VI. INTRODUCTION TO DATA STRUCTURES
- A. General Concepts
- B. Elementary Data Concepts
- C. Other Basic Data Concepts
- D. Dense Lists Revisited
- E. Restricted (special types) Lists
  1. Stacks
  2. Queues
- F. Linked Lists

A. General Concepts

1. "A *data structure* is a named list of related data with one or more logical relationships existing among them."
   The files and the tables of Assignment 2 are examples of data structures.

2. "An *elementary data item* represents an element that cannot be split into components. It can be addressed and accessed by use of an identifier (e.g., a data-name)."
   Example: The lowest level field is an elementary data item.

B. Elementary Data Structure Concepts

1. LIST: "A one-dimensional structure of a collection of data objects, usually of the same type". (NOTE: Nothing has been said about how the list is constructed or stored.)

2. DENSE LIST: "The elements of a *dense list* are located in contiguous storage locations". That is, the elements are stored right next to one another. The files and the elements in our tables we have used so far are good examples of dense lists.

C. Other basic Data Concepts

We have been stressing that the data-names must reflect the nature of the data it represents. Thus, the data-name (if properly chosen) can be considered to be an attribute.

1. DATA: Value or values, but with no intrinsic meaning.

2. INFORMATION: Data with attributes and organization
   a. Used to describe objects or entities
   b. One or more attri*bute-value* pairs are needed to describe an entity.
   Thus, in our applications, the properly chosen data-name and value stored under that name represent information.

3. VECTORS.
   1. "A term which describes a one-dimensional data storage with contiguous storage locations". For example, the elements defined for a table in COBOL are contiguous – one after another in physical locations. NOTE: Nothing is implied about the organization or how any data list might be stored.
   2. A *vector* has a fixed number of elements, although all elements may not be utilized.
   3. A *vector* requires a positive integer (such as that contained in a subscript or in an index) the value of which corresponds to the relative position of the re-occurring element of the structure (i.e., the vector).

      We generally refer to such a vector as a HOST VECTOR in that it may be the "host" to one or more lists. That is, one or more lists may simultaneously stored in a *host vector*.

      The creation of a *host vector* in COBOL is accomplished by defining a one-dimensional table. In so doing, we are simply defining a set of storage locations where we may place one or more lists. We may not use all of the table elements, nor are we required to do so.

D. Dense Lists Revisited

1. General View

   Consider an ordinary sequential file similar to the ones we have used in the past. *All* access is restricted to physical order. Consequently, one must proceed through all preceding records to get to a desired record.

2. Advantages
   - Simple to create
   - Simple logic to search; simply move along the physical positions, reading one data element at a time, examining each one.
   - Efficient use of storage space; there are no empty data storage elements. It has a high "data density".
   - It needs no auxiliary data structure or special procedures to process it.

3. Disadvantages
   - Requires a relatively long time to search any sequential list. The average number of "looks" required when searching a sequential list (of any type) is equal to $(n + 1)/2$, where n is the number of items in the list.
   - If the dense list is ordered, it is difficult to add or delete data elements (records if a file) and to maintain an ordered list and to keep the list dense. Several steps are involved in processing adds or deletes.

4. Appropriate Use
   - Efficient processing when a large percentage of elements (e.g., records) are processed during a run.
   - If a list is ordered, the data elements/records may be processed in the natural order of the list/file.
   - Effective when adds and deletes are relatively infrequent, i.e., the data is relatively "non-volatile". (These operations are often handled by a separate operation.)

5. Sorting (ordering)

   The elements in the dense list are arranged in logical order by arranging them in physical order based on the values of a specific key. This will enhance processing efficiency. (NOTE: We have examined already a form of the exchange sort in the BUBLSORT program of Assignment 1.)

E. Restricted Forms of Lists – These allow only certain operations. (We will use dense lists in our examples, but the assignment will use linked list implementations.)

1. STACK

   All additions and deletions are performed at one end of the list. It is customary to call this the *top* or the *head* of the stack. A separate data item called a STACK POINTER contains the "address" of the top element of the *stack*.

   If we consider the *stack* to be in a table, the value of the STACK POINTER is used as a subscript. It contains the occurrence number (i.e., the location) of the element at the top of the stack. The value of this pointer will change with the addition and with the deletion of an element.

   The operation to place a new element on the *stack* is called a PUSH, just as one PUSHes a plate on to the top of the stack of plates in a buffet. The deletion of an element is called a POP. These are the only two operations allowed on a sta*ck*.

   These stack operations can be summarized by Last In – First Out.

2. QUEUES

   A list for which the removal of an element takes place only at the top or the head of the list, and the addition of an element takes place only at the bottom or the tail of the list. Consequently, we need a "pointer" to hold the address (i.e., the location) of the element at the "head" of the *queue*, and another pointer the hold the address of the element at the "tail" of the *queue*. These are often have names like Q-HEAD-PTR and

Q-TAIL-POINTER.

We experience *queues* when we wait in line for the check-out stand at the grocery store, or waiting in line at the bank. In England, they use the phrase to "*queue* up", meaning to get in line.

These are the only operations allowed for a *queue*. They may be characterized as First In – First Out.

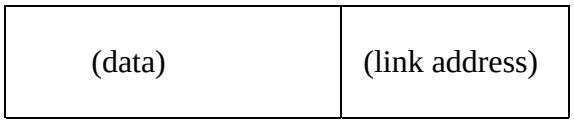
Note that, with both the *stack* and the *queue*, the data in these respective lists are not ordered. "Ordered-ness" is rarely a characteristic of these "restricted" lists.

F. LINKED LISTS

1. "No other logical data structure is more important than a linked list because most other logical data structures use extensions of the *linked list* (structure)" (from Ellzey).

   The *linked list* does not have the physical order of a dense list. It is logically connected, but not physically connected by adjacent, physical positions. (A *dense list* is logically connected because of the physical arrangement of the elements.)

   In a *linked list*, the data elements are "linked" from one to another. This is accomplished by providing in the list element the "address" or location of the next logical element.



   Consequently, all the elements belonging to the *linked list* are, conceptually, chained together. The connecting links are the addresses in each element.

   A separate data-name is required to store the address (location) of the first element of the *linked list*. It is frequently given a data-name similar to LIST-HD-PTR.

   Since a *linked list* is still a list, most dense list structures (e.g., data lists, stacks, queues) and processes (e.g., searches, adds, deletes) can be effectively implemented with l*inked lists*.

In a *linked list*, the physical order is not the logical order

When one moves from one element to the next in a *linked list*, one uses the link address contained in the current item to determine the location of the next logical item. (With a dense list, one simply "moves" to or reads the next item/record that is located in the next physical location.)