Department of Commerce (CA)

CORE PAPER-II-DATABASE SYSTEM CONCEPTS

SEMESTER:I SUB CODE: 18MCC12C

M.COM(CA)

UNIT4: Hierarchical approach-physical database-database decription-hierarchical sequence-external level of IMS-Logical databases-program communication block-IMS data manipulation-defining the program communication block-DL/I examples.

REFERENCE BOOK:

An introduction to database system-C.J. Dates

An introduction to database system-Bipin

PREPARED BY: DR. E.N. KANJANA,

ASST PROFESSOR.

Introduction

This chapter provides an overview of the network data model and hierarchical data model. The original network model and language were presented in the CODASYL Data Base Task Group's 1971 report; hence it is sometimes called the DBTG model. Revised reports in 1978 and 1981 incorporated more recent concepts. In this chapter, rather than concentrating on the details of a particular CODASYL report, we present the general concepts behind network-type databases and use the term **network model** rather than CODASYL model or DBTG model.

The original CODASYL/DBTG report used COBOL as the host language. Regardless of the host programming language, the basic database manipulation commands of the network model remain the same. Although the network model and the object-oriented data model are both navigational in nature, the data structuring capability of the network model is much more elaborate and allows for explicit insertion/deletion/modification semantic specification. However, it lacks some of the desirable features of the object models.

There are no original documents that describe the hierarchical model, as there are for the relational and network models. The principles behind the hierarchical model are derived from Information Management System (IMS), which is the dominant hierarchical system in use today by a large number of banks, insurance companies, and hospitals as well as several government agencies.

10.2 Network Data Modeling Concepts

There are two basic data structures in the network model: records and sets.

10.2.1 Records, Record Types, and Data Items

Data is stored in **records;** each record consists of a group of related data values. Records are classified into **record types,** where each record type describes the structure of a group of records that store the same type of information. We give each record type a name, and we also give a name and format (data type) for each **data item** (or attribute) in the record type. Figure 10.1 shows a record type STUDENT with data items NAME, SSN, ADDRESS, MAJORDEPT, and BIRTHDATE

We can declare a virtual data item (or derived attribute) AGE for the record type shown in Figure 10.1 and write a procedure to calculate the value of AGE from the value of the actual data item BIRTHDATE in each record.

A typical database application has numerous record types—from a few to a few hundred.

To represent relationships between records, the network model provides the modeling construct called *set type,* which we discuss next.

10.2.2 Set Types and Their Basic Properties

A **set type** is a description of a 1:N relationship between two record types. Figure 10.2 shows how we represent a set type diagrammatically as an arrow. This type of diagrammatic representation is called a **Bachman diagram.** Each set type definition consists of three basic elements:

- A name for the set type.
- An owner record type.
- A member record type.

The set type in Figure 10.2 is called MAJOR_DEPT; DEPARTMENT is the **owner** record type, and STUDENT is the **member** record type. This represents the 1:N relationship between academic departments and students majoring in those departments. In the database itself, there will be many **set occurrences** (or **set instances**) corresponding to a set type. Each instance relates one record from the owner record type—a DEPARTMENT record in our example—to the set of records from the member record type related to it—the set of STUDENT records for students who major in that department. Hence, each set occurrence is composed of:

- One owner record from the owner record type.
- A number of related member records (zero or more) from the member record type.

A record from the member record type *cannot exist in more than one set occurrence* of a particular set type. This maintains the constraint that a set type represents a 1:N relationship. In our example a STUDENT record can be related to at most one major DEPARTMENT and hence is a member of at most one set occurrence of the MAJOR_DEPT set type.

A set occurrence can be identified either by the *owner record* or by *any of the member records.* Figure 10.3 shows four set occurrences (instances) of the MAJOR_DEPT set type. Notice that each set instance *must* have one owner record but can have any number of member records (**zero** or more). Hence, we usually refer to a set instance by its owner record. The four set instances in Figure 10.3 can be referred to as the 'Computer Science', 'Mathematics', 'Physics', and 'Geology' sets. It is customary to use a different representation of a set instance (Figure 10.4) where the records of the set instance are shown linked together by pointers, which corresponds to a commonly used technique for implementing sets.

In the network model, a set instance is *not identical* to the concept of a set in mathematics. There are two principal differences:

- The set instance has one *distinguished element*—the owner record—whereas in a mathematical set there is no such distinction among the elements of a set.

- In the network model, the member records of a set instance are *ordered,* whereas order of elements is immaterial in a mathematical set. Hence, we can refer to the

first, second, i_{th}, and last member records in a set instance. Figure 10.4 shows an alternate "linked" representation of an instance of the set MAJOR_DEPT.

the record of ‘Manuel Rivera’ is the first STUDENT (member) record in the ‘Computer Science’ set, and that of ‘Kareem Rashad’ is the last member record. The set of the network model is sometimes referred to as an **owner-coupled set** or **co-set,** to distinguish it from a mathematical set.

Special Types of Sets

System-owned (Singular) Sets

One special type of set in the CODASYL network model is worth mentioning: SYSTEMowned sets.

Figure 10.5A singular (SYSTEM-owned) set ALL_DEPTS.

A **system-owned** set is a set with no owner record type; instead, the system is the owner. We can think of the system as a special "virtual" owner record type with only a single record occurrence. System-owned sets serve two main purposes in the network model:

• They provide *entry points* into the database via the records of the specified member record type. Processing can commence by accessing members of that record type, and then retrieving related records via other sets.

• They can be used to *order* the records of a given record type by using the set ordering specifications. By specifying several system-owned sets on the same record type, a user can access its records in different orders.

6

A system-owned set allows the processing of records of a record type by using the regular set operations. This type of set is called a **singular** set because there is only one set occurrence of it. The diagrammatic representation of the system-owned set ALL_DEPTS is shown in Figure 10.5, which allows DEPARTMENT records to be accessed in order of some field—say, NAME—with an appropriate set-ordering specification. Other special set types include recursive set types, with the same record serving as an owner and a member, which are mostly disallowed; multimember sets containing multiple record types as members in the same set type are allowed in some systems.

10.2.4 Stored Representations of Set Instances

A set instance is commonly represented as a **ring (circular linked list)** linking the owner record and all member records of the set, as shown in Figure 10.4. This is also sometimes called a **circular chain.** The ring representation is symmetric with respect to all records; hence, to distinguish between the owner record and the member records, the DBMS includes a special field, called the **type field,** that has a distinct value (assigned by the DBMS) for each record type. By examining the type field, the system can tell whether the record is the owner of the set instance or is one of the member records. This type field is hidden from the user and is used only by the DBMS.

In addition to the type field, a record type is automatically assigned a **pointer field** by the DBMS for *each set type in which it participates as owner or member.* This pointer can be considered to be *labeled* with the set type name to which it corresponds; hence, the system internally maintains the correspondence between these pointer fields and their set

types. A pointer is usually called the **NEXT pointer** in a member record and the **FIRST pointer** in an owner record because these point to the next and first member records, respectively. In our example of Figure 10.4, each student record has a NEXT pointer to the next student record within the set occurrence. The NEXT pointer of the *last member record* in a set occurrence points back to the owner record. If a record of the member record type does not participate in any set instance, its NEXT pointer has a special **nil**

pointer. If a set occurrence has an owner but no member records, the FIRST pointer points right back to the owner record itself or it can be **nil.**

The preceding representation of sets is one method for implementing set instances. In general, a DBMS can implement sets in various ways. However, the chosen representation must allow the DBMS to do all the following operations:

• Given an owner record, find all member records of the set occurrence.

• Given an owner record, find the first, i_{th}, or last member record of the set occurrence. If no such record exists, return an exception code.

• Given a member record, find the next (or previous) member record of the set occurrence. If no such record exists, return an exception code.

• Given a member record, find the owner record of the set occurrence.

The circular linked list representation allows the system to do all of the preceding operations with varying degrees of efficiency. In general, a network database schema has many record types and set types, which means that a record type may participate as owner and member in numerous set types. For example, in the network schema that appears later as Figure 10.8, the EMPLOYEE record type participates as owner in four set TYPES—MANAGES, IS_A_SUPERVISOR, E_WORKSON, and DEPENDENTS_OF—and participates as member in two set types—WORKS_FOR and SUPERVISEES. In the circular linked list representation, six additional pointer fields are added to the EMPLOYEE record type. However, no confusion arises, because each pointer is labeled by the system and plays the role of FIRST or NEXT pointer for a *specific set type.*

10.2.5 Using Sets to Represent M:N Relationships

A set type represents a 1:N relationship between two record types. This means that *a record of the member record type can appear in only one set occurrence.* This constraint is automatically enforced by the DBMS in the network model. To represent a 1:1 relationship, the extra 1:1 constraint must be imposed by the application program.

An M:N relationship between two record types cannot be represented by a single set type. For example, consider the WORKS_ON relationship between EMPLOYEEs and

PROJECTs. Assume that an employee can be working on several projects simultaneously and that a project typically has several employees working on it. If we try to represent this by a set type, neither the set type in Figure 10.6(a) nor that in Figure 10.6 (b) will represent the relationship correctly. Figure 10.6(a) enforces the incorrect constraint that a PROJECT record is related to only one EMPLOYEE record, whereas Figure 10.6(b) enforces the incorrect constraint that an EMPLOYEE record is related to only one PROJECT record. Using both set types E_P and P_E simultaneously, as in Figure 10.6(c), leads to the problem of enforcing the constraint that P_E and E_P are mutually consistent inverses, plus the problem of dealing with relationship attributes.

The correct method for representing an M:N relationship in the network model is to use two set types and an additional record type, as shown in Figure 10.6(d). This additional record type—WORKS_ON, in our example—is called a **linking** (or **dummy**) record type. Each record of the WORKS_ON record type must be owned by one EMPLOYEE record through the E_W set and by one PROJECT record through the P_W set and serves to relate these two owner records.

Figure 10.6 Representing M:N relationships. (a)–(c) Incorrect representations. (d) Correct representation using a linking record type.

This is illustrated conceptually in Figure 10.6(e).

Figure 10.6(f) shows an example of individual record and set occurrences in the linked list representation corresponding to the schema in Figure 10.6(d). Each record of the WORKS_ON record type has two NEXT pointers: the one marked NEXT(E_W) points

9

to the next record in an instance of the E_W set, and the one marked NEXT(P_W) points to the next record in an instance of the P_W set. Each WORKS_ON record relates its two owner records. Each WORKS_ON record also contains the number of hours per week that an employee works on a project. The same occurrences in Figure 10.6(f) are shown in Figure 10.6(e) by displaying the W records individually, without showing the pointers.

To find all projects that a particular employee works on, we start at the EMPLOYEE record and then trace through all WORKS_ON records owned by that EMPLOYEE, using the FIRST(E_W) and NEXT(E_W) pointers. At each WORKS_ON record in the set occurrence, we find its owner PROJECT record by following the NEXT(P_W) pointers until we find a record of type PROJECT. For example, for the E2 EMPLOYEE record, we follow the FIRST(E_W) pointer in E2 leading to W1, the NEXT(E_W) pointer in W1 leading to W2, and the NEXT(E_W) pointer in W2 leading back to E2. Hence, W1 and W2 are identified as the member records in the set occurrence of E_W owned by E2. By following the NEXT(P_W) pointer in W1, we reach P1 as its owner; and by following the NEXT(P_W) pointer in W2 (and through W3 and W4), we reach P2 as its owner. Notice that the existence of direct OWNER pointers for the P_W set in the WORKS_ON records would have simplified the process of identifying the owner PROJECT record of each WORKS_ON record.

10

Figure 10.6 ***(Continued)*** **(e) Some instances. (f) Using linked representation.**

In a similar fashion, we can find all EMPLOYEE records related to a particular PROJECT. In this case the existence of owner pointers for the E_W set would simplify processing. All this pointer tracing is done *automatically by the DBMS;* the programmer has DML commands for directly finding the owner or the next member.

Notice that we could represent the M:N relationship as in Figure 10.6(a) or Figure 10.6(b) if we were allowed to duplicate PROJECT (or EMPLOYEE) records. In Figure 10.6(a) a PROJECT record would be duplicated as many times as there were employees working on the project. However, duplicating records creates problems in maintaining consistency among the duplicates whenever the database is updated, and it is not recommended in general .

11

10.3 Constraints in the Network Model

In explaining the network model so far, we have already discussed "structural"

constraints that govern how record types and set types are structured. In the present section we discuss "behavioral" constraints that apply to (the behavior of) the members of sets when insertion, deletion, and update operations are performed on sets. Several constraints may be specified on set membership. These are usually divided into two main categories, called **insertion options** and **retention options** in CODASYL terminology. These constraints are determined during database design by knowing how a set is required to behave when member records are inserted or when owner or member records are deleted. The constraints are specified to the DBMS when we declare the database structure, using the data definition language. Not all combinations of the constraints are possible. We first discuss each type of constraint and then give the allowable combinations.

10.3.1 Insertion Options (Constraints) on Sets

The insertion constraints—or options, in CODASYL terminology—on set membership specify what is to happen when we insert a new record in the database that is of a member record type. A record is inserted by using the STORE command. There are two options:

- AUTOMATIC: The new member record is *automatically connected* to an appropriate set occurrence when the record is inserted.
- MANUAL: The new record is not connected to any set occurrence. If desired, the programmer can explicitly (*manually*) connect the record to a set occurrence subsequently by using the CONNECT command.

For example, consider the MAJOR_DEPT set type of Figure 10.2. In this situation we can have a STUDENT record that is not related to any department through the MAJOR_DEPT set (if the corresponding student has not declared a major). We should therefore declare the MANUAL insertion option, meaning that when a member STUDENT record is inserted in the database it is not automatically related to a DEPARTMENT record through the MAJOR_DEPT set. The database user may later insert the record "manually" into a set instance when the corresponding student declares a

major department. This manual insertion is accomplished by using an update operation called CONNECT, submitted to the database system.

The AUTOMATIC option for set insertion is used in situations where we want to insert a member record into a set instance automatically upon storage of that record in the database. We must specify a criterion for *designating the set instance* of which each new record becomes a member. As an example, consider the set type shown in Figure 10.7(a), which relates each employee to the set of dependents of that employee. We can declare the EMP_DEPENDENTS set type to be AUTOMATIC, with the condition that a new DEPENDENT record with a particular EMPSSN value is inserted into the set instance owned by the EMPLOYEE record with the same SSN value.

Figure 10.7 Different set options. (a) An AUTOMATIC FIXED set. (b) An AUTOMATIC MANDATORY set.

10.3.2 Retention Options (Constraints) on Sets

The retention constraints—or options, in CODASYL terminology—specify whether a record of a member record type can exist in the database on its own or whether it must always be related to an owner as a member of some set instance. There are three retention options:

• OPTIONAL: A member record can exist on its own *without being* a member in any occurrence of the set. It can be connected and disconnected to set
13
occurrences at will by means of the CONNECT and DISCONNECT commands of the network DML.
• MANDATORY: A member record *cannot* exist on its own; it must *always* be a member in some set occurrence of the set type. It can be reconnected in a single operation from one set occurrence to another by means of the RECONNECT command of the network DML.
• FIXED: As in MANDATORY, a member record *cannot* exist on its own. Moreover, once it is inserted in a set occurrence, it is *fixed;* it *cannot* be reconnected to another set occurrence.

We now illustrate the differences among these options by examples showing when each option should be used. First, consider the MAJOR_DEPT set type of Figure 10.2. To provide for the situation where we may have a STUDENT record that is not related to any department through the MAJOR_DEPT set, we declare the set to be OPTIONAL. In Figure 10.7(a) EMP_DEPENDENTS is an example of a FIXED set type, because we do not expect a dependent to be moved from one employee to another. In addition, every DEPENDENT record must be related to some EMPLOYEE record at all times. In Figure 10.7(b) a MANDATORY set EMP_DEPT relates an employee to the department the employee works for. Here, every employee must be assigned to exactly one department at all times; however, an employee can be reassigned from one department to another.

By using an appropriate insertion/retention option, the DBA is able to specify the behavior of a set type as a constraint, which is then *automatically* held good by the system.
14

Figure 10.8 A network schema diagram for the COMPANY database.

10.4 Data Manipulation in a Network Database

In this section we discuss how to write programs that manipulate a network database—including such tasks as searching for and retrieving records from the database; inserting, deleting, and modifying records; and connecting and disconnecting records from set occurrences. A **data manipulation language (DML)** is used for these purposes. The DML associated with the network model consists of record-at-a-time commands that are embedded in a general-purpose programming language called the **host language.** Embedded commands of the DML are also called the **data sublanguage.** In practice, the most commonly used host languages are COBOL and PL/I. In our examples, however, we show program segments in PASCAL notation augmented with network DML commands.
15

10.4.1 Basic Concepts for Network Database Manipulation

To write programs for manipulating a network database, we first need to discuss some basic concepts related to how data manipulation programs are written. The database system and the host programming language are two separate software systems that are linked together by a common interface and communicate only through this interface. Because DML commands are record-at-a-time, it is necessary to identify specific records of the database as **current records.** The DBMS itself keeps track of a number of current

records and set occurrences by means of a mechanism known as **currency indicators.** In addition, the host programming language needs local program variables to hold the records of different record types so that their contents can be manipulated by the host program. The set of these local variables in the program is usually referred to as the **user work area (UWA).** The UWA is a set of program variables, declared in the host program, to communicate the contents of individual records between the DBMS and the host program. For each record type in the database schema, a corresponding program variable with the same format must be declared in the program.

Currency Indicators

In the network DML, retrievals and updates are handled by moving or **navigating** through the database records; hence, keeping a trace of the search is critical. Currency indicators are a means of keeping track of the most recently accessed records and set occurrences by the DBMS. They play the role of position holders so that we may process new records starting from the ones most recently accessed until we retrieve all the records that contain the information we need. Each currency indicator can be thought of as a record pointer (or record address) that points to a single database record. In a network DBMS, several currency indicators are used:

• *Current of record type:* For each record type, the DBMS keeps track of the most recently accessed record of that record type. If no record has been accessed yet from that record type, the current record is undefined.

• *Current of set type:* For each set type in the schema, the DBMS keeps track of the most recently accessed set occurrence from the set type. The set occurrence is specified by a single record from that set, which is either the owner or one of the

member records. Hence, the current of set (or current set) points to a record, even though it is used to keep track of a set occurrence. If the program has not accessed any record from that set type, the current of set is undefined.

• *Current of run unit (CRU):* A run unit is a database access program that is executing (running) on the computer system. For each run unit, the CRU keeps track of the record most recently accessed by the program; this record can be from *any* record type in the database.

Each time a program executes a DML command, the currency indicators for the record types and set types affected by that command are updated by the DBMS.

Status Indicators

Several **status indicators** return an indication of success or failure after each DML command is executed. The program can check the values of these status indicators and take appropriate action—either to continue execution or to transfer to an error-handling routine. We call the main status variable DB_STATUS and assume that it is implicitly declared in the host program. After each DML command, the value of DB_STATUS indicates whether the command was successful or whether an error or an exception occurred. The most common exception that occurs is the **END_OF_SET (EOS)** exception.

10.5 Hierarchical Database Structures

10.5.1 Parent-Child Relationships and Hierarchical Schemas

The hierarchical model employs two main data structuring concepts: records and parentchild relationships. A **record** is a collection of **field values** that provide information on

an entity or a relationship instance. Records of the same type are grouped into **record types.** A record type is given a name, and its structure is defined by a collection of named **fields** or **data items.** Each field has a certain data type, such as integer, real, or string.
A **parent-child relationship type (PCR type)** is a 1:N relationship between two record types. The record type on the 1-side is called the **parent record type,** and the one on the N-side is called the **child record type** of the PCR type. An **occurrence** (or **instance**) of
17
the PCR type consists of *one record* of the parent record type and a number of records (zero or more) of the child record type.
A **hierarchical database schema** consists of a number of hierarchical schemas. Each **hierarchical schema** (or **hierarchy**) consists of a number of record types and PCR types.
A hierarchical schema is displayed as a **hierarchical diagram,** in which record type names are displayed in rectangular boxes and PCR types are displayed as lines connecting the parent record type to the child record type. Figure 10.9 shows a simple hierarchical diagram for a hierarchical schema with three record types and two PCR types. The record types are DEPARTMENT, EMPLOYEE, and PROJECT. Field names can be displayed under each record type name, as shown in Figure 10.9. In some diagrams, for brevity, we display only the record type names.

Figure 10.9 A hierarchical schema.

We refer to a PCR type in a hierarchical schema by listing the pair (parent record type, child record type) between parentheses. The two PCR types in Figure 12.1 are (DEPARTMENT, EMPLOYEE) and (DEPARTMENT, PROJECT). Notice that PCR types *do not* have a name in the hierarchical model. In Figure D.01 each *occurrence* of the (DEPARTMENT, EMPLOYEE) PCR type relates one department record to the records of the *many* (zero or more) employees who work in that department. An *occurrence* of the (DEPARTMENT, PROJECT) PCR type relates a department record to the records of projects controlled by that department. Figure 10.10 shows two PCR occurrences (or instances) for each of these two PCR types.
18

Figure10.10 Occurrences of Parent-Child Relationships.
(a) Two occurrences of the PCR type (DEPARTMENT, EMPLOYEE).
(b) Two occurrences of the PCR type (DEPARTMENT, PROJECT).

10.5.2 Properties of a Hierarchical Schema

A hierarchical schema of record types and PCR types must have the following properties:
1. One record type, called the **root** of the hierarchical schema, does not participate as a child record type in any PCR type.
2. Every record type except the root participates as a child record type in *exactly one* PCR type.
3. A record type can participate as parent record type in any number (zero or more) of PCR types.
4. A record type that does not participate as parent record type in any PCR type is called a **leaf** of the hierarchical schema.
5. If a record type participates as parent in more than one PCR type, then *its child record types are ordered.* The order is displayed, by convention, from left to right in a hierarchical diagram.

The definition of a hierarchical schema defines a **tree data structure.** In the terminology

of tree data structures, a record type corresponds to a **node** of the tree, and a PCR type corresponds to an **edge** (or **arc**) of the tree. We use the terms *node* and *record type,* and *edge* and *PCR type,* interchangeably. The usual convention of displaying a tree is slightly different from that used in hierarchical diagrams, in that each tree edge is shown separately from other edges (Figure 10.11). In hierarchical diagrams the convention is
19
that all edges emanating from the same parent node are joined together (as in Figure 10.9). We use this latter hierarchical diagram convention.
Figure 10.11 A tree representation of the hierarchical schema in Figure 10.9.
The preceding properties of a hierarchical schema mean that every node except the root has exactly one parent node. However, a node can have several child nodes, and in this case they are ordered from left to right. In Figure 10.9 EMPLOYEE is the first child of DEPARTMENT, and PROJECT is the second child. The previously identified properties also limit the types of relationships that can be represented in a hierarchical schema. In particular, M:N relationships between record types *cannot* be directly represented, because parent-child relationships are 1:N relationships, and a record type *cannot participate as child* in two or more distinct parent-child relationships.
An M:N relationship may be handled in the hierarchical model by allowing duplication of child record instances. For example, consider an M:N relationship between EMPLOYEE and PROJECT, where a project can have several employees working on it, and an employee can work on several projects. We can represent the relationship as a (PROJECT, EMPLOYEE) PCR type. In this case a record describing the same employee can be duplicated by appearing once under *each* project that the employee works for. Alternatively, we can represent the relationship as an (EMPLOYEE, PROJECT) PCR type, in which case project records may be duplicated
Example Consider the following instances of the EMPLOYEE:PROJECT relationship:
20

Project	**Employees Working on the Project**
A	E1, E3, E5
B	E2, E4, E6
C	E1, E4
D	E2, E3, E4, E5

If these instances are stored using the hierarchical schema (PROJECT, EMPLOYEE) (with PROJECT as the parent), there will be four occurrences of the (PROJECT, EMPLOYEE) PCR type—one for each project. The employee records for E1, E2, E3, and E5 will appear *twice each* as child records, however, because each of these employees works on two projects. The employee record for E4 will appear three times—once under each of projects B, C, and D and may have number of hours that E4 works on each project in the corresponding instance.
To avoid such duplication, a technique is used whereby several hierarchical schemas can be specified in the same hierarchical database schema. Relationships like the preceding PCR type can now be defined across different hierarchical schemas. This technique, called **virtual relationships,** causes a departure from the "strict" hierarchical model. We

discuss this technique in next section.

10.6 Integrity Constraints and Data Definition in the Hierarchical Model

10.6.1 Integrity Constraints in the Hierarchical Model

A number of built-in **inherent constraints** exist in the hierarchical model whenever we specify a hierarchical schema. These include the following constraints:

1. No record occurrences except root records can exist without being related to a parent record occurrence. This has the following implications:

a. A child record cannot be inserted unless it is linked to a parent record.

b. A child record may be deleted independently of its parent; however, deletion of a parent record automatically results in deletion of all its child and descendent records.

c. The above rules do not apply to virtual child records and virtual parent records.

2. If a child record has two or more parent records from the *same* record type, the child record must be duplicated once under each parent record.

3. A child record having two or more parent records of *different* record types can do so only by having at most one real parent, with all the others represented as virtual parents. IMS limits the number of virtual parents to one.

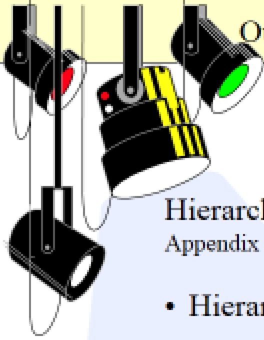
4. In IMS, a record type can be the virtual parent in *only one* VPCR type. That is, the number of virtual children can be only one per record type in IMS.

10.7 Summary

1. There are two basic data structures in the network model: records and sets.
2. Data is stored in **records;** each record consists of a group of related data values.
3. Records are classified into **record types,** where each record type describes the structure of a group of records that store the same type of information.
4. To represent relationships between records, the network model provides the modeling construct called *set type.*
5. Each set type definition consists of three basic elements: • A name for the set type. • An owner record type. • A member record type.
6. The constraints in network data model are usually divided into two main categories, called **insertion options** and **retention options** in CODASYL terminology.
7. A **data manipulation language (DML)** is used for inserting, deleting, and modifying records; and connecting and disconnecting records from set occurrences.
8. The hierarchical model employs two main data structuring concepts: records and parent-child relationships.

9. A **record** is a collection of **field values** that provide information on an entity or a relationship instance.
10. A **parent-child relationship type (PCR type)** is a 1:N relationship between two record types.

11. The definition of a hierarchical schema defines a **tree data structure.**
12. To a hierarchical schema, many **hierarchical occurrences,** also called **occurrence trees,** exist in the database. Each one is a **tree structure** whose root is a single record from the root record type.
13. An **occurrence tree** can be defined as the subtree of a record whose type is of the root record type.
14. A **hierarchical database occurrence** as a sequence of all the occurrence trees that are occurrences of a hierarchical schema.
15. A number of built-in **inherent constraints** exist in the hierarchical model whenever we specify a hierarchical schema.

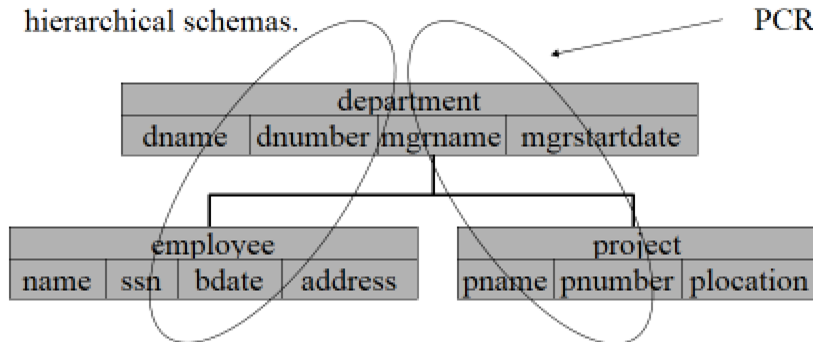


Hierarchical Database Management Systems

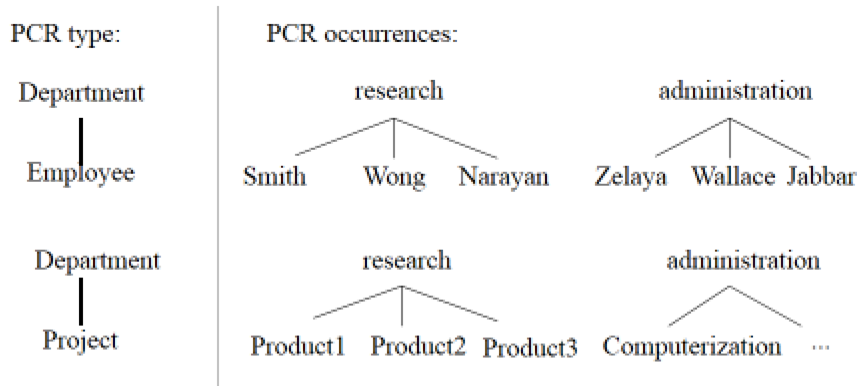
Appendix D – 3rd ed. (Appendix E – 4th ed. Appendix D – 5th and 6th ed.)

- Hierarchical Schemas
  - record type
  - parent-child relationship
  - hierarchical occurrence trees
  - linearized form of hierarchical occurrence
  - Virtual parent-child relationships
- Data definition in the hierarchical model
- Data manipulation language for the hierarchical model

- A hierarchical schema consists of record types and PCR types.
  - A record is a collection of field values.
  - Records of the same type are grouped into record types.
  - A PCR type (parent-child relationship type) is a 1:N relationship between two record types.
- A hierarchical database schema consists of a number of hierarchical schemas.



- PCR occurrence
  - Each PRC occurrence relates a record of a type (e.g., a department) to some records of another type (e.g., employee).

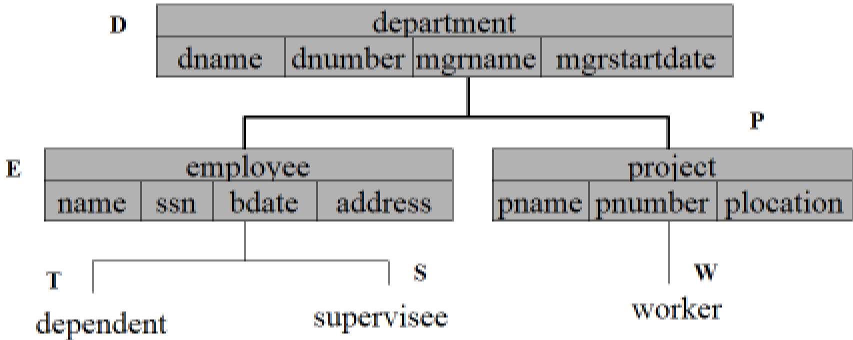


- Properties of a Hierarchical Schema

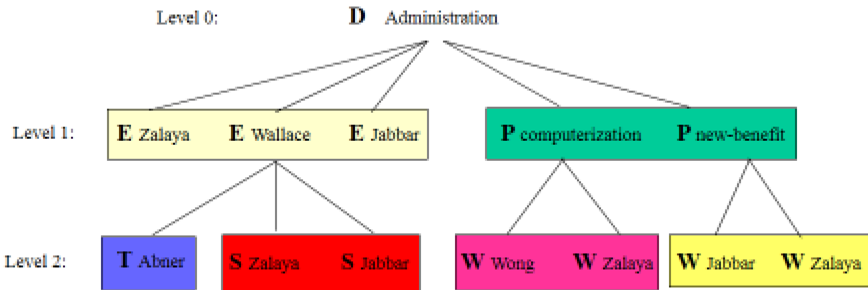
1. One record type, called the root of the hierarchical schema, does not not participate as a child record type in any PCR type.
2. Every record type except the root participants as a child record type in exactly one PCR type.
3. A record type can participate as parent record type in any number (zero or more) of PCR types.
4. A record type that does not participate as parent record type in any PCR is called a leaf of the hierarchical schema.
5. If a record type participate as parent in more than one PCR type, then its child record types are ordered. The order is displayed, by convention, from left to right in a hierarchical diagram.

- Hierarchical occurrence
  Each hierarchical occurrence, called an occurrence tree, is a tree structure whose root is a single record from some record type. Each subtree of the root is again a hierarchical occurrence.

  - type indicator



- hierarchical occurrence



- linearized form of a hierarchical occurrence

```
procedure Pre_order_traversal (root_record)
  begin
    output(root_record);
    if no child node then return;
    else for each child_record of root_record in left to right order do
            Pre_order_traversal (child_record)
  end
```

```
D  administration
E  Zelaya
E  Wallace
T  Abner
S  Zelaya
S  Jabbar
E  Jabbar
P  computerization
W  Wong
W  Zelaya
P  new-benefit
W  Jabbar
W  Zelaya
```

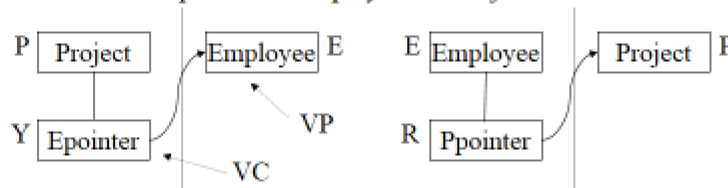

- Virtual Parent-child Relationships
  - Problems with hierarchical model
    1. M:N relationship
       causes redundancy
    2. The case where a record type participates as child in more than one PCR type
       causes redundancy
    3. N-ary relationships with more than two participating record type
       can not be modelled
  - Method dealing with the three problems:
    virtual record type
    virtual PCR relationship

- Virtual Parent-child Relationships
  - virtual record type

    A virtual (or pointer) record type VC is a record type with the property that each of its records contains a pointer to a record of another type VP.
    VC plays the role of "virtual child" and VP of 'virtual parent" in a "virtual parent-child relationship" (VPCR).

    A record of a VC type is a pointer to a record of some VP type.
    A record of a VP type is a "real" record.

    Example:
    M:N relationship between Employee and Project:



- Virtual Parent-child Relationships
  - intersection data in a virtual record

    An employee may participate in several projects. But for each project, he/she may work for different hours per week.
    Therefore, the data representing "different hours per week" should be included in the virtual records since each pointer to an employee record may have a different value. Such data are called intersection data.

  - VPCR

    The relationship between a virtual child and the corresponding virtual parent is called a Virtual Parent-Child-Relationship.

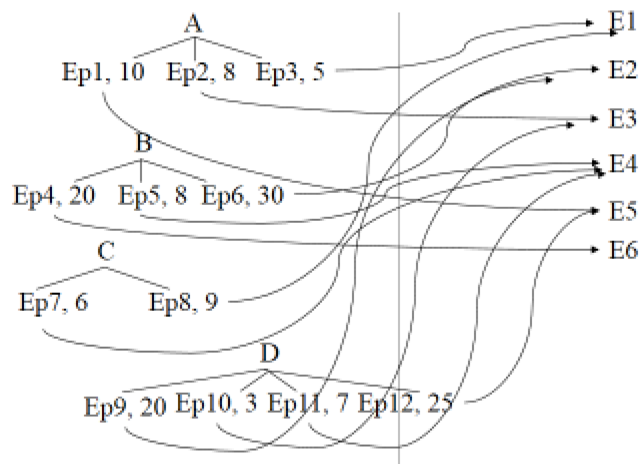
- Virtual Parent-child Relationships
  - Example:

    The relationship:

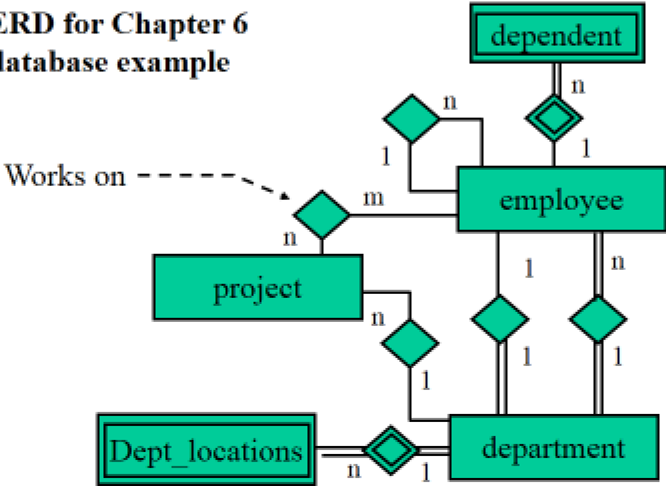
Project	Employees working on the project
A	E1, E3, E5
B	E2, E4, E6
C	E1, E4
D	E2, E3, E4, E5

can be stored as follows:

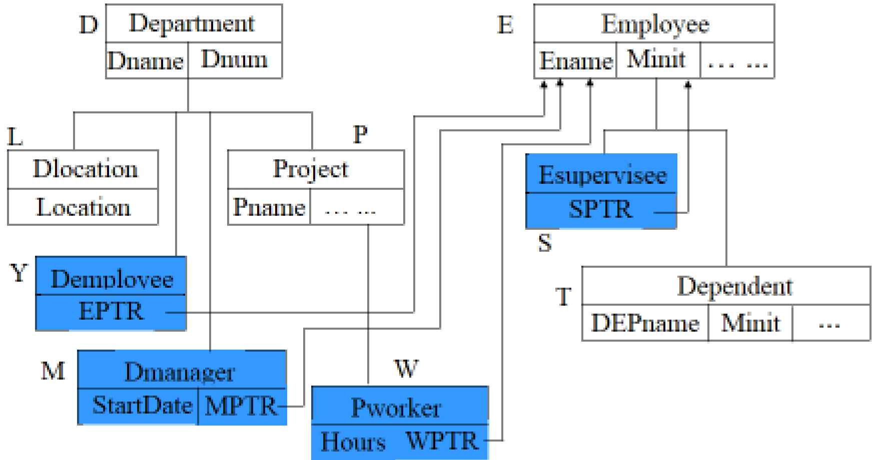
- Virtual Parent-child Relationships
  - Example:



ERD for Chapter 6 database example



- Virtual Parent-child Relationships
  - Hierarchical schema using VPCR - for a Company database



- Data Definition in the Hierarchical Model
  - Hierarchical data definition language (HDDL)
    - record type
      data item of a record type
      key clause
      parent
    - virtual record type
      virtual parent
    - CHILD NUMBER clause (the left-to-right order)
    - ORDER BY clause
      (the order of individual records of the same record type)
      sequence key

- Data Definition in the Hierarchical Model
  - Example

```
SCHEMA NAME = COMPANY

HIERARCHIES = HIERARCHY1, HIERARCHY2

RECORD
    NAME = EMPLOYEE
    TYPE = ROOT OF HIERARCHY2
    DATA ITEMS =
        FNAME       CHARACTER 15
        MINIT       CHARACTER 1
        LNAME       CHARACTER 15
        SSN         CHARACTER 9
        BDATE       CHARACTER 9
        ADDRESS     CHARACTER 30
```

- Data Definition in the Hierarchical Model
  - Example

```
        SEX             CHARACTER 1
        SALARY          CHARACTER 10
    KEY = SSN           CHARACTER 10
    ORDER BY LNAME, FNAME

RECORD
    NAME = DEPARTMENT
    TYPE = ROOT OF HIERARCHY1
    DATAITEMS =
        DNAME           CHARACTER 15
        DNUMBER         INTEGER
    KEY = DNAME
    KEY = DNUMBER
    ORDER BY DNAME
```


- Data Definition in the Hierarchical Model
  - Example

```
RECORD
    NAME = DLOCATION
    PARENT = DEPARTMENT
    CHILD NUMBER = 1
    DATA ITEMS =
        LOCATION        CHARACTER 15

RECORD
    NAME = DMANAGER
    PARENT = DEPARTMENT
    CHILD NUMBER = 3
    DATA ITEMS =
        MGRSTARTDATE         CHARACTER 9
        MPTR            POINTER WITH VIRTUAL PARENT = EMPLOYEE
```

- Data Definition in the Hierarchical Model
  - Example

```
RECORD
    NAME = PROJECT
    PARENT = DEPARTMENT
    CHILD NUMBER = 4
    DATA ITEMS =
        PNAME           CHARACTER 15
        PNUMBER         INTEGER
        PLOCATION       CHARACTER 15
    KEY = PNAME
    KEY = PNUMBER
    ORDER BY PNAME
```

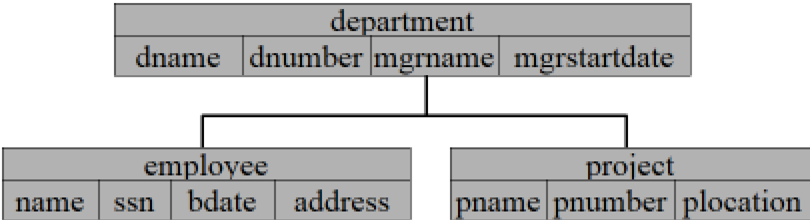

- Data Definition in the Hierarchical Model
  - Example

```
RECORD
    NAME = PWORKER
    PARENT = PROJECT
    CHILD NUMBER = 1
    DATA ITEMS =
        HOURS     CHARACTER 4
        WPTR      POINTER WITH VIRTUAL PARENT = EMLPOYEE

RECORD
    NAME = DEMPLOYEES
    PARENT = DEPARTMENT
    CHILD NUMBER = 2
    DATA ITEM =
        EPTR      POINTER WITH VIRTUAL PARENT = EMPLOYEE
```

- Data Manipulation in the Hierarchical Model
  - Update: INSERT, DELETE, REPLACE

```
$GET FIRST PATH DEPARTMENT, DEMPLOYEE
WHERE DNAME = 'Reseach';
 while DB_STATUS = 0 do
    begin
    $GET HOLD VIRTUAL PARENT EMPLOYEE OF DEMPLOYEES
    P_EMPLOYEE.SALARY := P_EMPLOYEE.SALARY * 1.1;
    $REPLACE EMPLOYEE FROM P_EMPLOYEE
    $GET NEXT DEMPLOYEES WHERE PARENT DEPARTMENT
    end;
```

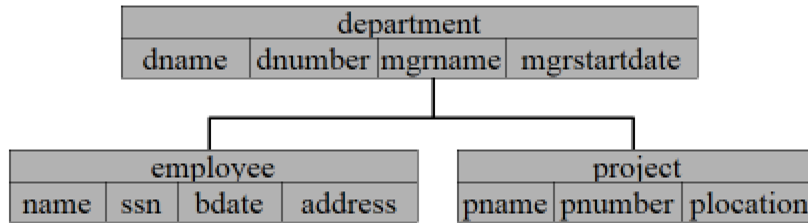



Two parent/child relationships are in the above schema:

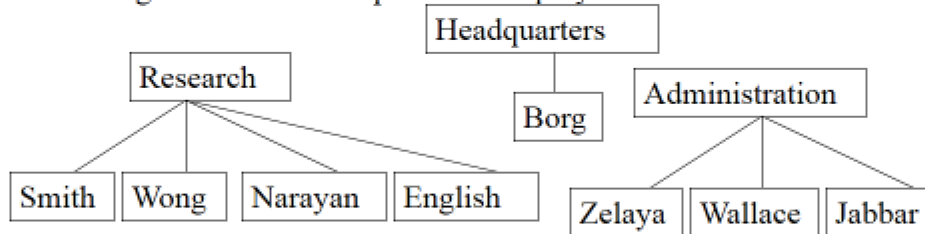
•department/employee

•department/project

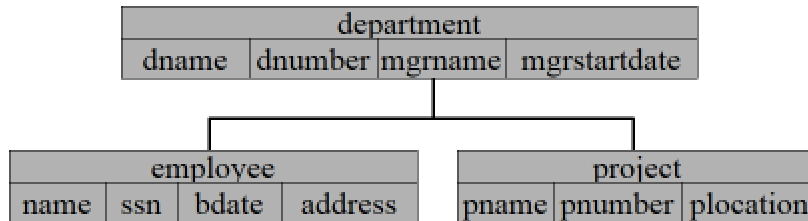
Parent/child instances — Hierarchical DBMS



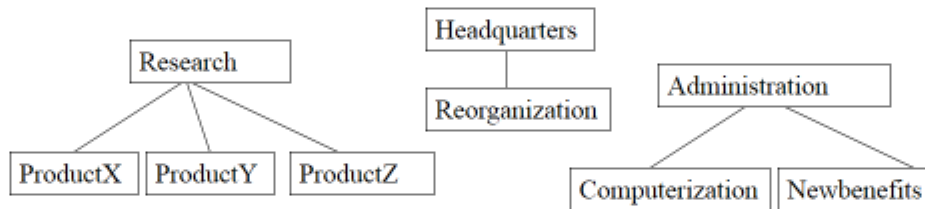
Using the data we had previously seen in Ch 7, we can depict the following 3 instances of department/employee:



Parent/child instances — Hierarchical DBMS

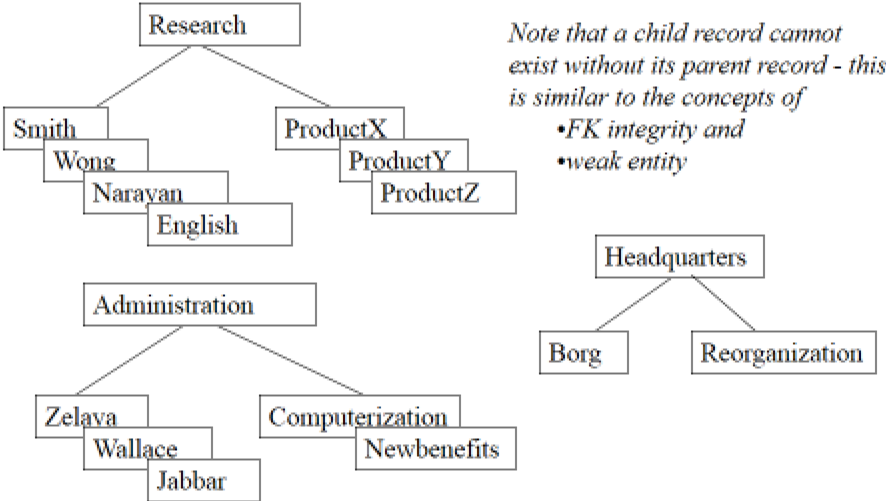


Using the data we had previously seen in Ch 7, we can depict the following 3 instances of department/project:

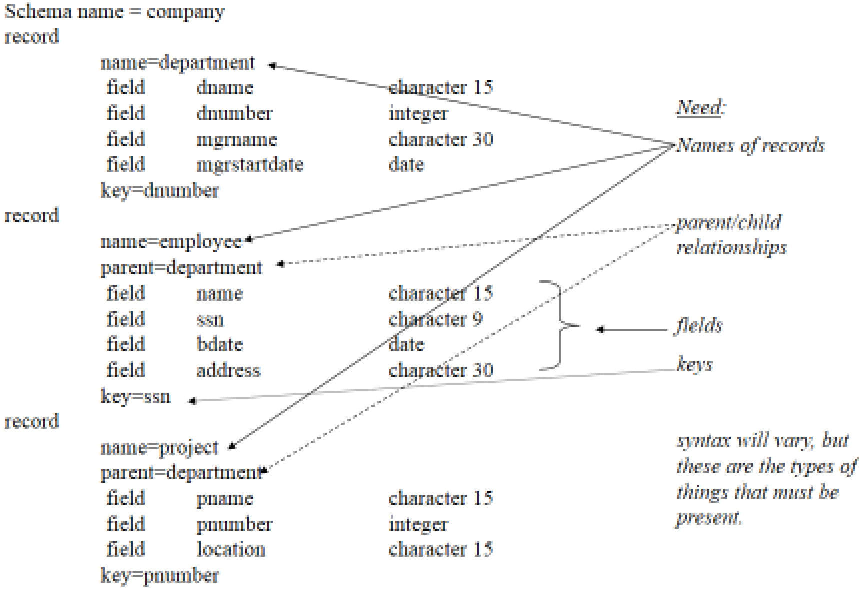


Hierarchical records | Hierarchical DBMS

In the following 3 hierarchical records are depicted. This is another way that such information is often depicted in practice.



Schema definition | Hierarchical DBMS



Data manipulation — Hierarchical DBMS

GU, GN, GNP, ISRT, DLET, REPL are IMS command names

Navigational - not set-oriented - you retrieve one record at a time

<u>Retrieval</u>

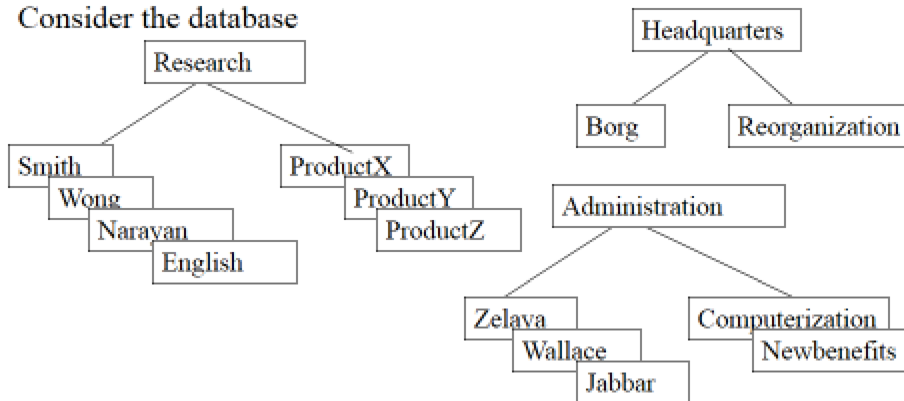
- GU, Get unique — retrieve a specific record
- GN, Get next — using your current position, get the next record in the database
- GNP, Get next within parent — using your current position, get the next child record for that parent

<u>Updating</u>

- ISRT, Insert
- DLET, Delete
- REPL, Replace

Navigating through the database — Hierarchical DBMS

Consider the database




```
GU Department (dname=headquarters)
Loop
        GNP
        exit when status code = ????
End Loop
```

Program would retrieve the Department record and all of its dependents