DOI: 10.5281/zenodo.4644565

OPTIMIZATION OF APPLICATION OBJECTS USED IN THE ECONOMIC ENVIRONMENTS

Emilia VASILE, PhD Professor
Athenaeum University, Bucharest, Romania
rector@univath.ro

Dănuţ-Octavian SIMION, PhD Associate Professor
Athenaeum University, Bucharest, Romania
danut_so@yahoo.com

Abstract: *The paper presents the Optimization of application objects used in the economic environments. The dynamic character of the economic system necessarily determines a dynamic character of the economic information system. Initially, variations in the behavior of the economic system may be disruptive to the information system, but due to its existence as a cyber system, the information system can adapt and function in full accordance with the economic system. A part of the phases of the informational process (for example the collection of data from the primary evidence) is performed within the operational (managed) subsystem of the economic system. It can be appreciated that the subsystem in which the information process takes place together with a part of the operational subsystem forms the managed subsystem of the cybernetic information system. The character of the cyber system is strengthened by the fact that the economic information system has its own objectives, methods, techniques and resources. At the level of the economic information system, at least two interdependent subsystems can be identified, which ensure the stability of the entire system: the managerial information subsystem and the operational information subsystem. There are many types of data in the economical environment such as simple chained lists that offer a lot of flexibility in storing data. Using these types of data permits the optimization of economical flows designed for economical applications that are implemented in organizations that invests in flexible and scalable information systems.*

Keywords: *economic system, simple chained lists, economical data, information processes, business flows, application modules, analysis of data.*
JEL Classification: *C23, C26, C38, C55, C81, C87*

1. Introduction

An organization must maintain appropriate relationships with other economic, political, and social systems in its environment. The systems group includes several factors such as customers, suppliers, competitors, shareholders, trade unions, financial institutions, government agencies and communities, each with its own objectives relative to the organization concerned. Information systems are those that facilitate the interaction between the organization seen as a system and each of the factors listed.

On the other hand, considering the definition of a cybernetic system (characterized by the existence of at least two subsystems between which there is self-regulation by reverse connection) and analyzing this for the economic information system we can see that it has a cybernetic character. Seen as a system, any organization comprises six interdependent system components: Input - economic resources such as human, financial, material, machinery, land, facilities, energy and information, which are taken from its environment and used in system activities.

Transformation function - organizational processes such as research, development, production, marketing, sales, which transform input into output.

Output - the results of the transformation function, which consist of products and services, payments of employees and suppliers, dividends, contributions, taxes and information to the external system (environment).

Feedback - is the defining element of a cybernetic system, which provides the function of self-regulation, when the output does not correspond to the objectives set (Z) within the system represented by the economic organization (Sandner and Vukics, 2020; Orszag, 2020).

Control - management is the control component of an organizational system, which aims at the functions of the enterprise so that the performance of the system reaches the organizational objectives (such as profitability, market share or social responsibility).

Environment - any economic organization is an open, adaptable system that shares input and output elements with other systems in its environment (Sanderson, 2019; Steeb and Solms, 2021).

2. The simple chained lists used to store and model data in applications

Simply chained lists are homogeneous dynamic data structures. Unlike massive ones, lists are not allocated as homogeneous blocks of memory, but as separate elements of memory. Each node of the list contains, apart from the useful information, the address of the next element. This organization allows only sequential access to list items.

To access the list you must know the address of the first element (called the head of the list); the following items are accessed by scrolling through the list (Reddy, 2020; Grant ,2021).

List structure - In order to ensure a greater degree of generality to the list, an alias was created for the useful data (in our case an integer):

```
// Data associated with a
// item in a list
typedef int Date
```

If you want to store another type of data, you only need to change the declaration of the alias Data.

A self-referenced structure is used to store the list. This structure will take the form of:

```
// The structure of an element
// from a simply chained list
struct Element
{
  // the actual data stored
  Value data;
  // link to the next node
  Next item *;
};
```

If the element is the last in the list, the next pointer will have the value NULL.

The declaration of the list is made in the form:

```
// declare life list
Element * cap = NULL;
```

List operations - The main operations with lists are:

Browse and display the list

The list is traversed starting from the pointer to the first element and advancing using the pointers in the structure to the end of the list (pointer NULL).

```
// Scroll and display the simple list
void Display (Item * cap)
{
  // as long as we have elements
  // in the list
  while (cap! = NULL)
  {
   // displays the current item
   cout << cap-> value << endl;
```

```
   // advance to the next item
   cap = cap-> next;
  }
}
```

Insert item - Inserting an item can be done at the beginning or end of the list.

a) Insertion at the beginning

This is the simplest case: you just need to allocate the item, related to the first item in the list and reposition the head of the list:

```
// Insert element at the beginning of a
// simply chained lists
void InsertStart (Item * & cap, Wave data)
{
  // Node allocation and value initialization
  Element * elem = new Element;
  element-> value = wave;

  // link node in list
  elem-> next = head;

  // move the head of the list
  cap = elem;
}
```

b) Insertion at the end of the list

In this case you must first go through the list and then add the item and link to the rest of the list. Also, the case if the list is empty must be considered.

```
// Insert element at the end of a
// simply chained lists
void InsertEnd (Item * & head, Wave data)
{
  // Node allocation and initialization
  Element * elem = new Element;
  element-> value = wave;
  elem-> next = NULL;

  // if we have a live list
  if (cap == NULL)
   // just change the head of the list
   cap = elem;
  else
 {
   // scroll through the list to the last node
   Element * node = head;
   while (nod-> next! = NULL)
    nod = nod-> next;
```

```
   // add the new item to the list
   nod-> next = element;
  }
}
```

c) inserare dupa un element dat

```
void InsertInterior (Element * & cap, Element * p, Date wave)
{
  // Node allocation and initialization
  Element * elem = new Element;
  element-> value = wave;
  elem-> next = NULL;

     // life list
  if (cap == NULL)
  {
   cap = elem;
   return;
  }

  // insert at the top of the list
  if (cap == p)
  {
   elem-> next = head;
   cap = elem;
   return;
  }

  // insert inside
  elem-> next = p-> next;
  p-> next = item;
}
```

Item search - Searching for an item in a list involves going through the list to identify the node according to a criterion. The most common criteria are those related to the position in the list and the useful information contained in the node. The result of the operation is the address of the first element found or NULL.

a) Search by position

Advance the pointer with the specified number of positions:

```
// Search for item by position
Item * SearchPosition (Item * head, int position)
{
  int i = 0; // current position

  // scroll through the list to
  // required position or up to
```

```
  // end of list
  while (cap! = NULL && i <position)
  {
   cap = cap-> next;
   i ++;
  }

  // if the list contains the item
  if (i == position)
   return cap;
  else
   return NULL;
}
```

b) Search by value

The list is scrolled until it is exhausted or the element is identified:

```
// Search for item by value
Item * SearchValue (Item * head, Wave data)
{
  // scroll through the list until you find it
  // item or list exhaustion
  while (cap! = NULL && cap-> value! = wave)
   cap = cap-> next;

  return cap;
}
```

Delete item

a) Deleting an item from the list (other than the list head)

In this case we need the address of the predecessor of the element to be deleted. The connections in the sense of short-circuiting the deleting element are modified, after which the memory corresponding to the deleting element is released:

```
// delete an item from the list
// receiving as a parameter the address of the predecessor
void DeleteElementInterior (Predecessor Element *)
{
  // save the reference to the delete element
  Element * deSters = predecessor-> next;

  // short-circuit the element
  predecessor-> next = predecessor-> next-> next;

  // and delete it
  delete deSters;
}
```

b) Deleting an item from a certain position

If the item is the first in the list, then the head of the list is modified, otherwise the item is searched and deleted using the previously defined function:

```
void Delete Position (Item * & head, int position)
{
  // if the list is empty we don't do anything
  if (cap == NULL)
   return;

  // if it is the first element, then
  // wipe it and move the head
  if (position == 0)
  {
   Element * deSters = none;
   cap = cap-> next;
   delete deSters;
   return;
  }

  // if it's inside, then we use
  // delete function
  Element * predecessor = SearchPosition (head, position-1);
  DeleteElementInterior (predecessor);
}
```

c) deletion after a value

Search for the element's predecessor and use the element deletion function:

```
void DeleteValue (Item * & cap, Date wave)
{
  // if the list is empty we don't do anything
  if (cap == NULL)
   return;

  // if it is the first element, then
  // wipe it and move the head
  if (head-> value == wave)
  {
   Element * deSters = none;
   cap = cap-> next;
   delete deSters;
   return;
  }

  // looking for the predecessor
  Element * elem = none;
  while (element-> next! = NULL && element-> next-> value! =
```

```
wave)
   elem = elem-> next;


  // if it was found, then we delete it
  if (elem-> next! = NULL)
   DeleteElementInterior (elem);
}
```

3. Methods of implementation for stored data

Queues and stacks are logical data structures (implementation is done using other data structures) and homogeneous (all elements are of the same type). Both structures have two basic operations: adding and removing an element. Apart from these operations, other useful operations can be implemented: vacuum structure test, obtaining the first element without extracting it (Sandner and Vukics, 2020; Steeb, 2021). The fundamental difference between the two structures is the access discipline. The stack uses a LIFO (Last In First Out) access discipline, and the queue uses a FIFO (First In First Out) discipline (Del Nero, 2020; Chand, 2020).

Stacks and tails can be implemented in several ways. The most used implementations are those using massive and lists. Both approaches have advantages and disadvantages.

To implement a stack using massive we need a massive V of size n to store the elements. The last element of the mass will be used to store the number of elements of the stack (Reddy, 2020; Steeb, 2021).

If the stack is empty, then the element Vn-1 will have the value 0. Using this representation, the basic operations can be implemented in constant time.

The algorithms for implementing basic operations (in pseudocode) are:

```
add (elem, V, n)
if v [n-1] = n-1 // check if the stack is not full
  return "full stack"
v [v [n-1]] = element // we add the element in bulk
v [n-1] = v [n-1] + 1 // we increase the number of elements
return "success"

  delete (V, n)
if v [n-1] = 0 // check if the stack is not empty
  return "empty stack"
elem = v [v [n-1] - 1] // extract the element from the massif
v [n-1] = v [n-1] + 1 // we decrease the number of elements
return elem
```

The queue can be implemented using a circular vector of size n (element n-4 is followed by element 0). The last two elements contain the start and end

indices of the queue, and the penultimate element is a marking used to be able to differentiate between empty tail and full tail cases.

The algorithms that implement the basic operations for a queue stored in the presented form are:

```
add (elem, V, n)
v [n-2] = (v [n-2] + 1) mode (n-2) // move the tail head
if v [n-1] = v [n-2] // check full queue
  return "full queue"
V [V [n-1]] = element // we add the element
return "success"

delete (V, n)
if v [n-1] = v [n-2] // empty queue check
  return "empty tail"
v [n-1] = (v [n-1] + 1) mode (n-2) // move the end index
return V [V [n-1]] // return the element
```

The second way to implement stacks and queues is to use dynamically allocated lists.

In the case of the stack, we will use a simple chained list organized as in the following:

Each node consists of useful information and a link to the next item. The type of information stored in the stack is indicated by the user by defining the TipStiva type. The empty stack is represented by a null pointer. Items are added before the first item (by moving the top of the stack). The extraction is also done from the top of the stack (Sandner and Vukics, 2020; Grant ,2021).

The source code for the library that implements the operations on the dynamically allocated stack is:

```
// An element from the stack
struct NodStiva
{
 TipStiva Date; // user defined type
 NodStiva * Next; // link to the next item

 // constructor for initializing a node
 NodStiva (TypeStiva data, NodStiva * next = NULL):
  Date (s), Next (next) {}
};

// The stack is stored as a
// pointer to the first element
typedef NodStiva * Stiva;

// Create the stiva life
StCreare stack ()
{
 return NULL;
}
```

```
// Check if a stack is empty
bool StEGoala (Hold & hold)
{
 return stack == NULL;
}

// Add an item to the stack
void StAdauga (Hold & Stack, DateStype Tip)
{
 stiva = new NodStiva (date, stiva);
}

// Returns a copy of the top of the stack
TipStave StVarf (Hold & hold)
{
 // Case 1: stiva vida
 if (StEGoala (stack)) // if the stack is empty, then
  return ActivateType (); // return the default value for the
stack type

 // Case 2: empty stack
 return stack-> Date; // turn the top of the stack
}

// Extract the element from the top of the stack
StExtrage Hold (Hold & Hold)
{
 // Case 1: stiva vida
 if (StEGoala (stack)) // if the stack is empty, then
  return ActivateType (); // return the default value for the
stack type

 // Case 2: empty stack
 NodStiva * nodDeSters = hold; // save a reference to the de-
lete node
 TypeStop Res = Stack-> Date; // save the data to be returned

 stiva = stiva-> Next; // we advance to the top of the list

 delete nodDeSters; // delete the delete node

 return rez; // return the results
}
```

Achieving good economic efficiency by enterprises is conditioned by the existence of scientific leadership based on a good knowledge of economic laws, operational and accurate knowledge of supply and demand in the internal and external market, the dynamics of commodity prices, technological trends and how use of the resources at their disposal (Reddy 2020; Orszag 2020).

4. Conclusions

Starting from the fact that, on the one hand, mathematical models represent the scientific component of an information system, and on the other hand, taking into account the facilities offered by the use of information and communication technology (ICT) as a component of the information system, it trace is a real tool in the scientific management of economic activity (Steeb and Solms, 2021); Grant, 2021). There are some of the arguments put forward in favor of the management of economic organizations using information systems such as offering the possibility to simulate economic processes and phenomena both at microeconomic level and at macroeconomic level. Mathematical models can be developed and implemented regarding the forecast of economic development, different plan variants can be elaborated and then the optimal variant can be chosen (Sanderson, 2019; Steeb and Solms, 2021). At the microeconomic level, with the help of SI the available resources are harmoniously correlated with the proposed objectives, ex: planning of overhauls and capital repairs, scheduling scheduling and production tracking, inventory management. The efficient way to store data in stacks, lists or queues offers many advantages and flexibility for the business logic in applications.

References

Grant, R. 2021. *An interface between Stata and C++, with big data and machine-learning applications*; ideas.repec.org.

Orszag, J.M. 2020. *Fortran, C and C++ Code for Econometrics and Optimisation Applications.* Oxford University Press.

Reddy, M. 2020. *API design for C++*. Elsevier; books.google.ro.

Sanderson, C. 2019. *An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*; www.researchgate.net.

Sandner, R., Vukics, A. 2020. **C**++QEDv2 Milestone 10: A C++/Python application-programming framework for simulating open quantum dynamics. *Computer Physics Communications,* 185(9), pp. 2380-2382. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0010465514001349?via%3Dihub.

Steeb,W-H., Solms, F. 2021. *Applications Of C++ Programming Administration, Finance and Statistics*; https://ideas.repec.org/b/wsi/wsbook/2798.html.