Available online at www.elixirpublishers.com (Elixir International Journal)

Computer Science and Engineering

Elixir Comp. Sci. & Engg. 54A (2013) 12737-12740

ISSN: 2229-712X

An Efficient Solution for Implementation of Property Lists in Programming Languages

Hassan Rashidi

Department of Statistics, Mathematics, and Computer Science, Allameh Tabatabe'i University, Tehran, Iran.

ARTICLE INFO

Article history:
Received: 9 July 2012;
Received in revised form:
15 March 2013;
Accepted: 15 March 2013;

Keywords
Programming Languages,
Property List,
Link List,
Set,
Hash,
Tree.

ABSTRACT

Supporting different data structures and their variations in both static and dynamic aspects are one of the challenges in programming languages. One of the data structures is property list of which applications use it as a convenient way to store, organize, and access standard types of data. In this paper, the solution methods for implementation Property List as Link List, Hash and Tree are reviewed. Then an efficient way to implement the property list as Set is presented and compared with the existing methods.

Introduction

Many applications require a mechanism for storing variable-size data objects of information in some situations [1]. A variable-size data objects is one in which the number of components in object may change dynamically during program execution. Some of the major types of variable-size data structures are list, list structure, stack, queue, tree, directed graph and property list.

This paper focuses on property lists. They are natural to use when the number and type of components in an object are not known in advance. Property List data structure supports many real-time applications when they read some data from an input device or change attributes of objects during program execution.

A *record* with a varying number of components is usually termed a *property list* if the number of components may vary without restriction [1]. In a property list, both the component names (field names) and their values must be stored. Each field name is termed a *property name*; the corresponding value of the field is the *property value*.

A common representation for a property list is as an ordinary linked list, with the property names and their values alternating in a single long sequence, as illustrated in Figure 1. In this Figure, the odd number items are property names, and the even are property values. There are three commands to process a property list:

- Inserting a new element to the list: When a new property is inserted in the property list, two components are inserted: the property name and its value.
- Removing an element from the list: To remove a particular property value (e.g., the value for the 'Name' property in Figure 1), the list is searched, looking only at the property names, until the desired property is found. A pair of component is then deleted from the list.
- Finding a value in the list: To select a particular property value (e.g., the value for the 'Age' property in Figure 1), the list is searched, looking only at the property names, until the desired property is found. The next list component is then the value for that property.

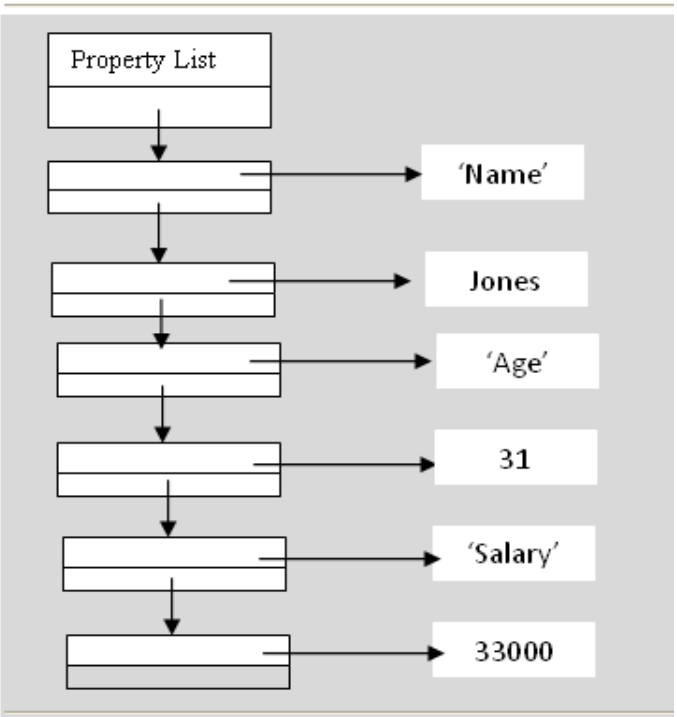

Figure 1: A storage representation of Property List

Property lists are used extensively by applications and other software on Mac OS X and iOS [3]. For example, the Mac OS X Finder (through bundles) uses property lists to store file and directory attributes. Applications on iOS use property lists in

Tele:
E-mail addresses: hrashi@googlemail.com

their Settings bundle to define the list of options displayed to users.

Property list is a simple XML format, designed by Apple for OSX as a format for storing lists of key-value pairs [4]-[7]. In this operation system, most applications store their Preferences as property list files. The property-list programming interfaces for Cocoa and Core Foundation allow the user to convert hierarchically structured combinations of these basic types of objects to and from standard XML. The user can save the XML data to disk and later use it to reconstruct the original objects.

Property lists are not part of the LISP language, but are an abstraction of common list patterns and usages, and are often defined as LISP library functions. Each item in a list is tagged with a name preceding the item like $(n_1\ val_1\ n_2\ val_2\ ...\ n_k\ val_k)$. In this list n_i is the property name of i^{th} element and val_i is the property value of i^{th} element.

This paper presents an efficient way for implementation of Property List in programming languages. Section 2 makes a literature review over the related work in implementation of property lists. Section 3 presents the detail of the efficient method and makes a comparison on the methods. Section 4 is considered for summary and conclusion.

1. The Related Work

A property list is a structured data representation used by Cocoa and Core Foundation [3] as a convenient way to store, organize, and access different types of data. Property lists organize data into named values and lists of values using several object types. This type of data structure gives the user the means to produce data that is meaningfully structured, transportable, storable, and accessible, but still as efficient as possible.

In this section, the existing methods for implementation of Property List are reviewed.

2.1. Property List as Link list

The first choice for implementation of property list is link list. For situations where a user needs to store small amounts of persistent data, for example less than a few hundred kilobytes, property lists offer a uniform and convenient means of organizing, storing, and accessing the data. In these situations, the simplest property-list implementation is a linked list (like Figure 1). The users can either have the alternating elements be the keys and values (LISP does this), or they can have each element be a structure containing pointers to the key and value. The linked list implementation is appropriate when the users:

- are just using the pattern to allow user annotations on object instances.
- don't expect many such annotations on any given instance.
- are not incorporating inheritance, serialization or meta-properties into their use of the pattern.

Logically a property list is an unordered set, not a sequential list, but when the set size is small enough a linked list can yield the best performance. The performance of the link list is O(N), so for long property lists the performance can deteriorate rapidly.

If the user needs a way to store large complex graphs of objects, objects not supported by the property-list architecture, or objects whose mutability settings must be retained, use *Archiving and Serializations*[3]. Archiving and serializations are two ways in which the user can create architecture-independent byte streams of hierarchical data. Byte streams can then be written to a file or transmitted to another process, perhaps over a network. When the byte stream is decoded, the hierarchy is regenerated. Archives provide a detailed record of a collection of interrelated objects and values. Serializations record only the simple hierarchy of property-list values.

2.2. Property List as Hash

The next most common implementation choice is a *hash-table*, which yields amortized constant-time on the operations of finding, inserting and removing for a given list, albeit at the cost of more memory overhead and a higher fixed per-access cost, i.e. the cost of the hash function. When a confliction occurs on inserting a new element in the list, the solutions of *Rehashing*, *Sequential Search* and *Bucketing* [1] may be used.

In most systems, a hash-table imposes too much overhead when objects are expected to have only a handful of properties, up to perhaps two or three dozen [6]. A common solution is to use a hybrid model, in which the property list begins life as a simple array or linked list, and when it crosses some predefined threshold (perhaps 40 to 50 items), the properties are moved into a hash-table. So if we need a tolerant constant-time on access an item and want to maintain the insertion order, we can't do better than a *LinkedHashMap* [5], a truly wonderful data structure. Java 6.0 implemented this solution. The complexity of this solution is O(1). However, the costs of hash-function and its overheads for solving confliction are inevitable.

2.3 Property List as Binary Tree

The third choice for implementation of property list is binary tree. If a language needs to impose a sort order on property names, it must use an ordered-map implementation, typically an ordered binary tree such as a splay tree or red/black tree [6]. A splay tree can be a good choice because of the low fixed overhead for insertion, lookup and deletion operations, but with the tradeoff that its theoretical worst-case performance is that of a linked list. A splay tree can be especially useful when properties are not always accessed uniformly. If a small subset M of an object's N properties are accessed most often, the amortized performance becomes O(log M), making it a bit like an Least Recently Used (LRU) cache [2].

2. Property List as Set

This section presents a new and an efficient solution for implementation of property lists. In some situations, the property-list architecture may prove insufficient [3] and inefficient. An efficient solution for implementation of property list is to present it as a set rather than a list because elements are accessed randomly by subscript (attribute name) rather than sequentially. A root property-list object is at the top of this hierarchy with a couple of pointers like Figure 2.

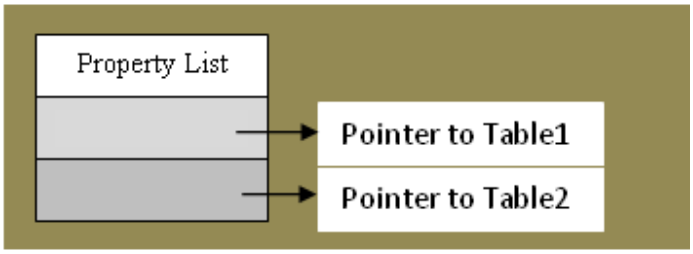

Figure 2: Representation of a property list object

In this solution, we define a data structure alike Table 1, consisting of *SetContents* to store the contents and Pointers to i^{th} element. We make OR operation together all the properties in the set and store this value in *SetContents*. The number of elements in the property list is variable with a limit on the maximum. For a 32-bit wordsize of memory, there will be 32 such pointers. The names are stored in a global table alike Table 2 with a bit value, representing its position in the set. If a

machine has a 32-bit word size, then up to 32 properties can be stored, with bit values of: 1, 10, 100, 1000, and so on. The Bit Strings are shown for clarity; i.e. they can be removed practically because they are i^n (n=0, 1, 2,.., 31) respectively. The commands to process the property list in this solution method are as follows:

- On lookup operation, we check if the bit string of the property name requested *AND SetContents* = 0, then the property is not defined in the set. If the value is 1, then the bit position defines the location containing a pointer to its attribute value.
- When a new element is to be inserted in the list, we must make a lookup as above. If the property name is in the list, duplication is not possible. If the result of the lookup is negative, the property name is put into an empty position in the Table 2. Then, we make *OR* operation the *SetContents* with the corresponding bit string of the property name. After that the corresponding Printer to the property value is set in Table1.
- When an existing element is to be removed from the list, we must make a lookup again. If the result of the lookup is positive, then we make an AND operation of *SetContents* with 00000..00 and store the result in *SetContents*. After that the memory for property name and property vale are freed and they set to *Null* in Tables 1 and 2. If the result of the lookup is negative (The result of 0), the element requested doesn't exist in the list.

Table 1: The Data Structure for the set

SetContents
Pointer to Property Value$_1$
Pointer to Property Value$_2$
……
……
Pointer to Property Value$_{32}$

Table 2: A Global Table with a bit string, representing the position of each property value

Bit String	Pointers to Property Name	
00…..01		→ Property Name$_1$
00…..10		→ Property Name$_2$
……		
……		
10…..00		→ Property Name$_{32}$

Since the property lists are based on an abstraction for expressing simple hierarchies of data, they can support the application programs. Some types are for primitive values and others are for containers of values. The primitive types are *strings, numbers, binary data, dates,* and *boolean* values. The containers are *arrays* and *dictionaries.* The arrays are indexed collections of values and the dictionaries are collections of values each identified by a key. The containers can contain other containers as well as the primitive types. Thus the user might have an array of dictionaries, and each dictionary might contain other arrays and dictionaries, as well as the primitive types. A root property-list object is at the top of this hierarchy, and in almost all cases is a dictionary or an array like Figure 3. Note, however, that a root property-list object does not have to be a dictionary or array; for example, the user could have a single *string, number,* or *date,* and that primitive value by itself can constitute an array or a dictionary.

Property list is extensively used in Markup Languages. From the basic abstraction, languages derive both a static representation of the property-list data and a dynamic (runtime) representation of the property list [3]. The static representation of a property list, which is used for storage, can be either XML or binary data. The binary version is a more compact form of the XML property list. In XML, each type is represented by a certain element. The runtime representation of a property list is based on objects corresponding to the abstract types. Both he static and dynamic representation can use our implementation.

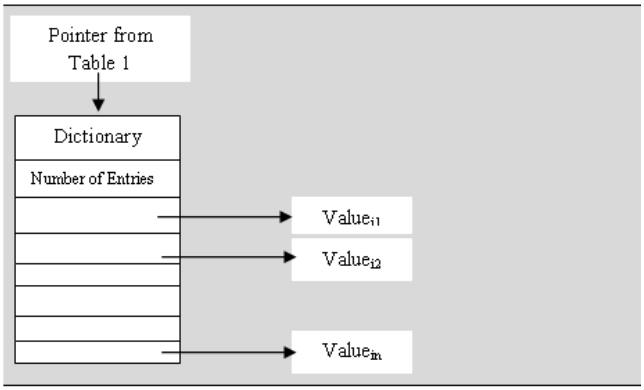

Figure 3: An array or dictionary of collections of values

The property lists are also used to define the SQL Queries in Database Management Systems and JDBC2 Components [8]. The solution presented here can by used in these systems and can create more efficient Dynamic Data Object.

The performance of this solution for all operations is O(1). On insertion and deletion operation of an element, although the performance of this solution is as the same as the hash-function, it benefits from the lookup operation and practically outperforms the hash because of no overheads.

Table 3 makes a summary on the main features and performance of the four solutions discussed in this paper. In the Link List solution, the property list is presented as an unordered set and has a lower difficulty for implementation. After that, the hash and binary tree are with a high and medium difficulty for implementation, respectively. Both solutions have some overheads in run-time during the operations. As shown in the table, our solution has no overheads on operations and its implementation is easy.

Table 3: A comparison among the solutions methods for implementation of property list

Solutions	Main Features	Complexity of Operations		
		Lookup	Insertion	Deletion
Link List	An unordered set, Low difficulty to implement	O(N)	O(1)	O(N)
Hash	High fixed overhead, High difficulty to implement	O(1)	O(1)	O(1)
Binary Tree	Low fixed overhead, Medium difficulty to implement	O(Log N)	O(Log N)	O(Log N)
Solution in this Paper (Set)	No overhead, Too easy to implement	O(1)	O(1)	O(1)

4. Summary and Conclusion

Basic differences among the languages refer to the types of data allowed, in the types operations available, and in the mechanisms provided for the implementation. Modern high programming languages need some data structures and their variations in both static and dynamic aspects. In this paper, the solution methods for implementation Property List as Link List, Tree and Hash are reviewed. Then a solution to implement the

property list as Set was presented. The solution proposed has more efficiency than the existing methods. In the construction of large application programs, the programmer is almost inevitably concerned with the design and implementation of new data types. The solution presented in this paper for implementation of Property List, supports the users so that they can have more flexible and efficient dynamic data objects. It can be used in both, programmer and programming languages levels.

References

[1]. Pratt T.W. and Zelkowitz M. (2001), "Programming Languages, Design and Implementation", 4th Edition Prentice Hall.

[2]. Pfaff B. (2004), “Performance Analysis of BSTS in System Software”, ACM Sigmetrics Performance Evaluation Review, Volume 32(1), pp. 410–411.

[3]. Mac OS X Reference Library, available on web at http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/PropertyLists/.

[4]. XML and Property Lists, available on web at http://www.satimage.fr/software/en/smile/xml/index.html.

[5]. Java Platform Standard Edition 6, available on web at http://download.oracle.com/javase/6/docs/api/java/util/LinkedHashMap.html.

[6]. Language features at the Emerging Languages camp, available on web at http://planets.sun.com/AllRuby/group/jruby/.

[7]. Lindfors J. (2002), " JMX: Managing J2EE with Java™ Management Extensions", Sams Publishing.

[8]. Stark S., and the JBoss Group (2002), "JBoss Administration and Development", 2nd Edition, JBoss.