Report Number: WUCS-91-33

1991-06-01

DNA Mapping Algorithms: Abstract Data Types - Concepts and Implementation

Will Gillett and Liz Hanks

The conceptual aspects of and the implementation details of a set of self-identifying abstract data types (ADT) are described. Each of the ADTs constitutes a specific class of object, upon which a set of well-defined access functions is available. The intent of these ADTs is to supply a paradigm in which a class of object is available for manipulation, but in which the underlying implementation is hidden from the application programmer. Specific ADTs are the described in some detail. The tagged architecture used to achieve the self-identifying property of the ADTs is presented, and a set of required system-backbone... Read complete abstract on page 2.

DNA Mapping Algorithms: Abstract Data Types - Concepts and Implementation

Will Gillett and Liz Hanks

Complete Abstract:

The conceptual aspects of and the implementation details of a set of self-identifying abstract data types (ADT) are described. Each of the ADTs constitutes a specific class of object, upon which a set of well-defined access functions is available. The intent of these ADTs is to supply a paradigm in which a class of object is available for manipulation, but in which the underlying implementation is hidden from the application programmer. Specific ADTs are the described in some detail. The tagged architecture used to achieve the self-identifying property of the ADTs is presented, and a set of required system-backbone access function is defined. Their combination is shown to produce a robust system in which complex aggregate ADT classes can be flexibly created and managed with little effort on the part of the application programmer. Memory management and statistics reporting techniques are presented.

DNA Mapping Algorithms: Abstract Data Types - Concepts and Implementation

Will Gillett and Liz Hanks

WUCS-91-33

June 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This work was supported by the James S. McDonnell Foundation under Grant 87-24 and NIH under grant 1 RO1 HG00180-01.

ABSTRACT

The conceptual aspects of and the implementation details of a set of self-identifying abstract data types (ADTs) are described. Each of the ADTs constitutes a specific class of object, upon which a set of well-defined access functions is available. The intent of these ADTs is to supply a paradigm in which a class of object is available for manipulation, but in which the underlying implementation is hidden from the application programmer.

Specific ADTs are described in some detail. The tagged architecture used to achieve the self-identifying property of the ADTs is presented, and a set of required system-backbone access function is defined. Their combination is shown to produce a robust system in which complex aggregate ADT classes can be flexibly created and managed with little effort on the part of the application programmer. Memory management and statistics reporting techniques are presented.

TABLE OF CONTENTS

TABLE OF FIGURES

1. Introduction

This reports describes the conceptual aspects of and the implementation details of a set of self-identifying abstract data types, which will be referred to here as ADTs. Each of the ADTs constitutes a specific **class** of object, such as integer, list, or tree, upon which a set of well-defined functions should be applicable. The intent of ADTs is to supply a paradigm in which a class of object is available for manipulation, but for which the underlying implementation is hidden from the application programmer.

The DNA Mapping encapsulation of ADTs, as implemented with a self-identifying (tagged) architecture, constitutes a paradigm very similar to that of object-oriented programming (OOP). The major difference between standard object-oriented programming and the DNA Mapping ADTs is that inheritance is present in most object-oriented system but is not present in our ADTs.

1.1. Software Engineering Reasoning for ADTs

Although the underlying implementation of ADTs is based on a combination of standard, well-understood data structures (arrays, structs, pointers, and intrinsic data types), the application programmer is not allowed to know this underlying structure or access any of the actual physical components directly. Instead, a set of access functions is supplied, with which the application programmer can manipulate the conceptual components of an ADT.

For instance, given an ADT intended to simulate a list of objects, **adt_list**, the application programmer may want to know the cardinality of a specific list. In order to extract this information, an access function, say `list_cardinality`, would be supplied. When invoked, this function would return the number of objects currently in the list -- but without the application programmer knowing how the underlying implementation is achieved. For example, it might be that the cardinality is actually held as an explicit field of the underlying data structure representing the conceptual list, and `list_cardinality` simply references this field causing no computation to be done. In this case the cardinality field probably would be incremented each time an object is inserted into the list and decremented each time an object is deleted from the list. This underlying implementation requires an explicit memory location to maintain the datum continually being updated. An alternative approach is not to maintain explicit information about the cardinality, but instead to search through the list, counting the number of objects present each time the `list_cardinality` function is invoked. This implementation option requires no extra memory to hold the explicit cardinality. Each has its own advantage. One is efficient in time, and the other is efficient in space.

Another option for holding lists is to implement them either as a sequential array or as a dynamically allocated linked list. Each of these possibilities has its advantages and disadvantages. The linked list approach may be efficient in time with respect to insertions and deletions, but may take much more memory than the array approach. The array approach may be efficient with respect to memory, but has severe time penalties for insertion and deletion of objects and may require significant reallocation as the list grows in size.

The ADT paradigm claims that all of these underlying implementation decisions should be hidden from the application programmer. The question as to why this is an desirable property to have is an appropriate and important one. The basic answer is the following. The application programmer should program in a conceptual space that does not change unless the application itself changes. In other words, the "things" manipulated by the programmer should be perceived to be "things" that are important to the application and not primarily important to the details of some specific programming language (i.e., "things" that are important to the solution of the application). Once the solution to a problem (application) has been found, it should not have to be changed just because something in the underlying implementation changes. (A more global example of this has to do with the reason that high-level languages are used to solve problems. Specifically, if a solution to a problem has been found using the language C, then we would like that solution to remain applicable even though the eventual executable program is run on a machine other than the one upon which it was developed.) In the conceptual world of ADTs, we would like the solution to a problem to remain valid, even though the underlying implementation of a list is changed from an array approach to a linked list approach. Specifically, with respect to the application, the sequence of insertions into and deletions from some list will be identical, independent of the mechanism chosen to implement the list (i.e., array or linked list). If, for some reason, the original mechanism chosen for implementing lists is found to be inappropriate or deficient and a subsequent decision is made to change the original mechanism for implementing lists, the application code itself should not have to be changed just because the underlying implementation of lists has changed.

By hiding the underlying implementation of the ADTs, implementation decisions can be deferred or changed without affecting the quality or validity of the application code being developed. This allows prototype systems to be developed quickly. If a quick-and-dirty but inefficient initial implementation of a set of ADTs can be created, then it may be possible to create the basic **functionality** of an application quickly, allowing the fine-tuning and time and space optimization to be deferred until later in the project.

Besides the ability to change the underlying implementations of the ADTs without affecting the application code built upon them, simply the use of mathematically well-understood abstractions (implementable as ADTs) is an extremely powerful conceptual tool in the programming of solutions to applications. If ADTs simulating lists, sets, pairs, trees, stacks, etc. can be implemented, then the application programmer can use these powerful conceptual tools as actual tools available in the concrete form of programming constructs. Specifically, many problems themselves are described by using abstract mathematical concepts. More importantly, abstract algorithms for solving the problems are often expressed using these abstract mathematical concepts. The ability to implement an abstract algorithm quickly, without resorting to time-consuming translation into standard mundane programming constructs, makes it possible to determine the validity of ideas quickly. It also enhances the ability to change, reconfigure, upgrade, manage, document and verify the application code produced.

At the inception of the DNA Mapping project, it was clear that whatever specific code was produced at the start of the project would not be applicable by the end of the project, let alone years after the end of the project. This is true for a myriad of reasons. The biological laboratory protocols will change over time; the problems themselves will change over time; the questions being asked by the biologist will change over time. It was evident that whatever code was produced would have to be extremely reconfigurable in order to adapt to (a) changes whose origin and nature are already known but whose specifics are not currently known and (b) changes whose origin and nature cannot be anticipated. The incorporation of ADTs within the system has largely been due to this realization. It is hoped that by the use of very abstract high-level constructs, large components of the software will be extractable, modifiable, and reusable over a long period of growth and learning about DNA mapping.

1.2. Self-identifying Properties of ADTs

The self-identifying properties of the ADTs implemented within the system are paramount to the operation of the system, because they allow the **type** of an object to be determined at execution time instead of at compile time. The self-identification is achieved by attaching a **tag** to each instance of an ADT at execution time. Each instance of an ADT in a specific class of ADT is given the **same** tag as all other instances in that class of ADT. The tags used across the different classes of ADTs are uniquely different, i.e., the tag of an instance uniquely identifies the class of ADT to which the instance belongs. This tag is not a compile-time concept; it is an execution-time concept. It is present and identifiable at execution time. This means that, given a pointer to an arbitrary instance of an ADT, it is possible to determine the class of ADT to which it belongs at execution time.

This self-identifying property of ADTs allows combinations (or aggregates) of ADTs to be put together in arbitrary ways. For instance, it is possible to create a list that contains objects which are instances from different classes of ADTs. In other words, a list can have a combination of objects present in it, such as integer ADTs, float ADTs, and string ADTs. The ADT classes are not restricted to intrinsic data types, however. A list can contain another list as one of the objects present in it. In fact, there is no limit to the recursive inclusion of ADTs as components of other ADTs. For instance, it is possible to have a pair whose left element is a string ADT and whose right element is a list of trees. It also is possible to have a list of stacks of pairs of trees.

The use of ADTs with an execution-time tag present does restrict what can be generically used as components of ADTs. In most cases, the generic components must be ADTs themselves. In other words, raw (unencapsulated) C data types such as integer, float, and string are not allowed, since they have no tag present to identify their type at execution time. This does present minor restrictions on the use of ADTs. In general the solution to these restrictions is to "wrap" raw intrinsic C data types into an ADT and use the ADT counterparts as components of other (aggregate) ADTs. These ADT classes are supplied for: integer, float, string, and Boolean.

1.3. System-backbone Access Functions

For each ADT class, a set of required system-backbone access functions must be supplied. Each ADT class is declared to the system at execution time; these required access functions are made available to the system at this declaration time. They include (a) how to create an instance of the ADT, (b) how to destroy an instance of the ADT (i.e., return the, possibly aggregate, memory associated with the instance), (c) how to make a copy of an instance of the ADT, (d) how to pretty print an instance of the ADT (i.e., print the contents of the instance in a high-level conceptual manner), (e) how to debug print an instance of the ADT (i.e., print the contents of the instance in a low-level detailed manner), and (f) how to compare two instances of the ADT to determine whether one is less than, equal to, or greater than the other.

Given these access functions and the self-identifying properties of the ADTs, the application programmer can manipulate aggregate ADTs in very generic ways. For instance, the programmer can pretty print a list by calling the routine `list_pprint`. In this process, appropriate punctuation (square brackets and a comma) are used to delimit the objects in the list, as one might expect. However, the actual printing of the objects themselves cannot be accomplished by the `list_pprint` routine directly because it does not have the knowledge required to pretty print every type of object present in the system. Instead, since the tag of each object is available at execution time, the appropriate pretty print routine applicable to the specific object present can be "looked up" and invoked to produce the desired results.

Similar logic holds for destroying, copying, debug printing and comparing compound objects. In fact, since this idea of recursively operating on a compound object (based on the tag of its components) is so pervasive that macros which accept an object and call the appropriate specific function based on the value of the tag found are supplied to the application programmer. These macros include `DESTROY`, `COPY`, `PPRINT`, `DPRINT`, and `COMPARE`.

The ideas used for building complex data structures using these types of ADTs are very similar to those used while producing complex data structures in LISP. In LISP, whenever a list of objects is desired, a left-spined or right-spined tree is created with the objects "hanging" from the tree; here a specific generic list is created. However, the application programmer need put no effort into understanding or implementing the details of the list mechanism. In LISP, if a pair of objects is desired, a `cons` is performed making one object the left son of a tree and the other object the right son of the tree; here an explicit pair is created. In general the conceptual viewpoint for creating complex data structures using ADTs is similar to that found in using LISP.

1.4. Exposition

In order to expose the nature of the ADTs present in the system, the details of four specific ADTs will be presented: **adt_int** (integer), **adt_string** (character string), **adt_pair** (a 2-tuple), and **adt_dll** (doubly-linked list). **adt_int** is included because it is a very simple fixed-size intrinsic data type. **adt_string** is included to show the variation involved for a variable-length data type. **adt_pair** is included because it is probably the simplest aggregate ADT present in the system. **adt_dll** is included to show the complexity of a variable-length, multiple-component ADT.

The reader should be aware that the code presented in this report is not exactly the same as that present in the DNA Mapping software. In general, the code shown here is intended to convey the important ideas and some of the details of ADTs. However, the code has been simplified to eliminate system detail that might confuse this exposition.

The reader also should be aware that the use of ADTs does have some disadvantages. The power of these ADTs allows the application programmer to do many unorthodox and inappropriate things. For instance, it is possible to include as a member of a list the list itself (since actual pointers are used). If such a hideous recursive situation occurs, an action as simple as a pretty print will produce infinite output. Since memory is dynamically allocated in creating ADTs and the user of the ADT is responsible for destroying the ADT (i.e., returning the memory), the application programmer must be vigilant to destroy each instance of an ADT which is no longer needed. Failure to comply with this maxim will cause memory exhaustion and the execution will abort. Consider the prospect of leaving just one word of memory inappropriately allocated in the middle of a loop which is executed a million times.

Section 2 presents a number of required system-backbone access function that must be present for every ADT class declared to the system. The presence of these functions along with the self-identifying property of the ADTs allows the effective management of complex aggregate ADTs. Section 3 shows how to implement the normal object-operation access functions associated with ADTs. Section 4 describes how an ADT class is declared to the system at execution time, how statistics about the status of instances of ADTs can be obtained, and how memory management is performed. Section 5 briefly describes a set of ADTs present in the DNA Mapping system. These ADTs are subdivided into three categories based on their generality. Section 6 discusses the usefulness and utility of the ADT paradigm.

2. Required System-backbone Access Functions

In order to use an ADT within the system, it must first be declared to the system. In this declaration a number of required system-backbone access function are made available to the system, the size of an instance is given, the pointer to the null object is specified, and the tag to be used for this ADT is returned for external use. The details of this declaration will be deferred to Section 4.1. Here, we are interested in the access functions that are presented during this declaration. They are the following functions: create, destroy, pprint, dprint, copy, and compare.

2.1. Creating

As a simple introduction to creating an instance of an ADT, consider **adt_int**. Assume that the structure of an **adt_int** has been declared as in Figure 1. In this figure, the macro `MEM_NULL` refers to the NULL pointer for a generic ADT of unknown class; its current value is `0` in compliance with most null pointers. The macro `INT_VAL` is used to mask the underlying implementation from the programmer. Note that, in this case, the programmer is not the application programmer but instead the creator of the ADT class itself. Such hiding by macros has been found to be invaluable for allowing structural changes to

```
struct  int_node {
            int int_fld;
        };
typedef struct int_node     STR_INT_TYPE, *INT_TYPE;
#define INT_NULL    ((INT_TYPE)MEM_NULL)
#define INT_VAL(p)  ((int)(p)->int_fld)
```

Figure 1: Declarations for **adt_int**

occur to the underlying implementation while minimizing the amount of code that must be modified.

A simple routine for creating an instance of an **adt_int** can be constructed, as shown in Figure 2. In this type of create routine, the actual value of the desired ADT is presented at creation time. This is simple because the value being inserted into the ADT is a scalar, in contrast to creation of an aggregate list. Here, the integer `i` is an input parameter. The first executable statement simply insures that initialization (mostly declaration) has been done. The invocation of `mem_man_fixedsize_allocate` allocates the correct amount of memory for a data structure of this class. (Here, the prefix `mem_man_` stands for "memory management".) The details of this will be deferred to Section 4.3.1. However, the general idea will be discussed here. In this case, two full words of memory are allocated; one for the tag and one for the integer itself. (The size of the **adt_int** is known to the system because it was declared at initialization time and thus need not be specified here. This is in contrast to a variable-size data structure, in which a call to `mem_man_varsize_allocate` will be performed.) The correct tag for an **adt_int** is placed in the first word of the memory allocated, and a pointer to the second word (the actual **adt_int** structure as declared in Figure 1) is returned through the function name. The macro `INT_VAL` is used to insert the value of the input parameter into the memory which has been allocated. A pointer to the user-accessible portion of this instance of an **adt_int** is returned through the function name.

In this ADT convention, the system knows that if it is manipulating a pointer to an instance of an ADT, then it can find the tag corresponding to this instance one word to the left of the current pointer. However, the current pointer points to exactly the structure which was declared by the programmer. Thus, this pointer can be used as if there were no tag present at all.

Now consider a more complex situation in which a variable amount of memory must be allocated based on the "size" of the incoming argument. Such a situation is typified by **adt_string**. Consider the declarations of Figure 3. Here, notice that the struct `string_node` actually contains no space for the

```
INT_TYPE
int_create(i)
    int i;
{
    INT_TYPE     p;

    if(!int_init_done) int_ds_init();
    p = (INT_TYPE)mem_man_fixedsize_allocate(int_tag);
    INT_VAL(p) = i;

    return(p);
}
```

Figure 2: Creation of an Instance of **adt_int**

```
struct   string_node{
             int size_fld;
         };
typedef struct string_node  STR_STRING_TYPE, *STRING_TYPE;
#define STNG_NULL    ((STRING_TYPE)MEM_NULL)
#define STNG_SIZE(p)     ((int)(p)->size_fld)
#define STNG_VAL(p)  ((STRING)((MEM_TYPE)(p)+sizeof(STR_STRING_TYPE)))
#define STNG_LEN(p)      ((int)strlen(STNG_VAL(p)))
#define STNG_IS_EMPTY(p)     (STNG_LEN(p) == 0)
```

Figure 3: Declarations for **adt_string**

variable-size string that eventually will reside in the instance of the **adt_string**. Instead there is only an integer `size_fld` (size field) which will be used to indicate the size of the (user-specified) variable-size memory allocated for this instance of the **adt_string**. The actual string data will be concatenated just after this one-word field. Note again the use of macros to mask the underlying implementation.

The code for creating an instance of an **adt_string** is shown in Figure 4. Here, the type `STRING` is equivalent to `char *`. The structure of the code here is similar to that of Figure 2, but there is some variation. The length of the (user-specified) portion of the data structure must be computed. This is the length of the character string (i.e., the call to `strlen`) plus 1 (for the `\0` at the end of the string) plus the length of the `size_fld` itself. The correct amount of memory is allocated by calling `mem_man_varsize_allocate`. Note that in this variable-size case, the size of the desired object must be specified (as the second argument, `size`). The `size_fld` is set by using the `STNG_SIZE` macro. Then the value of the incoming string is copied to just after the `size_fld` component of the struct. The pointer to the (user-specified) data structure is returned through the function name.

Several conventions have been used here that are important to the construction of ADTs. It should be clear that there are two routines used for allocating memory: `mem_man_fixedsize_allocate` for fixed-size ADTs, and `mem_man_varsize_allocate` for variable-size ADTs. Correspondingly, there are two deallocation functions: `mem_man_fixedsize_deallocate` and `mem_man_varsize_deallocate`. For fixed-size ADTs there is no question about how much memory to deallocate when an instance is returned to the system. However, for the deallocation of a variable-size ADT, the original amount of memory allocated at creation time must be specified at destruction time, so that the correct amount of memory is returned. This is the basic reason for the including the `size_fld` in the structure of an **adt_string**. It's value will be used at deallocation time. The decision to make it the first field of the struct is an important one. This is because there are two ways of returning an instance of a variable-size ADT to the system. `mem_man_varsize_deallocate` may be called, in which case the size of the (user-specified) data structure must be specified. Alternatively `mem_man_fixedsize_deallocate` can be called if the builder of the ADT class has used the convention to make the size of the variable-size data structure the first field of the data structure. In this case, if the instance is known to be of variable length, `mem_man_fixedsize_deallocate` simply looks in the first field of the structure for the appropriate length and deallocates that amount of memory.

Now consider an aggregate ADT, **adt_pair**, which implements a 2-tuple. The declarations for an **adt_pair** are shown in Figure 5. Here again, the standard use of macros masks the underlying structural implementation.

```
STRING_TYPE
string_create(s)
    STRING      s;
{
    STRING_TYPE     p;
    int         size;

    if(!string_init_done) string_ds_init();
    size = sizeof(STR_STRING_TYPE) + strlen(s) + 1;
    p = (STRING_TYPE)mem_man_varsize_allocate(string_tag,size);
    STNG_SIZE(p) = size;
    strcpy(STNG_VAL(p),s);

    return(p);
}
```

Figure 4: Creation of an Instance of **adt_string**

```
struct pair {
    MEM_TYPE left_element_fld;
    UTIL_TYPE left_util_fld;
    MEM_TYPE right_element_fld;
    UTIL_TYPE right_util_fld;
};
typedef struct pair STR_PAIR_TYPE, *PAIR_TYPE;
#define PAIR_NULL  (PAIR_TYPE)MEM_NULL
#define PAIR_LEFT_ELEMENT(p)   ((MEM_TYPE)(p)->left_element_fld)
#define PAIR_RIGHT_ELEMENT(p)  ((MEM_TYPE)(p)->right_element_fld)
#define PAIR_LEFT_UTIL(p)      ((p)->left_util_fld)
#define PAIR_RIGHT_UTIL(p)     ((p)->right_util_fld)
#define PAIR_IS_EMPTY(p)       (PAIR_LEFT_ELEMENT(p) == MEM_NULL &&
                                   PAIR_RIGHT_ELEMENT(p) == MEM_NULL)
```

Figure 5: Declarations for **adt_pair**

Note that in the declaration of the struct `pair`, there are more than two fields, i.e., more than just the `left_element_fld` and the `right_element_fld`. The two extra fields are referred to as utility fields, and indicate whether or not the corresponding component object should be destroyed in the process of destroying the pair itself. This idea of "destroyable or not" comes from the concept of "ownership". Specifically, certain aggregate ADTs will deal with managing data (i.e., objects) for some other application. If the ADT is just managing data, then it does not "own" the data itself, and therefore when it (in this case, the pair) is destroyed and its management activities end, it should not destroy the data that it was managing because some other application "owns" the previously managed data and needs to manipulate it more. If the destruction of the managing pair caused the destruction of the data it is managing, then the application requesting the management would not operate correctly because its data would have been destroyed. The field `left_util_fld` is a `BOOLEAN` which indicates whether or not the object held in the field `left_element_fld` is "owned" by this instance of an **adt_pair**, i.e., whether or not the object held in the field `left_element_fld` should be destroyed when this instance of an **adt_pair** is destroyed. The field `right_util_fld` is a `BOOLEAN` which indicates whether or not the object held in the field `right_element_fld` is "owned" by this instance of an **adt_pair**, i.e., whether or not the object held in the field `right_element_fld` should be destroyed when this instance of an **adt_pair** is destroyed.

This kind of utility "ownership" field is present in many of the aggregating ADTs, such as list and set. Often, there are additional separate access functions for inserting objects into aggregate ADTs, for which the component objects being managed should not be destroyed when the aggregate ADT itself is destroyed. These access functions will contain the string "`_ddd`" as part of the name of the access function. The **ddd** is meant to connote "don't destroy data". Thus, a call such as `pair_create_ddd(obj1,obj2)` creates an instance of an **adt_pair** in which `obj1` and `obj2` will not be destroyed when the managing pair itself is destroyed, whereas a call such as `pair_create(obj1,obj2)` creates an instance of an **adt_pair** in which `obj1` and `obj2` will be destroyed when the aggregating pair itself is destroyed. In the case of the **adt_pair** there also are access functions for selectively allowing either the right or left field to be destroyed or not.

The code for creating an **adt_pair** is shown in Figure 6. Notice that the type of the input parameters is `MEM_TYPE`. This is the generic type for an ADT of unknown class. Since the ADT class of the objects that will be placed in the pair at execution time cannot be known at compile time, `MEM_TYPE` is the only appropriate type for the input parameters. The constant `UTIL_CAN_DESTROY` is used to indicate that the object held in the corresponding field should be destroyed when the pair created here is destroyed. There is a corresponding `UTIL_CANT_DESTROY` constant. To show the variations possible for utility management, the code for `pair_create_ddd_left` is shown in Figure 7. Notice that the flag indicating whether or not a component object should be destroyed can be modified after creation of the aggregating object.

```
PAIR_TYPE
pair_create(left,right)
      MEM_TYPE left,right;
{
    PAIR_TYPE a;

    if (!pair_init_done) pair_ds_init();
    a = (PAIR_TYPE) mem_man_fixedsize_allocate(pair_tag);
    LEFT_ELEMENT(a) = left;
    RIGHT_ELEMENT(a) = right;
    util_set_destruction(&PAIR_LEFT_UTIL(a),UTIL_CAN_DESTROY);
    util_set_destruction(&PAIR_RIGHT_UTIL(a),UTIL_CAN_DESTROY);

    return(a);
}
```

Figure 6: Creation of an Instance of **adt_pair**

```
PAIR_TYPE
pair_create_ddd_left(left,right)
      MEM_TYPE left,right;
{
    PAIR_TYPE a;

    a = pair_create(left,right);
    util_set_destruction(&PAIR_LEFT_UTIL(a),UTIL_CANT_DESTROY);
    util_set_destruction(&PAIR_RIGHT_UTIL(a),UTIL_CAN_DESTROY);

    return(a);
}
```

Figure 7: Code for `pair_create_ddd_left`

The last ADT class to be addressed is the **adt_dll**. Declarations used in the definition of the **adt_dll** are shown in Figure 8. The **adt_dll** is based on the concept of a `list_node`. In fact, there is a separate ADT class for `list_node` with its own access functions. Its direct implementation will be suppressed here, and it will be assumed that the reader can infer the meaning of any access function that occurs in the code from the name of the access function itself.

In the declaration of a `list_node` there are three major fields: the `left_fld`, the `right_fld`, and the `data_fld`. The `left_fld` of a node points to the node to its left. Similarly for the `right_fld`. The `data_fld` points to the instance of the ADT being held at this position. There are two other minor fields. The `util_fld` is used to indicate ownership, and specifies whether or not the object pointed to by the `data_fld` of this node should be destroyed when this node is destroyed. The `up_fld` indicates the context in which this node resides. It is essentially a "parent" indicator. In most cases this field points to the header node of the **adt_dll** in which it resides.

An **adt_dll** implements a doubly linked list with a **header**. The empty **adt_dll** contains one and only one node, the header node; the `left_fld` and the `right_fld` point to the header node itself. The header node is distinguished from all other `list_nodes` in that the `data_fld` of the header node points to the header node itself.

The code for creating an instance of an empty **adt_dll** is shown in Figure 9. Notice that the implementation style here is exactly the same as the previous creation routines. However, in this case the corresponding object being created is the **empty** object. Also, note the introduction of the concept of a

```
struct  list_node {
        MEM_TYPE                up_fld;
        struct  list_node       *left_fld;
        struct  list_node       *right_fld;
        MEM_TYPE                data_fld;
        UTIL_TYPE               util_fld;
    };
typedef struct list_node        STR_LIST_NODE_TYPE, *LIST_NODE_TYPE;
typedef STR_LIST_NODE_TYPE      STR_DLL_TYPE, *DLL_TYPE;
#define DLL_NULL                (DLL_TYPE)PNT_LIST_NODE_NULL
#define DLL_LEFT(p)             ((LIST_NODE_TYPE)(p)-> left_fld)
#define DLL_RIGHT(p)            ((LIST_NODE_TYPE)(p)-> right_fld)
#define DLL_DATA(p)             ((MEM_TYPE)(p)-> data_fld)
#define DLL_PARENT(p)           ((MEM_TYPE)(p)-> up_fld)
#define DLL_UTIL(p)             ((MEM_TYPE)(p)-> util_fld)
#define DLL_IS_HEAD(p)          (DATA(p) == (MEM_TYPE)(p))
```

Figure 8: Declarations for **adt_dll**

```
DLL_TYPE
dll_create(parent)
    MEM_TYPE        parent;
{
    DLL_TYPE    p;

    if (!dll_init_done) dll_ds_init();
    p = (DLL_TYPE) mem_man_fixedsize_allocate(dll_tag);
    LEFT(p) = p;
    RIGHT(p) = p;
    DATA(p) = (MEM_TYPE)p;
    PARENT(p) = parent;

    return(p);
}
```

Figure 9: Creation of an empty **adt_dll**

"parent" to indicate the context in which the specific instance of the **adt_dll** resides.

2.2. Destroying

The concept of destroying an object deals with returning the memory, which was allocated for "holding" that object, to the system for future used by other instances of ADTs. Since no part of the application should have a pointer to this returned memory after deallocation, whatever variable is used to hold the pointer to this object (in order to return the memory to the system) should be NULLed out after the destroy has been completed. In order to help enforce this convention, a pointer to the pointer to the object is handed to the destroy function, and it is the responsibility of the destroy function to NULL out the pointer it received upon completion of the return of the memory.

Consider the destruction of an **adt_int**, as shown in Figure 10. Here, note the call to `mem_man_fixedsize_deallocate`. Since this is a fixed-size data structure, the system knows the amount of memory that was allocated and therefore how much to deallocate. Thus, there is no need to supply the data structure size for deallocation. Note that a pointer to the pointer to the data structure is passed to this routine. The pointer to the data structure is set to NULL, just before returning.

```
void
int_destroy(p)
      INT_TYPE     *p;
{
      if (*p == INT_NULL) return;

      mem_man_fixedsize_deallocate((MEM_TYPE)*p);
      *p = INT_NULL;

      return;
}
```

Figure 10: Destruction of an Instance of **adt_int**

When destroying a variable-size object, only slight differences occur. Consider the destroy of an **adt_string**, as shown in Figure 11. Notice here that there is a calculation of the size of the memory which must be returned and the call to `mem_man_varsize_deallocate` with a second argument which specifies that size.

Notice that there are no component ADTs to be destroyed in the **adt_int** and the **adt_string**. For an example of an aggregate ADT which has component ADTs which may have to be destroyed, consider the destroy of an **adt_pair**, as shown in Figure 12. Here, the routine determines ownership by checking to see if it should destroy its two component objects. Any component that should be destroyed is destroyed, and then the memory for the pair itself is returned to the system.

As the last example of a destroy routine, consider the destroy of an **adt_dll**, as shown in Figure 13. Here, each list node in the **adt_dll** is selected and destroyed. The check to determine ownership is "hidden" in the call to `list_node_destroy`. Notice that it is possible that some of the objects placed in the **adt_dll** may be destroyable and other may be nondestroyable. A selective decision is made at each `list_node` as to which component objects are destroyable and which are not. After each of the nodes in the list has been destroyed, the memory for the header node is returned to the system.

In the destruction of an **adt_pair** and an **adt_dll**, the macro `DESTROY` has been used. This macro interrogates the tag of the object handed to it to determine which specific destroy routine should be used to return the appropriate amount of memory to the system. It should now be clear why the execution-time tagged architecture is so important to the effective use of recursively embeddable ADTs. Without the

```
void
string_destroy(p)
      STRING_TYPE *p;
{
      int          size;

      if (*p == STNG_NULL) return;

      size = STNG_SIZE(*p);
      mem_man_varsize_deallocate((MEM_TYPE)*p,size);
      *p = STNG_NULL;

      return;
}
```

Figure 11: Destruction of an Instance of **adt_string**

```
void
pair_destroy(pair)
    PAIR_TYPE    *pair;
{
    if (*pair == PAIR_NULL) return;

    if (util_can_destroy(&PAIR_LEFT_UTIL(*pair)))
        if (LEFT_ELEMENT(*pair) != MEM_NULL)
            DESTROY(&LEFT_ELEMENT(*pair));
    if (util_can_destroy(&PAIR_RIGHT_UTIL(*pair)))
        if (RIGHT_ELEMENT(*pair) != MEM_NULL)
            DESTROY(&RIGHT_ELEMENT(*pair));
    mem_man_fixedsize_deallocate((MEM_TYPE)*pair);
    *pair = PAIR_NULL;

    return;
}
```

Figure 12: Destruction of an Instance of **adt_pair**

```
void
dll_destroy(dll)
    DLL_TYPE    *dll;
{
    LIST_NODE_TYPE  ln;

    if (*dll == DLL_NULL) return;

    ln = RIGHT(*dll);
    while(!DLL_IS_HEAD(ln)) {
        list_node_destroy(&ln);
        ln = RIGHT(*dll);
    };
    mem_man_fixedsize_deallocate((MEM_TYPE)*dll);
    *dll = DLL_NULL;

    return;
}
```

Figure 13: Destruction of an Instance of **adt_dll**

execution-time tag present, it would be impossible to correctly manipulate (in this case, deallocate memory) the constituent components of a complex aggregate ADT.

2.3. Pretty Printing

The objective of pretty printing is to have the logical content of an instance of an ADT printed in a high-level manner, from which the reader can quickly extract the content. For scalar objects, this is relatively straightforward. Consider the code for `int_pprint` as shown in Figure 14. Notice that the logic is extremely simple. The integer value is extracted and printed. No new-line is printed; this is because the pretty print of the integer may be done in a much larger context, that of an aggregate ADT. Extraneous new-lines may destroy the overall contextual effect desired. Here, the macro `RERROR` prints an error message. The first argument to the macro, `ABORT_CODE`, indicates that this is a severe error, and after printing the error message the run should be aborted.

```
void
int_pprint(p)
      INT_TYPE     p;
{
      int    i;

      if (p == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));

      i = INT_VAL(p);
      printf("%d",i);

      return;
}
```

Figure 14: Pretty Print of an Instance of **adt_int**

The pretty print of a variable-size object is not much different, as shown in the code for `string_pprint` in Figure 15.

The pretty print of an aggregate ADT requires some punctuation. In the case of the **adt_pair**, the two objects of the 2-tuple are place between parentheses, with a comma between them. The pretty print routine for an **adt_pair**, `pair_pprint`, is shown in Figure 16. Note the use of the macro `PPRINT` to perform the pretty print on the two components, the types of which are unknown. Notice again that no new-line is printed which might destroy the contextual effect.

The pretty print for the **adt_dll** is shown in Figure 17. Note the call to `list_node_pprint` to achieve the pretty printing of the object at each selected position of the list.

In order to see the utility of the tag architecture and the declaration of the required system-backbone routines, now consider the code shown in Figure 18(a), which produces a relatively complex ADT structure. This code first creates an **adt_dll** containing two elements, the string "abc" and the integer 1. It then creates an **adt_pair** containing this **adt_dll** as its first element and the integer 2 as its second element. It then creates a second **adt_dll**, whose first object is the pair just created and whose second object is the string "def". The resulting ADT is pretty printed, and the output produced is shown in Figure 18(b).

```
void
string_pprint(p)
      STRING_TYPE        p;
{
      STRING             s;

      if (p == STNG_NULL) RERROR(ABORT_CODE,("NULL pointer"));

      s = STNG_VAL(p);
      printf("\"%s\"",s);

      return;
}
```

Figure 15: Pretty Print of an Instance of **adt_string**

```
void
pair_pprint(pair)
    PAIR_TYPE   pair;
{
    if (pair == PAIR_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    printf("(");
    PPRINT(LEFT_ELEMENT(pair));
    printf(",");
    PPRINT(RIGHT_ELEMENT(pair));
    printf(")");

    return;
}
```

Figure 16: Pretty Print of an Instance of **adt_pair**

```
void
dll_pprint(dll)
    DLL_TYPE    dll;
{
    LIST_NODE_TYPE  ln;

    if (dll == DLL_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    ln = RIGHT(dll);
    printf("<");
    while (!DLL_IS_HEAD(ln)) {
        list_node_pprint(ln);
        ln = RIGHT(ln);
    };
    printf(">");

    return;
}
```

Figure 17: Pretty Print of an Instance of **adt_dll**

```
dll1 = dll_create(MEM_NULL);
dll_insert_data_to_right(dll1,dll1,int_create(1));
dll_insert_data_to_right(dll1,dll1,string_create("abc"));
pair = pair_create(dll1,int_create(2));
dll2 = dll_create(MEM_NULL);
dll_insert_data_to_right(dll2,dll2,string_create("def"));
dll_insert_data_to_right(dll2,dll2,pair);
dll_pprint(dll2);
```

(a)

```
<(<"abc",1>,2),"def">
```

(b)

Figure 18: Example Input and Output for Pretty Print

2.4. Debug Printing

The objective of debug printing is to have the implementation content of an instance of an ADT printed in a low-level manner, from which the reader can extract the details of the implementation. This is for the purpose of discovering errors in the implementation. The overall idea behind debug printing is similar to that of pretty printing except that all the implementation details, such as explicit pointers into memory, are displayed.

Consider the debug printing for an **adt_int**, as shown in Figure 19. Notice that the value of the integer stored as well as the pointer to the memory at which it is stored are printed. Similar code is used for debug printing an **adt_string**, as shown in Figure 20.

A debug print for a simple aggregate ADT such as an **adt_pair** has a similar structure to its pretty print counterpart, as shown in the code for `pair_dprint` presented in Figure 21. Notice the recursive use of the macro `DPRINT`.

In order to keep things simple for complex aggregate ADTs, such as an **adt_dll**, the unbounded complexity of the structure is suppressed, as shown in the code for `dll_dprint` presented in Figure 22. Here, only the fields of the header node are printed in detail. The reason for simplifying the debug printing of a complex ADT such as this is as follows. The basic reason for doing a debug print is to find an error;

```
void
int_dprint(p)
      INT_TYPE    p;
{
      int   i;

      if (p == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));

      i = INT_VAL(p);
      printf("(int%x(i:%d))",p,i);

      return;
}
```

Figure 19: Debug Print of an Instance of **adt_int**

```
void
string_dprint(p)
        STRING_TYPE        p;
{
        if (p == STRING_NULL) RERROR(ABORT_CODE,("NULL pointer"));

        printf("(str%x(sz:%d)(s:\"%s\"))",p,STNG_SIZE(p),STNG_VAL(p));

        return;
}
```

Figure 20: Debug Print of an Instance of **adt_string**

```
void
pair_dprint(pair)
    PAIR_TYPE    pair;
{
    if (pair == PAIR_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    printf("(");
    DPRINT(LEFT_ELEMENT(pair));
    printf(",");
    DPRINT(RIGHT_ELEMENT(pair));
    printf(")");

    return;
}
```

Figure 21: Debug Print of an Instance of **adt_pair**

```
void
dll_dprint(dll)
    DLL_TYPE     dll;
{
    if (dll == DLL_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    printf("(dll%x(u:%x)(l:%x)(r:%x))",dll,PARENT(dll),LEFT(dll),RIGHT(dll));

    return;
}
```

Figure 22: Debug Printing of an Instance of **adt_dll**

usually this error involves an erroneous pointer. The process of debug printing should not produce an error in its own right. If one of the pointers in a `list_node` is in error, then a debug print routine which follows that erroneous pointer may cause an execution error. In order to eliminate this kind of "random search" through memory, only the top-level header node is debug printed. During debugging, it is possible to "march down" the separate `list_nodes` of the **adt_dll** to discover the pointer error.

The code shown in Figure 23(a) produces the debug print output shown in Figure 23(b).

```
dll1 = dll_create(NULL);
int1 = int_create(1);
int_dprint(int1);
dll_insert_data_to_right(dll1,dll1,int1);
string1 = string_create("abc");
string_dprint(string1);
dll_insert_data_to_right(dll1,dll1,string1);
dll_dprint(dll1);
int2 = int_create(2);
int_dprint(int2);
pair = pair_create(dll1,int2);
pair_dprint(pair);
dll2 = dll_create(NULL);
string2 = string_create("def");
string_dprint(string2);
dll_insert_data_to_right(dll2,dll2,string2);
dll_insert_data_to_right(dll2,dll2,pair);
dll_dprint(dll2);
```

(a)

```
(int2198b1(i:1))
(str21988d(sz:8)(s:"abc"))
(dll219828(u:0)(l:2197e8)(r:219899))
(int2198f5(i:2))
((dll219828(u:0)(l:2197e8)(r:219899)),(int2198f5(i:2)))
(str2198d1(sz:8)(s:"def"))
(dll2198dd(u:0)(l:2198b9)(r:219909))
```

(b)

Figure 23: Example Input and Output for Debug Print

2.5. Copying

The objective of making a copy of an instance of an ADT is essentially the same as that of the assignment operator in most programming languages, i.e., to make a copy of the object so that the original can be modified without changing the copy. For scalar ADTs the copy routines are simple, as shown in Figures 24 and 25 for an **adt_int** and **adt_string**, respectively. Note the seemingly unnecessary appearance of an "extra" input parameter named `parent`. In fact, this parameter is not used in these two copy routines, because there is no parent field to be set. However, in more complex ADTs, there may be a parent field that has to be set when a newly created instance is being copied into its new context. In this case the parent input parameter is necessary. The reason that the `parent` parameter is present here where it is not needed is because all required system-backbone routines must have exactly the same calling protocol. In other words, every copy routine must have exactly two input parameters, of which the parent is first and the object to be copied is second. The COPY macro also takes exactly these two parameters. Without this kind of uniform calling convention, it is impossible to develop a recursive mechanism for handling all the different classes of ADTs.

The code for copying an aggregate ADT uses the same recursive concepts as present in the rest of the system. The code for copying an **adt_pair** is shown in Figure 26. Notice the continued use of the first parameter, which in this case is called `dummy`, since it is known that the incoming value will not be used.

```
INT_TYPE
int_copy(parent,in)
      MEM_TYPE    parent;
      INT_TYPE    in;
{
      INT_TYPE    out;

      if (in == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));

      out = (INT_TYPE)mem_man_fixedsize_allocate(int_tag);
      INT_VAL(out) = INT_VAL(in);

      return(out);
}
```

Figure 24: Copying an Instance of **adt_int**

```
STRING_TYPE
string_copy(parent,in)
      MEM_TYPE          parent;
      STRING_TYPE       in;
{
      STRING_TYPE       out;

      if (in == STNG_NULL) RERROR(ABORT_CODE,("NULL pointer"));

      out=(STRING_TYPE)mem_man_varsize_allocate(string_tag,STNG_SIZE(in));
      STNG_SIZE(out) = STNG_SIZE(in);
      strcpy(STNG_VAL(out),STNG_VAL(in));

      return(out);
}
```

Figure 25: Copying an Instance of **adt_string**

```
PAIR_TYPE
pair_copy(dummy,pair_in)
    MEM_TYPE    dummy;
    PAIR_TYPE   pair_in;
{
    PAIR_TYPE   pair_out;

    if (pair_in == PAIR_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    pair_out = pair_create(COPY(MEM_NULL,LEFT_ELEMENT(pair_in)),
                     COPY(MEM_NULL,RIGHT_ELEMENT(pair_in)));

    return(pair_out);
}
```

Figure 25: Copying an Instance of **adt_pair**

The copy routine for a more complex aggregate ADT, an **adt_dll**, is shown in Figure 26. Note that here the `parent` parameter is used, because there is a parent field present in the ADT itself. It should now be clear why the uniform calling protocol for all copy routines is necessary. Also note the recursive

```
DLL_TYPE
dll_copy(parent,dll_in)
    MEM_TYPE    parent;
    DLL_TYPE    dll_in;
{
    DLL_TYPE        dll_out;
    LIST_NODE_TYPE  ln_in;
    LIST_NODE_TYPE  ln_out;

    if (dll_in == DLL_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    dll_out = dll_create(MEM_NULL);
    PARENT(dll_out) = parent;
    ln_in = RIGHT(dll_in);
    while (!DLL_IS_HEAD(ln_in)) {
        ln_out = list_node_copy((MEM_TYPE)dll_out,ln_in);
        list_node_insert_node_to_right((MEM_TYPE)dll_out,
            LEFT(dll_out),ln_out);
        ln_in = RIGHT(ln_in);
    };

    return(dll_out);
}
```

Figure 26: Copying an Instance of **adt_dll**

call to `list_node_copy` as the objects of the original input **adt_dll** are scanned and copied.

2.6. Comparing

For each ADT, a compare function must be supplied. This compare function is required to take two instances of the same ADT class and determine whether the first is less than, equal to, or greater than the second. Note that the creator of the new ADT class need only produce the compare function for two instances of the new ADT class being created. There is a macro, COMPARE, with extended logic which takes two instances of ADTs (from potentially different ADT classes) and determines their order. This COMPARE macro constitutes the computation of a **total order** of all instances of all ADT classes that reside in the system. Its logic will be presented in the discussion of aggregate ADTs.

The compare function for the ADTs can be used in a number of ways. The COMPARE macro is actually used internally in the implementation of some of the ADTs themselves. For instance the current implementation of **adt_set** uses the COMPARE macro to essentially sort the objects that are in the instance of an **adt_set**. Also, it is sometimes useful to sort objects to facilitate subsequent processing; but the criteria for the sort is unimportant. The generic COMPARE function, since it computes a total order on all instances of ADTs, can be used in such a generic sort.

The compare functions for **adt_int** and **adt_string** are given in Figures 27 and 28, respectively. Here, the type `COMPARE_TYPE` is an `enum` containing three values: `less_than_code`, `equal_code`, and `greater_than_code`. The routine `system_int_compare` takes a single integer value as input and returns: `less_than_code` if the integer is less than 0, `equal_code` if the integer is equal to 0, and `greater_than_code` if the integer is greater than 0. In this situation, a specific ordering of the instances of ADTs is specified by the code shown. For instance, for **adt_int** `less_than_code` is returned if the first parameter is less than the second parameter. However, the code can be changed to reflect the reverse ordering and the system will still operate as desired. The total ordering of all instances of ADTs is preserved as long as each separate compare function maintains a total

```
COMPARE_TYPE
int_compare(p1,p2)
      INT_TYPE    p1;
      INT_TYPE    p2;
{
      COMPARE_TYPE      ans;

      if (p1 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
      if (p2 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));

      ans = system_int_compare(INT_VAL(p1) - INT_VAL(p2));

      return(ans);
}
```

Figure 27: Comparing Instances of **adt_int**

```
COMPARE_TYPE
string_compare(p1,p2)
      STRING_TYPE p1,p2;
{
      COMPARE_TYPE      ans;

      if (p1 == STNG_NULL) RERROR(ABORT_CODE,("NULL pointer"));
      if (p2 == STNG_NULL) RERROR(ABORT_CODE,("NULL pointer"));

      ans = system_int_compare(strcmp(STNG_VAL(p1),STNG_VAL(p2)));

      return(ans);
}
```

Figure 28: Comparing Instances of **adt_string**

ordering within its single ADT class.

As a simple example of how to write a compare function for an aggregate ADT, consider the function `pair_compare` shown in Figure 29. Here, the compare is biased toward the first component of the 2-tuple. If the first component of the first parameter is less than the first component of the second parameter, then the first parameter is declared to be less than the second parameter. Similarly, if the first component of the first parameter is greater than the first component of the second parameter, then the first parameter is declared to be greater than the second parameter. However, if the first component of each of the parameters is equal, then the second component must be interrogated to resolve the apparent tie. If the second component of the first parameter is less than the second component of the second parameter, then the first parameter is declared to be less than the second parameter. Similarly, if the second component of the first parameter is greater than the second component of the second parameter, then the first parameter is declared to be greater than the second parameter. However, if the second component of each of the parameters is equal, then the two parameters must be equal, because both of their components are equal.

Note the use of the macro COMPARE, which allows comparisons between instances of objects from different ADT classes. COMPARE works in the following simple way. Each ADT class has a unique tag associated with it. COMPARE compares the tags of the two instances of ADTs which are its input parameters. If the tags are different, the instance with the numerically smaller tag is declared to be less than the instance with the larger tag. If the tags are identical, then the instances are from the same ADT class, and the appropriate (unique) compare function is "looked up" and invoked to compare the two instances.

```
COMPARE_TYPE
pair_compare(pair1,pair2)
    PAIR_TYPE   pair1;
    PAIR_TYPE   pair2;
{
    COMPARE_TYPE        ans;

    if (pair1 == PAIR_NULL) RERROR(ABORT_CODE,("NULL pointer"));
    if (pair2 == PAIR_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    ans = COMPARE(LEFT_ELEMENT(pair1),LEFT_ELEMENT(pair2));
    if (ans == equal_code)
        ans = COMPARE(RIGHT_ELEMENT(pair1),RIGHT_ELEMENT(pair2));

    return(ans);
}
```

Figure 29: Comparing Instances of **adt_pair**

The compare function for an **adt_dll** is a bit more complicated, but retains the same idea as that used for the **adt_pair**. It is shown in Figure 30. Here, the logic has one more level of complexity. In this case, first the cardinalities of the two lists are compared. If the cardinalities are different, then the one with the smaller cardinality is declared to be less than the one with the larger cardinality. If the cardinalities are identical, then the next level of test is applied. The corresponding objects in the two lists are compared starting at the beginning of each list. If the first objects in the two lists are not equal, then the declared

```
COMPARE_TYPE
dll_compare(dll1,dll2)
    DLL_TYPE    dll1;
    DLL_TYPE    dll2;
{
    LIST_NODE_TYPE  ln1;
    LIST_NODE_TYPE  ln2;
    COMPARE_TYPE        ans;

    if (dll1 == DLL_NULL) RERROR(ABORT_CODE,("NULL pointer"));
    if (dll2 == DLL_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    ans = system_int_compare(dll_cardinality(dll1) - dll_cardinality(dll2));
    if (ans == equal_code) {
        ln1 = RIGHT(dll1);
        ln2 = RIGHT(dll2);
        while((ans == equal_code) && !DLL_IS_HEAD(ln1)) {
            ans = COMPARE(ln1,ln2);
            ln1 = RIGHT(ln1);
            ln2 = RIGHT(ln2);
        };
    };

    return(ans);
}
```

Figure 30: Comparing Instances of **adt_dll**

order of the two lists is determined by the order of the first two objects. If the first objects in the two lists are equal, then attention is focused on the second objects in the two lists. The order of the two lists is determined by discovering the first set of corresponding objects in the two lists which are not equal. If all the corresponding objects in the list turn out to be equal, then the two lists themselves are declared to be equal.

3. Object-operation Access Functions

The access functions presented so far constitute the required system-backbone access function which must be supplied for every ADT class. These access functions are used for managing the objects, but have nothing to do with the operations on the abstract data types for which they were created in the first place. For instance, a stack ADT might have a push operation and a pop operation associated with it.

Once the basic ADT class has been created and the system-backbone access functions produced, the creator of the ADT can supply any number of object-operations on the ADT desired. For example, for the **adt_int**, the four basic arithmetic functions are shown in Figure 31. Obviously, more such access functions can be added for other arithmetic operations, such as **mod** and **exp**.

The concatenation function is shown for **adt_string** in Figure 32.

Three object-operation access function that might be useful for **adt_pair** are shown in Figure 33.

4. Management of ADTs

This section discusses how ADTs are managed within the DNA Mapping system. This includes how ADT classes are declared to the system, how the system maintains statistics about ADT instance utilization and reports this to the user, and how internal memory management is performed.

4.1. Declaring an ADT Class

Each ADT class used within the system must be declared to the system at execution time, usually done at initialization of the entire DNA Mapping system. This is done by invoking a number of routines associated with the memory management subsystem. Consider the declaration of **adt_int**, as shown in Figure 34. The first executable statement simply insures that initialization is done no more than once. The three routines used to declare the properties of an ADT class are: `mem_man_declare`, `mem_man_set_destroy_function`, and `mem_man_set_copy_function`.

The first argument to `mem_man_declare` simply supplies a character string name for identification purposes during printing of tables. The second argument is an output parameter through which the system-created tag for the specific ADT class is returned. The third argument specifies the size of (i.e., the number of bytes of memory required) an instance of the ADT. If the ADT class is of fixed-size for all instances, then a positive integer is input; if the ADT class has variable-size instances, then a special code is used for this argument to indicate that it is of variable length. The next two arguments supply a pointer to the debug print and pretty print routines. The next two arguments supply a pointer to the create and compare routines. The last argument specifies the encodement used for the null pointer for the specific ADT class. Currently, all of these null pointers are encoded as 0.

The `mem_man_set_destroy_function` sets the destroy function for the specific ADT class. The `mem_man_set_copy_function` sets the copy function for the specific ADT class.

```
INT_TYPE
int_add(p1,p2)
        INT_TYPE        p1;
        INT_TYPE        p2;
{
        INT_TYPE        ans;
        if (p1 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
        if (p2 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
        ans = int_create(INT_VAL(p1) + INT_VAL(p2));
        return(ans);
}

INT_TYPE
int_sub(p1,p2)
        INT_TYPE        p1;
        INT_TYPE        p2;
        INT_TYPE        ans;
        if (p1 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
        if (p2 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
        ans = int_create(INT_VAL(p1) - INT_VAL(p2));
        return(ans);
}

INT_TYPE
int_mult(p1,p2)
        INT_TYPE        p1;
        INT_TYPE        p2;
{
        INT_TYPE        ans;
        if (p1 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
        if (p2 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
        ans = int_create(INT_VAL(p1) * INT_VAL(p2));
        return(ans);
}

INT_TYPE
int_div(p1,p2)
        INT_TYPE        p1;
        INT_TYPE        p2;
{
        INT_TYPE        ans;
        int              i1;
        int              i2;
        if (p1 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
        if (p2 == INT_NULL) RERROR(ABORT_CODE,("NULL pointer"));
        i1 = INT_VAL(p1);
        i2 = INT_VAL(p2);
        if (i2 == 0) RERROR(ABORT_CODE,("Attempt to divide by zero"));
        ans = int_create(i1/i2);
        return(ans);
}
```

Figure 31: Arithmetic functions for **adt_int**

```
STRING_TYPE
string_concat(s1,s2)
    STRING_TYPE   s1;
    STRING_TYPE   s2;
{
    STRING_TYPE     p;
    int             size;

      if (s1 == STRING_NULL) RERROR(ABORT_CODE,("NULL pointer"));
      if (s2 == STRING_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    size = sizeof(STR_STRING_TYPE)
           + strlen(STNG_VAL(s1))+strlen(STNG_VAL(s2))+1;
    p = (STRING_TYPE)mem_man_varsize_allocate(string_tag,size);
    STNG_SIZE(p) = size;
    strcpy(STNG_VAL(p),STNG_VAL(s1));
    strcat(STNG_VAL(p),STNG_VAL(s2));

    return(p);
}
```

Figure 32: Concatenation for **adt_string**

```
MEM_TYPE
pair_get_left_element(pair)
    PAIR_TYPE           pair;
{
    MEM_TYPE            ans;

    if (pair == PAIR_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    ans = LEFT_ELEMENT(pair);

    return(ans);
}

void
pair_set_left_element(pair,data)
    PAIR_TYPE           pair;
    MEM_TYPE            data;
{
    if (pair == PAIR_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    LEFT_ELEMENT(pair) = data;
    util_set_destruction(&PAIR_LEFT_UTIL(pair),UTIL_CAN_DESTROY);

    return;
}

PAIR_TYPE
pair_create_by_swap(p)
    PAIR_TYPE   p;
{
    PAIR_TYPE   newp;

    if (p == PAIR_NULL) RERROR(ABORT_CODE,("NULL pointer"));

    newp = pair_create_ddd(pair_get_right_element(p),
                      pair_get_left_element(p));

    return(newp);
}
```

Figure 33: Object-operations for **adt_pair**

```
void
int_ds_init()
{
	if (int_init_done) return;
	if (int_tag == NULL_TAG) {
		mem_man_declare("int_ds",&int_tag,sizeof(STR_INT_TYPE),
				int_dprint,int_pprint,(MFUNC_TYPE)
				int_create,int_compare,(NULL_TYPE)INT_NULL);
	};
	mem_man_set_destroy_function(int_tag,int_destroy);
	mem_man_set_copy_function(int_tag,(MFUNC_TYPE)int_copy);
	int_init_done = TRUE;

	return;
}
```

Figure 34: Declaration of **adt_int**

The memory management system accepts all of these parameters and places them in a table, the entries of which will be used during the remainder of the execution. This table is implemented as an array of a struct. The structure of the struct is shown in Figure 35. All of the fields present in the actual DNA Mapping software are not present here; only those which are pertinent to the discussion at hand have been included. The first nine fields included in this struct have already been discussed in the previous few paragraphs. The remainder will be introduced in subsequent subsections.

The declaration for a variable-size ADT class is very similar, as shown in Figure 36 for **adt_string.** Note that there is only one slight change to the scheme; the third argument to `mem_man_declare` is `VAR_SIZE_DS_CODE`, which stands for "variable size data structure code". The value of this constant is -1, and indicates to the system that this is a variable-size ADT class.

The declarations for **adt_pair** and **adt_dll** are very similar, and are shown in Figures 37 and 38.

```
struct adt_tab_entry{
	STRING				name_fld;
	int				size_fld;
	VFUNC_TYPE			dprint_fld;
	VFUNC_TYPE			pprint_fld;
	NULL_TYPE			null_pnt_fld;
	MFUNC_TYPE			create_fld;
	VFUNC_TYPE			destroy_fld;
	CFUNC_TYPE			compare_fld;
	MFUNC_TYPE			copy_fld;
	int				total_obj_fld;
	int				max_obj_fld;
	int				cur_obj_fld;
	PNT_AVAIL_TAB_ENTRY_TYPE	avail_tab_fld;
    };
```

Figure 35: Memory Management Table Structure

```
void
string_ds_init()
{
	if (string_init_done) return;
	if (string_tag == NULL_TAG) {
		mem_man_declare("string_ds",&string_tag,VAR_SIZE_DS_CODE,
				string_dprint,string_pprint,
				(MFUNC_TYPE)string_create,
				string_compare,
				(NULL_TYPE)STRING_NULL);
	};
	mem_man_set_destroy_function(string_tag,string_destroy);
	mem_man_set_copy_function(string_tag,(MFUNC_TYPE)string_copy);
	string_init_done = TRUE;

	return;
}
```

Figure 36: Declaration of **adt_string**

```
void
pair_ds_init()
{
    if(pair_init_done) return;
    if (pair_tag == NULL_TAG) {
        mem_man_declare("pair_ds",&pair_tag,
                sizeof(STR_PAIR_TYPE),
                pair_dprint,pair_pprint,pair_create,
                pair_compare,(NULL_TYPE)PAIR_NULL);
    };
    mem_man_set_destroy_function(pair_tag,pair_destroy);
    mem_man_set_copy_function(pair_tag,(MFUNC_TYPE)pair_copy);
    pair_init_done = TRUE;

    return;
}
```

Figure 37: Declaration of **adt_pair**

```
void
dll_ds_init()
{
    if(dll_init_done) return;
    if (dll_tag == NULL_TAG) {
        mem_man_declare("dll_ds",&dll_tag,sizeof(STR_DLL_TYPE),
                dll_dprint,dll_pprint,dll_create,
                dll_compare,(NULL_TYPE)DLL_NULL);
    };
    mem_man_set_destroy_function(dll_tag,dll_destroy);
    mem_man_set_copy_function(dll_tag,(MFUNC_TYPE)dll_copy);
    list_node_ds_init();
    dll_init_done = TRUE;

    return;
}
```

Figure 38: Declaration of **adt_dll**

4.2. Statistics on Instances of ADTs

The user can print a table which presents statistics about the current status of the ADTs. An example of such a table is shown in Figure 39. The last column indicates the name of the ADT class as specified during its declaration; it is included as a reference to externally name the ADT class. The first column indicates how many instances of the specific ADT class are currently allocated. The value printed here is extracted from the `cur_obj_fld` field of the internal ADT table (cf. Figure 35). This field is incremented every time an instance of the specific ADT class is created and decremented every time an instance of the specific ADT class is destroyed. The second column indicates the maximum number of instances of the ADT class that have ever been allocated an one specific time; it is essentially a "high water mark" indicating how much of a resource bottleneck this particular ADT class represents. The value printed here is extracted from the `max_obj_fld` field of the internal ADT table (cf. Figure 35). This field is modified every time an instance of the specific ADT class is created to calculate the current maximum. The third column indicates how many instances of the specific ADT class have ever been allocated. The value printed here is extracted from the `total_obj_fld` field of the internal ADT table (cf. Figure 35). This field is incremented every time an instance of the specific ADT class is created; it is never decremented. The fourth column indicates the (user-specified) size, in bytes, of the memory required to hold an instance of the ADT. Notice that several ADT classes have a -1 in this column, indicating that they are variable-size ADTs. The fifth column indicates the actual tag used internally for the specific ADT class. In fact, this tag is a pointer (in memory) to the entry in the internal ADT table for this specific ADT class. In other words, the tag is actually a pointer to a record (i.e., struct) which contains all the functions needed to access a specific instance of the ADT class and all the data fields necessary to maintain running statistics about the ADT class as a whole.

Notice that there are a number of lines at the bottom of the table which refer to the amount of memory which has been allocated from the operating system. The first line indicates the total memory that has been allocated from the operating system. This includes more than just ADT memory; certain DNA Mapping system tables are dynamically allocated, but are not built by using ADTs. The second line indicates how much memory has been allocated for use by ADTs. Specifically, the DNA Mapping system does its own memory allocation and management. Large blocks of memory (by default set at a size of 1K) are allocated from the operating system, and memory for individual instances of ADTs is allocated from these blocks. No memory is ever returned to the operating system. The third line indicates how large a block will be allocated the next time a block of memory is needed from the operating system. The size of the block allocated from the operating system can be controlled by invocation of the function `mem_man_numK`. The fourth line indicates how much memory is currently available in the last block that was allocated. The last line indicates how much miscellaneous memory has been allocated. (Miscellaneous memory is a special class of memory which cannot be returned even internally to the DNA Mapping

current # used	max # used	ADT TAB total # used	size	tag	name
0	166	779	8	216488	langwrap_ds
0	0	0	-1	210030	bipart_ds
0	0	0	12	20ffd8	bigraph_ds
0	0	0	16	20ff80	digraph_ds
0	0	0	16	20ff28	edge_ds
0	0	0	16	20fed0	vertex_ds
0	0	0	-1	20fe78	perm_ds
0	1	17	-1	20fe20	marray_ds
0	0	0	16	20fdc8	hashtab_ds
0	0	0	4	20fd70	interval_ds
0	0	0	16	20fd18	rlist_ds
0	0	0	12	20fcc0	bag_ds
0	1	234	-1	20fc68	combo_ds
0	7	69	-1	20fc10	bitv_ds
0	0	0	20	20fbb8	focus_ds
0	0	0	4	20fb60	point_ds
0	0	0	4	20fb08	line_ds
0	379	4649	12	20fab0	prop_ds
0	6	102	16	20fa58	window_ds
0	253	2171	16	20fa00	vfrag_ds
0	553	3551	16	20f9a8	rfrag_ds
0	0	0	16	20f950	heap_ds
0	1	43	-1	20f8f8	odometer_ds
0	57	5291	16	20f8a0	pair_ds
0	671	5784	32	20f848	tree_ds
0	0	0	16	20f7f0	socket_ds
0	1	164	8	20f798	stk_ds
0	998	28829	20	20f740	list_ds
0	672	11008	8	20f6e8	set_ds
0	998	30067	20	20f690	dll_ds
0	1061	31988	20	20f638	list_node_ds
0	72	4742	4	20f5e0	float_ds
0	0	0	1	20f588	bool_ds
0	307	4456	-1	20f530	string_ds
0	4	93	4	20f4d8	int_ds
12	12	12	-1	20f480	misc_mem

```
Total memory allocated is 140248 bytes.
Total ADT memory allocated is 137216 bytes.
Current block allocation size is 1K bytes.
936 bytes in current block.
53 bytes of misc memory.
```

Figure 39: Statistics about ADTs and Memory Management

system.)

The data in this table can be used to determine the memory resources used in any specific computational activity. This table is extremely useful in the process of "zeroing out" memory. At the end of a computation, the "currently allocated" column of this table should be 0 for all ADT classes, except miscellaneous memory. If a particular entry is not 0, then instances of that ADT class are still allocated, and the application has failed to deallocate (destroy) unused objects. Great care must be taken to destroy all instances of ADTs which are no longer useful. Failure to do so can cause memory exhaustion and cause execution of an otherwise correctly executing program to abort.

4.3. Memory Management

The DNA Mapping system does its own memory management internally. No calls to `malloc`, `calloc`, or `realloc` are present; thus, the use of `free` is not possible. Instead, all allocation of memory from the operating system is done through invocations to the system routine `sbrk`. (The application programmer should not use `malloc` or any of its variations, because `sbrk` and `malloc` are incompatible with each other.) Large blocks of memory (by default set at a size of 1K) are allocated from the operating system, and memory for individual instances of ADTs is allocated from these blocks. The size of the block allocated from the operating system can be controlled by invocation of the function `mem_man_numK`. No memory is ever returned to the operating system.

Memory nodes which are returned to the system for future use are stacked in **avail lists** for future reallocation. There is a separate avail list for each individual node size which has been returned to the system. An array of headers for these avail lists is maintained.

4.3.1. Allocation

As a request for a memory node of a specific size is processed, the system first checks to see if there is an avail list for this size node and whether there is a memory node currently available. If a memory node is available, it is extracted from the avail list and returned to whoever requested the memory. If there is no such node available in an appropriate avail list, the current memory block is checked to determine if there is enough memory left in the block to satisfy the request. If there is enough memory present, the correct amount is extracted from the current memory block and returned to whoever requested the memory. If there is not enough memory present in the current memory block, a new block of memory is allocated from the operating system and the required amount of memory is extracted from it. This is the basic logic as expressed in `mem_man_get_mem`, as shown in Figure 40. Here, the two input parameters are (a) the tag of the object for which memory is being requested and (b) a pointer to an avail list header where memory nodes of exactly the correct size may be present.

The first executable statement computes the actual size of the memory needed, including the memory for the tag. The second statement, invoking `avail_get`, attempts to allocate memory from the appropriate avail list. The rest of the code is applicable only if memory could not be extracted from the avail list. `next_memory` is a pointer into the current memory block where the next (currently unallocated) available byte of memory will be found. `last_memory` is a pointer to the next byte after the current memory block. `numK_to_sbrk` is a global variable which indicates how much memory to allocate from the operating system; it can be modified by invoking `mem_man_numK`. The routine `mem_alloc` invokes `sbrk` to get the next memory block from the operating system. The routine `mem_man_cannot_alloc` reports memory exhaustion and offers the user several interactive options.

The routine `mem_man_get_mem` is invoked by `mem_man_fixedsize_allocate` and `mem_man_varsize_allocate` as shown in Figures 41 and 42, respectively. In `mem_man_fixedsize_allocate` the macro `AVP_FIELD` extracts the appropriate avail list header from the internal ADT table. This is the last field, `avail_tab_fld`, of the struct shown in Figure 35. For fixed-size ADT classes, this field is initialized at declaration time to point to an avail list with appropriately sized memory nodes. This size is computed by adding the user-specified data structure size to the size of a tag.

Memory is allocated and a pointer to it resides in the variable `ans`. The value of the tag is inserted at the beginning of the memory, and the variable `ans` is incremented by the length of a tag so that `ans` now points to the user-specified portion of the memory which has been allocated. This is the value returned in the last executable statement. The three executable statements just prior to the return update the

```
MEM_TYPE
mem_man_get_mem(tag,av)
    TAG_TYPE                    tag;
    PNT_AVAIL_TAB_ENTRY_TYPE    av;
{
    int         size;
    MEM_TYPE    ans;
    int         mem_to_allocate;

    size = SIZE_FIELD(av);
    ans = avail_get(av);        /* try to get memory for avail list */
    if (ans == MEM_NULL) {      /* try to do block allocation */
        if ((next_memory==MEM_NULL)||(next_memory+size>last_memory)) {
            mem_to_allocate = MAX(size,numK_to_sbrk*1024);
            if ((next_memory = mem_alloc(mem_to_allocate))
                != MEM_FAIL_CODE) {
                    total_adt_mem += mem_to_allocate;
                    last_memory = next_memory + mem_to_allocate;
                    ans = next_memory;
                    next_memory += size;
            }
            else {
                mem_man_cannot_alloc(tag);
            };
        }
        else {
            ans = next_memory;
            next_memory += size;
        };
    };

    return(ans);
}
```

Figure 40: Code for mem_man_get_mem

```
MEM_TYPE
mem_man_fixedsize_allocate(tag)
    TAG_TYPE    tag;
{
    MEM_TYPE        ans;

            /* get some memory */
    ans = mem_man_get_mem(tag,AVP_FIELD(tag));
            /* insert the tag and increment to user portion */
    *((PNT_TAG_TYPE)ans)++ = tag;
            /* increment # of nodes allocated */
    CURO_FIELD(tag)++;
    TOTALO_FIELD(tag)++;
            /* calculate max */
    MAXO_FIELD(tag) = MAX(CURO_FIELD(tag), MAXO_FIELD(tag));

    return(ans);
}
```

Figure 41: Code for mem_man_fixedsize_allocate

appropriate statistics about (a) the current number of instances allocated, (b) the total number of instance allocated, and (c) the maximum number of instances allocated.

The code for mem_man_varsize_allocate is very similar. There are only two differences. First, the (user-specified) size is specified as an input parameter. Second, in the call to mem_man_get_mem, the appropriate avail list header must be computed. This cannot be preprocessed at declaration time, as with a fixed-size ADT class, because the size of each instance cannot be known at declaration time. The macro ACT_SIZE (i.e., actual size) takes its input parameter and adds the size of a tag. The routine avail_tab_lookup searches the array of avail list headers for one corresponding to memory nodes of the appropriate size; if none is present, one is inserted. The rest of the code is identical to

```
MEM_TYPE
mem_man_varsize_allocate(tag,size)
    TAG_TYPE    tag;
    int         size;
{
    PNT_ADT_TAB_ENTRY_TYPE  p;
    MEM_TYPE        ans;

            /* get some memory */
    ans = mem_man_get_mem(tag,avail_tab_lookup(ACT_SIZE(size)));
            /* insert the tag and increment to user portion */
    *((PNT_TAG_TYPE)ans)++ = tag;
            /* increment # of nodes allocated */
    CURO_FIELD(tag)++;
    TOTALO_FIELD(tag)++;
            /* calculate max */
    MAXO_FIELD(tag) = MAX(CURO_FIELD(tag), MAXO_FIELD(tag));

    return(ans);
}
```

Figure 42: Code for mem_man_varsize_allocate

mem_man_fixedsize_allocate.

4.3.2. Deallocation

As instances of ADTs are destroyed, their memory is deallocated or returned by calling one of two routines: mem_man_fixedsize_deallocate and mem_man_varsize_deallocate. The code for mem_man_fixedsize_deallocate is shown in Figure 43. The logic for this routine is somewhat complicated by the fact that instances of both fixed-size and variable-size ADTs can be returned through the use of this routine. (Recall that the convention for variable-size ADTs is to make the first field of the user-specified data structure a field which hold the size of the memory which was allocated.) The TAG_OF macro "looks up" the tag of the instance of the ADT which was input, node. (Once the tag is known, the size of the ADT can be determined. If the size is greater than zero, the ADT is a fixed-size ADT; if the size is -1, the ADT is a variable-size ADT.) The SIZE_FIELD macro extracts the field size_fld from the internal ADT table (cf. Figure 35) to determine whether it is a fixed-size ADT or a variable-size ADT. If size is greater than 0 (i.e., a fixed-size ADT), then the macro AVP_FIELD extracts the appropriate avail list header which was determined during the declaration of the ADT class. Otherwise this is a variable-size ADT and the appropriate avail list header must be computed. The SIZE_OF macro looks in the first word of the (user-specified) data structure to extract the size of the memory present. To this, the macro ACT_SIZE adds the size of the tag, and the routine avail_tab_lookup searches the array of avail list headers for the one which corresponds to the appropriate size; a header is inserted if one of appropriate size is not present. Some statistics bookkeeping is done. The variable node which originally pointed to the user-specified portion of the data structure is decremented to point to the tag which occurs just to the left. This memory pointer is then inserted into the appropriate avail list, as specified by the avail list header which was previously computed.

The logic of mem_man_varsize_deallocate, as shown in Figure 44, is much simpler. Here, the tag is "looked up", and the appropriate statistics bookkeeping is done. The correct avail list header is computed, and the memory is inserted into the appropriate avail list for potential future use.

```
void
mem_man_fixedsize_deallocate(node)
    MEM_TYPE    node;
{
    TAG_TYPE                        tag;
    PNT_AVAIL_TAB_ENTRY_TYPE        avl;
    int                             size;

    tag = TAG_OF(node);
            /* determine size and check */
    size = SIZE_FIELD(tag);
    if (size > 0) avl = AVP_FIELD(tag);
    else avl = avail_tab_lookup(ACT_SIZE(SIZE_OF(node)));
            /* decrement # of allocated nodes */
    CURO_FIELD(tag)--;
            /* move back to beginning of memory */
    --((PNT_TAG_TYPE)node);
            /* put memory on avail list */
    avail_put(avl,node);

    return;
}
```

Figure 43: Code for mem_man_fixedsize_deallocate

```
void
mem_man_varsize_deallocate(node,size)
    MEM_TYPE    node;
    int     size;
{
    TAG_TYPE        tag;
    PNT_ADT_TAB_ENTRY_TYPE  p;

    tag = TAG_OF(node);
            /* decrement # of allocated nodes */
    CURO_FIELD(tag)--;
            /* move back to beginning of memory */
    --((PNT_TAG_TYPE)node);
            /* put memory on avail list */
    avail_put(avail_tab_lookup(ACT_SIZE(size)),node);

    return;
}
```

Figure 44: Code for `mem_man_varsize_deallocate`

5. System ADTs

At this point, only four ADTs have been discussed in detail: **adt_int**, **adt_string**, **adt_pair**, and **adt_dll**. (**adt_list_node** also has been mentioned in passing.) In this section, a number of other ADT classes present in the DNA Mapping system are discussed briefly. No code is presented, and only basic ideas and intent are discussed. The ADT classes are subdivided into three categories: (a) very general ADTs that might be included in almost any application, (b) general ADTs of a less universal nature, and (c) ADTs specific to the DNA Mapping project itself.

5.1. General Purpose ADTs

In this section, a number of generic ADTs that might occur in almost any application are presented briefly.

Two standard scalar data types are Boolean and float. There are two ADTs, **adt_bool** and **adt_float**, which encapsulate these scalars. It should be obvious what their intent is, and they will not be discussed further.

A very important generic ADT is **adt_list** which is an extension of **adt_dll**. In fact, **adt_list** is built on top of **adt_dll** (i.e., uses **adt_dll** as a component in its implementation). **adt_list** implements an aggregate which can be thought of dually (i.e., concurrently) as either a linked list or an array. For instance, it is possible to ask for the 9th object in the list, or to request that an object be inserted after the 9th object in the list. Of course, it still is possible to extract and insert data at either end of the list. **adt_list** also has one other important new feature, the concept of a **current** position. This gives **adt_list** the property of having a **state**. For instance, it is possible to place the current pointer at the beginning of a list and incrementally "march through" the list by advancing the current pointer. There are many access functions available for **adt_list**, such as concatenation of lists, sorting of a list, reversing a list, etc. There are currently more than 100 access functions and 50 access macros defined on **adt_list**.

adt_set is another interesting ADT. It implements the abstract mathematical concept of a set, in which no duplicates are allowed. **adt_set** is built on top of **adt_list**. In order to make operations such as union, intersection, and membership efficient, the elements of a set are sorted (inside an **adt_list**) so that searching and merging can be done efficiently. Here, the macro COMPARE is used to determine the order of the ADT instances.

The concept of a stack is achieved through the introduction of **adt_stk**. This also is built on top of **adt_list**. The standard operations of push, pop, empty, etc. are supplied as object-operations.

The concept of an array of ADTs is achieved through the introduction of **adt_marray** (i.e., memory array). In this ADT any finite length array can be allocated with user-specified upper and lower index bounds. In this implementation random access to an arbitrary index position is achieved in constant time (as in a random access array), whereas the array indexing feature of **adt_list** requires "marching through" the elements of a linked list to find the correct index position.

The concept of a bit vector is achieved through the introduction of **adt_bitv**. In this ADT any finite length bit vector (i.e., array of Boolean) can be allocated with user-specified upper and lower index bounds. This is useful for implementing the simulation of sets for which the members are known a priori. For instance, the adjacency matrix of an arbitrary graph easily can be implemented by construction an **adt_marray** of **adt_bitvs**.

The concept of a hash table is achieved through the introduction of **adt_hashtab**. This is implemented on top of **adt_array**. This ADT supplies the standard operations of inserting data based on a key and then subsequently searching for the data which is associated with a specific key.

5.2. Other General ADTs

In this section ADTs of general interest, but of a less general nature are introduced. Here, only the general concept associated with the ADT is presented.

There are a number of ADTs which address the mathematical formalism of a graph. Specifically, **adt_digraph** simulates the realization of a directed graph. Similarly, **adt_bigraph** simulates the realization of a bipartite graph. Both of these are built on top of two other ADTs, **adt_edge** and **adt_vertex**.

Another mathematical abstraction is achieved through **adt_bag**. The mathematical concept of a **bag** is similar to that of a set, except that the same object can occur more than once in a bag whereas an object can occur no more than once in a set.

Two operations associated with graphics have been implemented. **adt_point** implements the concept of a point in Cartesian coordinates, i.e., a 2-tuple of floats. **adt_line** implements the concept of a line segment, i.e., a 2-tuple of **adt_points**.

Three ADTs have been implemented to simulate combinatorial concepts. Specifically, **adt_perm** implements the concept of permutations. In other words, given a set of objects (instances of ADTs), **adt_perm** will produce, one at a time, every permutation possible for that set of objects. Similarly, **adt_combo** implements the concept of combinations. In other words, given a set of objects, **adt_combo** will produce, one at a time, every combination possible (of a given size) for that set of objects. **adt_odometer** simulates the idea of a generalized odometer (as in the odometer of a car). It is given a list of lists of objects. Each element of the top-level list (which is a list itself) corresponds to a wheel of the odometer. The objects in each secondary list correspond to the "characters" that reside on the wheel. **adt_odometer** returns, one at a time, a list of objects corresponding to what would appear on the concatenation of the wheels as the least significant wheel continues to turn, recursively causing subsequent wheels to turn.

An interesting form of a list has been introduced through the inclusion of **adt_rlist** (i.e., reversible list). The general implementation of a list, as expressed through **adt_list**, gives all of the functionality needed for manipulating lists, including concatenation of lists and reversal of a list. However, within its implementation the operations of concatenation and reversal are not of O(1) time complexity. It was not possible to modify the implementation of **adt_list** to make these operations of O(1) time complexity and still retain the generality that had been built into **adt_list**, for which there are a large number of access functions. **adt_rlist** was created to supply a restricted form of a list in which both the concatenation function and the reversal function are of O(1) time complexity.

5.3. DNA Mapping ADTs

There also are several ADTs which are directly related to DNA Mapping concepts. For instance, **adt_rfrag** implements the concept of what is referred to as a **real fragment**. A real fragment corresponds to an actual fragment present in a clone as identified through the process of electrophoresis. The term "real fragment" is meant to imply that the fragment is known to exist (via visual inspection of the electrophoresis gel). A real fragment has: (a) a name, (b) a measured length, and (c) estimates as to the right and left error bounds on the measured length.

In contrast, **adt_vfrag** implements the concept of what is referred to as a **virtual fragment**. A virtual fragment corresponds to a fragment present in a map unit. A virtual fragment is derived from one or more real fragments which have been inferred to come from the identical region of a genome. However this inference is a working hypothesis, and the virtual fragment (as derived from its constituent real fragments) may not exist at all. The length of a virtual fragment is the average of the lengths of the real fragments which constitute it.

Several forms of tree are implemented by **adt_tree**. The most important form of tree is the **sequence-set tree** (SST). SSTs are used to encode the group/fragment information present in a map unit. Specifically, in a map unit there is a sequence of groups, each group containing one or more fragments. The order of the groups is known (i.e., **sequence**), but the order of the fragments within each group is not known (i.e., **set**). For instance, the SST shown in Figure 45 corresponds to the map unit shown in Figure 46.

In performing DNA mapping and attempting to map a clone into a map unit, a window of interest (just slightly larger than the clone) is "dragged" across the map unit at group boundaries. At each window position, an attempt is made to map the clone within the portion of the map unit defined by the window. Since the concept of a map unit is encoded as an SST, a windowing mechanism is needed for defining a contiguous subregion of the tree. **adt_window** supplies this physical mechanism. Within an instance of

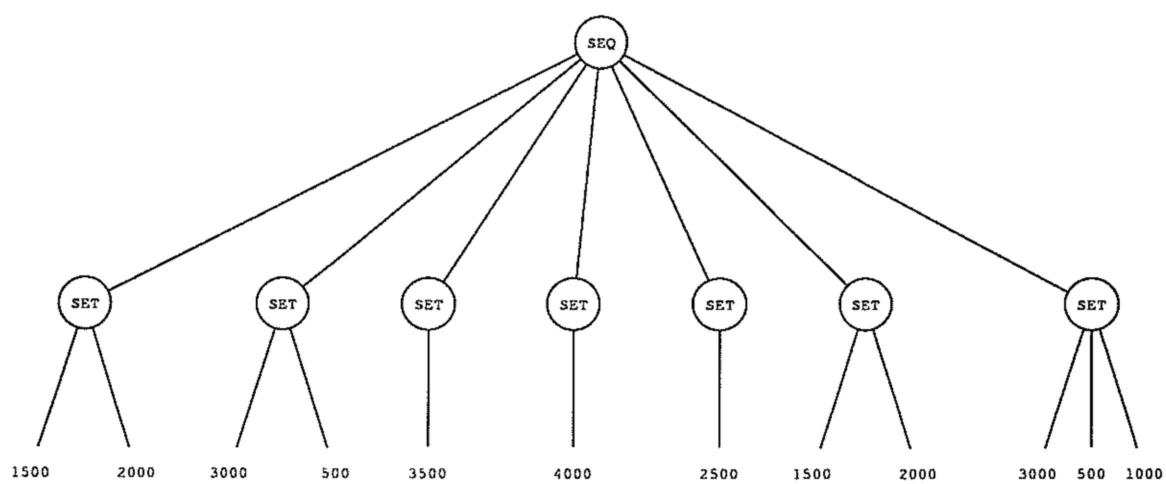

Figure 45: A Sequence-set Tree

Figure 46: A Map Unit

adt_window two pointers identify two set nodes within an SST. The groups corresponding to the set nodes (inclusively) between these two pointers define the window in the map unit.

5.4. Pseudo ADTs

Besides the formal ADTs that have been discussed, there are a number of **pseudo ADTs** that have been produced. They are referred to as **pseudo ADTs** because the do not formally have a distinct tag associated with them, but instead are built from a combination of other formal ADTs. Four of the current **pseudo ADTs** are **function**, **mu**, **partition**, and **relation**. **function** implements the mathematical concept of a function. In other words, a set (i.e., **adt_set**) of 2-tuples (i.e., **adt_pair**) in which the set of values occurring in the first component must be unique with respect to the second component. **mu** implements a number of useful functions on map units, expressed as sequence-set trees (SSTs). (The reason that this is separated from the implementation of SSTs is that SSTs are an abstraction which can have trees of any height or mixture of node types, whereas map units have exactly three tiers of rigidly structured nodes.) **partition** implements the concept of a partition of a set. **relation** implements the mathematical concept of a relation, i.e., a set of 2-tuples for which there is no restriction on the uniqueness of the components.

6. Conclusions

The use of the ADTs described in this report has made it possible to quickly create prototypes of many of the DNA Mapping algorithms. The self-identifying nature of the execution-time tagged architecture combined with the required system-backbone access functions produce a framework in which it is possible to create, copy, compare, print, and destroy data structures of unbounded complexity. Their utility stems from their object-oriented nature, making it possible to construct arbitrarily complex aggregates of data structures without the intellectual or physical effort normally required to understand and implement the details at each level of the aggregation.

The aggregate nature of many of the general purpose ADTs makes it possible to create a data "bridge" between different computational components of a complex algorithm. The aggregate output of one routine (say a list of objects) can be made available as the input to any number of routines for subsequent processing. The aggregate can be passes through routines and be perceived as a single object until it is necessary to unbundle the collection to extract its parts for individual processing. The concept is very similar to that of a **pipe** in the UNIX operating system. In the case of a **pipe**, UNIX uses a file (either on disk or in memory) to communicate information between processes. This file is an aggregate of information, the syntax of which both the creator and the receiver must understand. The receiver can choose to pass on the information to a subsequent process without any modifications, or it can choose to "filter" the information passed through it in any number of ways. The aggregate ADTs supply a similar encapsulation mechanism between computational components of an application, using well-known concepts and structures. In a similar manner that the concept and use of a **pipe** encourages modularity and communication, the ADT paradigm enhances the ability to modularize and parameterize the implementation of an application. This modularity allows algorithms to be reconfigured quickly into variations of the original algorithm.

Once the basic idea behind ADTs is understood, it is not difficult to design and implement new ADT classes. It often is appropriate to use already implemented ADTs as constituent components of a new ADT being created. For instance, in the current ADTs implemented within the DNA Mapping system, **adt_dll** is built on top of **adt_list_node**, **adt_list** is built on top of **adt_dll**, and **adt_set** is built on top of **adt_list**. Many other ADTs are interdependent also.

Since the ADT classes are declared at execution time, the entire system need not be recompiled as a new ADT class is introduced. No compile-time tables keep track of what ADTs are or are not present; thus, no compile-time objects have to be changed to reflect the introduction of a new ADT. The creator of a new ADT class need only program the required system-backbone and object-operation access function and include the declaration of the new ADT within the initialization activities of the system.

Although the use of ADTs supplies a very flexible and powerful framework for the application programmer, it also forces significant responsibility onto the application programmer. The programmer must be careful not to produce "circular" data structures whose recursive nature will confuse the simple-minded aggregate ADT management framework. The programmer must also be vigilant to destroy all instances of ADTs which are no longer needed. This requirement often dictates the style and structure of the algorithms produced. This can be thought of as either an advantage or a disadvantage. It has been found that the discipline imposed by "memory zeroing" restrictions often produces algorithms of much better structure than would have been produced if the restrictions were not present. This tends to make the resulting algorithms more modular, modifiable, verifiable, and reconfigurable.