A CLASSIFICATION OF DATA TYPES

Nell B. Dale and Henry M. Walker*

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-90-17 June 1990

---

* Department of Mathematics, Grinnell College.

A Classification of Data Types

by *Nell Dale*
Department of Computer Sciences
The University of Texas at Austin

and *Henry M. Walker*[1]
Department of Mathematics
Grinnell College

Abstract

There is considerable variation in the terminology that is used in discussing the subject of (abstract) data types. Further, discussions of individual data types often combine several types into unnecessarily complex or interlinked structures and sometimes refer to a single data type in inconsistent ways. This paper resolves many of these problems by proposing a unified classification of a wide range of data types.

Introduction: The authors taught parallel sections of a junior level course entitled Data Structures (CS7 in *Curriculum '78*) last year. After discussing the topic for hours on end, and, yes, even arguing for hours, we came to two conclusions: Data Structures is an inadequate and misleading title and the traditional paradigm used to present the material needs reorganizing.

Data structures refers to the study of data and how to represent data objects within a program; that is, the implementation of structured relationships. Over the last ten years the focus has broadened considerably. We are now interested in the study of the abstract properties of classes of objects in addition to how these objects might be represented in a program. Johannes J. Martin puts in very succinctly: "... depending on the point of view, a data object is characterized by its type (for the user) or by its structure (for the implementer)."[2]

The topic of Data Structures has now been subsumed under the broader topic of Abstract Data Types (ADTs): the study of classes of objects whose logical behavior is defined by a set of values and a set of operations.

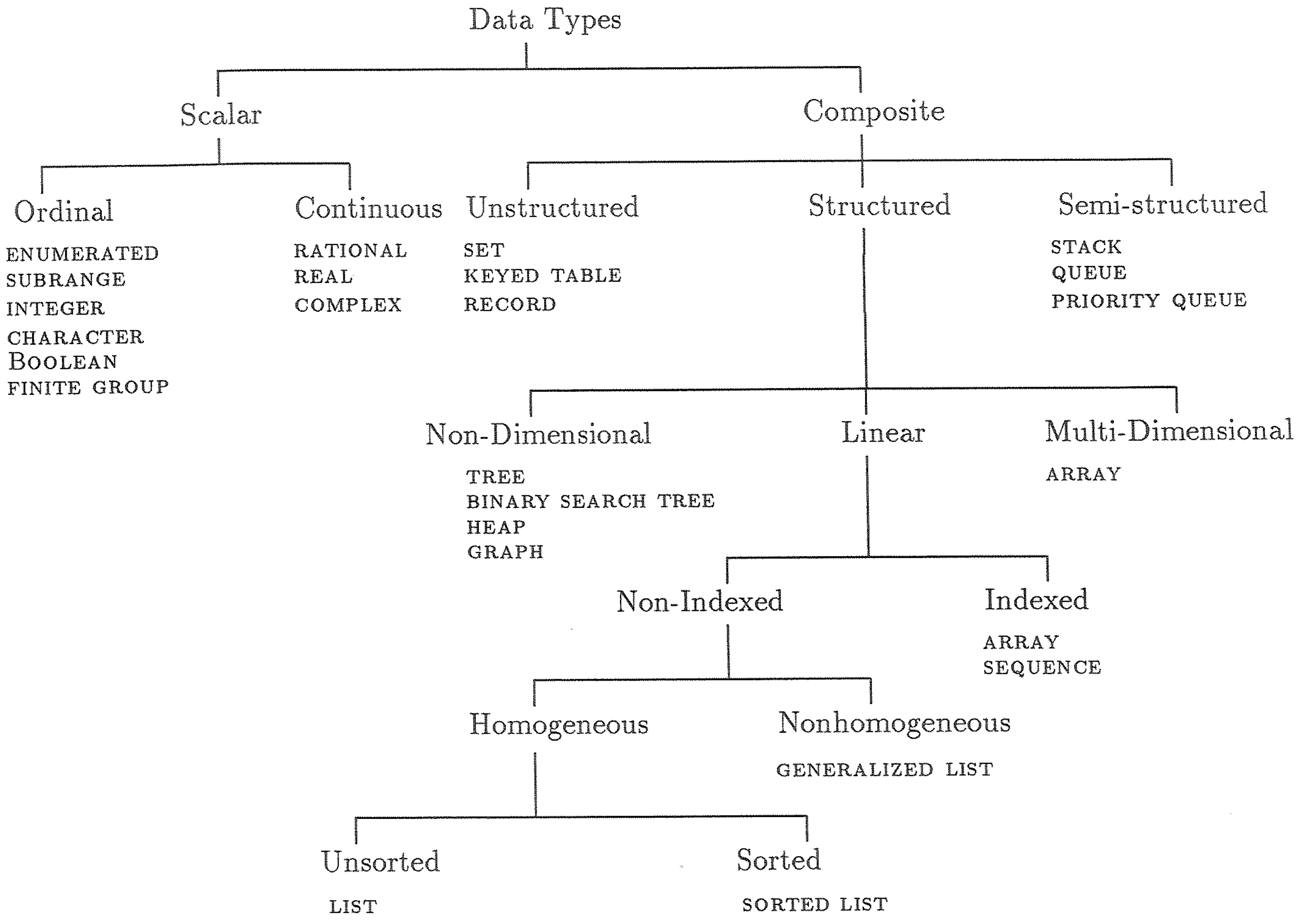
The traditional paradigm for studying Data Structures is based on characteristics of the implementation of the structures. For example, a stack and a queue are classified as restrictive versions of a list where access is limited to one or both ends of the list. The properties of a stack and a queue can certainly be represented this way. However, the *user* does not care about ends and restricted access. In fact the user does not care what happens when an item is stored in a stack or a queue; the user is only interested in what is returned on a pop or a dequeue: the item that is returned is the last one inserted in the case of a stack or the first one inserted in the case of a queue.

---

[1] During much of the discussion that led to this paper, the author was on sabbatical leave from Grinnell College and was working as a Senior Lecturer in the Department of Computer Sciences, The University of Texas at Austin.

[2] Johannes J. Martin, *Data Types and Data Structures*, Prentice-Hall International Series in Computer Science, C. A. R. Hoare, Series Editor.

Figure 1: Classification Tree



The study of ADTs requires that we step back and view data types from the *functional view of the user*.[3] The classification proposed in this paper is based on such a view. Section 1 describes this classification in words, following a top-down organization. Then, within this hierarchy, Section 2 uses axiomatic specifications to define each data type. This specification clarifies individual operations for each data type and formalizes the similarities and differences between data types. Overall, this hierarchical, axiomatic specification of data types provides a framework to review various traditional classifications of data structures.

---

[3] While it is common to include the word *abstract* in the phrase "abstract data type", the perception of abstraction is often in the eye of the beholder. For example, an integer data type often is based on the mathematical notion of these numbers, using operations such as addition, subtraction, multiplication, and division. However, even for integers, choices may be made for implementation. Integers may be limited to a specific range, or storage may be allocated dynamically to accommodate integers of any size. In this paper, therefore, we frequently drop the term "abstract", but bear in mind that we are always referring to the logical properties of a data type.

Table 1: Summary of Data Type Specifications

- **Data Types:** Data objects together with specified operations.
  - **Scalar:** Base type involves a single, elementary data object.
    - **Ordinal:** Data values are ordered and discrete.
      - *Operations:* Pred, Succ, Relational Operators $(<, \leq, >, \geq, =, \neq)$, Arithmetic
    - **Continuous:** Data values are not discrete.
      - *Operations:* Relational Operators $(<, \leq, >, \geq, =, \neq)$, Arithmetic
  - **Composite:** Data type combines one or more elementary data objects.
    - **Unstructured:** No relationship or ordering is specified or implied among objects.
      - *Operations:* Store, IsThere, Delete, Find
    - **Structured:** An explicit relationship is specified among objects.
      - **Non-Dimensional:** No linear order or indexing is specified or implied; the data may or may not be partially ordered.
      - **Linear:** A linear or total ordering is specified or implied.
        - **Non-Indexed:** No index variable or value is available.
          - *Operations:* Store, Delete, GetFirst, GetNext
          - **Homogeneous:** All data objects held within the structure have the same type.
            - **Unsorted:** The linear ordering specification is external to the data; the ordering may be supplied by the user or programmer.
            - **Sorted:** The linear ordering may be inferred by the data itself.
          - **Nonhomogeneous:** Individual data objects that make the structure may have different types.
        - **Indexed:** A single index variable or value is given or implied.
          - **Array:** A linear stream of data of bounded length.
            - *Operations:* Store, Retrieve
          - **Sequence:** A linear data stream whose length is not bounded.
            - *Operations:* Store, Insert, Delete
      - **Multi-Dimensional:** Indexing requires multiple variables or values.
        - *Operations:* Store, Retrieve
    - **Semi-Structured:** A relationship is implied, but not stated among objects.
      - *Operations:* Store, Look, GetAnother (destructive)

Section 1: A Description of Data Types

This section organizes many standard data types by identifying several properties that distinguish these data types from each other. These properties then are used to develop a unified, hierarchical classification for a wide range of data types. The results of this classification are shown graphically in Figure 1. Table 1 presents a parallel summary of the classification scheme using tabular form.

Data Types $\Longrightarrow$ Scalar + Composite

At a top level, data types may be divided into two major categories, those involving individual data values and those which combine several values within a single object. The first of these is called the *scalar data type*, and normally the operations on this type allow the combining of two values into a third or the modifying of one value to yield another. In contrast, data types which combine several values are called *composite data types*, and the operations on such composite types normally may be defined in terms of primitive *store, retrieve* and/or *delete* operations.

Scalar $\Longrightarrow$ Ordinal + Continuous

Scalar data type in turn may be subdivided into two types, *ordinal data types* and *continuous data types*. In ordinal data types, data models view the data values as being separate, discrete objects. Data values within continuous data types, on the other hand, are viewed as being part of a continuous, connected region of a line, plane, or n-dimensional space.

Ordinal Data Type involves discrete values, which often come with a total ordering. Such types are built into many standard programming languages and include *enumerated types*, *integers*, *characters*, *Boolean type* and *subranges* of these types. While all of these traditional types are ordered in a natural way, this scalar type could also include finite groups which do not have such a natural ordering. (For finite groups, however, an ordering can be imposed on the finite values, merely by chosing one value to be smallest, another value next smallest, etc.) For each of these cases, typical operations take advantage of this ordering and include *predecessor* and *successor* operations and the relational operations $(<, \leq, >, \geq, =, \neq)$. Since many ordinal data types involve numbers, it is common for these types also to include various arithmetic operations, such as addition, subtraction, multiplication, and division.

Continuous Data Type includes values such as the *rational numbers*, the *real numbers*, and the *complex numbers*. In each case, the values are not discrete, and it is inappropriate to ask what value immediately precedes another. (For example, it is hard to say what real or rational number immediately precedes 1/2 .) Continuous data types normally involve numbers, and operations customarily include the arithmetic functions addition, subtraction, multiplication, and division. Some of these types also are ordered, in which case relational operations may be defined.

Composite $\Longrightarrow$ Unstructured + Structured + Semi-Structured

Composite data types involve the combining of several individual objects into a larger whole. Further, each of these composite types depends upon the fundamental notions of storing, retrieving, and deleting objects from the composite structures.

In some cases, the combining of objects imposes additional structure on the objects. For example, the objects may be placed in an order, the objects may be indexed, or various relationships among data objects may be specified. Such types may be called *structured data types*.

At the other extreme, no relationships may be stated or implied among the various objects, and such data types are called *unstructured*.

At an intermediate level, some data types place some restrictions on the order in which data may be stored or retrieved. Such data types may be called *semi-structured*.

Unstructured Data Types place no constraints on the data objects being stored or retrieved, except that these objects may be required to all be from the same underlying type. Further, no relationship among objects is stated or implied. Such data types includes *sets*, *keyed tables* (sometimes called *symbol tables*), and *records*. Operations for these unstructured data types often are limited to simple statements to *store* data, to *retrieve* an object (without removing it from the structure), to *delete* an object, and to determine if an object is present (the Boolean *IsThere* operation.)

A **Semi-Structured Data Type** implicitly specifies which data object is accessed whenever a retrieve or delete operation is executed. For example, this implicit specification may involve the time when objects were stored, or it may depend upon the order of store operations, although factors other than time or order may be used. These data types include *stacks*, *(FIFO) queues*, and *priority queues*. Operations for semi-structured data types always involve *store* and *delete* capabilities, although these operations often are given different names. These data types also provide a *Empty* operation to test if any data are presently stored. In some cases, provision also is made to examine some or all of the data objects being stored.

Structured $\Longrightarrow$ Non-Dimensional + Linear + Multi-Dimensional

Structured data types all impose some additional connections or structure upon individual data objects. In a *non-dimensional data type*, the connections between objects are given explicitly for various ordered pairs of objects.

In other cases, the structure is imposed by considering one or more variables or by utilizing a natural ordering of data values. A structure based upon one variable or upon a total ordering of the objects in the structure is called a *linear data type*, while a structure based on two or more variables or values is said to be *multi-dimensional*.

A **Non-Dimensional, Structured Data Type** organizes data types by requiring each application to specify relations between objects, either explicitly or implicitly. In a *tree data type*, this relationship follows a strict hierarchy. In a *search tree*, the order of individual data objects implies a further structure. In a *heap*, additional structure is given both by the order of individual data objects and by the overall shape of the tree structure (although there can be some debate about whether shape should be an axiomatic issue or a matter of implementation.) In a *graph data type*, all relationships between pairs of data objects must be stated explicitly. In any of these objects, a total ordering of the data objects in the structure is neither stated nor implied. Operations for nondimensional data types include means to *store*, *retrieve*, and *delete* individual data objects and to specify appropriate connections among objects. Additional operations are present to move from one object to another within the structure.

A **Multi-Dimensional Data Type** uses two or more specified variables to index individual objects. *Multi-dimensional arrays* fall into this category. Operations for this data type often are limited to a simple *store* and *retrieve* capability.

Linear $\Longrightarrow$ Non-Indexed + Indexed

The data objects in a linear data type come with a total ordering which may be either specified explicitly or implied. In an *indexed data type*, this ordering is specified explicitly by a given variable, and the value of this variable serves as the mechanism to refer to each object. For example, data objects may be referenced by a subscript in the notation $A_1, A_2, A_3, A_4, \ldots$. In a *non-indexed data type*, the ordering of data objects may be inferred from the data themselves, but storage and retrieval need not depend upon a separate index variable.

An **Indexed Data Type** uses a single, specified variable to index all data objects in the structure. If a fixed amount of space for this structure is declared initially and a store to one position has no effect on the data elsewhere in the structure, then the structure is called an *array*. Otherwise, the amount of space may be considered as being unlimited or a store in one position *position i* may change indexing of later items (data item a_{i+1} is relabeled a_{i+2}, for example). In such case, the structure is called a *sequence*.

Non-Indexed $\Longrightarrow$ Nonhomogeneous + Homogeneous

A non-indexed, linear data type contains data that have been ordered. However, the ordering may be imposed separately by a user or programmer or it may be due to a known ordering of the data objects themselves. At one extreme, the data type may be used to store objects of differing individual types, and the resulting composite type is called *nonhomogeneous*. In these instances, individual data objects cannot be compared with each other, and any sequencing of these objects must be done by a user or programmer. In contrast, in a *homogeneous* data type, all individual objects that make up the structure have the same type.

With either type of structure, it should be noted that the ordering may be used to index various objects, giving index 1 (or 0) to the first item, index 2 (or 1) to the next, and so forth. However, a user or programmer cannot necessarily refer to the n^{th} item unless it is already known that at least that many objects have been stored. A non-indexed data type does not have pre-defined storage areas for various objects. Thus, it is inappropriate to consider an operation such as *Store(Object, Position)* as being a primitive operation, since the given position may not be defined at any given time. Indexing may be inferred as a side effect of the linear ordering of data, but the indexing is not a fundamental part of the data type.

A **Nonhomogeneous Data Type** consists of a linear sequence of data objects, and these individual objects may come from several different data types. In this situation, any *store* operation must specify where a new object is to be placed. Other operations may include a means to *delete* an object, to find the first object *GetFirst*, to move from one object to the next *GetNext*, and to determine the type of a specified object *GetType*. This data type includes the *generalized list data type*, when data objects may be drawn from either a single underlying type or from generalized lists of such objects.

Homogeneous $\Longrightarrow$ Unsorted + Sorted

A homogeneous data type involves a linear ordering of data. In this situation, it is possible that the ordering may be inferred by the data objects themselves, the structure is said to be *sorted*. If instead the sequencing of data objects is imposed only by the creation of the data structure (by a user or a programmer), then the data type is said to be *unsorted*.

An **Unsorted Data Type** requires the user or programmer to specify where each subsequent data object will be placed within the structure. For example, it might be specified that a new data object be placed first or last in the structure or the object might be placed before or after a specified object. Thus, for these structures, a simple *store* operation places a object at the beginning of the list. Other operations may include a means to *delete* an object, to find the *length* of the list, to obtain the first (or *head*) of the list, or to return the *tail* of the list (the list with the head deleted). This unordered data type category includes the *list data type*, when all data objects are drawn from a single underlying type.

An **Sorted Data Type** depends upon an ordering inherent in the data objects themselves. In such instances, it is customary to refer to the data in the structure as being *sorted*. Here, the first object in the structure is the one that happens to be the minimum (or maximum) of the objects stored. The second object is the second smallest (or second largest) object being stored. Such an object is often called a *sorted list*. Operations for such structures involve *store* or *insert*, *delete*, find the first object on the list (*head*), and obtain the list with the first object deleted *Tail*).

Section 2: The Axiomatic Specification of Data Types

This section presents a formal definition of each composite data type presented in Section 1, expanding the general description given in the previous section. This axiomatic specification of data types complements and clarifies the hierarchical, verbal description given earlier.

Notes on Axiomatic Specifications

In an axiomatic specification, the abstract data type being defined is called the *designated domain* or the *carrier domain*. The other data types involved in the operations on the carrier domain are called *auxiliary domains*. There are four types of operations that can be defined on an abstract data type: *primitive constructors, constructors, observers,* and *iterators*.

Primitive constructors return an instance of the carrier domain without taking one as input. That is, a primitive constructor creates a new instance of the abstract data type. This new instances is either empty or has no defined values stored within it. For obvious reasons, these operations are often called *Create*. *Create* may be parameterless or it make take types as parameters that set certain bounds on instances of the type.

Constructors take objects of the carrier domain as input and return objects of the carrier domain. *Observers* are operators that take an instance of the carrier domain and return results of a different type. That is, they observe the instance without changing it.

Iterators are operations that allow the user to view the items stored in an instance of the carrier domain. In some data types, separate iterators are not necessary; in some data types, iterators have no meaning; in some data types iterators must be explicitly defined using an additional auxiliary data type to hold the items to be viewed.

Exceptions

We need a way to express two situations that produce exceptions which come up in our specifications of data types. The first situation is where the application of an operation causes an error. Trying to pop a stack when the stack is empty is such a situation. The second situation occurs in cases where the *Create* operation takes a parameter that represents size. Here, we view the operation as creating the shell or structure where places for all future values exist from the beginning rather than creating an empty structure which grows and shrinks as values are stored into it. When a retrieval operation is applied to a place where no value has been stored, an exception has occurred.

To handle these two situations, we introduce two constants, *Error* and *Undefined*, which are a part of every data type. *Error* is returned when the application of the operation causes an error. *Undefined* is returned when the slot in a structure exists, but the contents of a slot have not been given a value.

UNSTRUCTURED DATA TYPES

From a user's view point, an *unstructured data type* is a composite data type where items are put into a collection of items and retrieved from that collection of items. There is no relationship among the items in the composite structure other than that they reside in the same structure. The *set* is the classic unstructured data type. There are, however, three other common data types that also belong in this category: the *bag*, the *keyed table* (sometimes called a *symbol table*), and the *record*.

Set

The user of the abstract data type *set* expects it to model the mathematical data type set. An empty set is created and items of the component type are inserted into the set and deleted from the set. The only parameters needed for either operation are the set and the item. Because the binary operations *Difference*, *Union*, and *Intersection* are so commonly used, they have been included in the set of functions.

The names that we choose for the operations are, for the most part, obvious. *Store* is the exception: We usually think of inserting an item into a set. We choose to use the verb *Store* here to be consistent with its use in later data types. (*Insert* is reserved for a second operator that adds an item to an abstract data but has a side effect of altering the position of other items within the structure.)

```
structure  Set (of Item)
interface  Create ⟶ Set
           Store(Set, Item) ⟶ Set
           IsEmpty(Set) ⟶ Boolean
           Card(Set) ⟶ Integer
           IsIn(Set, Item) ⟶ Boolean
           Delete(Set, Item) ⟶ Set
           Difference(Set, Set) ⟶ Set
           Union(Set, Set) ⟶ Set
           Intersection(Set, Set) ⟶ Set
      end
   axioms  for i1, i2 in Item, S, T in Set, let
           Store(S, i1) =
                 IF IsIn(S, i1)
                    THEN  S
           IsEmpty(Create) = True
           IsEmpty(Store(S, i1)) = False
           Card(Create) = 0
           Card(Store(S, i1)) = 1 + Card(S)
           IsIn(Create, i1) = False
           IsIn(Store(S, i2), i1) =
                  IF i1 = i2
                     THEN  True
                     ELSE  IsIn(S, i1)
           Delete(Create, i1) = Create
           Delete(Store(S, i2), i1) =
                  IF i1 = i2
                     THEN  S
                     ELSE  Store(Delete(S, i1), i2)
```

```
        Difference(Create, T) = Create
        Difference(Store(S, i1), T) =
               IF IsIn(T, i1)
                  THEN  Difference(S, T)
                  ELSE  Store(Difference(S, T), i1)
        Union(Create, T) = T
        Union(Store(S, i1), T) = Store(Union(S, T), i1)
        Intersection(Create, T) = Create
        Intersection(Store(S, i1), T) =
               IF IsIn(T, i1)
                  THEN  Store(Intersection(S, T), i1)
                  ELSE  Intersection(S, T)
   end
```

A set does not have duplicates. This is represented in the constraint on the *Store* operation. The *Store* does nothing if the item is already in the set. This property could have been expressed in another way: We could have put no constraints on *Store* and let *Delete* take care of removing all the extra duplicates. *Union* and *Card* would have to be altered accordingly.

```
Delete(Create, a) = Create
Delete(Store(S, i2), i1) =
       IF i1 = i2
          THEN  Delete(S, i1)
          ELSE  Store(Delete(S, i1), i2)
Union(Create, T) = T
Union(Store(S, i1), T) =
       IF IsIn(T, i1)
          THEN  Union(S, T)
          ELSE  Store(Union(S, T), i1)
Card(Create) = 0
Card(Store(S, i1)) = 1 + Card(Delete(S, i1))
```

Since there is no structure in a set that we can use to allow us to move from one element to another in order to view each one, we must define an iterator which takes each of the items in the collection and puts them in an unsorted, non-indexed, linear list. Once in the list, we can view each item in turn. The operations on unsorted, non-indexed, linear lists will be defined in a later section. The constructor operation which takes a list and an item and returns a list is the *Make* operation.

Iterator for ADT Set: Augment Domain with ADT UnsortedList

```
Members(Set) ⟶ List
Members(Create) = Create
Members(Store(S, i1)) = Make(Members(S), i1)
```

Notice that *Create* is used twice in the same axiom. There is no ambiguity, however, because the context determines which empty structure is being defined. The *Members* operation takes a parameter of type *Set*, so the *Create* on the left of the equal sign refers to an empty set. Members returns a parameter of type *UnsortedList*, so the *Create* on the right of the equal sign refers to the empty unsorted list.

If we use the second set of axioms where there is no constraint on *Store*, the general axiom for *Members* would be

```
Members(Store(S, i1)) = Make(Members(Delete(S,i1)), i1)
```

Bag

A *Bag* is a counted set. That is, duplicate items are allowed in the set. The axioms for a *Bag* are identical to the first axioms defined on the set data type with the constraint on the *Store* removed. We will not repeat the axioms.

Record

The *Record Data Type* is a non-homogeneous data type where $< field, value >$ pairs are stored. The *Create* operation takes a set of fields and returns the structure with the fields defined but the associated values undefined.

```
structure  Record (of <Field, Value>)
interface  Create(Set of Field) ⟶ Record
           Store(Record, Field, Value) ⟶ Record
           Find(Record, Field) ⟶ Value
      end
   axioms  for fields ⊆ Set of Field,  f1, f2 in fields, v in Value,
               R in Record, let
           Find(Create(fields), f1) = Undefined
           Find(Store(R, f2, v), f1) =
                  IF f1 = f2
                     THEN  v
                     ELSE  Find(R, f1)
      end
```

Since the constants in the fields are known in advanced, there is no need to define an iterator. The same is true for a set if the component type is limited in size. However, because this is not the general case we included an iterator.

Keyed Tables

In the abstract data type *keyed table*, the items in the unstructured composite are $< name, value >$ pairs. The pairs are inserted into the collection, removed by specifying the name, and the collection is searched for the value associated with a name. There are no common binary operations applied to keyed tables.

In one sense the *keyed table* is like the *record*: The items contained in the collection are made up of pairs where the first one of the pair is used to identify and retrieve the second. The difference between a *record* and a *keyed table* is that the fields in a *record* come from a closed, predefined type. The *Create* operation creates a structure where the fields exist waiting for the associated values to be stored. The names in a $< name, value >$ pair may or may not be known in advanced. The *Create* operation is not parameterized; it returns an empty collection.

Keyed Table

```
structure   KeyedTable (of <Name, Value>)
interface   Create ⟶ KeyedTable
            Store(KeyedTable, Name, Value) ⟶ KeyedTable
            Delete(KeyedTable, Name) ⟶ KeyedTable
            Find(KeyedTable, Name) ⟶ Value
            IsIn(KeyedTable, Name) ⟶ Boolean
            IsEmpty(KeyedTable) ⟶ Boolean
      end
   axioms   for n1, n2 in Name, v in Value, KT in KeyedTable, let
            Delete(Create, n1) = Create
            Delete(Store(KT, n2, v), n1) =
                   IF n1 = n2
                      THEN  Delete(KT, n1)
                      ELSE  Store(Delete(KT, n1), n2, v)
            Find(Create, n1) = Error
            Find(Store(KT, n2, v), n1) =
                   IF n1 = n2
                      THEN  v
                      ELSE  Find(KT, n1)
            IsIn(Create, n1) = False
            IsIn(Store(KT, n2, v), n1) =
                   IF n1 = n2
                      THEN  True
                      ELSE  IsIn(KT, n1)
            IsEmpty(Create) = True
            IsEmpty(Store(KT, n1, v)) = False
      end
```

The axioms state that trying to find a $< name, value >$ pair when no pair has been stored with that name is an error condition.

In the following iterator definition we use the data type *SortedList* to hold the names to impose an order on the way in which the names will be viewed.

Iterator for ADT KeyedTable: Augment Domains with ADT SortedList

```
ListOfNames(KeyedTable) ⟶ SortedList
ListOfNames(Create) = Create
ListOfNames(Store(KT, n1, v))
         = Insert(ListOfNames(KT), n1)
```

Summary of Unstructured Types

The *unstructured composite data types* can be characterized by patterns in their axioms: there is only one constructor operation that increases the number of elements in the collection and one operation that reduces the number of elements in the collection. The operation that deletes an item from the collection takes two parameters: the name of the structure and a parameter that specifies which item to remove, the item, a field, or a name.

One of the types, the *record*, is a fixed size. The *Create* operation takes a parameter that determines the size. In such a case, the *IsEmpty* operation has no meaning. The equivalent operation would be *IsUndefined* which would return true if all of the fields were undefined. However, this does not seem to be a particularly useful operation.

SEMI-STRUCTURED DATA TYPES

From the user's view point, a *semi-structured data type* is a collection of items which has a special or designated item but no logical relationship exists among the rest of the items in the collection. The operation that deletes an item or views an item knows that it is this designated item that is to be deleted or returned. The *stack*, the *FIFO queue*, and *priority queue* fall into this category.

Stack

In the *stack*, the designated item is the last item that was put into the collection. The axioms for the *stack* are used frequently to describe the axiomatic approach to specifying the behavior of an abstract data type. The operations even have their very own names.

```
structure  Stack (of Item)
interface  Create ⟶ Stack
           Push(Stack, Item) ⟶ Stack
           Pop(Stack) ⟶ Stack
           Top(Stack) ⟶ Item
           IsEmpty(Stack) ⟶ Boolean
      end
   axioms  for all S in Stack, i in Item, let
           Pop(Create) = Error
           Pop(Push(S, i)) = S
           Top(Create) = Error
           Top(Push(S, i)) = i
           IsEmpty(Create) = True
           IsEmpty(Push(S, i)) = False
      end
```

FIFO Queue

In the *FIFO queue*, the designated item is the first item put into the collection. The operation that puts an item into the *FIFO queue* is usually called *Enq*; the operation that removes an item is usually called *Deq*; and the operation that views the designated item is usually called *First*.

```
structure  Queue (of Item)
interface  Create ⟶ Queue
           Enq(Queue, Item) ⟶ Queue
           Deq(Queue) ⟶ Queue
           First(Queue) ⟶ Item
           IsEmpty(Queue) ⟶ Boolean
      end
```

```
axioms   for Q in Queue, i in Item, let
         Deq(Create) = Error
         Deq(Enq(Q, i)) =
               IF IsEmpty(Q)
                  THEN  Create
                  ELSE  Enq(Deq(Q), i)
         First(Create) = Error
         First(Enq(Q, i)) =
               IF IsEmpty(Q)
                  THEN  i
                  ELSE  First(Q)
         IsEmpty(Create) = True
         IsEmpty(Enq(Q, i)) = False
    end
```

Priority Queue

In the *priority queue*, the items are made up of < *item*,*priority* > pairs. The designated item is the item with the highest priority. The operation that deletes the designated item is usually called *Serve* in a priority queue, and the operation that views the designated item is usually called *Next*. In the following axioms we use the relational operator greater than (>) to compare priorities. Note that highest priority may or may not mean greatest value.

```
structure  PQueue (of <Item, Priority>)
 interface Create ⟶ PQueue
           Enq(PQueue, Item, Priority) ⟶ PQueue
           Serve(PQueue) ⟶ PQueue
           Next(PQueue) ⟶ Item
           IsEmpty(PQueue) ⟶ Boolean
      end
  axioms   for PQ in PQueue, i1, i2 in Item, p1, p2 in Priority, let
           Serve(Create) = Error
           Serve(Enq(Create, i1, p1)) = Create
           Serve(Enq(Enq(PQ, i1, p1), i2, p2))
                 IF p1 > p2
                    THEN  Enq(Serve(Enq(PQ, i1, p1)), i2, p2)
                    ELSE  Enq(Serve(Enq(PQ, i2, p2)), i1, p1)
           Next(Create) = Error
           Next(Enq(Create, i1, p1)) = i1
           Next(Enq(Enq(PQ, i1, p1), i2, p2))
                 IF p1 > p2
                    THEN  Next(Enq(PQ, i1, p1))
                    ELSE  Next(Enq(PQ, i2, p2))
           IsEmpty(Create) = True
           IsEmpty(Enq(PQ, i1, p1) = False
```

Summary of Semi-Structured Data Types

The *semi-structured data types* can be characterized by the fact that the operation that deletes an item takes no item as a parameter. The item to be removed (or viewed) is the designated item. It is the definition of the designated item that distinguishes among these data types. Since the names of the operations are different across the data types in this category, they are summarized below.

Operation	**Stack**	**FIFO Queue**	**Priority Queue**
Insert Item	Push	Enq	Enq
Delete Designated Item	Pop	Deq	Serve
Observe Designated Item	Top	First	Next

It is interesting to note that the *priority queue* can be used to simulate both the *stack* and the *FIFO queue*. If a time stamp is attached to each item to use as the priority, the designated item in the *stack* is the one with the most recent time stamp and the designated item in the *FIFO queue* is the one with the oldest time stamp.

STRUCTURED DATA TYPES

Linear Data Types

Linear data types are those that the user views as having a first, a next, and a last. If it is a property of the type that the user access the items by position, then the *linear data type* is called an *indexed linear list*. If the items are only accessed by moving from one to the next, the data type is called a *non-indexed linear list*. Often the term *linear list* is used to refer to all *linear data types*, leading to confusion about such operations as "storing an item" and "inserting an item". We think that there are two distinct categories of data types.

Indexed

There are two types of *indexed linear structures*: the *array* where an index is permanently bound to an item and the *sequence* where an index reflects the item's current position in the list.

Array

The *array data type* is a collection of $< index, value >$ pairs where the bounds on the index set are known at the time an instance of an *array data type* is created. Like the *record data type*, a newly created instance of an *array data type* is not empty; the structure exists, but the value for each $< index, value >$ pair is undefined. An attempt to store an $< index, value >$ pair where the index is not within the bounds of the array causes an error.

```
structure  Array (of <Index, Value>)
 interface Create(Index, Index) ⟶ Array
           LoBound(Array) ⟶ Index
           HiBound(Array) ⟶ Index
           Store(Array, Index, Value) ⟶ Array
           Retrieve(Array, Index) ⟶ Value
      end
```

```
axioms  for all A in Array, i1, i2, i3, i4 in Index, and
        v in Value, let
        LoBound(Create(i3, i4)) ⟶ i3
        LoBound(Store(A, i1, v)) ⟶ LoBound(A)
        HiBound(Create(i3, i4)) ⟶ i4
        HiBound(Store(A, i1, v)) ⟶ HiBound(A)
        Store(A, i1, v) =
              IF (i1 < LoBound(A)) or (i1 > HiBound(A))
                 THEN  Error
        Retrieve(Create(i3, i4), i1) =
              IF (i1 < LoBound(A)) or (i1 > HiBound(A))
                 THEN  Error
                 ELSE  Undefined
        Retrieve(Store(A, i2, v), i1) =
              IF (i1 < LoBound(A)) or (i1 > HiBound(A))
                 THEN  Error
                 ELSE IF (i1 = i2)
                     THEN  v
                     ELSE  Retrieve(A, i1)
    end
```

Sequences

The items in a *sequence* are also $< index, value >$ pairs. However, the index represents the value's current position in the *sequence* and may change as other items are inserted into the *sequence*. The *Create* operation has no parameters; it returns an *empty sequence*. There are two operations that add items to the collection of items: the *Store* operation which puts an item into a specified place in the *sequence* and the *Insert* operation which inserts an item into a specified position and shifts all those items in that position and subsequent positions down one.

An error condition occurs if an attempt is made to store or insert an item into a position in the list that does not already exist or is greater than the length of the list plus 1. Therefore we will need an operation that will return the length of the *sequence*.

```
structure  Sequence (of <Index, Value>)
interface  Create ⟶ Sequence
           Store(Sequence, Index, Value) ⟶ Sequence
           Length(Sequence) ⟶ Integer
           Insert(Sequence, Index, Value) ⟶ Sequence
           Delete(Sequence, Index) ⟶ Sequence
           Retrieve(Sequence, Index) ⟶ Value
           Find(Sequence, Value) ⟶ Index
           Replace(Sequence, Index, Value) ⟶ Sequence
      end
```

```
axioms for all S in Sequence, i1, i2 in Index,
       and v1, v2 in Value, let
       Length(Create) = 0
       Length(Store(S, i1, v1)) = 1 + Length(S)
       Store(S, i1, v1) =
              IF i1 > Length(S) + 1
                 THEN  Error
                 ELSE  IF i1 <= Length(S)
                       THEN  Replace(S, i1, v1)
       Insert(Create, i2, v2) =  Store(Create, i2, v2)
       Insert(Store(S, Length(S) + 1, v1), i2, v2) =
              IF i2 > 1 + Length(S)
                 THEN  Error
                 ELSE IF i2 = Length(S) + 1
                    THEN Store(Store(S, i2, v2), i2+1, v1)
                    ELSE Store(Insert(S, i2, v2), Length(S)+2,v1)
       Retrieve(Create, i2) = Error
       Retrieve(Store(S, i1, v1), i2) =
              IF i2 = i1
                 THEN  v1
                 ELSE  Retrieve(S, i2)
       Find(Create, v1) = Error
       Find(Store(S, i1, v2), v1) =
              IF v2 = v1
                 THEN i1
                 ELSE Find(S, v1)
       Delete(Store(S, Length(S) + 1, v1), i2) =
              IF i2 > Length(S) + 1
                  THEN  Error
                  ELSE IF i2 = Length(S) + 1
                       THEN  S
                       ELSE Store(Delete(S, i2), Length(S), v1)
       Replace(Create, i2, v2) = Error
       Replace(Store(S, Length(S) + 1, v1), i2, v2) =
              IF i2 = Length(S) + 1
                 THEN Store(S, Length(S) + 1, v2)
                 ELSE Store(Replace(S, i2, v2), Length(S)+1, v1)
```

Summary of Indexed Linear Structures

Arrays are used so often to represent *sequences* in a program that they are frequently thought to be the same thing. There are, however, two distinctions between *arrays* and *sequences*. *Arrays* are fixed size; the *Create* operation returns the structure with the index part of the $< index, value >$ pairs already defined. Once a value is stored with an index, the value stays bound to the index until another value is stored with that same index.

In a *sequence*, the *Create* operation returns an *empty sequence*. The binding of the index and the value is only temporary. It represents the current position of the value within the *sequence*.

Arrays and *records* are similar. Both have their structure built by the create operation. The *array*, however, is *linear* because the index type is ordered, and the *record* is *unstructured* because the set of fields is not ordered.

We have not included the *string* as a separate abstract data type. It is our feeling that the string is a special case of the *sequence*. The additional operations normally associated with strings such as concatenate, substring, and pattern match can be specified using the sequence axioms.

Non-Indexed

Homogeneous

The *non-indexed homogeneous linear structured data type* is what most users mean when they say "linear list." The property of position that is used in the *indexed linear structured data type* is implicitly there, but is immaterial to the user. The operations do not use position as a parameter. All of the items in the list are of the same type.

The two data types in this category differ in the same way that the two data types in the *indexed* category do: one has only one operation that increases the size of the structure, and the other has two. The result is that the items in the first data type are unordered. The items in the second data type are inserted into their sorted position in the list.

Unsorted

```
structure  UnsortedList (of Item)
interface  Create ⟶ UnsortedList
           Make(UnsortedList, Item) ⟶ UnsortedList
           Delete(UnsortedList, Item) ⟶ UnsortedList
           Head(UnsortedList) ⟶ Item
           Tail(UnsortedList) ⟶ UnsortedList
           IsEmpty(UnsortedList) ⟶ Boolean
           IsThere(UnsortedList, Item) ⟶ Boolean
           Length(UnsortedList) ⟶ Integer
      end
   axioms  for i1, i2 in Item, L in UnsortedList, let
           Head(Create) = Error
           Head(Make(L, i1)) = i1
           Tail(Create) = Error
           Tail(Make(L, i1)) = L
           Delete(Create, i1) = Create
           Delete(Make(L, i2), i1) =
                 IF i1 = i2
                    THEN L
                    ELSE Make(Delete(L, i1), i2)
           IsEmpty(Create) = True
           IsEmpty(Make(L, i1)) = False
           IsThere(Create, i1) = False
           IsThere(Make(L, i2), i1) =
                 IF i1 = i2
                    THEN  True
                    ELSE  IsThere(L, i1)
           Length(Create) = 0
           Length(Make(L, i1)) = 1 + Length(L)
      end
```

Sorted

```
structure  SortedList (of Item)
interface  Create ⟶ SortedList
           Make(SortedList, Item) ⟶ SortedList
           Insert(SortedList, Item) ⟶ SortedList
           Delete(SortedList, Item) ⟶ SortedList
           Head(SortedList) ⟶ Item
           Tail(SortedList) ⟶ SortedList
           IsEmpty(SortedList) ⟶ Boolean
           IsThere(SortedList) ⟶ Boolean
           Length(SortedList) ⟶ Integer
      end
   axioms  for i1, i2 in Item, L in SortedList, let
           Head(Create) = Error
           Head(Make(L, i1)) = i1
           Tail(Create) = Error
           Tail(Make(L, i1)) = L
           Insert(Create, i1) = Make(Create, i1)
           Insert(Make(L, i2), i1) =
                 IF i1 < i2
                     THEN  Make(Insert(L, i1), i2)
                     ELSE  Make(Make(L, i2), i1)
           Delete(Create, i1) = Error or Create
           Delete((Make(L, i2 ), i1) =
                 IF i1 = i2
                    THEN  L
                    ELSE Make(Delete(L, i1), i2)
           IsEmpty(Create) =  True
           IsEmpty(Make(L, i1)) = False
           IsThere(Create, i1) = False
           IsThere(Make(L, i2), i1) =
                  IF i1 = i2
                     THEN  True
                     ELSE IF i1 < i2
                        THEN IsThere(L, i1)
                        ELSE False
           Length(Create) = 0
           Length(Make(L, i1)) = 1 + Length(L)
       end
```

Summary of Non-Indexed Linear Structure

The only difference in the sorted and unsorted specifications is the additional operator *Insert* in the sorted version.

Non-Homogeneous

The only traditional data type in this category is the abstract data type usually referred to as a *generalized list*. A *generalized list* is a *linear structured data type* where the items in the structure can either be data items (often called *atoms*) or other generalized lists.

```
     type  ComponentType = (Atom, GenList)
structure  GenList (of ComponentType)
interface  Create ⟶ GenList
           IsEmpty(GenList) ⟶ Boolean
           WhichType(Component) ⟶ ComponentType
           Make(GenList, Component) ⟶ GenList
           Concat(GenList, GenList) ⟶ GenList
           Head(GenList) ⟶ Component
           Tail(GenList) ⟶ GenList
      end
   axioms  for c in Component, GL1, GL2 in GenList,
           IsEmpty(Create) ⟶ True
           IsEmpty(Make(c, GL1)) = False
           Head(Create) = Error
           Head(Make(GL1, c)) = c
           Tail(Create) = Error
           Tail(Make(GL1, c)) = GL1
           Concat(Create, GL1) = GL1
           Concat(Make(GL2, c), GL1) =
                  Make(Concat(GL2, GL1), c)
      end
```

Augment Domains with ADT List

```
List(GenList) ⟶ UnsortedList
List(Create) ⟶ Create
List(Make(Gl1, c)) =
       IF WhichType(c)) = Atom
          THEN  Make(List(Gl1), c))
          ELSE  Concat(List(Gl1), List(c))
```

Multi-Dimensional

The specifications for *multi-dimensional arrays* are a direct extension of the specifications for *arrays*. We will give only the specification for *two-dimensional arrays* here.

```
structure  TwoDArray (of <Index, Index, Value>)
interface  Create(Index, Index, Index, Index) ⟶ Array
           LoBound1(Array) ⟶ Index
           HiBound1(Array) ⟶ Index
           LoBound2(Array) ⟶ Index
           HiBound2(Array) ⟶ Index
           Store(Array, Index, Index, Value) ⟶ Array
           Retrieve(Array, Index, Index) ⟶ Value
      end
```

```
axioms  for all A in TwoDArray, i1, i2, i3, i4, i5, i6 in Index,
            and v in Value, let
        LoBound1(Create(i3, i4, i5, i6)) ⟶ i3
        LoBound1(Store(A, i1, i2, v)) ⟶ LoBound1(A)
        HiBound1(Create(i3, i4, i5, i6)) ⟶ i4
        HiBound1(Store(A, i1, i2, v)) ⟶ HiBound1(A)
        LoBound2(Create(i3, i4, i5, i6)) ⟶ i5
        LoBound2(Store(A, i1, i2, v)) ⟶ LoBound2(A)
        HiBound2(Create(i3, i4, i5, i6)) ⟶ i6
        HiBound2(Store(A, i1, i2, v)) ⟶ HiBound2(A)
        Store(A, i1, i2, v) =
              IF (i1 < LoBound1(A)) or (i1 > HiBound1(A))
                 or (i2 < LoBound2(A)) or (i2 > HiBound2(A))
                 THEN  Error
        Retrieve(Create(i3, i4, i5, i6), i1, i2) =
              IF (i1 < LoBound1(A)) or (i1 > HiBound1(A))
                 or (i2 < LoBound2(A)) or (i2 > HiBound2(A))
                 THEN  Error
                 ELSE  Undefined
        Retrieve(Store(A, i1, i2, v), i3, i4) =
              IF (i3 < LoBound1(A)) or (i3 > HiBound1(A))
                 or (i4 < LoBound2(A)) or (i4 > HiBound2(A))
                 THEN  Error
                 ELSE IF (i1 = i3) and (i2 = i4)
                     THEN  v
                     ELSE  Retrieve(A, i3, i4)
    end
```

Non-Dimensional

This category contains two basic data types, the *tree* and the *graph*. In fact, a *tree* is technically a restricted type of graph, although the user of the abstract data type *tree* will probably not view it that way. Similarly, a *search tree* and a *heap* can be considered as restricted type of *tree*, but users often consider these special trees as separate abstract data types in their own right. In the next two sets of axioms, we focus on *binary trees* and *binary search trees*. A natural extension of these axioms would produce *k-way trees* and *k-way search trees*.

Binary Tree

As in the case of *sorted* and *unsorted lists* and *sequences* and *arrays*, there are different types of *binary trees*. Here, a simple *binary tree* has only a *store* operation, while both *binary search trees* and *heaps* have *insert* operations.

Binary Tree

```
structure  BinTree (of Item)
 interface Create ⟶ BinTree
           Make(BinTree, Item, BinTree) ⟶ BinTree
           LeftTree(BinTree) ⟶ BinTree
           RightTree(BinTree) ⟶ BinTree
           Data(BinTree) ⟶ Item
           IsEmpty(BinTree) ⟶ Boolean
     end
```

```
axioms  for all BT1, BT2 in BinTree, i1 in Item, let
        LeftTree(Create) = Error
        LeftTree(Make(BT1, i1, BT2)) = BT1
        RightTree(Create) = Error
        RightTree(Make(BT1, i1, BT2)) = BT2
        Data(Create) = Error
        Data(Make(BT1, i1, BT2)) = i1
        IsEmpty(Create) = True
        IsEmpty(Make(BT1, i1, BT2)) = False
    end
```

Iterator for Binary Tree: Augment the Domains with ADT Queue (of Data)

```
PreOrder(BinTree) ⟶ Queue
Concat(Queue, Queue) ⟶ Queue
Concat(Q, Create) = Q
Concat(Q, Enq(P, i1)) = Enq(Concat(Q, P), i1)
PreOrder(Create) = Create
PreOrder(Make(BT1, i1, BT2)) =
  Concat(Concat(Enq(Create, i1), PreOrder(BT1)),
         PreOrder(BT2))
```

Binary Search Trees

When the operations for a *BinTree* are augmented with an *Insert* operation that inserts the item into is proper place in the *tree*, the result is a *binary search tree*. Because of the length of these axioms, we will only show the *Insert* and the corresponding *Delete*. We also need an operation that returns the maximum element in a *binary search tree* in order to specify the *Delete* operation correctly. This operation can be made part of the interface or can be kept hidden from the user. Here, we will add it to the interface.

```
structure  BSTree (of Item)
interface  .
           .
           Insert(BSTree, Item) ⟶ BSTree
           Delete(BSTree, Item) ⟶ BSTree
           MaxEl(BSTree) ⟶ Item
     end
```

```
axioms   for all i1, i2 in Item, BST1, BST2 in BSTree, let
         Insert(Create, i1) = Make(Create, i1, Create)
         Insert(Make(BST1, i2, BST2), i1) =
                IF i1 < i2
                   THEN  Make(Insert(BST1, i1), i2, BST2)
                   ELSE  Make(BST1, i2, Insert(BST2, i1))
         MaxEl(Create) =  Error
         MaxEl(Make(BST1, i1, BST2))
                   IF IsEmpty(BST2)
                      THEN  i1
                      ELSE  MaxEl(BST2)
         Delete(Create, i1) = Create
         Delete(Make(BST1, i2, BST2), i1) =
                IF i1 = i2
                   THEN  IF IsEmpty(BST1)
                            THEN  BST2
                         ELSE  IF IsEmpty(BST2)
                                  THEN  BST1
                         ELSE  Make(Delete(BST1, MaxEl(BST1)),
                                    MaxEl(BST1), BST2)
                   ELSE  IF i1 < i2
                         THEN Make(Delete(BST1, i1), i2, BST2)
                         ELSE Make(BST1, i2, Delete(BST2, i1))
```

Heaps

Just as one ordering of data within a tree yields a *binary search tree*, another ordering produces a *partially-ordered tree*. In most cases, users want *partially-ordered trees* to have a special, balanced shape, and the resulting structure is called a *heap*. It should be noted, however, that there is some debate about whether shape is an abstract property (to be given in axioms) or whether shape is an implementation issue. In the context of *binary search trees*, shape gives rise to the notion of *AVL trees*, but many consider these height-balanced trees to vary only in implementation from other *binary search trees*; there may not be any change in axioms from one to the other. Similarly, *partially-ordered trees* and *heaps* can be viewed as having similar properties. When users often rely upon shape in their thinking and analysis, however, it may be helpful to incorporate this notion of shape in the axioms themselves. The following axioms for *heaps* illustrate how shape can be built into an axiomatic specification of a data type.

```
structure  Heap (Of Item)
 interface Create ⟶ Heap
           Make(Heap, Item, Heap) ⟶ Heap
           Left(Heap) ⟶ Heap
           Right(Heap) ⟶ Heap
           Data(Heap) ⟶ Item
           Insert(Heap, Item) ⟶ Heap
           Delete(Heap) ⟶ Heap
           TopHeap(Heap) ⟶ Item
           IsEmpty(Heap) ⟶ Boolean
      end
```

```
axioms  for all H1, H2 in Heap, i2 in Item, let
        Left(Create) = Error
        Left(Make(H1, i1, H2)) = H1
        Right(Create) = Error
        Right(Make(H1, i1, H2)) = H2
        Data(Create) = Error
        Data(Make(H1, i1, H2)) = i1
        IsEmpty(Create) = True
        IsEmpty(Make(H1, i1, H2)) = False
        TopHeap(Create) = Error
        TopHeap(Make(H1, i1, H2)) = i1
```

The axioms for *Delete* and *Insert* require several auxiliary operations which are included in what follows.

```
Height(Heap) ⟶ Integer
Full(Heap) ⟶ Heap
Last(Heap) ⟶ Item
DelLast(Heap) ⟶ Heap
ReHeap(Heap) ⟶ Heap
```

```
Height(Create) = 0
Height(Make(H1, d1, H2)) = Max(Height(H1), Height(H2)) + 1
Full(Create) = True
Full(Make(H1, i1, H2)) =
      IF Height(H1) <> Height(H2)
         THEN  False
         ELSE  Full(H1) AND Full(H2)
Insert(Create, i1) = Make(Create, i1, Create)
Insert(Make(H1, i2, H2), i1) =
      IF NOT Full(H1) OR Full(Make(H1, i2, H2))
         THEN
           IF i1 < i2
              THEN  Make(Insert(H1, i1), i2, H2)
              ELSE  Make(Insert(H1, i2), i1, H2)
         ELSE
           IF i1 < i2
              THEN Make(H1, i2, Insert(H2, i1))
              ELSE Make(H1, i1, Insert(H2, i2))
Last(Create, i1, Create) = i1
Last(Make(H1, i1, H2))   =
      IF Full(Make(H1, i1, H2)) OR NOT Full(H2)
         THEN  Last(H2)
         ELSE  Last(H1)
DelLast(Create, i1, Create) = Create
DelLast(Make(H1, i1, H2))   =
      IF Full(Make(H1, i1, H2)) OR NOT Full(H2)
         THEN  Make(H1, i1, DelLast(H2))
         ELSE  Make(DelLast(H1), i1, H2)

Delete(Create) = Error
Delete(Make(Create, i1, Create)) = Create
Delete(Make(H1, i1, H2)) =
        ReHeap(DelLast(Make(H1, Last(Make(H1, i1, H2)) , H2)
```

```
ReHeap(Make(Create, i1, Create) = Make(Create, i1, Create)
ReHeap(Make(H1, i1, Create)) =
       IF i1 < Data(H1)
          THEN Make(Make(Create, i1, Create), Data(H1), Create))
          ELSE Make(H1, i1, Create)
ReHeap(Make(H1, i1, H2)) =
      IF (i1 > Data(H1)) AND (i1 > Data(H2))
          THEN Make(H1, i1, H2)
          ELSE
           IF Data(H1) > Data(H2)
              THEN
     Make(ReHeap(Make(Left(H1), i1, Right(H1))), Data(H1), H2)
              ELSE
     Make(H1, Data(H2), ReHeap(Make(Left(H2), i1, Right(H2))))
```

Graphs

Mathematically, a *Graph* is a nonempty set of *Vertices* and a set of *Edges*. In the axiomatic specification, it is more convenient to allow the vertex set to be empty and to view the edge set as a collection of $< Vertex, Vertex, Weight >$ triples. Most axioms for graphs then are completely analogous to those for sets, with one set of axioms for the vertex set and one for the edge set. The relationship among these operations is given in the following table.

Operation	**Set**	**Vertex Set**	**Edge Set**
Insert Item	Store	StoreVertex	StoreEdge
Determine if Set Empty	IsEmpty	VerticesEmpty	EdgesEmpty
Count Items	Card	VertCard	EdgeCard
Delete Item	Delete	DelVertex	DelEdge
Find Item in Set	IsIn	IsVertex	IsEdge

The axioms for a *graph* distinguish between the vertex set and the edge set using both operations *StoreVertex* and *StoreEdge* as primitives, subject to the constraint that an edge may be stored only if both of its vertices are present. Individual axioms for graphs now generally follow the same form as for sets, except that the deletion of a vertex from a *graph* also deletes all the edges associated with the deleted vertex.

```
structure  Graph (of <vertex> and <Vertex, Vertex, Weight>)
interface  Create ⟶ Graph
           StoreVertex(Graph, Vertex) ⟶ Graph
           VerticesEmpty(Graph) ⟶ Boolean
           VertCard (Graph) ⟶ Integer
           DelVertex(Graph, Vertex) ⟶ Graph
           IsVertex(Graph, Vertex) ⟶ Boolean
           StoreEdge(Graph, Vertex, Vertex, Weight) ⟶ Graph
           EdgesEmpty(Graph) ⟶ Boolean
           EdgeCard (Graph) ⟶ Integer
           DelEdge(Graph, Vertex, Vertex) ⟶ Graph
           IsEdge(Graph, Vertex, Vertex) ⟶ Boolean
           GetWeight(Graph, Vertex, Vertex) ⟶ Weight
     end
```

```
axioms  for all v, v1, v2, v3, v4 in Vertex, w in Weight,
                G in Graph, let
        StoreVertex(G, v) =
                 IF IsVertex(G, v)
                    THEN G
        VerticesEmtpy(Create) = True
        VerticesEmpty(StoreVertex(G, v)) = False
        VerticesEmpty(StoreEdge(G, v1, v2, w)) = VerticesEmpty(G)
        VertCard(Create) = 0
        VertCard(StoreVertex(G, v1)) = 1 + VertCard(G)
        VertCard(StoreEdge(G, v1, v2, w)) = VertCard(G)
        DelVertex(Create, v) = Create
        DelVertex(StoreVertex(G, v1), v) =
                IF (v1 = v)
                   THEN G
                   ELSE StoreVertex(DelVertex(G, v), v1)
        DelVertex(StoreEdge(G, v1, v2, w), v) =
                IF (v1 = v) OR (v2 = v)
                   THEN DelVertex(G, v)
                   ELSE StoreEdge(DelVertex(G, v), v1, v2, w)
        IsVertex(Create, v) = False
        IsVertex(StoreVertex(G, v1), v) =
                IF v1 = v
                   THEN True
                   ELSE IsVertex(G, v)
        IsVertex(StoreEdge(G, v1, v2, w) = IsVertex(G, w)
        StoreEdge(G, v1, v2, w) =
                 IF IsVertex(G, v1) AND IsVertex(G, v2)
                    THEN IF IsEdge(G, v1, v2)
                            THEN G
                            ELSE <null>
                    ELSE G
        EdgesEmtpy(Create) = True
        EdgesEmpty(StoreVertex(G, v)) = EdgesEmpty(G)
        EdgesEmpty(StoreEdge(G, v1, v2, w)) = False
        EdgeCard(Create) = 0
        EdgeCard(StoreVertex(G, v1)) = EdgeCard(G)
        EdgeCard(StoreEdge(G, v1, v2, w)) = 1 + EdgeCard(G)
        DelEdge(Create, v) = Create
        DelEdge(StoreVertex(G, v1), v2, v3) =
                StoreVertex(DelEdge(G, v2, v3), v1)
        DelEdge(StoreEdge(G, v1, v2, w), v3, v4) =
                IF (v1 = v3) AND (v2 = v4)
                   THEN G
                   ELSE StoreEdge(DelEdge(G, v3, v4), v1, v2, w)
        IsEdge(Create, v1, v2) = False
        IsEdge(StoreVertex(G, v), v1, v2) = IsEdge(G, v1, v2)
        IsEdge(StoreEdge(G, v3, v4, w), v1, v2) =
                IF (v1 = v3) AND (v2 = v4)
                   THEN True
                   ELSE IsEdge(G, v1, v2)
```

```
        GetWeight(Create, v1, v2) = Undefined
        GetWeight(StoreVertex(G, v), v1, v2) = GetWeight(G, v1, v2)
        GetWeight(StoreEdge(G, v3, v4, w), v1, v2) =
                IF (v3 = v1) AND (v4 = v2)
                    THEN  w
                    ELSE  GetWeight(G, v1, v2)
    end
```

Graph algorithms often depend upon a list of vertices that are adjacent to a given one. The appropriate axioms use the data type *list* to hold vertex names.

Augment the Domains with the ADT List (of Edges)

```
FromEdges(Graph, Vertex) ⟶ List
ToEdges(Graph, Vertex) ⟶ List

FromEdges(Create, v1) = Create
FromEdges(StoreVertex(G, v), v1) = FromEdges(G, v1)
FromEdges(StoreEdge(G, v3, v4, w), v1) =
     IF v1 = v3
        THEN  Store(FromEdges(G, v1), v4)
        ELSE  FromEdges(G, v1)
ToEdges(G, v1) = Create
ToEdges(StoreVertex(G, v), v1) = ToEdges(G, v1)
ToEdges(StoreEdge(G, v3, v4, w), v1) =
     IF v1 = v4
        THEN  Store(ToEdges(G, v1), v3)
        ELSE  ToEdges(G, v1)
```

Conclusions: This paper has presented a coherent classification of data types, based upon a hierarchical tree organization. Throughout the tree, properties or constraints are added in moving from each node to its children, and each node inherits all properties of its ancestors. The result is a unified classification of a wide variety of data types. Within this framework, this paper presented a careful, axiomatic specification of each data type. This specification clarified the individual operations for each data type and allowed similarities and differences of operations on various data types to be highlighted.