(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 August 2003 (07.08.2003)

PCT

(10) International Publication Number
WO 03/065252 A1

(51) International Patent Classification[7]: G06F 17/30, 12/06

(21) International Application Number: PCT/US03/03067

(22) International Filing Date: 3 February 2003 (03.02.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/353,487 1 February 2002 (01.02.2002) US

(71) Applicant and
(72) Inventor: FAIRWEATHER, John [US/US]; 1649 Wellesley Drive, Santa Monica, CA 90405 (US).

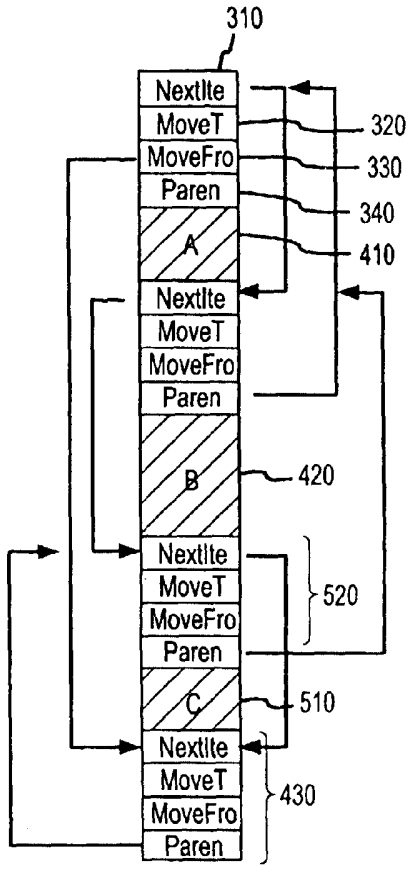
(74) Agents: THIESSEN, Kendall, I. et al.; Gibson, Dunn & Crutcher LLP ,, 1801 California Street, Suite 4100, Denver, CO 80202 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, SE, SI,

[Continued on next page]

(54) Title: SYSTEM AND METHOD FOR MANAGING MEMORY



(57) Abstract: A new memory tuple is described that creates both a handle as well as a reference (510) to an item (430) within the handle (430). The reference is created using an offset value that defines the physical offset of the data within the memory block. Thereafter, if references are passed in terms of their offset value, this value will be the same in any copy of the handle regardless of the machine. In a distributed computing environment, equivalence between handles is established in a single transaction between two communicating machines. Thereafter, the two machines can communicate about specific handle contents simply by using offsets.

WO 03/065252 A1

SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:

— *of inventorship (Rule 4.17(iv)) for US only*

Published:

— *with international search report*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

SYSTEM AND METHOD FOR MANAGING MEMORY

Inventor: John Fairweather

BACKGROUND OF THE INVENTION

The Macintosh Operating system ("OS"), like all OS layers, provides an API where applications can allocate and de-allocate arbitrary sized blocks of memory from a heap. There are two basic types of allocation, viz: handles and pointers. A pointer is a non-relocatable block of memory in heap (referred to as *p in the C programming language, hereinafter "C"), while a handle is a non-relocatable reference to a relocatable block of memory in heap (referred to as **h in C). In general, handles are used in situations where the size of an allocation may grow, as it is possible that an attempt to grow a pointer allocation may fail due to the presence of other pointers above it. In many operating systems (including OS X on the Macintosh) the need for a handle is removed entirely as a programmer may use the memory management hardware to convert all logical addresses to and from physical addresses.

The most difficult aspect of using handle based memory, however, is that unless the handle is 'locked', the physical memory allocation for the handle can move around in memory by the memory manager at any time. Movement of the physical memory allocation is often necessary in order to create a large enough contiguous chunk for the new block size. The change in the physical memory location, however, means that one cannot 'de-reference' a handle to obtain a pointer to some structure within the handle and pass the pointer to other systems as the physical address will inevitably become invalid. Even if the handle is locked, any pointer value(s) are only valid in the current machine's memory. If the structure is passed to another machine, it will be instantiated at a different logical address in memory and all pointer references from elsewhere will be invalid. This makes it very difficult to efficiently pass references to data. What is needed, then, is a method for managing memory references such that a reference can be passed to another machine and the machine would be able to retrieve or store the necessary data even if the physical address of the data has been changed when transferred to the new machine or otherwise altered as a result of changes to the data.

SUMMARY OF THE INVENTION

The following invention provides a method for generating a memory reference that is capable of being transferred to different machine or memory location without jeopardizing access to relevant data. Specifically, the memory management system and method of the present invention creates a new memory tuple that creates both a handle as well as a reference to an item within the handle. In the latter case, the reference is created using an offset value that defines the physical offset of the data within the memory block. If references are passed in terms of their offset value, this value will be the same in any copy of the handle regardless of the machine. In the context of a distributed computing environment, all that then remains is to establish the equivalence between handles, which can accomplished in a single transaction between two communicating machines. Thereafter, the two machines can communicate about specific handle contents simply by using offsets.

The minimum reference is therefore a tuple comprised of the handle together with the offset into the memory block, we shall call such a tuple an 'ET_ViewRef' and sample code used to create such a tuple 100 in C is provided in Figure 1. Once this tuple has been created, it becomes possible to use the ET_ViewRef structure as the basic relocatable handle reference in order to reference structures internal to the handle even when the handle may move. The price for this flat memory model is the need for a wrapper layer that transparently handles the kinds of manipulations described above during all de-referencing operations, however, even with such a wrapper, operations in this flat memory model are considerably faster that corresponding OS supplied operations on the application heap.

BRIEF DESCRIPTION OF THE FIGURES

Figure 1 illustrates sample code used to create the minimum reference 'tuple' of the present invention;

Figure 2 illustrates a drawing convention that is used to describe the interrelationship between sub-layers in one embodiment of the present invention;

Figure 3 illustrates a sample header block that may be used to practice the present invention;

Figure 4 illustrates a simple initial state for a handle containing multiple structures;

Figure 5 illustrates the type of logical relationships that may be created between structures in a handle following the addition of a new structure;

Figure 6 illustrates a sample of a handle after increasing the size of a given structure within the handle beyond its initial physical memory allocation;

Figure 7 illustrates the manner in which a handle could be adapted to enable unlimited growth to a given structure within the handle;

Figure 8 illustrates the handle after performing an undo operation;

Figure 9 illustrates a handle that has been adapted to include a time axis in the header field of the structures within the handle;

Figure 10 illustrates the manner in which the present invention can be used to store data as a hierarchical tree; and

Figure 11 illustrates the process for using the memory model to sort structures within a handle.

DETAILED DESCRIPTION

Descriptive Conventions

In order to graphically describe the architectural components and interrelations that comprise the software, this document adopts a number of formalized drawing conventions. In general, any given software aspect is built upon a number of sub-layers. Referring now to figure 2, a block diagram is provided that depicts these sub-layers as a 'stack' of blocks. The lowest block is the most fundamental (generally the underlying OS) and the higher block(s) are successive layers of abstraction built upon lower blocks. Each such block is referred to interchangeably as either a module or a package.

The first, an opaque module 200, is illustrated as a rectangular in Figure 2A. An opaque module 200 is one that cannot be customized or altered via registered plug-ins. Such a form generally provides a complete encapsulation of a given area of functionality for which customization is either inappropriate or undesirable.

The second module, illustrated as T-shaped form 210 in Figure 2B, represents a module that provides the ability to register plug-in functions that modify its behavior for particular purposes. In Figure 2A, these plug-ins 220 are shown as 'hanging' below the horizontal bar of the module 210. In such cases, the module 210 provides a complete 'logical' interface to a certain functional capability while the plug-ins 220 customize that functionality as desired. In general, the plug-ins 220 do not provide a callable API of their own. This methodology provides the benefits of customization and flexibility without the negative effects of allowing application specific knowledge to percolate any higher up the stack than necessary. Generally, most modules provide a predefined set of plug-in behaviors so that for normal operation they can be used directly without the need for plug-in registration.

In any given diagram, the visibility of lower layers as viewed from above, implies that direct calls to that layer from higher-level layers above is supported or required as part of normal operation. Modules that are hidden vertically by higher-level modules, are not intended to be called directly in the context depicted.

Figure 2C illustrates this descriptive convention. Module 230 is built upon and makes use of modules 235, 240, and 245 (as well as what may be below module 245). Module 230, 235 and 240 make use of module 245 exclusively. The functionality within module 240 is completely hidden from higher level modules via module 230, however direct access to modules 250 and 235 (but not 245) is still possible.

In Figure 2D, the Viewstructs memory system and method 250 is illustrated. The ViewStructs 250 package (which implements the memory model described herein) is layered directly upon the heap memory encapsulation 280 provided by the TBFilters 260, TrapPatches 265, and WidgetQC 270 packages. These three packages 260, 265, 270 form the heap memory abstraction, and provide sophisticated debugging and memory tracking capabilities that are discussed elsewhere. When used elsewhere, the terms ViewStructs or memory model apply only to the contents of a single handle within the heap.

To reference and manipulate variable sized structures within a single memory allocation, we require that all structures start with a standard header block. A sample header block (called an ET_Hdr) may be defined in C programming language as illustrated in Figure 3. For the purpose of discussing the memory model, we shall only consider the use of ET_Offset fields 310, 320, 330, 340. The word 'flags' 305, among other things, indicates the type of record follows the ET_Hdr. The 'version' 350 and 'date' fields 360 are associated with the ability to map old or changed structures into the latest structure definition, but these fields 350, 360 are not necessary to practice the invention and are not discussed herein.

Referring now to Figure 4, Figure 4 illustrates a simple initial state for a handle containing multiple structures. The handle contains two distinct memory structures, structure 410 and structure 420. Each structure is preceded by a header record, as previously illustrated in Figure 3, which defines its type (not shown) and its relationship to other structures in the handle. As can be seen from the diagram, the 'NextItem' field 310 is simply a daisy chain where each link simply gives the relative offset from the start of the referencing structure to the start of the next structure in the handle. Note that all references in this model are relative to the start of the referencing structure header and indicate the (possibly scaled) offset to the start of the referenced structure header. The final structure in the handle is indicated by a header record 430 with no associated additional data where 'NextItem = 0'. By following the 'NextItem' daisy chain it is possible to examine and locate every structure within the handle.

As the figure illustrates, the 'parent' field 340 is used to indicate parental relationships between different structures in the handle. Thus we can see that structure B 420 is a child of structure A 410. The terminating header record 430 (also referred to as an ET_Null record) always has a parent field that references the immediately preceding structure in the handle. Use of the parent field in the terminating header record 430 does not represent a "parent" relationship, it is simply a convenience to allow easy addition of new records to the handle. Similarly, the otherwise meaningless 'moveFrom' field 330 for the first record in the handle contains a relative reference to the final ET_Null. This provides an expedient way to locate the logical end of the handle without the need to daisy chain through the 'nextItem' fields for each structure.

Referring now to Figure 5, Figure 5 illustrates the logical relationship between the structures after adding a third structure C 510 to the handle. As shown in Figure 5, structure C 510 is a child of B 420 (grandchild of A 410). The insertion of the new structure involves the following steps:

1) If necessary, grow the handle to make room for C 510, C's header 520, and the trailing ET_Null record 430;

2) Overwrite the previous ET_Null 430 with the header and body of structure C 510.

3) Set up C's parent relationship. In the illustrated example, structure C 510 is a child of B 420, which is established by pointing the 'parent' field of C's header file 520 to the start of structure B 420.

4) Append a final ET_Null 530, with parent referenced to C's header 520.

5) Adjust the 'moveFrom' field 330 to reflect the offset of the new terminating ET_Null 530.

In addition to adding structures, the present invention must handle growth within existing structures. If a structure, such as structure B 420, needs to grow, it is often problematic since there may be another structure immediately following the one being grown (structure C 510 in the present illustration). Moving all trailing structures down to make enough room for the larger B 420 is one way to resolve this issue but this solution, in addition to being extremely inefficient for large handles, destroy the integrity of the handle contents, as the relative references within the original B structure 420 would be rendered invalid once

such a shift had occurred. The handle would then have to be scanned looking for such references and altering them. The fact that structures A 410, B 420, and C 510 will generally contain relative references over and above those in the header portion make this impractical without knowledge of all structures that might be part of the handle. In a dynamic computing environment such knowledge would rarely, if ever, be available, making such a solution impractical and in many cases impossible.

For these reasons, the header for each structure further includes a moveFrom and moveTo fields. Figure 6 illustrates the handle after growing B 420 by adding the enlarged B' structure 610 to the end of the handle. As shown, the original B structure 420 remains where it is and all references to it (such as the parent reference from C 510) are unchanged. B 420 is now referred to as the "base record" whereas B' 610 is the "moved record". Whenever any reference is resolved now, the process of finding the referenced pointer address using C code is:

```
src = address of referencing structure header
dst = src + ET_Offset value for the reference
if ( dst->moveTo )
   dst = dst + dst->moveTo     -- follow the move
```

Further whenever a new reference is created, the process of finding the referenced pointer using C code is:

```
src = address of referencing structure header
dst = address of referenced structure header
if ( dst->moveFrom )
   dst = dst + dst->moveFrom;
ref value = dst - src
```

Thus, the use of the moveto and movefrom fields ensures that no references become invalid, even when structures must be moved as they grow.

Figure 7 illustrates the handle when B 420 must be further expanded into B" 710. In this case the 'moveTo' of the base record 420 directly references the most recent version of the structure, in this example B" 710. Correspondingly, the record B'' 710 now has a 'moveFrom' 720 field that references the base record 420. B's moveFrom 720 still refers back to B 420 and indeed if there were more intermediate records between B 420 and B'' (such as B' 610 in this example) the 'moveTo' and 'moveFrom' fields for all of the records 420, 610, 710 would form a doubly linked list. Once each of these records 420, 610, 710

have been linked, it is possible to re-trace through all previous versions of a structure using these links. For example, one could find all previous versions of the record starting with B" 710 by following the 'movefrom' field 720 to the base record 420 and then following the 'nextItem' link of each record until a record with a 'moveFrom' referencing the base record 420 is found. Alternatively, and perhaps more reliably, one could look for structures whose 'moveTo' field references record 420 and then work backward through the chain to find earlier versions.

This method, in which the last 'grown' structure moves to the end of the handle, has the beneficial effect that the same structure is often grown many times in sequence and in these cases we can optionally avoid creating a series of intermediate 'orphan' records. References occurring from within the bodies of structures may be treated in a similar manner to those described above and thus by extrapolation one can see that arbitrarily complex collections of cross-referencing structures can be created and maintained in this manner all within a single 'flat' memory allocation.

The price for this flat memory model is the need for a wrapper layer that transparently handles the kinds of manipulations described above during all de-referencing operations, however, even with such a wrapper, operations in this flat memory model are considerably faster that corresponding OS supplied operations on the application heap. Regardless of complexity, a collection of cross-referencing structures created using this approach is completely 'flat' and the entire 'serialization' issue is avoided when passing such collections between processors. This is a key requirement in a distributed data-flow based environment.

In addition to providing the ability to grow and move structures without impacting the references in other structures, another advantage of the 'moveTo'/'moveFrom' approach is inherent support for 'undo'. FIGURE 8 illustrates the handle after performing an 'undo' on the change from B' to B''. The steps involved for 'undo' are provided below:

```
src = base record (i.e., B)
dst = locate 'moved' record (i.e. B'') by following 'moveTo' of base record
prev = locate last record in handle whose 'moveTo' references dst
src->moveTo = prev – src;
```

The corresponding process for 'redo' (which restores the state to that depicted after B'' was first added) is depicted below:

```
src = base record (i.e., B)
```

```
dst = locate 'moved' record (i.e. B') by following 'moveTo' of base record
if ( dst->moveTo )
   nxt = dst + dst->moveTo
src->moveTo = nxt - src;
```

This process works because of the fact that 'moveTo' fields are only followed once when referencing via the base record. The ability to trivially perform undo/redo operations is very useful in situations where the structures involved represent information being edited by the user, it is also an invaluable technique for handling the effects of a time axis in the data.

One method for maintaining a time axis is by using a date field in the header of each structure. In this situation, the undo/redo mechanism can be combined with a 'date' field 910 in the header that holds the date when the item was actually changed. This process is illustrated in Figure 9 (some fields have been omitted for clarity).

This time axis can also be used to track the evolution of data over time. Rather than using the 'moveTo' fields to handle growing structures, the 'moveTo' fields could be used to reference future iterations of the data. For example, the base record could specify that it stores the high and low temperatures for a given day in Cairo. Each successive record within that chain of structures could then represent the high and low temperatures for a given date 910, 920, 930, 940. By using the 'date' fields 910, 920, 930, 940 in this fashion, the memory system and method can be used to represent and reference time-variant data, a critical requirement of any system designed to monitor, query, and visualize information over time. Moreover, this ability to handle time variance exists within the 'flat' model and thus data can be distributed throughout a system while still retaining variance information. This ability lends itself well to such things as evolving simulations, database record storage and transaction rollback, and animations.

Additionally, if each instance of a given data record represents a distinct version of the data designed for a different 'user' or process, this model can be used to represent data having multiple values depending on context. To achieve this, whatever variable is driving the context is simply used to set the 'moveTo' field of the base record, much like time was used in the example above. This allows the model to handle differing security privileges, data whose value is a function of external variables or state, multiple distinct sources for the same datum, configuration choices, user interface display options, and other multi-value situations.

A 'flags' field could also be used in the header record and can be used to provide additional flexibility and functionality within the memory model. For example, the header could include a 'flag' field that is split into two parts. The first portion could contain arbitrary logical flags that are defined on a per-record type basis. The second portion could be used to define the structure type for the data that follows the header. While the full list of all possible structure types is a matter of implementation, the following basic types are examples of types that may be used and will be discussed herein:

kNullRecord – a terminating NULL record, described above.

kStringRecord – a 'C' format variable length string record.

kSimplexRecord – a variable format/size record whose contents is described by a type-id.

kComplexRecord – a 'collection' element description record (discussed below)

kOrphanRecord – a record that has been logically deleted/orphaned and no longer has any meaning.

By examining the structure type field of a given record, the memory wrapper layer is able to determine 'what' that record is and more importantly, what other fields exist within the record itself that also participate in the memory model, and must be handled by the wrapper layer. The following definition describes a structure named 'kComplexRecord' and will be used to illustrate this method:

```
typedef struct ET_Complex      //    Collection element record
{
   ET_Hdr                             hdr;          // Standard header
   ...
   ET_Offset /* ET_SimplexPtr */      valueR;       // value reference
   ET_TypeID                          typeID;       // ID of this type
   ET_Offset /* ET_ComplexPtr */      nextElem;     // next elem. link
   ET_Offset /* ET_ComplexPtr */      prevElem;     // prev. elem. link
   ET_Offset /* ET_ComplexPtr */      childHdr;     // First child link
   ET_Offset /* ET_ComplexPtr */      childTail;    // Last child link
   ...
} ET_Complex;
```

The structure defined above may be used to create arbitrary collections of typed data and to navigate around these collections. It does so by utilizing the additional ET_Offset fields listed above to create logical relationships between the various elements within the handle.

Figure 10 illustrates the use of this structure 1010 to represent a hierarchical tree 1020. The ET_Complex structure defined above is sufficiently general, however, that virtually any collection metaphor can be represented by it including (but not limited to) arrays (multi-dimensional), stacks, rings, queues, sets, n-trees, binary trees, linked lists etc. The 'moveTo', 'moveFrom' and 'nextItem' fields of the header have been omitted for clarity. The 'valueR' field would contain a relative reference to the actual value associated with the tree node (if present), which would be contained in a record of type ET_Simplex. The type ID of this record would be specified in the 'typeID' field of the ET_Complex and, assuming the existence of an infrastructure for converting type IDs to a corresponding type and field arrangement, this could be used to examine the contents of the value (which could further contain ET_Offset fields as well).

As Figure 10 illustrates, 'A' 1025 has only one child (namely 'B' 1030), both the 'childHdr' 1035 and 'childTail' 1040 fields reference 'B' 1030, this is in contrast to the 'childHdr' 1045 and 'childTail' 1070 fields of 'B' 1030 itself which reflect the fact that 'B' 1030 has three children 1050, 1055, 1060. To navigate between children 1050, 1055, 1060, the doubly-linked 'nextItem' and 'prevItem' fields are used. Finally the 'parent' field from the standard header is used to represent the hierarchy. It is easy to see how simply by manipulating the various fields of the ET_Complex structure, arbitrary collection types can be created as can a large variety of common operations on those types. In the example of the tree above, operations might include pruning, grafting, sorting, insertion, rotations, shifts, randomization, promotion, demotion etc. Because the ET_Complex type is 'known' to the wrapper layer, it can transparently handle all the manipulations to the ET_Offset fields in order to ensure referential integrity is maintained during all such operations. This ability is critical to situations where large collections of disparate data must be accessed and distributed (while maintaining 'flatness') throughout a system.

Figure 11 illustrates the process for using the memory model to "sort" various structures. A sample structure, named ET_String 1100, could be defined in the following manner (defined below) to perform sorting on variable sized structures:

```
typedef struct ET_String                        // String Structure
{
   ET_Hdr      hdr;                             // Standard header
   ET_Offset /* ET_StringPtr */  nextString; // ref. to next string
   ...
   char        theString[ 0 ];          // C string (size varies)
} ET_String;
```

Prior to the sort, the 'nextString' fields 1110, 1115, 1120, 1125 essentially track the 'nextItem' field in the header, indeed 'un-sort' can be trivially implemented by taking account of this fact. By accessing the strings in such a list by index (i.e., by following the 'nextString' field), users of such a 'string list' abstraction can manipulate collections of variable sized strings. When combined with the ability to arbitrarily grow the string records as described previously (using 'moveTo' and 'moveFrom'), a complete and generalized string list manipulation package is relatively easy to implement. The initial 'Start' reference 1130 in such a list must obviously come from a distinct record, normally the first record in the handle. For example, one could define a special start record format for containers describing executable code hierarchies. The specific implementation of these 'start' records are not important. What is important, however, is that each record type contain a number of ET_Offset fields that can be used as references or 'anchors' into whatever logical collection(s) is represented by the other records within the handle.

The process of deleting a structure in this memory model relates not so much to the fields of the header record itself, but rather to the fields of the full structure and the logical relationships between them. In other words, the record itself is not deleted from physical memory, rather it is logically deleted by removing from all logical chains that reference it. The specific manner in which references are altered to point "around" the deleted record will thus vary for each particular record type. Figure 12 illustrates the situation after deleting "Dog" 1125 from the string list 1100 and 'C' 1050from the tree 1020.

When being deleted, the deleted record is generally 'orphaned'. In order to more easily identify the record as deleted, a record may be set to a defined record type, such as 'kOrphanRecord'. This record type could be used during compression operations to identify those records that have been deleted. A record could also be identified as deleted by confirming that it is no longer referenced from any other structure within the handle. Given the complete knowledge that the wrapper layer has of the various fields of the structures within the handle, this condition can be checked with relative ease and forms a valuable double-check when particularly sensitive data is being deleted.

The compression process involves movement of higher structures down to fill the gap and then the subsequent adjustment of all references that span the gap to reduce the reference offset value by the size of the gap being closed during compression. Once again, the fact that

the wrapper layer has complete knowledge of all the ET_Offset fields within the structures in the handle make compression a straightforward operation.

The foregoing description of the preferred embodiment of the invention has been presented for the purposes of illustration and description. For example, the term "handle" throughout this description is addressed as it is currently used in the Macintosh OS. This term should not be narrowly construed to only apply to the Macintosh OS, however, as the method and system could be used to enhance any sort of memory management system. The descriptions of the header structures should also not be limited to the embodiments described. While the defined header structures provide examples of the structures that may be used, the plurality of header structures that could in fact be implemented is nearly limitless. Indeed, it is the very flexibility afforded by the memory management system that serves as its greatest strength. For these reasons, this description is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. In particular due to the simplicity of the model, hardware based implementations can be envisaged. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended hereto.

CLAIMS

I claim the following inventions:

1. A method for storing data, comprising the steps of:

obtaining a non-relocatable reference to a possibly relocatable block of memory;

creating one or more memory structures to be stored within the block of memory, such memory structures having space allocated for a header and a data portion;

creating a header for a memory structure, wherein such header includes a field for linking to the next memory structure in the memory block and a field for identifying additional data structures unique to a particular type of memory structure; and

storing said header within the corresponding memory structure.

2. The method of claim 1, wherein the method further comprises the step of assigning values to the fields in the header using one or more offset values, said offset values representing the physical memory offset between the start of the block of memory and the memory structure being linked, or the signed physical offset between the headers of the referencing and referenced structures involved in the reference.

3. The method of claim 1, wherein the method further comprises the step of assigning values to the fields in the header using one or more offset values, said offset values representing the signed physical offset between the headers of the referencing and referenced structures involved in the reference.

4. The method of claim 1, wherein the process of requesting a non-relocatable block of memory includes requesting a handle from the memory manager in the Macintosh operating system environment.

5. The method of claim 1, wherein the process for creating a header for a memory structure includes creating headers that have a field for storing a date associated with the memory structure.

6. The method of claim 1, wherein the process for creating a header for a memory structure includes creating headers that have a field for storing the parent-child relationship between two or more memory structures.

7. The method of claim 1, wherein the method further includes

a header type that indicates the end of the list of memory structures within the memory block, and

a starting memory block, which holds entry point references into the other structures held within the memory block.

8. The method of claim 1, wherein the step for creating a header for a memory structure includes creating headers that have a field for identifying the type of data, as distinct from the type of header, associated with the memory structure.

9. The method of claim 1, further including the step of creating a field within a header to identify orphaned memory structures.

10. The method of claim 1, wherein the step of creating a header includes assigning values to one or more fields within the header, such values including the physical offset between the start of the block of memory to the start of each memory structure.

11. The method of claim 1, wherein the step of creating a header includes assigning values to one or more fields within the header, such values including the signed relative memory offset between the referencing and referenced headers.

12. A system for storing data, the system comprising:

obtaining a non-relocatable reference to a block of memory;

one or more memory structures stored within the block of memory, such memory structures having space allocated for a header and a data portion;

a header, stored within the memory structure, that includes fields for linking to one or more memory structures that are related to the first memory structure; and

13. The system of claim 12, wherein the field for linking to one or more memory structures that are related to the first memory structure includes a move to field for linking to alternate memory structures of the referencing memory structure within the block of memory.

14. The system of claim 13, wherein the 'moved' fields for linking to/from another memory structure are comprised of an offset value, wherein the offset value is the physical memory offset between the start of the memory block and the start of the linked memory structure.

15. The system of claim 12, wherein the 'moved' fields for linking to/from another memory structure are comprised of an offset value, wherein the offset value is the signed physical memory offset between the referencing and referenced headers of the two structures involved.

16. The system of claim 13, wherein the header is further comprised of a field for storing a date.

17. The system of claim 16, wherein the field included within the header for storing the date is assigned the date upon which the data contained within the data portion of the memory structure was first collected, stored, or updated.

18. The systems of 16, wherein the 'moved' fields stores a series of memory structures relating to a single logical base structure can be altered to cause the base to reference one of a number of alternative versions of the structure.

19. The systems of 18, wherein the control of the particular value of the 'moved' fields is based on an algorithm.

20. The system of claim 18, wherein the system further comprises means for providing wrapper support, such wrapper support means enabling alteration of the 'moved' references to an earlier state using a specific date, current state, or any other algorithm

21. The system of claim 18, wherein the system further comprises means for providing wrapper support, such wrapper support means enabling alteration of the 'moved' references to the current state.

22. The system of claim 18, wherein the system further comprises means for providing wrapper support, such wrapper support means enabling alteration of the 'moved' references to an earlier state using an algorithm.

```
typedef  int32              ET_Offset;
typedef  struct
{
        Handle              theView;
        ET_offset    theOffset;
} ET_ViewRef,  *ET_ViewRefPtr;

#define VIEWREF_TO_PTR(vr).((charPtr).(*vr.theView) + vr.theOffset)
#define PTR_TO_VIEWREF(hdl,ptr,vr)
    (vr,theView = hdl; vr. theOffset = (charPtr)ptr - (charPtr) *hdl;}
```

FIG.1

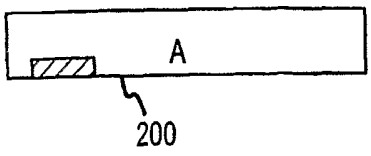


FIG.2A

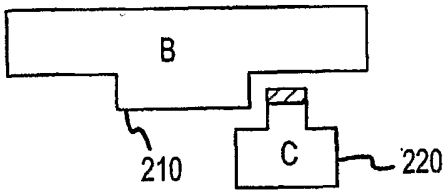


FIG.2B

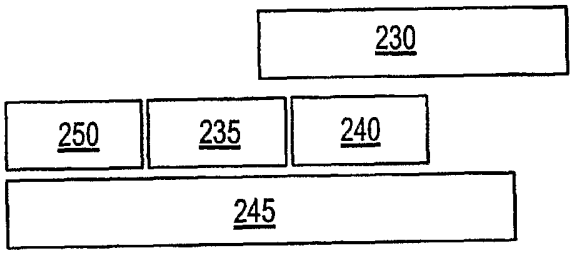


FIG.2C

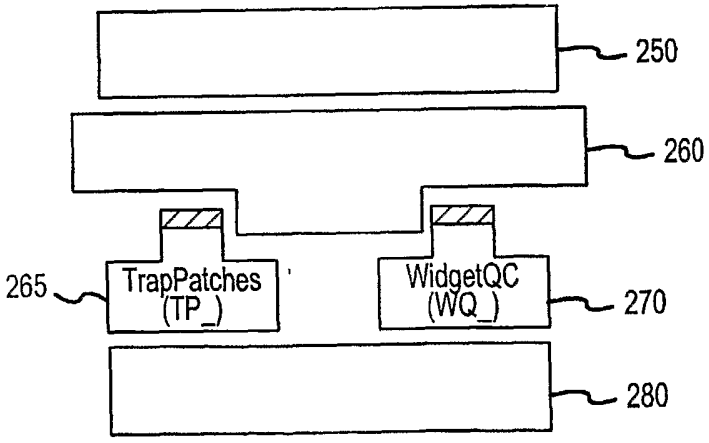


FIG.2D

```
typedef struct ET_Hdr
{
    int32       flags;          // Static & Dynamic flags used by Mitopia™
    ET_Offset   nextItem;       // ref to next item in handle
    ET_Offset   moveTo;         // ref to where structure has been moved to
    ET_Offset   moveFrom;       // ref to where structure has been moved from
    ET_Offset   parent;         // ref to parent or referencing structure
    UnsChar     version;        // structure version number & creator ID
    UnsChar     date[3];        // date of creation
} ET_Hdr;
```

FIG.3

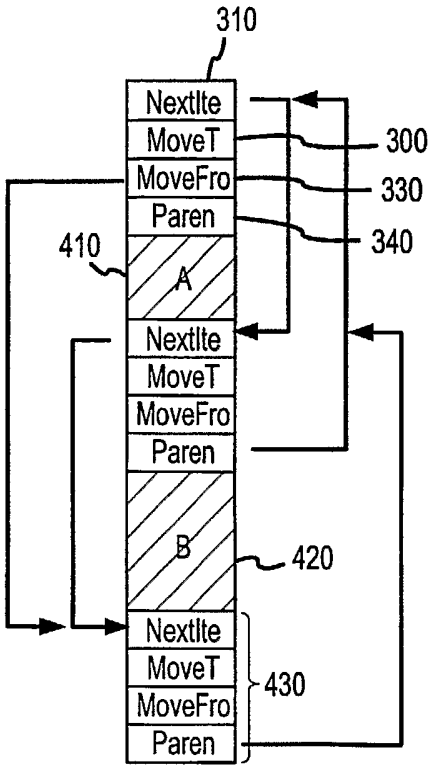

310
NextIte
MoveT
MoveFro
Paren
300
330
340
410
A
NextIte
MoveT
MoveFro
Paren
B
420
NextIte
MoveT
MoveFro
Paren
430

FIG.4

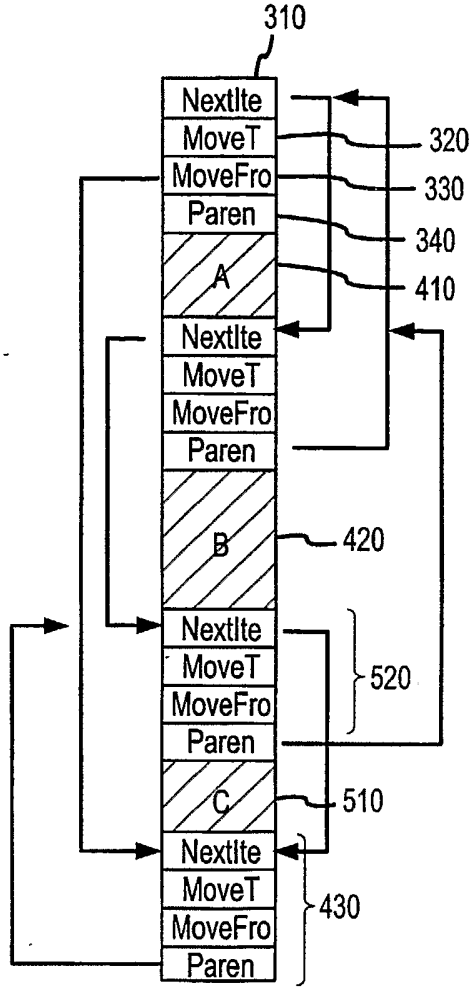

310
NextIte
MoveT
MoveFro
Paren
320
330
340
410
A
NextIte
MoveT
MoveFro
Paren
B
420
NextIte
MoveT
MoveFro
Paren
520
C
510
NextIte
MoveT
MoveFro
Paren
430

FIG.5

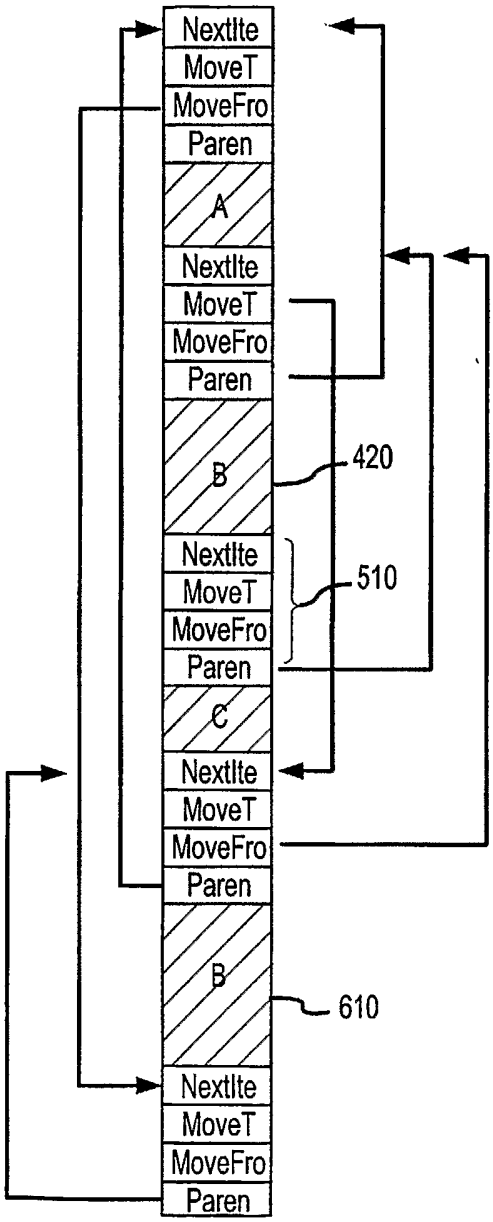

NextIte
MoveT
MoveFro
Paren
A
NextIte
MoveT
MoveFro
Paren
B
420
NextIte
MoveT
MoveFro
Paren
510
C
NextIte
MoveT
MoveFro
Paren
B
610
NextIte
MoveT
MoveFro
Paren

FIG.6

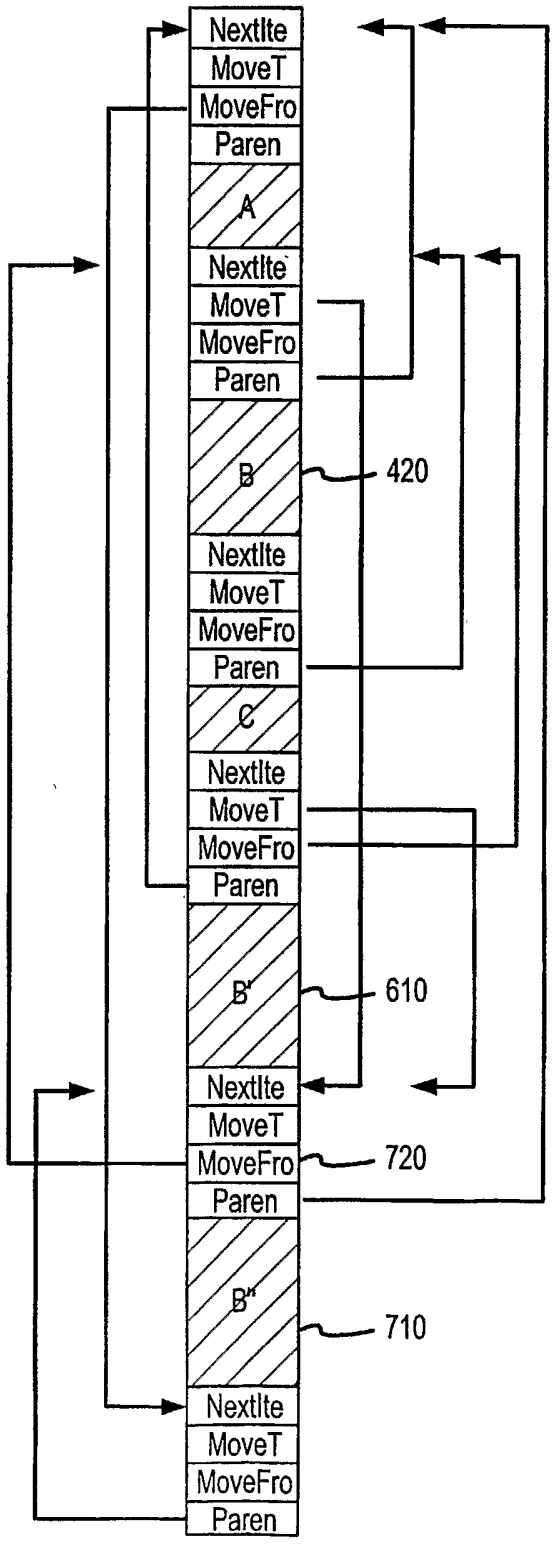

NextIte
MoveT
MoveFro
Paren
A
NextIte
MoveT
MoveFro
Paren
B
420
NextIte
MoveT
MoveFro
Paren
C
NextIte
MoveT
MoveFro
Paren
B'
610
NextIte
MoveT
MoveFro
720
Paren
B"
710
NextIte
MoveT
MoveFro
Paren

FIG.7

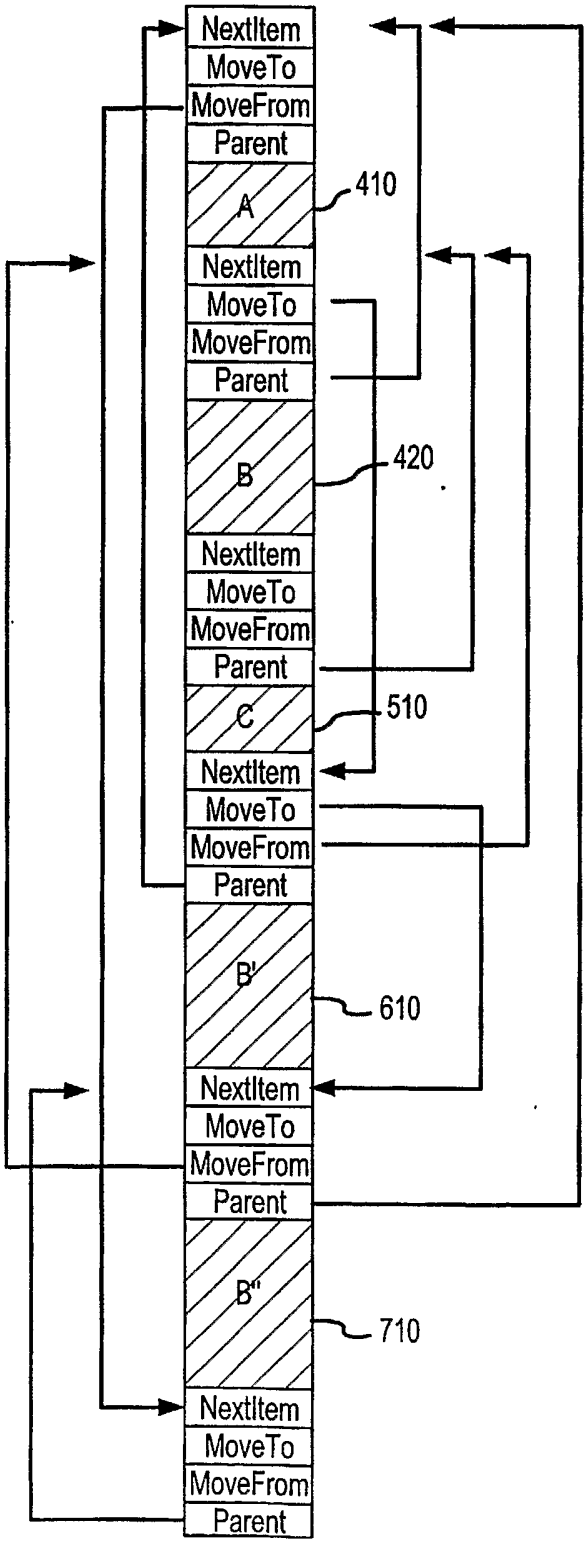
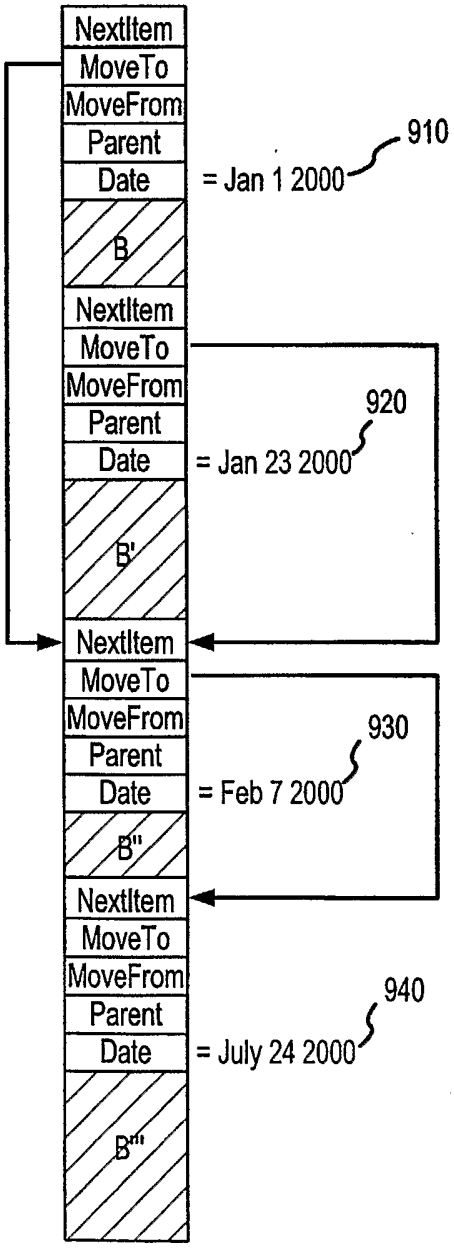
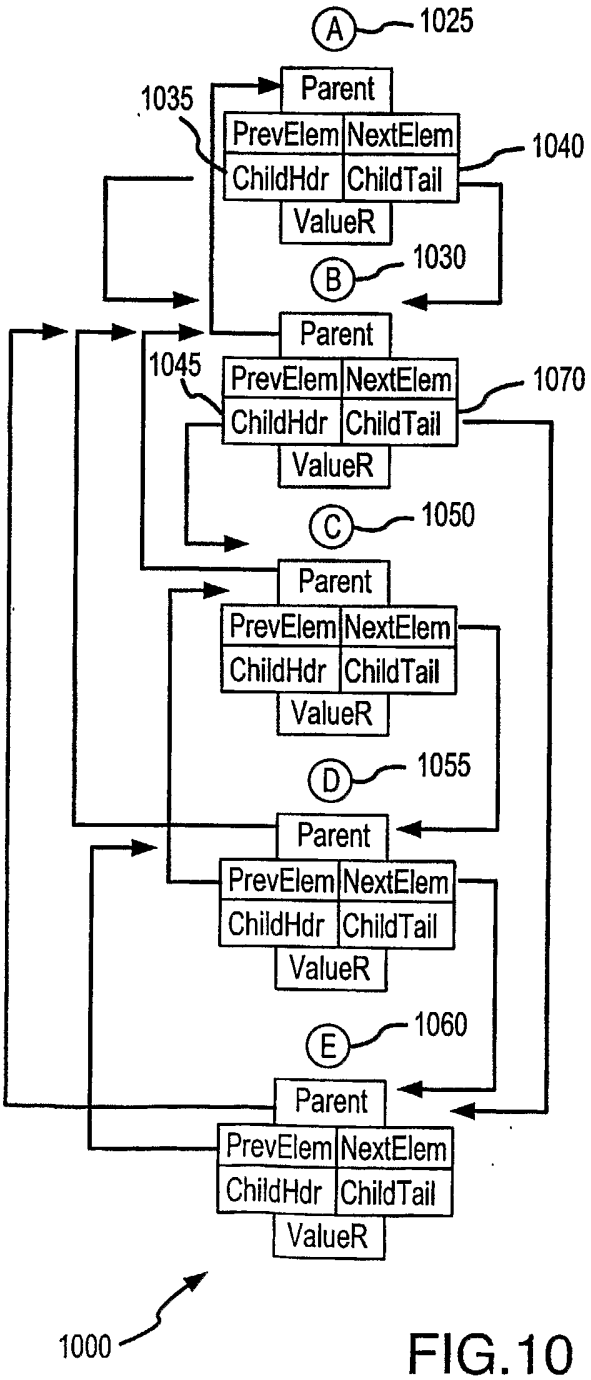

NextItem
MoveTo
MoveFrom
Parent
A
410
NextItem
MoveTo
MoveFrom
Parent
B
420
NextItem
MoveTo
MoveFrom
Parent
C
510
NextItem
MoveTo
MoveFrom
Parent
B'
610
NextItem
MoveTo
MoveFrom
Parent
B''
710
NextItem
MoveTo
MoveFrom
Parent

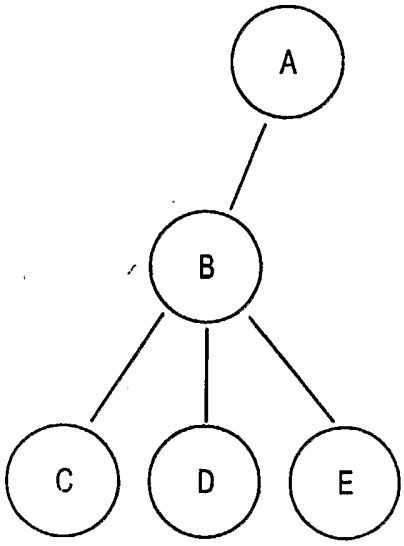
FIG.8



TIME AXIS DATE = MARCH 4 2000

FIG.9





EQUIVALENT TREE STRUCTURE

FIG.10

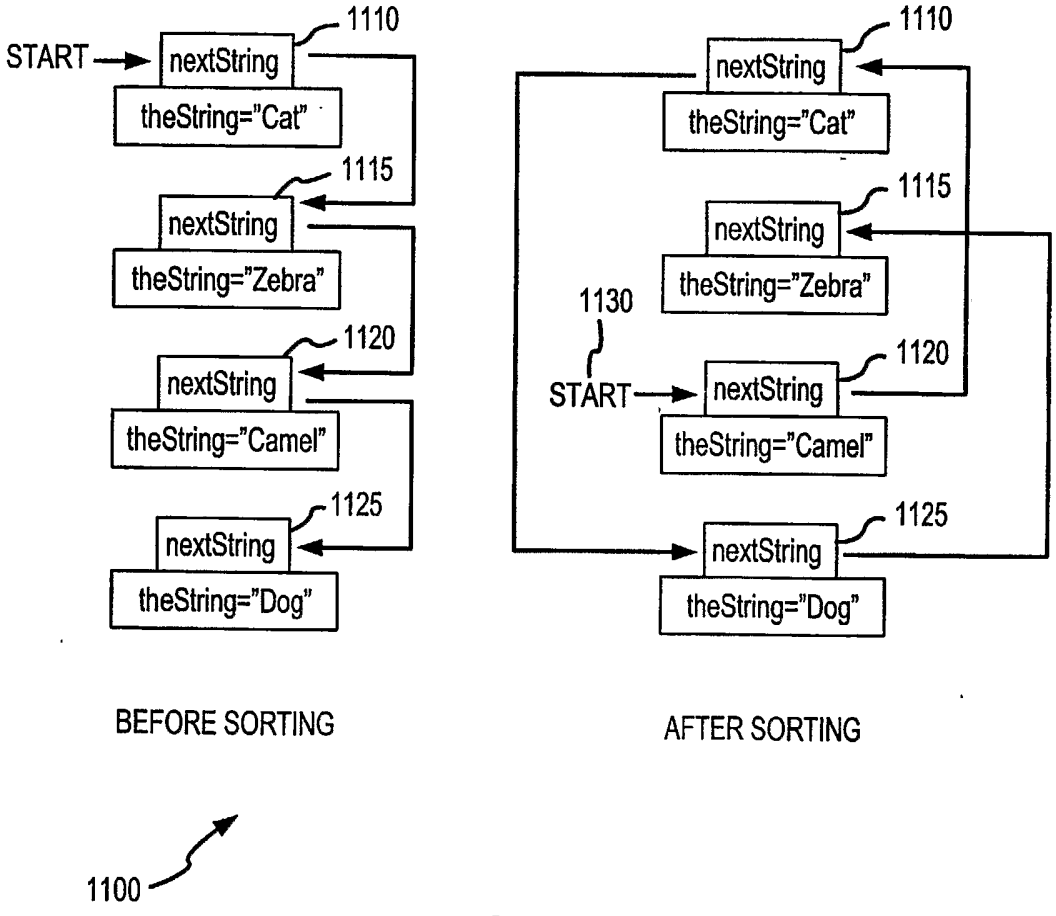

START
nextString
1110
theString="Cat"
nextString
1115
theString="Zebra"
nextString
1120
theString="Camel"
nextString
1125
theString="Dog"
BEFORE SORTING
1130
START
AFTER SORTING
1100

FIG.11

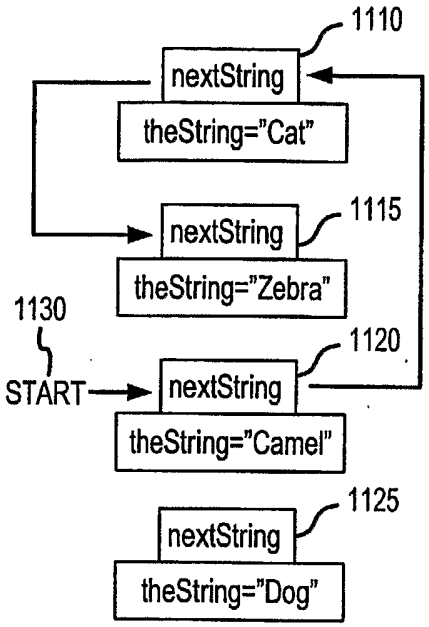


FIG.12A

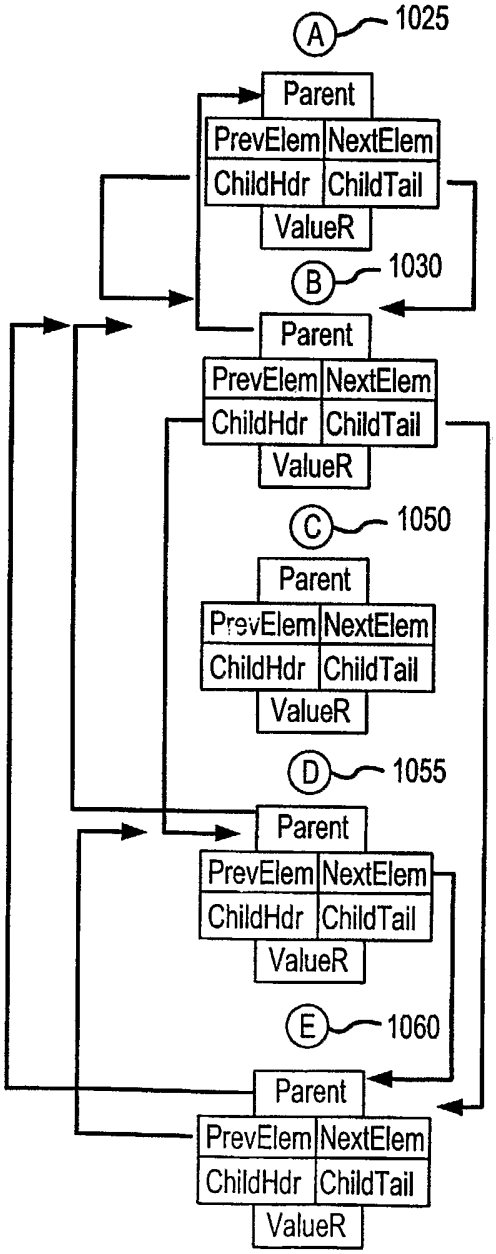


FIG.12B

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US03/03067

A. CLASSIFICATION OF SUBJECT MATTER

IPC(7) :G06F 17/30, 12/06
US CL : 707/100, 101, 103; 711/161
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 707/100, 101, 103; 711/161

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
NONE

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

WEST

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 5,680,616 A (WILLIAMS et al.) 21 October 1997, the entire paper is relevant	1-22
Y	US 5,628,007 A (NEVAREZ) 06 May 1997, the entire paper is relevant	1-22
Y, P	US 6,490,666 B1 (CABRERA et al.) 03 December 2002, the entire paper is relevant	1-22
Y, P	US 6,502,097 B1 (CHAN et al.) 31 December 2002, the entire paper is relevant	1-22

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 31 MAY 2003	Date of mailing of the international search report 19 JUN 2003
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer THUY PARDO Telephone No. (703) 305-1091

Form PCT/ISA/210 (second sheet) (July 1998)★