PERIODICA POLYTECHNICA SER. TRANSP. ENG. VOL. 31, NO. 1–2, PP. 97–105 (2003)

CONSTRUCTION OF OPTIMAL DATABASES FOR RAILWAY-STATION RESTORATION KNOWLEDGE MANAGEMENT SYSTEMS

István PÁLYI

Department of Control and Transport Automation
Budapest University of Technology and Economics
H–1521 Budapest, Hungary
Phone: +36 1 463 1013, Fax: +36 1 463 3087

Received: October 1, 2002

Abstract

This paper presents database structures with fast access time and easy extensibility for railway-station data storage. These structures will help to build optimal strategies for reconstruction plans and provide data for design software, such as the world famous ARCHICAD, which is straightforward in using in designing architectural restoration, based on pre-existent architectural plans. The implementation of the modern database structure was realized with the aid of an object-oriented programming language, due to the flexibility of its file-management system.

Keywords: railway-station architecture, mathematical models, object-oriented environment, dynamical data-storage systems.

1. Introduction

1.1. Object Oriented Programming

In Hungary, the reconstruction of railway stations represents an important task for the near future. Unfortunately, these impressive architectural monuments had a lot to suffer during this century. Many of them were destroyed during the war, others were reconstructed without the necessary expert knowledge. Meanwhile, the existing railway stations, as well as those that can be reconstructed, represent an important part of the architectural treasure of Hungary. Their preservation and restoration is on one hand a task of great importance from cultural and historical point of view, and on the other hand it is an issue closely related to the continuous effort to improve the image of the country and to increase its turistic potential.

Our paper was written in the spirit of these considerations, noting also that during the last decade the planning of architectural restoration in industrial and transportation fields experienced an astonishing development compared to previous periods of time. This work relays on the properties of certain programming languages (closely related to the mathematical formalisms), namely the fact that dynamic data storage facilitates the management of large databases. The paper also presents the most important steps in the implementation of a database, as well as

the fundamental notions related to this problem. The graph theoretical model used in the building restoration process is based on the database model presented below, and will be thoroughly described in a future paper. In the designing and implementing the application, the modern paradigms of object-orientation programming were followed, which also facilitate the ulterior development of this project.

1.2. The Object, as Algebraical Structure

The object is a new mathematical notion, fundamental for information technology.

Definition 1 The non-empty set Q is called object, if the following conditions are defined:

$S : Q \to Q$	finite set S of selector mappings, and
$k : Q \times S \times Q \to Q$	constructive mapping.

Observation 1

1. The operations (S) project the data structure to its parts (for example, the extraction of an element is a selection operation with respect to the extracted element).
2. The constructive operation (k) creates and updates a data structure, or maps it to a similar data structure.

1.3. The Typical Structure of an Object in Information Technology

An object is the aggregation of FIELDS/ATTRIBUTES (representing data) and METHODS.

1.4. The Four Basic Properties of an Object

1. Encapsulates data and code (Field + method).
2. Is endowed with inheritance.
3. Is endowed with polymorphism.
4. Presents the closeness property.

1.5. The Origins of Object-Oriented Programming (OOP)

In the history of information technology and computer science, the first reference to object oriented programming (OOP) goes back as far as 1970. The first programs developed according to this paradigm were:

1. SMALLTALK Pr. developed at Xerox (which was quite slow)
2. C++ (fast, but not completely OOP)
3. Other programs (LISP, SIMULA, MODULA-2, ADA)
4. For the case of Turbo Pascal, first the 5.5 version introduced OOP features.

In 1985, Larry Tesler was the one who first formulated this idea, in his work called 'Object Pascal Report'. The 5.5 version (1985) was fast and fully object-oriented.

2. The Structure of the Database

The model assumes n – (railway station) buildings and is able to store for each building the restoration costs, the expected duration of the work, the direct and indirect profit gained as a consequence of restorations, the technical drawings' data necessary for the design, as well as physical parameters, and also textual information for further processing. The database model is a dynamic chain, which can be updated any time, by inserting a new element.

2.1. Creation of the Database and the Use of Dynamic Variables (Pointers)

Unfortunately, static variables cannot be used to implement our method, due to the dimension of the problem. The use of static variables presents the following drawbacks:

- The size of static variables is decided at compile-time.
- During run-time, the size cannot be modified.

When creating the database, it was taken into consideration that the source code of the Turbo Pascal program is divided into multiple segments, having each a maximal size of 64 kilobytes. So, one can write a code larger than 64 Kbytes, only if the program is divided in several parts, called modules/units.

The 64 kilobytes upper limit of the segment data represents a very serious restriction for solving our problem. The memory occupied by static data defined in the program (global variables and typed constants) cannot exceed 64 Kilobytes. This restriction can be relaxed only by the use of dynamical variables.

Furthermore, the stack has also a maximal dimension of 64 Kbytes, which is a quantitative upper bound for the local variables used in functions and procedures.

On the other hand, a program can use a 640 kilobytes heap!

Thereby it is necessary that the program is able to manage itself a certain amount of memory in order to have a good performance. This is done in the following way:

1. If we need memory for one of the variables, a memory chunk is allocated and used.
2. If the allocated memory is not needed anymore it can be freed (made available for further use).

In this way the dynamical management of the memory is realized. In Turbo Pascal, the memory where the dynamic variables are allocated is called *heap*. The size of this memory chunk is much larger than the size of those allocated to other segments of the program. The only problem that has to be clarified is how to access the heap. This task is solved by the use of a new type of variable, called pointer.

Definition 2 The pointer is a special (4 byte long) variable, which takes as value a memory address.

2.2. *Declaring a Pointer:* `var p: ^Integer`

The ^ symbol can be followed by any standard or user-defined type. After declaration, the pointer `p` is not initialized, so it does not contain any valid address (it does not point anywhere). A value can be assigned to the pointer in several ways, the most being straightforward the use of the standard procedure *new*, which allocates memory and sets the address of this memory as the value of the pointer-parameter `p` in the instruction: `new(p);`.

In Pascal exists a predefined constant nil, used to denote the null pointer. The following assignment is allowed: `p:=nil;`. If memory allocation was successful, the memory location pointed by `p` can be used as in the following example:

```
Example:  p^ :=77;
          p^ :=p^ +15;
```

It is easy to notice that by using the symbol ^ after the name of the pointer, we mean the value stored at the location indicated by the pointer, and not the value of the pointer itself. If the integer variable `p^` is not needed anymore, then by the aid of the procedure *dispose* we can erase it: `dispose(p);`. After calling the *dispose* procedure, as well as before calling *new*, the use of `p^` may lead to a programming error. An essential characteristic of pointers is that they are bounded to a certain type. This type determines the amount of memory allocated during the dynamic allocation of the pointer. The real advantage of this feature can be observed when using array variables, with size of several kilobytes. (If there is no sufficient space on the heap, the program will stop with the following run-time error (203) 'Heap overflow error'). The error message can be avoided, if before calling *new* we check if there is enough free space on the heap. One can compare the value returned by the function 'maxavail', representing the maximal size of the free blocks of memory available, with the size required for the storage of the integer variable of our program:

```
if maxavail < sizeof(integer) then halt else new(p);
```

By the aid of the 'memavail' function we can find out the total size of the available free space on the heap.

2.3. Array on the Heap

The array of type 'realarray' (of size 60.000 byte) is placed on the heap by the aid of the variable 'p'. The array itself is denoted p^, and the elements of the array can be accessed via the expression: `p^ [i]`.

```
program pltomb;
type realarray = array [1.  .10000] of real;
point= ^ realarray,
var p:  point;
i:  integer,
begin
new(p);
for i := 1 to 10000 do p^ [i]:=i*pi ;
dispose(p)
end.
```

2.4. The Use of Pointer Array

If we would like to use more than one array of the type described in the previous section, or even as many as would fit into the memory, then it is advisable to use an array of pointers, instead of some pointer-to-array type variables. This way, `p[i]^` will reference the `i`-th array, while the `j`-th element of the `i`-th array is denoted by the expression `p[i]^[j]`.

3. List-Container Structure and Dynamic Chains

The optimal dynamical management of data space can be achieved by the use of list structures. For the Pascal implementation of lists the record type is used. *The list constructed in this way is a chain of records.* A list consists of elements, which contain data and a pointer to the next element in the list.

3.1. More Information on Pointers

Generally, pointers are defined having a certain variable type, but also pointers without any type information exist:

```
var p1:^ integer; { with type }
p2 :pointer; { without type }
```

In both cases the following operations are valid.
Assignment to the object defined by the pointer (the object pointed by the pointer):

```
p1^ :=57;
```

Assignment of the pointer:

```
p2:=nil; p1:=p2;
```

Comparison: For pointers to the same type the operations = and <> are valid.

The nil constant pointer, as well as the pointers defined with no type information can be compared with any other pointer:

```
if (p1 = p2) and (pl <> nil) then halt;
```

3.2. The Use of the Addr Function and the @ Operator

The *Addr* function, and the corresponding @ operator return the address of any Pascal object, and the result may be assigned to a pointer (of arbitrary type). E.g.:

```
var
ip:  ^ integer,
i:  integer,
begin
i:=5;
ip:=@i; {or ip:=Addr(i);}
ip^ :=i+6;
writeln(i);
{The output of the program:  11,}
end.
```

4. The Principles for the Use of a List of n Buildings

The list formed by n buildings is determined by the following type definition and by *Fig. 1*:
type

```
link = ^building;
building = record
    ....
    next :  link;
end;
```

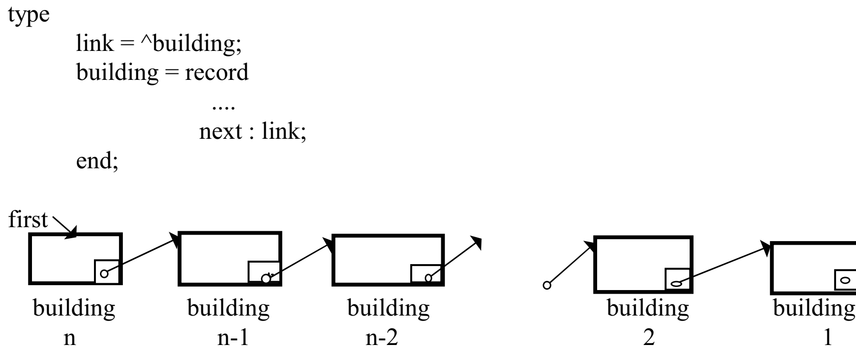

Fig. 1. Chained list

The first element – the head – of the list is pointed by the *link* typed variable, called *first*. The last element of the list points to nil. If we assume that the input file contains *n* characteristics (numbers), the previous chain can be constructed, for example, by the following code:

```
var first, p:  link; i:  integer;
first:=nil;
for i:= 1 to n do
begin read(s); new(p); p^ .next:=first; p^
     .characteristic=s;
     first:=p;
end.
```

In order to solve the insertion problem described above, the first step is the definition of a pointer variable, which we shall call newp. Next, by using the instruction:

```
new (newp);
```

we allocate memory for a new *building*-type variable. In the following step, the new variable – pointed by the pointer newp, has to be inserted in the chain, at the location following the element pointed by *pt* (see *Fig.* 2).
Insertion consists of actualization of pointers' values:

```
newp^.next:=pt^.  next;
pt^ .next:=newp;
```

The result is presented in *Fig.* 3.

The element that follows the one pointed by the auxiliary pointer variable *pt* can be erased (ignored) by performing the following assignment:

```
pt^.next:=pt.next^.next.
```

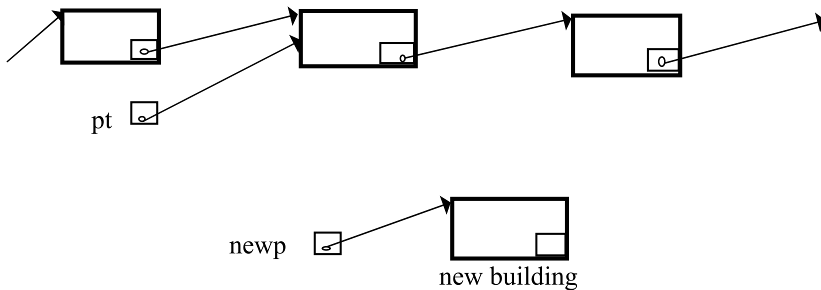

Fig. 2. Before insertion

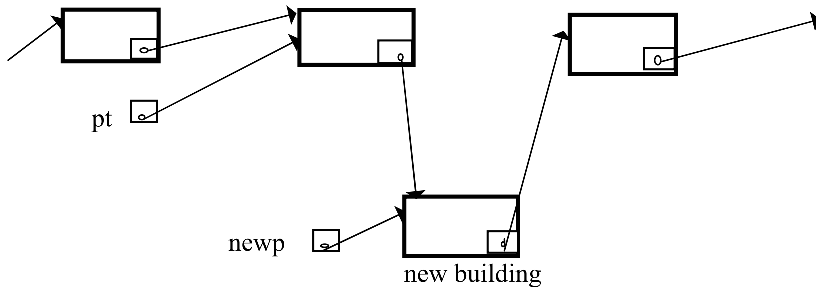

Fig. 3. After insertion

Often, it is advisable to process a list by using two pointer variables pointing to successive elements of the list. In order to delete an element, one pointer $p1$ is assigned the address of the element previous to the one that has to be deleted, and the other, $p2$ to the element that should be deleted.

5. Summary

The paper is a detailed presentation of information needed by the restoration data storage method. The aim of the method used by the author is to ensure fast access even for large databases containing the railway station building data.

The records in the database may contain image data, as well as physical parameters, textual information etc. The data is stored in a dynamic chain structure, allocated on the heap, and can easily be retrieved, evaluated, and continuously updated by using this structure. Besides the fast access to data, the purpose of this investigation is the construction of a railway station architecture database necessary for a decision making information system that implements optimal restoration strategy.

References

[1] JENSEN, K. – WIRTH, N., *PASCAL User Manual and Report*, Springer-Verlag, New York Inc. 1980.
[2] *TURBO PASCAL Object-Oriented Programming Guide*, Borland Inc., 1988.
[3] PIRKO, J., Turbo Pascal Objektum-Orientált Programozás, LSI központ., 1990.
[4] KUBINSZKY, M., A magyar vasút és az építési szabványtervezés kialakulása, *Közlekedéstudományi Szemle 6.*, **XLVIII** (1998), (in Hungarian).
[5] PÁLYI, I., Dinamikus lánc létrehozása vasúti indóházak optimális adatbázisának kialakítására, *Közlekedéstudományi Szemle 4.*, **L.** (2000), pp. 125–129, (in Hungarian).
[6] MARCEL, P., Modeling and Querying Multidimensional Databases: An Overview, *Networking and Information Systems Journal*, **2** No. 5–6 (1999), pp. 515–548.
[7] PARSAYE, K. – CHIGNELL, M., *Intelligent Database Tools and Applications*, John Wiley & Sons, Inc., New York, 1995.
[8] YOURDON, E. – ARGILA, C., *Case Studies in Object Orientated Analysis and Design*, Prentice Hall, 1996.
[9] PÉTER, T. – ZIBOLEN, E., Analysis of Model in Vehicle Dynamics in Computer Algebraic Environment, *6th Mini Conf. on Vehicle System Dynamics, Identification and Anomalies*, Budapest, Nov. 7–10, 1998, pp. 305–314.
[10] PÉTER, T., Mathematical Transformations of Road Profile Excitation for Variable Vehicle Speeds. *Studies in Vehicle Engineering and Transportation Science. A Festschrift in Honor of Professor Pál Michelberger on Occasion of his 70th Birthday*. Hungarian Academy of Sciences – Budapest Univ. of Technology and Economics 2000, pp. 51–69.
[11] MICHALETZKY, GY. – BOKOR, J. – VÁRLAKI, P., *Representability of Stochastic Systems*, Akadémiai Kiadó, Budapest, 1998; p. 236.
[12] KÓCZY, L. T. – NÁDAI, L. – VÁRLAKI, P., Fuzzy System Identification for Cognitive and Decision Processes, *IEEE SMC'98*, San Diego, USA, 1998.