLIENT, BOOK_COPY, and BOOK_DUE as shown in Figure 8.27

The DBMS maintains a currency indicator for each of the record types and set types and one for the run unit. We give these indicators in the form of a table in Figure 8.28. The initial values of the currency indicators for a run unit that uses the subschema shown above is given in the table. In this case there is one currency indicator for each of the record types BOOK_COPY, CLIENT, and BOOK_DUE; a currency indicator for each of the set types *BORROWED* and *BOOK_COPY_LENT;* in addition, there is an indicator for the run unit. The null values indicate that the datab : has not been accessed.

Figure 8.28 Initial values for the currency indicator for run unit using subscheme *Circulation.*

Indicator	Current Value
Run unit	null
BOOK_COPY	null
CLIENT	null
BOOK_DUE	null
BORROWED	null
BOOK_COPY_LENT	null



```
while DB_Status = 0 and not done do
        if Call_No = 1234 then
            begin
              Due_Date := 12/12;
              modify BOOK_DUE;
              done := true;
              end
            else find for update duplicate
                    BOOK_DUE using CLIENT_No;
```

Adding a Record Occurrence

The store command is used to store a new occurrence of a record type in the database. The new occurrence is first created in the template space for the record type in the UWA and then we execute the command:

store <record type>

This method allows a single record occurrence to be created and stored at one time. The following statements add the new CLIENT Gold to the database:

```
CLIENT.Client_No := 237;
CLIENT.Name := 'Gold';
CLIENT.Address := 'Lynn';
store CLIENT;
```

If the new record occurrence belongs to a record type associated with a set type, there must be a mechanism to place the record occurrence in the appropriate set occurrence. We discuss this in Section 8.8.2.

Deleting a Record Occurrence

An existing record occurrence may be deleted from the data base by use of the erase command. However, before the record is deleted, we have to locate it using the find command with the for update clause. As before, this informs the DBMS that the record may be updated, which in this case means deletion. The following statements delete the CLIENT Gold added in the previous example:

```
CLIENT.Client_No := 237;
find for update any CLIENT using CLIENT.Client_No;
if DB_Status = 0 then
        erase CLIENT
    else    error_routine;
```

In this example, we use the **DB_Status** register to verify that the find operation was successfully completed before executing the next statement.

In the case where a record occurrence to be deleted is associated with one or more set occurrences (obviously of different types) as an owner, appropriate operations must be carried out on the members of these sets before the record is deleted. One of the actions could be to move the member record occurrences to other set

occurrences of the same set types (if the membership retention status for the member record type is mandatory or optional), or to remove the member record occurrences (if retention status is optional). If these operations are not performed, the DBMS will delete the record and the members of the sets of which the record is an owner would also be deleted or removed from the sets before the actual deletion. The **erase** statement has options that can be included to indicate the extent of deletion to be performed by the DBMS.

8.8.2 Operations on Sets

The DBTG set construct allows related records to be stored as a set. This construct also allows records to be retrieved via their association in one or more set types. A format of the find command can be used to locate the members in a set once we have located the owner record occurrence. Conversely, another format of the find command can be used to locate the owner record occurrence once the member re occurrence has been located.

Locating Records via Sets

To locate a member record occurrence of a member record type in a set, we first locate the appropriate set occurrence by locating the owner record occurrence. Once the owner record occurrence is located, we can locate the first member record occurrence of a given member type by the following format of the find statement:

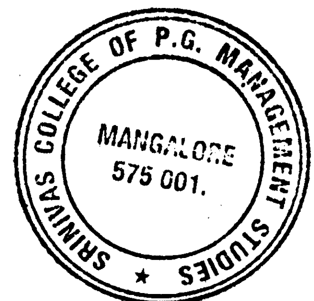
```
find first <member record type> within <set type>
```

The following statements locate the first BOOK_DUE by CLIENT 234 in the set occurrence of set type *BORROWED* owned by 234. The first find statement locates the owner record occurrence and also sets the currency indicator for the set type *BORROWED*. The second find statement locates the first member occurrence of the record type BOOK_DUE.

```
CLIENT.Client_No := 234;
find any CLIENT using CLIENT.Client_No;
find first BOOK_DUE within BORROWED;
```

To locate all the books borrowed by the CLIENT 234 we could use the following program segment:

```
CLIENT.CLIENT_No := 234;
find any CLIENT using CLIENT.Client_No;
find first BOOK_DUE within BORROWED;
while DB_Status = 0 do
    begin
      (* process the current member record *)
      find next BOOK_DUE within BORROWED;
      end
```



In this example we located the first member record occurrence by the **find first within** statement and the subsequent member record occurrences using the **find next within** statement. The order in which the members are located depends on the order specified for the insertion of the members in the set definition.

The clerk at the circulation desk, in addition to checking out the books that have been borrowed by a client, can identify the branch from which a particular copy of a book was borrowed. The following program segment locates and retrieves the name of the branch from which client 234 borrowed the first book.

```
CLIENT.Client_No := 234;
find any CLIENT using CLIENT.Client_No;
find first BOOK_DUE within BORROWED;
find owner within BOOK_COPY_LENT;
get BOOK_COPY;
display ('Branch_Id is', BOOK_COPY.Branch_Id);
```

In this example, we located the first BOOK_DUE, representing the first book borrowed by 234 as before. After locating this book we located its owner in the set *BOOK_COPY_LENT*. The latter owner is an occurrence of the record type BOOK_COPY containing the branch information.

The find first within and the find next within commands for locating members of a set could be used with a singular set in exactly the same manner. However, since there is only one occurrence of a singular set of a given set type and it is owned by the system, there is no need to locate the owner record occurrence before issuing the find first command.

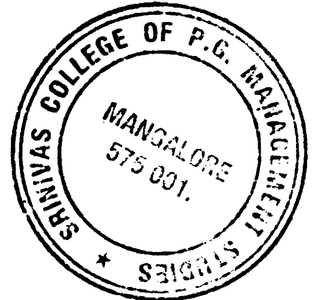
Set Manipulation

The DBTG data manipulation facility proposed a number of operations for manipulating sets. For instance, if a member record type is defined to have manual optional membership in a set type, the user could place a record occurrence of the record record type in a set occurrence. The user could also remove it from a set occurrence and then place it, if required, in another set occurrence at some later time.

For discussion purposes in this section, consider the subschema below, used by a clerk in the acquisition department of the library. The acquisition section of a library procures copies of new or existing books and assigns them to one or more branches; it may also transfer a copy of a book from one branch to another and remove a copy of a book from circulation.

```
Subschema name is Acquisition
        type BOOK = record
                Author: string;
                Title: string;
                Call_No: integer;
                end
        type BOOK_COPY = record
                Call_No: integer;
                Copy_No: integer;
                Branch_Id: string;
                Current_Status: string;
                end
```

```
set is COPY_STATUS
        owner is BOOK
        member is BOOK_COPY optional manual
        end
type BRANCH = record
        Br_Name: string;
        Address: string;
        Phone_No: string;
        end
set is COLLECTION
        owner is BRANCH
        member is BOOK_COPY manual optional
        end
```



Manual Set Manipulation

Let us see how to add a new title, *Anne of Green Gables* by Montgomery, to the collection. The steps involved are the following:

1. Add a record occurrence for the new title to the record type BOOK.
2. Add a record occurrence to the record type BOOK_COPY for every copy that is acquired.
3. Insert the newly created occurrences of BOOK_COPY into the *COPY_STATUS* set occurrence, where the newly inserted occurrence of BOOK is the owner.

The first step is performed using the following statements:

```
BOOK.Author := 'Montgomery';
BOOK.Title := 'Anne of Green Gables';
BOOK.Call_No := 1238;
store   BOOK;
```

Assuming that three copies are acquired and that one copy is to be assigned to each of the three branches of the library, the following statements perform this step.

```
BOOK_COPY.Call_No := 1238;
for i := 1 to 3 do
   begin
     BOOK_COPY.Copy_No := i;
     case i of
          1:BOOK_COPY.Branch_Id := 'Lynn';
          2:BOOK_COPY.Branch_Id := 'Revere';
          3:BOOK_COPY.Branch_Id := 'Salem'
          end
     BOOK_COPY.Current_Status := 'in transit':
     store BOOK_COPY;
     end
```

Note: In the above example, we have embedded the DML statements in an application program in a high level language.

At this point the database contains one record occurrence for the new book and three record occurrences, one for each copy of the book. Now we have to place each of these three occurrences of the record type BOOK_COPY in the set type *COPY_STATUS* wherein the owner is BOOK = 1238. The DBTG command to insert a new member into a set occurrence is the connect command, which specifies the record type that has to be inserted into the set type. The currency indicators have been appropriately initialized to point to the correct member record type occurrence and the correct owner record type occurrence.

The following statements insert the members of the record type BOOK_COPY in the set occurrence of the set type *COPY_STATUS* wherein the owner is BOOK = 1238:

```
BOOK.Call_No := 1238;
find any BOOK using BOOK.Call_No;
            (* establish the pointer for the set type
                            COPY_STATUS *)
BOOK_COPY.Call_No := 1238;
find any BOOK_COPY using BOOK_COPY.Call_No
            retaining currency for COPY_STATUS;
while DB_Status = 0 do
    begin
      connect BOOK_COPY to COPY_STATUS;
      find duplicate BOOK_COPY using
          BOOK_COPY.Call_No
      retaining currency for COPY_STATUS;
      end
```

In the above program segment implementation we used the format of the find statement, which suppresses the updating of the currency indicator for the set type *COPY_STATUS*. Without the **retaining currency** clause, for example, the second find statement would have updated the currency indicator for the set type *COPY_STATUS* to point to the record occurrence of the record type BOOK_COPY. The reason for this is that the record type BOOK_COPY appears as a member of the set type *BOOK_COPY_STATUS* in the subschema.

An alternate method of connecting the record occurrences of the record type, wherein we locate the owner for each insertion, is given below:

```
BOOK.Call_No := 1238,
BOOK_COPY.Call_No := 1238;
find any BOOK_COPY using BOOK_COPY.Call_No;
        (* establish the currency indicator for the
                record type BOOK_COPY *)
while DB_Status = 0 do
    begin
      find any BOOK using BOOK.Call_No;
      (* establish the currency indicator for the
          set type COPY_STATUS *)
      connect BOOK_COPY to COPY_STATUS;
      find duplicate BOOK_COPY using
          BOOK_COPY.Call_No;
      end
```

The reason we do not use the retaining clause in this case is that the find statement for the record type BOOK will set the currency indicator of the record type BOOK as well as the set type *COPY_STATUS* and run unit. However, it will not update the currency indicator for the record type BOOK_COPY.

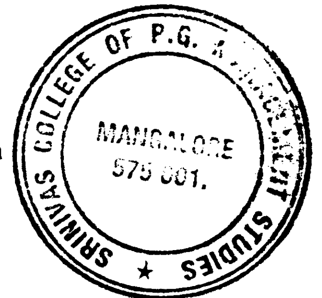
We can combine the operation of storing the record occurrence for BOOK_COPY with connecting the occurrence in the appropriate set occurrence in the set type *COLLECTION*, as illustrated by the following program segment:

```
BOOK.Call_No := 1238;
BOOK_COPY.Call_No := 1238;
for i : = 1 to 3 do
  begin
    BOOK_COPY.Copy_No := i;
    case i of
        1:BOOK_COPY.Branch_Id := 'Lynn ;
        2:BOOK_COPY.Branch_Id := 'Revere';
        3:BOOK_COPY.Branch_Id := 'Salem'
        end
    BOOK_COPY.Current_Status := 'in transit';
    store BOOK_COPY;
    BRANCH.Br_Name := BOOK_COPY.Branch_Id;
    find any BRANCH using BRANCH.Br_Name;
            (* establish the pointer for the set type
                COLLECTION *)
    connect BOOK_COPY to COLLECTION;
    end
```

An occurrence of a record type declared in the set definition to be an optional member of a set could be removed using the disconnect statement. However, before this statement is issued the currency indicator for the record type must be updated to point to the specific occurrence of the record type that is to be removed from the set occurrence. The currency indicator of the set type must also be updated to point to the owner record occurrence of the set type wherein the record is a member.

To remove the *Copy_No* = 3 of the book with *Call_No* = 1238 from the set occurrence of the set type *COPY_STATUS*, we could use the following statements:

```
done := false;
BOOK.Call_No := 1238;
find for update any BOOK using BOOK.Call_No;
    (* establish the currency indicator for the set type
        COPY_STATUS *)
find first BOOK_COPY within COPY_STATUS;
    (* now find its member until Copy_No = 3 is found then
        disconnect it *)
while DB_Status = 0 and not done do
    if BOOK_COPY.Copy_No = 3 then
        begin
            disconnect BOOK_COPY from COPY_STATUS;
            done := true;
            end
```


COLLEGE OF P.G. ... SRINIVAS ... STUDIES
MANGALORE
575 001.

The CLIENT 234 returning the BOOK 1237 would require the circulation clerk to delete the appropriate BOOK_DUE record. The deletion of the record would detach the record from the two set occurrences. In the following application program section, we illustrate the location of the member record occurrence of the record type BOOK_DUE via the set occurrence of the set type *BORROWED* owned by CLIENT 234. We use this member record occurrence of BOOK_DUE in locating the owner record occurrence in the set *BOOK_COPY_LENT* and modify the data-item *Current_Status* of the record type BOOK_COPY. Before issuing the erase instruction we reestablish the currency indicator of the run unit to the record occurrence of BOOK_DUE by locating it as a member in *BOOK_COPY_LENT*.

```
done := false;
CLIENT.Client_No := 234;
find any CLIENT using CLIENT.Client_No;
find first BOOK_DUE within BORROWED;
while DB Status = 0 and not done do
    if BOOK_DUE.Call_No = 1237 then
      done := true
    else find next BOOK_DUE within BORROWED;
if done then
  begin
    find for update owner within BOOK_COPY_LENT;
    BOOK_COPY.CURRENT_Status := in;
    modify BOOK_COPY;
    find for update first BOOK_DUE within BOOK_COPY_LENT;
    disconnect BOOK_DUE from BOOK_COPY_LENT;
    erase BOOK_DUE;
    end
  else
    error routine
```

Deletion of an Owner Record Occurrence

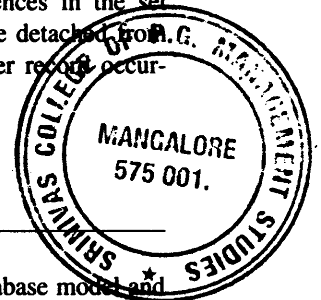
The retention status for the sets BORROWED has been defined as mandatory. An attempt, as shown below, to delete the occurrence of the CLIENT 234, which is the owner of a nonempty set, will fail until all the members in the set are deleted.

```
CLIENT.CLIENT_No := 234;
find for update any Client using CLIENT.Client_No;
erase CLIENT;
```

However, if the retention status for the member record type in the set *BOOK_COPY_LENT* had been defined as fixed, an attempt to delete the occurrence of the owner record type BOOK_COPY (i.e., the record 1237 2 Lynn LENT) would have been successful. When the owner record occurrence is deleted in a set having member record types with the fixed retention status, the member record occurrences will be deleted as well. Furthermore, if the member records are themselves owner of set types with membership retention status fixed, the deletion will be done recursively. The deletion of the member records would have some very undesirable effects if the

member record occurrences were members of other set types. In such a case, the preferable action for the DBMS would be to disconnect these member records from the owner record being deleted.

The retention status for the members of the set *BOOK_COPY_LENT* has been defined as optional. An attempt to delete a record occurrence of BOOK_COPY with a nonempty set would be successful. The member record occurrences in the set *BOOK_COPY_LENT* owned by the occurrence of BOOK_COPY are detached from the set occurrence prior to the deletion of the owner. These member record occurrences would continue to exist in the database.



8.9 Concluding Remarks

The NDM as defined in the DBTG was the first formally defined database model and led to the implementation of a large number of DBMSs from commercial software houses. These systems were designed to run on mainframe and midsize computers.

The advantage of the model is that the data structure diagrams give the user a clear pictorial means of understanding the database structure. The sets and the relationships between record types involved in the sets are predefined. These predefined relationships are usually implemented at the physical level with the use of link structure. This results in faster access to related records than is possible in the relational case using a simple join operation to navigate dynamically through the various relations.

The NDM builds indexes on user (DBA) specified key data-items for direct access to records or groups of records. Once one of the owner record occurrences is located by the use of a selection criterion based on a key, the record occurrences of the member record type(s) can be retrieved relatively quickly.

On the minus side, the query language is procedural and requires the user to navigate through the database by specifying sets, owners, and members. This in turn means that the user has to be cognizant of the structure of the database.

Notwithstanding the above, the model continues to be used extensively for corporate databases in many organizations.

With the current interest in the relational approach, a large number of network-based DBMSs are redesigned to offer the user an optional relational interface, thus combining convenience for the user and at the same time avoiding some of the inefficiencies of the relational approach.

8.10 Summary

The network data model represents entities by records and expresses relationships between entities by means of sets implemented by the use of pointers or links. The model allows the representation of an arbitrary relationship. The DBTG proposal places a number of restrictions on the use of the links.

The basic data definition structure of the DBTG proposal includes records and sets. Record types are representations of entity types and are made up of data-items, vectors, and repeating groups.

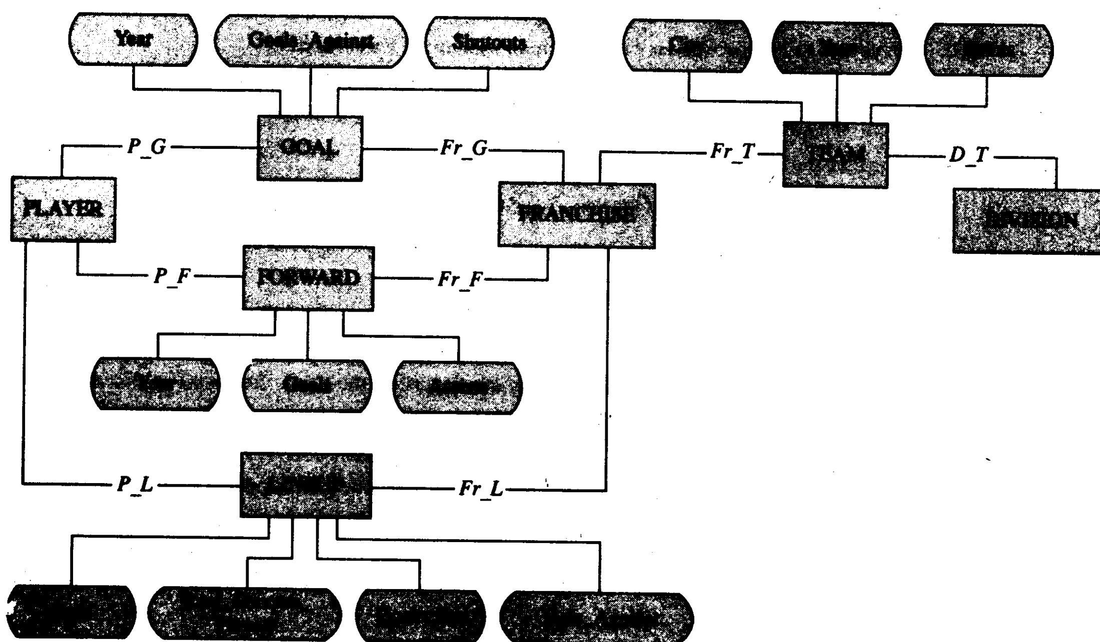
(d) A set occurrence is empty when it has no member record occurrences.
(e) A set type can have only one record type as its member.
(f) A set can represent only a certain relationship between entities; however, not all possible relationships between entities can be conveyed through a set.
(g) Data independence and data integrity suffer due to the set concept.

8.9 Consider a network database with a schema corresponding to the data structure diagram of Figure 8.22, where all the sets have an automatic fixed membership status. Can data ever be inserted in such a database? Amplify your answer with adequate explanations.

8.10 Draw the data structure diagram of the complete library database system discussed in this chapter and comment on the statement that it is a purely hierarchical structure.

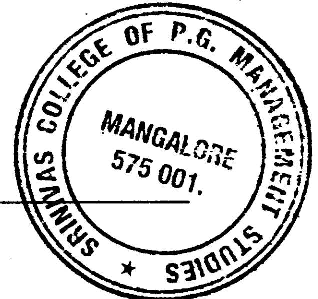
8.11 Consider the database for the UHL that we discussed in Chapter 2. Let us add to the database the requirement of keeping the statistics on the performance of the various lineups during a season. This extension is illustrated in Figure B. A lineup is the group of players from a franchise that plays together for certain shifts during a game. There can be a number of different lineups used during a game and lineups may change from game to game during a season. Here we have added the intersection record LINEUP and the sets P_L and Fr_L. Thus, the relationship between a player and lineup is one to many; similarly the relationship between a lineup and the franchise is also one to many. Give the modified schema for the database and write a pseudocode program to find the best lineup for each player.

Figure B Extended network model for UHL database.



8.12 The following is an incomplete list of DBMSs marketed by various software houses. The names are registered trademarks of the respective companies. Choose one of these DBMSs and describe it in terms of the generalized features described in this chapter.

DBMS	Digital Equipment Corp.
DMS–11	Unisys
DMS–90	Unisys
DMS–1100	Unisys
IDMS	Cullinet
IDS II	Honeywell
IMAGE	Hewlett-Packard
TOTAL	Cincom



Bibliographic Notes

Several commercial database management systems based on what was to be the network approach were implemented in the late 1960s. The DBTG proposal evolved from these systems. The system that had the most influence on the proposal was the Integrated Data Store (IDS) system at General Electric (Bach 64). The IDS was the result of Bachman's early work and was developed under his supervision. Bachman is also credited with developing the data structure diagram for representing records and links used in the network data model (Bach 69). The data structure diagram, like the more recent E–R diagram, is an aid in the logical design of a database system.

The Database Task Group (DBTG) was set up as a special group within CODASYL. The DBTG group issued a final report in 1971 and this was the first standard specification for a database system. A number of commercial database management systems were based on this report. However, it has not been accepted as a standard by ANSI (American National Standards Institute). The DBTG was reconstituted as the Data Description Language Committee (DDLC), which produced a revised version of the scheme data description language (DDL). The ANSI-X3H2 committee received this report, modified it to some extent, and issued the 1981 DDL draft. This, too, has not been accepted to date because the draft lacks a data manipulation language to go with the DDL. In 1984, the X3H2 committee proposed NDL, a standard network database language based on the original DBTG specification. This too has yet to be standardized.

The DBTG proposal is discussed in the CODASYL DBTG 1971 report (CODA 71) and by Olle (Olle 78). Modifications to the original proposal and the DDL are presented in (Coda 78).

Since the DBTG proposal of 1971 there have been various modifications, not only by standards committees but also by software houses offering commercial DBMSs based on this model. Some examples are the DMS-1100 from Unisys (previously called Sperry Univac and which recently has merged with Burroughs) (Sper), TOTAL from Cincom (Cinc), and IDS II from Honeywell (Hone). Some of these systems are discussed in textbooks by Cardenas (Card 85), Date (Date 86), Kroenke (Kroe 83), Tsichritzis and Lochovsky (Tsic 77), and Ullman (Ullm 82)

Like the network data model, the hierarchical data model uses records and pointers or links to represent entities and the relationships among them. However, unlike the network data model, the data structure used is a rooted tree with a strict parent-to-child ordering. We are not going to concentrate on any one of the commercially available DBMSs based on the hierarchical model, although the discussion is somewhat oriented toward features included in IBM's IMS database management system, the most prominent system of this type.

9.1 The Tree Concept

Trees in the form of a family tree or genealogical tree trace the ancestry of an individual and show the relationships among the parents, children, cousins, uncles, aunts, and siblings. A tree is thus a collection of nodes. One node is designated as the root node; the remaining nodes form trees or subtrees.

An **ordered tree** is a tree in which the relative order of the subtrees is significant. This relative order not only signifies the vertical placement or level of the subtrees but also the left to right ordering. Figures 9.1a and b give two examples of ordered trees with **R** as the root node and **A, B,** and **C** as its children nodes. Each of the nodes **A, B,** and **C,** in turn, are root nodes of subtrees with children nodes **(D, E), (F),** and **(G, H, J),** respectively. The significance in the ordering of the subtrees in these diagrams is discussed below

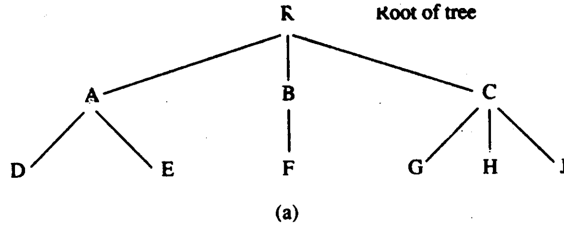
Traversing an ordered tree can be done in a number of ways. The order of processing the nodes of the tree depends on whether or not one processes the node before the node's subtree and the order of processing the subtrees (left to right or right to left). The usual practice is the so-called **preorder traversal** in which the node is processed first, followed by the leftmost subtree not yet processed, as shown below:

```
Procedure Preorder (node);
     process node
     left_child := leftmost child node not processed yet
     while left_child ≠ null do
          begin
               Preorder (left_child)
               left_child := leftmost child node not
                                   processed yet
               end
     end
```

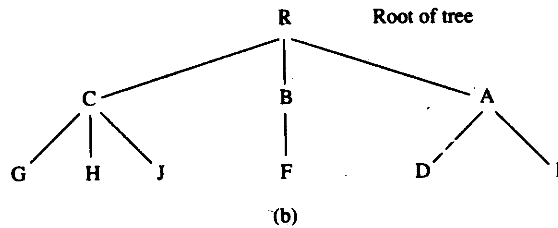
The preorder processing of the ordered tree of Figure 9.1a will process the nodes in the sequence **R, A, D, E, B, F, C, G, H, J.**

The significance of the ordered tree becomes evident when we consider the sequence in which the nodes could be reached when using a given tree traversing strategy. For instance, the order in which the nodes of the hierarchical tree of Figure 9.1b are processed using the preorder processing strategy is not the same as the order for Figure 9.1a, even though the tree of part b contains the same nodes as the tree of part a.

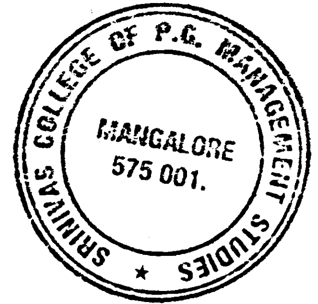
Figure 9.1 Ordered tree where (c) illustrates hierarchical pointers and (d) illustrates child/sibling pointers.

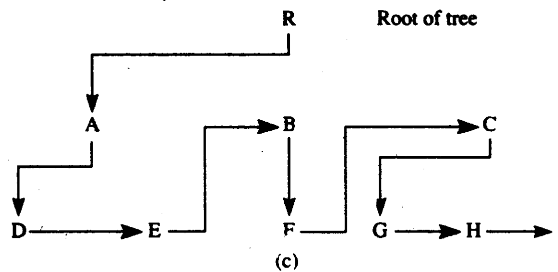


(a)

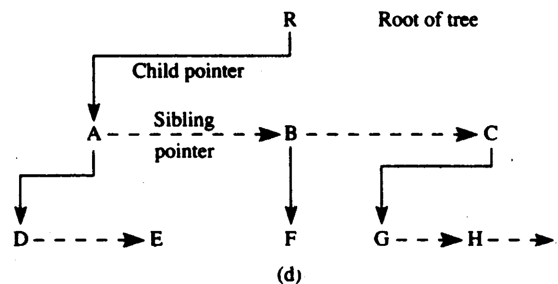


(b)


SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES
MANGALORE
575 001.



(c)



(d)

Two distinct methods can be used to implement the preorder sequence in the ordered tree. The first method, shown in Figure 9.1c uses hierarchical pointers to implement the ordered tree of part a. Here the pointer in each record points to the next record in the preorder sequence. The second method, shown in part d uses two types of pointers, the **child** and the **sibling pointers.** The child pointer is used to point to the leftmost child and the sibling pointer is used to point to the right sibling. The siblings are nodes that have the same parent and the right sibling of a node is the sibling that is immediately to the right of the node in question.

The record types DEPT_SECTION and EMPLOYEE in turn are the parents of the record types EMPL_ASSGNMNT (employee assignment) and *DS_ASSGND* (department or section assigned to), respectively. (Some instances of these hierarchical trees are given in Figures 9.4, 9.5 and 9.6.)

A many-to-many relationship can only be represented in the hierarchical data model by replication of the record concerned or by the use of **virtual records.** For instance, the many-to-many relationships between a BOOK and CLIENT or between DEPT_SECTION and EMPLOYEE, which were represented in the network model by introducing an intermediate record type and two sets, are represented in the hierarchical model by replication of the records or by the use of virtual records. Virtual records are basically pointers that point to the actual physical records in the database. We discuss virtual records in Section 9.2.1.

In Figure 9.3, LIBRARY is a dummy parent that holds together the three hierarchical trees *BOOK_TREE, CLIENT_TREE,* and *BRANCH_TREE.* A DBMS on a given computer system belonging to a library is supporting that library system, so there is no need to actually store a single occurrence of the record type LIBRARY. However, these disjointed trees can be considered to be connected to a single occurrence of the dummy LIBRARY node, and therefore the database contains a single hierarchical tree with this dummy LIBRARY node as the root node. Traversing this tree becomes equivalent to going through the entire database.

If the DBMS were to support the data for more than one library system, the LIBRARY node would actually exist and would form the root node of the subtrees *BOOK TREE, CLIENT_TREE,* and *BRANCH_TREE.* In this case, we would have a forest of trees and for each library system supported by the DBMS, there would exist in the database a tree with the corresponding library node as the root node.

Consider the following definitions for the record types BOOK and BOOK_COPY for the records in the first hierarchical tree, *BOOK_TREE:*

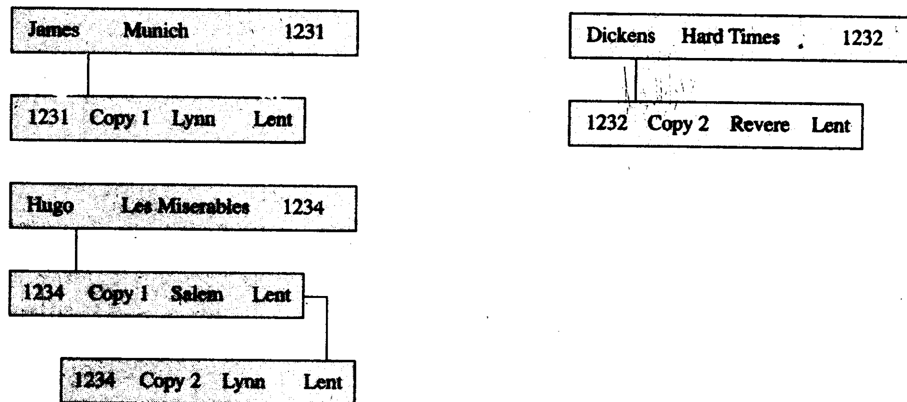
```
type BOOK = record
              Author: string;
              Title: string;
              Call_No: string;
              end

type BOOK_COPY = record
              Call_No: string;
              Copy_No: integer;
              Branch_Id: string;
              Current_Status: string;
              end
```

In Figure 9.4, we give some instances of the hierarchical trees for *BOOK_TREE.* One instance of the tree corresponds to the parent (James Munich 1231) of the record type BOOK; it has its child, the record type BOOK_COPY occurrence (1231 Copy 1 Lynn Lent). Another instance of this hierarchical tree consists of the parent record occurrence (Hugo Les Miserables 1234) and its two children record occurrences of the record type BOOK_COPY.

The record types in the second hierarchy with the root node CLIENT can be defined as follows:

Figure 9.4 Occurrences of *BOOK_TREE* hierarchical tree.

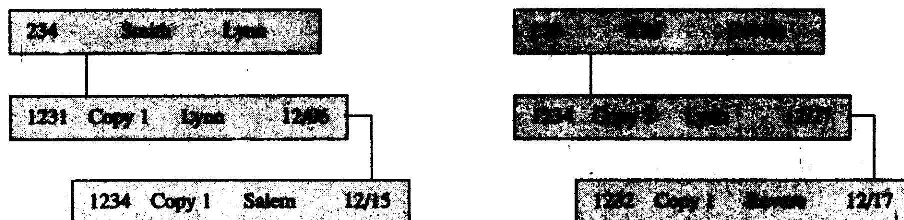



```
type    CLIENT = recora
            Client_No: string;
            Name: string;
            Address: string;
            end

ype     BOOK_DUE = record
            Call_No: string;
            Copy_No: integer;
            Branch_Id: string;
            Current_Status: string;
            Due_Date : string;
            end;
```

Figure 9.5 gives two occurrences of this hierarchy. The first tree corresponas to the CLIENT Smith who has borrowed two *BOOKs* with *Call_Nos* 1231 and 1234 with the *Due_Dates* of 12/06 and 12/15, respectively.

Figure 9.5 Occurrences of *CLIENT_TREE* hierarchical tree.



9.2.2 Expressing a Many-to-Many Relationship

Let us consider the method that we can use to express the relationship between BOOK and CLIENT. As we discussed in Section 8.1.1 this is a many-to-many relationship because the library may have more than one copy (BOOK_COPY) of a given title. However, since only one client can borrow a given copy at a given time, the relationship between a CLIENT and a BOOK_COPY is one-to-one.

In the network model we converted the many-to-many relationship between BOOK and CLIENT into a one-to-many set between BOOK and BOOK_COPY. We then introduced an intermediate record BOOK_DUE to hold the common data between CLIENT and BOOK_COPY and the two one-to-one sets between CLIENT and BOOK_DUE and BOOK_COPY and BOOK_DUE.

In the hierarchical model we can easily express the one-to-many relationship between BOOK and BOOK_COPY as a hierarchy that can be represented by a tree as follows:

```
tree is BOOK_TREE
          BOOK is parent
          BOOK_COPY is child
          end
```

Examples of this hierarchical tree are shown in Figure 9.4.

Similarly, we can express the one-to-many relationship between a client and the items she or he borrows by a hierarchical tree *CLIENT_TREE* as follows:

```
tree is CLIENT_TREE
          CLIENT is parent
          BOOK_DUE is child
          end
```

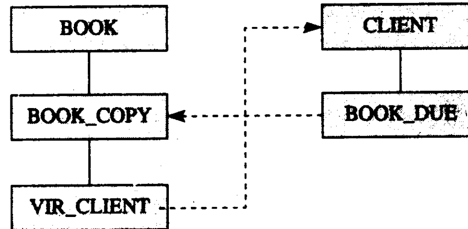
Examples of this hierarchical tree are shown in Figure 9.5.

Suppose the relationship between a BOOK_COPY and a CLIENT who borrows it is expressed by replication as shown in Figures 9.4 and 9.5. The data in BOOK_DUE, except for *Due_Date*, is a duplication of the corresponding data in BOOK_COPY. If a virtual record is used for BOOK_DUE, we could indicate this by the following definition:

```
type  BOOK_DUE = record
          {Call_No: string;
           Copy_No: integer;
           Branch_Id: string;
           Current_Status: string;}
              virtual of logical parent
                  BOOK_COPY in BOOK_TREE;
            Due_Date: string;
          end
```

This indicates that the data items enclosed in the brackets of the record BOOK_DUE are virtual and are derived from the physical record BOOK_COPY, which is defined as the logical parent of the record BOOK_DUE, BOOK_DUE being its logical child. The data item *Due_Date* in this case is the intersection data in the rela-

Figure 9.7 Using virtual records.



tionship between CLIENT and BOOK_COPY. Note that in the above example, the virtual record type BOOK_DUE in the hierarchical tree *CLIENT_TREE* contains data that is derived from a separate physical hierarchical tree, namely *BOOK_TREE*.

Similarly, to keep track of which CLIENT has borrowed a given BOOK_COPY, we can introduce a virtual record type VIR_CLIENT and a one-to-one relationship *BOOK_COPY_TREE* between BOOK_COPY and VIR_CLIENT as follows:

```
tree is BOOK_COPY_TREE
        BOOK_COPY is parent
        VIR_CLIENT is child
        end

type VIR_CLIENT = record
        { Client_No: string;
         Name: string;
         Address: string;}
           virtual of logical parent CLIENT in CLIENT_TREE;
        end
```

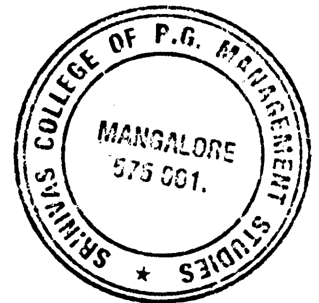

COLLEGE OF P.G. MANAGEMENT STUDIES
MANGALORE
575 001.
SRINIVAS

Figure 9.7 now includes the modified section of the hierarchical structure diagram of Figure 9.3, showing the many-to-many relationship between BOOK and CLIENT.

The problem with this hierarchy is that to determine the author and title, etc., of the volumes borrowed by client Smith, we have to go through the following inefficient series of operations:

- Go from the required occurrence of the record type CLIENT to the first occurrence of its child record type BOOK_DUE.
- Follow the pointer to the logical parent of BOOK_DUE to an occurrence of BOOK COPY and note the *Call_No*.
- Search the occurrences of BOOK with the same *Call_No* and retrieve the details pertaining to the *Author*, etc.
- Repeat for each child occurrence of BOOK_DUE belonging to Smith.

Such queries can be handled more efficiently if we add another dependent record to *CLIENT_TREE*, such as VIR_BOOK, defined to be virtual of the logical parent BOOK as follows on the next page.

ASSGND is a logical child of the record type DEPT_SECTION. This virtual record contains the intersection data *Hours*, which represents the hours worked by the employee during a work week for a given DEPT_SECTION. The intersection data is a replication of that in the virtual record EMPL_ASSGNMNT. Unlike the examples of the virtual record discussed in Section 9.2.2, the virtual records EMPL_ASSGNMNT and DS_ASSGND have as their logical parent a record in the same physical hierarchical tree, namely, the *BRANCH_TREE* of Figure 9.3.

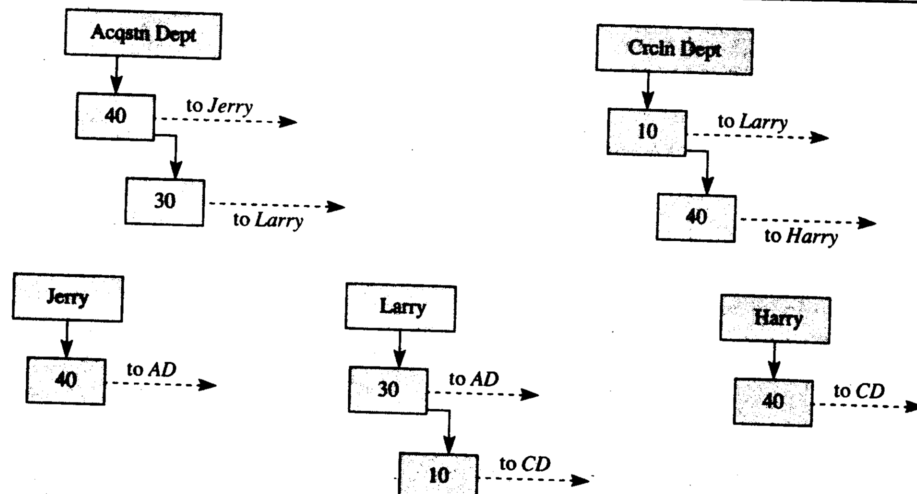
```
tree is EMPLOYEE_TREE
        EMPLOYEE is parent
        DS_ASSGND is child
        end

type EMPLOYEE = record
        Emp_Name: string;
        Home_Address: string;
        Phone_No: string;
        end

type DS_ASSGND = record
        {Ds_Name: string;
        Room_No: string;
        Phone_No: string;}
          virtual of logical parent
          DEPT_SECTION in BRANCH_TREE
        Hours: integer;
        end
```

Figure 9.11 gives some occurrences of the hierarchical trees *DS_TREE* and *EMPLOYEE_TREE*. The instance of *DS_TREE* rooted by the Acqstn Dept is shown

Figure 9.11 Sample occurrences of *DS_TREE* AND *EMPLOYEE_TREE*.



to have two occurrences of the dependent record type EMPL_ASSGNMNT. One of these contains the intersection data corresponding to employee Jerry and the other is for employee Larry. A pointer in each of these records, point to the logical parent.

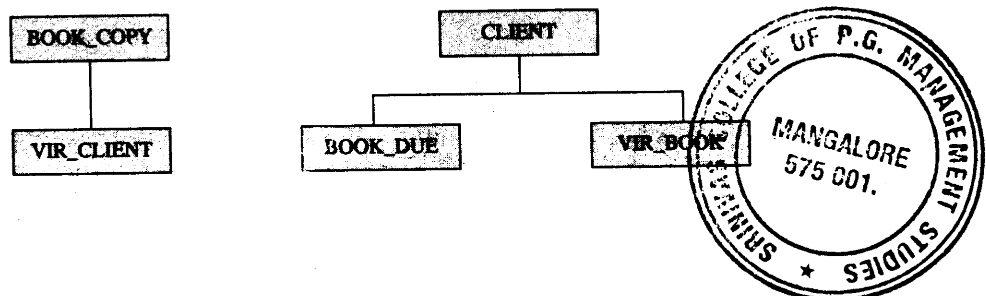
The above example is an illustration of a **paired bidirectional logical relationship** of the hierarchical model. In such a relationship a many-to-many correspondence between two record types is resolved by introducing two virtual records with these record types as the logical parents. In the above example, the record types are DEPT_SECTION and EMPLOYEE. EMPL_ASSGNMNT is a virtual record that is a physical child of DEPT_SECTION and a logical child of EMPLOYEE; DS_ASSGND is a physical child of EMPLOYEE and a logical child of DEPT_SECTION. Each of these virtual record types contains appropriate pointers to the logical parents and the intersection data, *Hours*, may be replicated as we have done. The replicated data is stored in the two virtual record types and could lead to inconsistencies. Since the DBMS is aware of this controlled redundancy it has the responsibility for ensuring that whenever one of the replicated values in the intersection data is changed, its twin value is also changed.

9.3 Data Definition

The hierarchical database consists of a collection of hierarchical trees (or set of spanning trees) which are described using a database description facility. Figure 9.12 gives part of the hierarchical definition tree for our library database example. The corresponding data definition is given below. The trees described could be actual physically stored trees or logical trees derived from the physically stored trees. In the latter case, the logical trees can be considered to be user or external views. The logical trees are also hierarchical and derived from one or more physical trees and could contain virtual records. Defining a new logical tree thus may involve implementing pointers for the virtual records and as such is a reorganization of the physical database. Such a reorganization is performed by the DBA. A virtual record in a hierarchical tree can be materialized from its logical parent record. The latter may or may not be in the same physical hierarchical tree.

We used a Pascal-like convention to define the database, wherein we introduced the tree structure by listing the root of the tree and all its children record types. For the sake of clarity and simplicity, we avoided the introduction of implementation-related details such as specifying the number and types of pointers. In the commercially available database management products based on the hierarchical data model, the data definition requires the specification of these details.

Figure 9.12 Logical database as viewed by circulation clerk.



SRINIVAS COLLEGE OF P.G. MANAGEMENT STUDIES MANGALORE 575 001.

9.4.2 Basic Data Manipulation

The basic data retrieval command in the hierarchical data model is the **get** command, which unlike in the network data model need not be preceded by a find command. The command retrieves the appropriate occurrence of the record type, places it in the corresponding record type template in the UWA, and sets the currency indicators for the relevant hierarchical tree. In this instance, the currency indicators will be the current record of the run unit and the parent of the current record retrieved. The record occurrence to be retrieved is specified by indicating the condition to be met by the retrieved record. The hierarchical path to be used for the retrieval may also be given to retrieve a record. For instance, the condition specified in the get command may involve the parent (or one of the grandparents) of the record being retrieved.

The first format of the get command that we will discuss is the **get first.** This format is sometimes called **get unique** or **get leftmost.** Note that the hierarchical tree is traversed using the preorder scheme. Consequently, the get first command will retrieve the first record that meets this condition. The syntax of this format of the get command is as follows:

get first <record type> **where** <condition>

The **where** <condition> clause is optional and if it is omitted, the first record of the specified record type is retrieved and placed in the corresponding record template within the UWA. Once the command is successfully executed, the DB-Status register contains a value of 0 and the currency indicators are set. If the command is not executed successfully, i.e., if no record exists in the database that satisfies the specified condition, the the DB-Status will contain an error code.

For the sample database given in Figure 9.13b the following statements will locate the first record type BOOK_DUE for CLIENT Smith, and if the record is successfully located then the values for the data items *Call_No* and *Due_Date* are displayed:

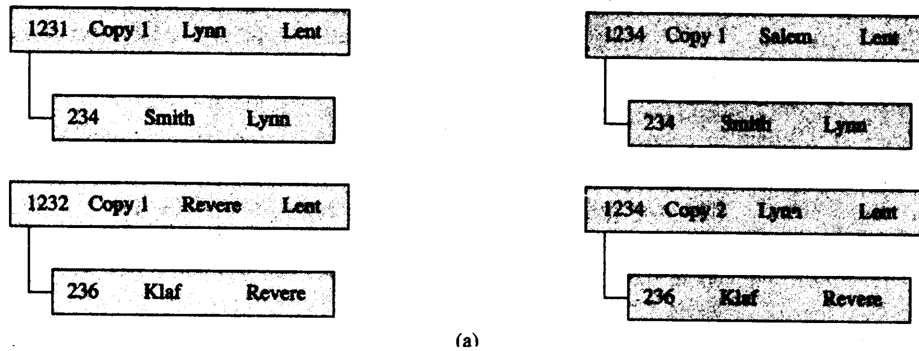
get first BOOK_DUE **where** CLIENT.*Name* = 'Smith';
if **DB-Status** = 0 *then*
  display (BOOK_DUE.*Call_No*, BOOK_DUE.*Due_Date*);

The above statements for the sample database will display 1231 12/06.

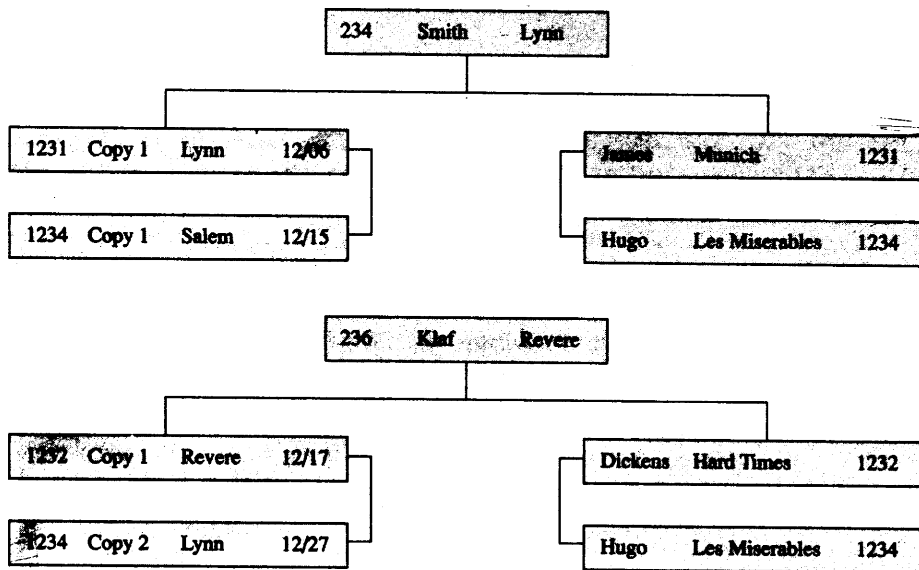
9.4.3 Sequential Retrieval

The **get next** statement is used in the hierarchical database to do sequential processing in preorder. Once the position for a run unit is established in the database with a get first statement, the get next statement performs the retrieval in the forward sense. If the database contains disjoint hierarchical trees, we assume that the DBMS provides a dummy record and these disjoint trees are considered the children of the DBMS supplied unique dummy root record occurrence. The order of these disjoint hierarchical trees is their order in the data definition. For our example, we assume that there is a dummy record LIBRARY, which is the root of the hierarchical trees *BOOK*

Figure 9.13 (a) Sample database: *BOOK_COPY_TREE;* (b) sample database: *CLIENT_TREE.*



(a)



(b)

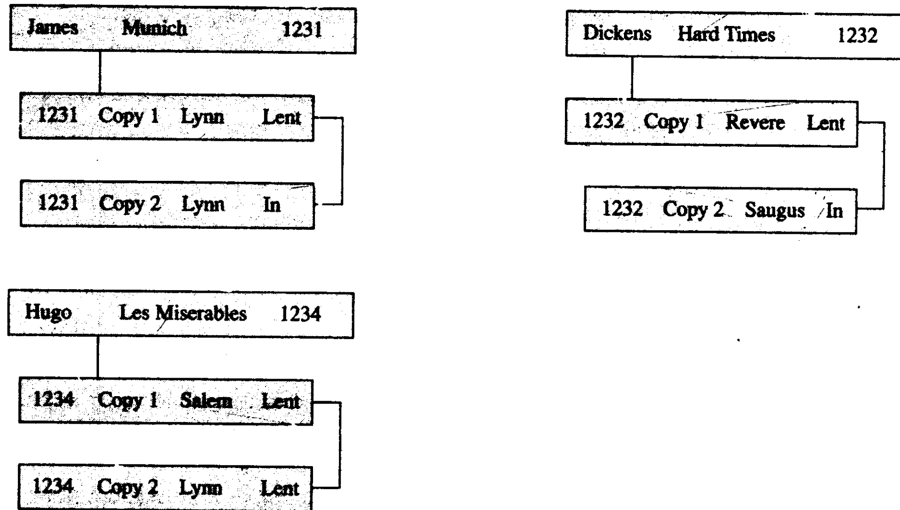
TREE, *CLIENT_TREE*, and *BRANCH_TREE*. The format of the get next statement is:

get next <record type> **where** <condition>

As in the **get first** statement, the **where** clause is optional; the <record type> specification is also optional. In case the **get first** statement appears without any options, the retrieval is of the next record in the database in preorder. If the <record type> is specified, the retrieval is of the next record of the specified type in the preorder. If both the <record type> and the **where** <condition> are included, the retrieval is the next record of the specified type that satisfies the <condition>.

Once we have located the first occurrence of the BOOK_DUE child of Smith, we can retrieve and display the subsequent occurrences using the following on the next page.

Figure 9.15 Sample database contents.



9.5.1 Insert

The format of the command to insert a new occurrence of a record type is given by:

insert <record type> **where** <condition>

When a new record is to be inserted in the database, the parentage of the record, unless it is at the root of a hierarchical tree, is specified with the where clause. Without the parentage information the DBMS will insert the record in the first possible position where the specified record type appears in the data definition. When the new record to be inserted is a child record type, we assume that it will be inserted in the first position in the preorder traversal, which will be to the left of the current leftmost child. The record to be inserted is first created in the record template in the UWA before the **insert** statement is executed.

The following statements create a new occurrence of the record type BOOK in the database:

BOOK.*Author* := 'Montgomery';
BOOK.*Title* := 'Anne of Green Gables';
BOOK.*Call_No* := 1235;
insert (BOOK);

Here we did not specify the parentage of the record type to be inserted because it is at the root of the hierarchical *BOOK_TREE*.

The following statements insert a copy of this new title into the database tree occurrence, corresponding to the new root record occurrence just inserted in the database. The parent record is specified in the where clause.

BOOK_COPY.*Call_No* := 1235;
BOOK_COPY.*Copy_No* := 1;

```
BOOK_COPY.Branch_Id := 'Lynn';
BOOK_COPY.Status := 'transit';
insert (BOOK_COPY)
     where (BOOK.Call_No = 1235);
```

Without the where <condition> clause, the record will be inserted in the data base, but since a child record cannot exist in a hierarchical database without a parent record, it is connected to the first possible position where such a record could exist. For our sample database, assuming that Figure 9.15 is the preorder of the BOOK_TREE hierarchy, the new BOOK_COPY will be inserted in the tree with the (James Munich 1231) root node if the insert statement did not have the where clause.

9.5.2 Modification and Deletion

A record that is to be modified or deleted from the database must first be retrieved using a locking form of the get statement as follows:

```
get hold first <record type>
```

The need to hold the record arises when there are a number of concurrent run units using the database. A run unit issuing the **get hold** locks out the other programs from the record occurrence and thus avoids the anomalies associated with concurrent updates (see Chapter 12).

The following statements modify the BOOK_COPY.*Branch_Id* of the second copy of the BOOK (James Munich 1231) from Lynn to Salem.

```
get first (BOOK)
     where BOOK.Call_No = 1231;
get hold first BOOK_COPY
     where BOOK_COPY.Copy_No = 2;
BOOK_COPY.Branch_Id := 'Salem';
BOOK_COPY.Status := 'transit';
replace;
```

The first statement locates the root node of the hierarchical tree occurrence where the required BOOK is the parent. The next statement retrieves and locks the child record occurrence of BOOK_COPY where the BOOK_COPY.*Copy_No* is 2. The fields to be modified are changed in the next two statements within the record template. The last statement replaces the record occurrence of BOOK_COPY with the modified record. After execution of the **replace** statement, the lock on the record occurrence of BOOK_COPY is removed.

The following statements **delete** the BOOK_COPY record occurrence of the second copy of the BOOK (Dickens Hard Times 1232).

```
get first (BOOK)
     where BOOK.Call_No = 1232;
get hold first BOOK_COPY
     where BOOK_COPY.Copy_No = 2:
delete;
```

The first statement locates the root node of the hierarchy tree occurrence, where the required BOOK is the parent. The next statement retrieves and locks the child record occurrence of BOOK_COPY, where the BOOK_COPY.*Copy_No* is 2. The last statement deletes the record occurrence of BOOK_COPY.

When a record to be deleted is a parent record occurrence of a hierarchical tree or subtree, all the children (and grand children) record occurrences are also deleted. This action is similar to the deletion of the owner record occurrence of a set in DBTG with fixed membership, wherein all occurrences of the member records are also deleted.

9.5.3 Updates of Virtual Records

Let us return to the logical database as viewed by the clerk at the circulation desk, given in Figure 9.12. The logical database contains a number of virtual records. Some parts of these records (excluding the intersection data portion) are derived from their logical parent records, which are actual physical records. For the data retrieval operations, the logical database can be processed exactly as if the virtual record were really a physical one. In other words, the virtual records are materialized from their logical parent records. An update operation, however, could have an effect on the underlying physical records. Some of these operations are disallowed, while other operations could cause these logical parent records to be inserted, modified, or deleted. The operations that are allowed and their effects are determined by the rules for the insert, delete, and replace operations on the record type related to a virtual record. IMS uses options and associated rules that could be called physical, logical, or virtual for each of these update operations. The effects of these are, in a way, similar to the effects of the DBTG membership insertion and retention options we discussed in Chapter 8. We summarize some of the possibilities below. Details of these rules can be found in the application manuals of the commercially available DBMs based on the hierarchical approach.

Inserting a new occurrence of a CLIENT record is allowed because it is a physical record in the logical view. The following statements create a new occurrence of the record type CLIENT in the database.

CLIENT.*Client_No* := '237';
CLIENT.*Name* := 'Cook';
CLIENT.*Address* := 'Peabody';
insert (CLIENT);

Here we need not specify the parentage of CLIENT because it is the root node of a hierarchical tree.

Inserting an instance of BOOK_DUE for a nonexistent CLIENT will not be allowed, since in the hierarchical data model a child record cannot exist without the parent record and such operations will fail.

Inserting an instance of BOOK_DUE for a nonexistent BOOK_COPY, depending on the rule specified for the logical parent BOOK_COPY, would fail or succeed. It will fail if the insert rule for BOOK_COPY is specified as physical. However, if the insert rule for BOOK_COPY is logical or virtual, then on insertion of BOOK_