Educational Issues
Въпроси на преподаването

ABSTRACT DATA TYPES

Lasko M. Laskov
New Bulgarian University (Bulgaria)

Abstract. Formation of notion is a fundamental process in education, and for that reason it is excessively studied in both psychology and pedagogy. The application of the pedagogical techniques has a key role in education in informatics (Asenova, 1990) and has significantly improved the results in introduction of complex notions in our practice. Such complex notion is abstract data type (ADT) which is a key concept in computer programming for developing data structures, data types, and have vast influence on the algorithms applied on them. In order to introduce the notion ADT we adopt a system of tasks that develop the needed knowledge through the important data structures of linked lists, queues and stacks.

Keywords: system of tasks; teaching through tasks; concept formulation; informatics education; computer programming; object-oriented programming; abstract data types

Introduction

Notions have a fundamental role in development of the scientific knowledge. For that reason, the process of notion formation has been studied by number of researchers in the fields of psychology and pedagogy (see (Usuva, 2011), (Vygotsky, 1987), (Aleksandrov, 1999), (Rubinstein, 1946), (Davydov, 1996) and the cited literature in these works). One of the basic goals of teaching is the formation of the notion apparatus in the learner. According to Usova (see (Usuva, 2011)) in the complicated spiral process of formation and development of a given notion, the following distinct 11 steps can be followed:

1. Specific perception: observation of the objects, demonstrations by the lecturer. The attention is drawn towards the features and links between the observed objects.

2. Discovery of common essential features of the class of observed objects.

3. Generalization. From the specific examples a transition to abstract conclusion for the common essential features of the class of objects.

4. The definition is formulated. Whenever possible, a generic term and a species distinction are used.

5. Consolidation of the essential features. It is achieved through a group of exercises aimed at:

– Essential features.

– Studying of objects that have common essential and unessential features. From all features, the essential are selected. The objects are distinct based on the unessential features. These unessential features of the examined objects help the distinction of similar notions.

6. Link of the examined notion with other notions. Conclusions are made organized, based on which these links are introduced. Graphs and formulas can be analyzed here.

7. Application of the notion in simple situations based on simple tasks. The exercises reveal also the links to other notions.

8. Classification of notions. The goal is to give the general links of the notion in a general system of notions. This helps the comprehension of the role of the classification for arrangement and systematization of the knowledge.

9. Application of the notion for solving of creative tasks. Complicated tasks are solved. A link to other systems of notions are given from the same discipline or from other disciplines.

10. Enrichment of the notion. The notion is enriched with new essential features towards its complete fullness. This process may continue in a spiral manner in time.

11. The studied notion is used as a formulation of new notions. In this manner the studied notion is developed continuously and is included in new links and new systems of notions.

The learners who have passed through the above stages during the assimilation of the notions show solid knowledge, understanding of the notions and skills to use them in nonstandard situations. This shows that these stages can be used as a goal in the development of system of tasks for given notion formulation. In (Assenova & Marinov, 2019) it is given such an example is presented in the case of notions formation in the education in mathematics.

We will note that some of the notions cannot be formulated in a single discipline. Besides that, it is not always necessary to fully formulate a given notion to serve the goals of the program. For example, many of the notions in the informatics education in the high-school remain on the level of perception, concept, or generalized concept. These are the first three stages, according to Usova. In this case the learners themselves generalize the essential features of the class of objects, and the generalization is separated from the precise examples that cause it. In this case, the level of abstraction that contains the minimal features that define the notion is not reached yet. In (Albertovna, 2017) the authors, referring to the studies of Vygotsky, call this level *pre-notion*. Pre-notion opens the possibilities for assimilation of given skills and technologies. This gives the basis for the development of an abstract definition.

The notion ADT

The notion abstract data type (ADT) is considered important in computer programming ever since the development of structured programming as a key concept

in controlling the complexity with which the programmer have to cope with. An early work by Dijkstra (Dijkstra, 1972) emphasizes the significance of the introduction of a structures in software development, especially in the case implementation of large projects. ADT are shown to play a key role in this process of structures incorporation (Guttag, 1977) as a mechanism that allows the separation of behavior of a data type from its actual implementation. It has been shown that the subroutines (in procedural and functional programming these are the functions), even though powerful in the case of operations separation, are not enough to describe well abstract objects. At that point the idea of two separate types of attributes in a computer program emerge: (i) objects and the set of operations that are defined on them; (ii) names and abstract meanings of the operations. ADT actually belong to the second type of attributes, as it is shown in (Guttag, 1977).

Data abstraction has become a significant part of the approach of computer program structure formulation. In (Abelson, 1996) it is pointed out that data abstraction is a methodology for structuring of a program in such way that many of its components become independent on the particular choice of implementation. The key idea is to create an abstraction that separates the way a data type is used from the way this data type is represented. In this way the complexity of the program implementation can be controlled, and something more, different underlying representations can be provided for the same data abstraction, each of them superior to the others in concrete situations. Thus, as described in (Sedgewick, 1997), ADT is a set of values and operations that are defined on those values that are accessible only through an interface. The definition given by Sedgewick is very important because it directly fits the more recent concepts of object-oriented programming (OOP). Here the key part is played by the concept of interface: data is never accessed directly by the client program, but only through the operations that are defined by the interface.

The same concept is also given directly in the context of OOP by (Horstmann, 2008), where ADT is presented as specification of fundamental operations that give the characteristics of the data type. The separation of the functionality of a data type from its implementation is clearly described using the mechanisms of the OOP, since data abstraction and encapsulation are the first two principles of this programming paradigm. Data abstraction in this case refers to the ability of the programming language to define new data types. In general in OOP these abilities are provided by classes. Encapsulation ensures that private parts of a class remain hidden from the user, while the public routines (in this case called member functions or methods) provide the public interface that is used to manipulate the type representatives. Of course, ADT is not a feature of OOP, it is a general concept that is applicable in any of the contemporary programming paradigms.

Following the above formulations, in the general case ADT can be defined as a concept of a data type in which its behavior is provided by a set of possible operations that are independent from the concrete implementation. Also, in different

applications, different underlying implementations of the ADT can be provided that do not affect the functionally of the type, but only underlying representation that remain hidden from the user.

There are two straightforward examples that are usually given to illustrate what ADT is, and these are queues and stacks (see for example (Goodrich, 2011)).

A *queue* is a linear data structure that manages a sequence of values in *first-in-first-out* (FIFO) out policy. It is defined by two operations that are the interface of the data type:

– *push* operation that adds an element into the back of the sequence;
– *pop* operation that removes an element from the front of the sequence.

A *stack* is also linear data structure but it manages a sequence of values in *last-in-first-out* (LIFO) out policy. Again, it is defined by two operations that are the interface of the data type:

– *push* operation that adds an element, but this time into the back of the sequence;
– pop operation that removes an element from the back of the sequence.

Even from the first glance it is obvious that these two data structures can have identical intrinsic representations, but they differ exactly in the interface that shape their behavior. They are very good as examples for ADT because the abstract barrier between their underlying implementation, and the characteristics of the data type provided by their fundamental operations is obvious. Actually a queue and a stack differ in the definition of their push operations.

Something more, in different particular applications, queues and stacks can have totally different intrinsic implementations. In the course of OOP usually the underlying implementation that is given is based on linked lists. On the other hand, very often in the context of competitive programming training (Skiena 2003) the underlying implementation is based on arrays.

In this paper we present a system of tasks that build the notion ADT, following the system presented by Usuva (Usuva, 2011)).

System of tasks

The notion ADT is constructed using the basic data structure linked list that is used for the implementation of queues and stacks in the context of OOP. The programming language that is used during the course is C++, however the presented tasks are fully applicable in the case of any of the other standard programming languages that support the OOP paradigm, like Java and Python for example. Even more, with few modifications, similar tasks can be presented in the case of procedural or functional programming.

By the time of the problems introduction, the learners are familiar with mechanisms of arrays and vectors, dynamic memory management and basic principles of OOP in C++. The goal of the system of tasks is to develop the notions linked list,

stack, and queue, and to build the notion ADT based on these examples. The expected are also the acquisition of additional knowledge in the mechanisms of programming in C++ that include automatic memory management (dynamic memory management in the context of classes) and friendship mechanism as well.

Definition of linked lists and STL class list

At that point of the curriculum, the learners are fully familiar with the notions of *arrays* and *vectors* in programming, as basic tools for representation of sequences of values of the same data type. The notion linked list is presented based on this background.

Following the first step from the system of Usuva *perception*, we give the basic characteristics of the object linked list and its link to the context of OOP. For this introduction we use the STL class list that is ready to use standard library features and it clearly shows the functionality of the data structure, and its usage.

Task 1. Implement a program that reads*n* number of strings, places them at the end of a linked list, and then print the contents of the linked list.

```
void readList(list<string>& slist, int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "> ";
        string buff;
        cin >> buff;
        slist.push_back(buff);
    }
}
```

Listing 1 Read STL linked list

Task 2. Rewrite the function in Listing 1 using an iterator, instead of the member function `push_back`(). The member function `insert`() takes as first parameter an iterator that points the position, and as second parameter, the value to be inserted.

```
void printList(list<string>& slist)
{
    for (list<string>::iterator pos = slist.begin();
            pos != slist.end();  ++pos)
    {
        cout << *pos << " ";
    }
    cout << endl;
}
```

Listing 2 Print contents of STL list

Task 2 introduces a helper term *iterator* that is common for all sequential data structures in the C++ standard library, inclusively vectors. This is also a common feature, as described in the educational system of Usuva.

Task 3. Add a function to the program from Task 2 that inserts a new node with data at specific location into the linked list. The location is given by its consecutive number.

```
void insert(list<string>& slist, int node, const
                                    string &data)
{
    list<string>::iterator pos = slist.begin();
    for (int i = 0; i < node; ++pos, i++);
    slist.insert(pos, data);
}
```

Listing 3 Insert new element at a given location

Task 4. Extend the program from the above problem with a capability to delete an element from the linked list given its consecutive number.

```
void remove(list<string>& slist, int node)
{
    list<string>::iterator pos = slist.begin();
    for (int i = 0; i < node; ++pos, i++);
    slist.erase(pos);
}
```

Listing 4 Delete an element at a given location in the list

Task 3 and Task 4 illustrate two important features: the s*equential access* that is provided by the linked lists on opposite of the *random access* that is provided by the arrays. This gives the link of the new notion linked list with a notion of the arrays and vectors, that at that point of the curriculum are fully developed. Thus, the notion linked list is described in the terms of perception and discovery of common features with existing notions. Next step is to give a definition of the notion, by providing of the implementation of the linked list itself.

Implementation of linked list

The implementation of a linked list of integer keys is composed by the following three classes:

– `Node` – a node of integer data and pointers to previous and next node;
– `Iterator` – an iterator that visits the nodes of the linked list;
– `LList` – the linked list itself.

Task 5. Implement the class Node. It contains an integer data field, a pointers of type Node to the previous and to the next node in the list. Implement a parameter constructor that sets the data field, and initializes the two pointers with nullptr.

```
class Node
{
    public:
        Node(int data);

    private:
        int data;
        Node* ptr_prev;
        Node* ptr_next;
};
```

Listing 5 Definition of the class Node

Task 6. Implement the class `LList`. In next exercises more members will be added to the class. Default constructor simply creates an empty list by setting both `ptr_frst` and `ptr_last` to `nullptr`.

```
class LList
{
    public:
        LList();
        void pushBack(int data);

    private:
        Node* ptr_frst;
        Node* ptr_last;
};
```

Listing 6 Definition of the class LList

Task 7. In class `LList`, implement the member function `pushBack`(). Create a new node `ptr_newn` that contains the data filed. We must also grant access of the class `LList` to the private members of the class Node. In C++ this is easily done by defining that `LList` is a *friend* class of Node in Node definition.

In Task 7 again a link is created with a new notion that in this case is an important mechanism of the programming language itself. This is the mechanism of *friend classes* and *friend functions* that grants to access to the private section of a class. It is also pointed that this approach destroys the encapsulation of the classes and thus proves that C++ is not a purely object-oriented language.

Task 8. Implement the destructor of the class `LList`. For each node in the list, starting from the first one, attach a temporary, move the first node to the next one, and then delete the temporary.

```
Node* ptr_tmp = ptr_frst;
ptr_frst = ptr_frst->ptr_next;
delete ptr_tmp;
ptr_tmp = nullptr;
```

Listing 7 Body of the loop that implements the destructor of LList

Task 9. Implement class `Iterator.` It contains two private fields: a pointer to `Node` that is the current position of the iterator, and a pointer to the linked list that contains the iterator. The default constructor sets both pointers to `nullptr`. The constructor with parameters is needed for the implementation of `begin`() and `end`() member functions of class `LList`.

```
class Iterator
{
    public:
        Iterator();
        Iterator(Node* ptr_pos, LList* ptr_cnt);
    private:
        Node* ptr_pos;
        LList* ptr_cnt;
};
```

Listing 8 Definition of the class iterator

Task 10. Add two member functions to `LList: begin`() and `end`(). They return an iterator, attached to the list object, that points to the beginning and the `end`, respectively. The function end() returns an iterator whose position pointer is set to `nullptr`. This is because it returns iterator to the past-the-end element which denotes the end of the list.

Task 11. In class Iterator add a member function `next`() that moves the iterator to the next node, and `prev`() that moves the iterator to the previous node. In `prev`() first verify that current position is not the first node. If current position is not nullptr, move to the previous node, otherwise go to the last node of the list. Implement a member function `get`() that returns the integer that is pointed by the iterator.

```
void Iterator::prev()
{
    assert(ptr_pos != ptr_cnt->ptr_frst);
    ptr_pos = ptr_pos ? ptr_pos->ptr_prev :
                              ptr_cnt->ptr_last;
}
```

Listing 9 Move the iterator to the previous position

With the later tasks, the notion linked list is more or less built and we are ready to move to the subsequent notions of queue and stack, that actually will lead us to the development of the notion ADT.

Queue and stack definition by the STL classes queue and stack

With the notion linked list developed, we are ready to use it as basis of the introduction of the two notions *queue* and *stack*. Firstly, the standard library classes are used to illustrate the application of the two terms, before the mechanism of their implementation is presented to the learner.

First, we start with the term queue. A queue is a sequence of elements of the same data type that are managed in the first-in-first-out (FIFO) policy. The elements in a queue are removed in the same order in which they have been inserted. Two operations are defined:

– push add an element into the back of the queue;
– pop remove an element from the front of the queue.

In STL queue is a template class defined in the header queue.

Task 12. Write a program that reads a sequence of personal names as strings, stores them in a queue, and prints them out in the standard output in the same order.

```
do
{ ... else
    {
        qnames.push(buff);
    }
} while (more);
```

Listing 10 Read strings and store them in a queue

```
while (!qnames.empty())
{
    cout << qnames.front() << " ";
    qnames.pop();
}
cout << endl;
```

Listing 11 Print the contents of the queue

The purpose of Task 12 is actually the perception of the notion without explaining at that point the intrinsic structure of its implementation. With the existing notion of

linked list, and the simple example of queue usage, the learner gets the idea of the characteristics of the object queue simply by naming the two interface operations push() and pop(). The presentation of the notion stack is absolutely analogous.

A stack is a sequence of elements of the same data type that are managed in the last-in-first-out (LIFO) policy. The elements in a stack are removed in the reversed order in which they have been inserted. The two operations push and pop are defined:

– push add an element into the back of the stack;
– pop remove an element from the back of the stack.

Task 13. Write a program that reads a sequence of personal names as strings, stores them in a stack, and prints them out in the standard output in the reversed order.

With the Task 13, also the similarities and differences between queues and stacks become immediately obvious that give the common features of the class of objects and specifics of the studied object, and at the same time, the link to the environment of notions. At that point the idea of ADT still does not exist, but with the next step that provides the implementation of both queue and stack based on linked list, the most important feature of abstract data types become visible, namely the abstract barrier between the data type underlying implementation and its usage provided by the interface of functionalities.

Queue and stack implementation

The implementation of queue that stores integer keys is based on singly linked list. The implementation consist of the following two classes:

– `Node` – node of the linked list composed by integer data field and a pointer to the next element;
– `Queue` – the class queue itself, based on a pointer to the head, and the tail of the list.

Task 14. Implement class Node that represents a node of singly linked list of integers. A node contains an integer data field and a pointer to the next node.

```
class Node
{
    public:
        Node(int data);

    private:
        int data;
        Node* ptr_next;

    friend class Queue;
};
```

Listing 12 Class Node definition

In this task the learner is implementing a data structure that is very similar to the one that was implemented for the purposes of the linked list. This gives the link to the existing notion of linked list, but also shows the difference between two distinct types of list: *singly linked list* and *doubly linked list*.

Task 15. Define the class `Queue` that will be implemented based on singly linked list. The first node of the list is called *head*, the last node is called *tail*. The queue front is located at the head of the list. The queue back is located at the tail of the list. The class Queue contains pointers to the head and to the tail of the list. The default constructor sets them to `nullptr` and creates an empty queue. Destructor is the same as in the case of `LList`.

Task 16. In `Queue` implement a member function `push`() that adds an element to the tail of the linked list, that represents the queue back:

1. Create a new node and store the integer data in it.
2. If the pointer to tail is not `nullptr`, the queue is not empty, and then attach the new node after the `ptr_tail`. Otherwise `ptr_head` must point to the new node.

3. The new node becomes the tail node of the queue by redirecting `ptr_tail` to point to it.

```
class Queue
{
    public:
        Queue();
        ~Queue();
        void push(int data);
        void pop();
        int front() const;
        bool empty() const;

    private:
        Node* ptr_head;
        Node* ptr_tail;
};
```

Listing 13 Class Queue definition

Task 17. In Queue implement a member function `pop`() that removes an element from the head of the linked list, that represents the queue front.

1. Assert that the queue is not empty, and attach a temporary pointer to the `ptr_head` that points the node to remove.
2. In the case the queue is composed by a single node, set both `ptr_head` and `ptr_tail` to `nullptr`.
3. In the case the queue is composed by more than one node, redirect `the ptr_head` pointer to the next node.

4. Finally, delete the temporary pointer, that stores the address of the former list head node.

Tasks 15 and 16 introduce the principle of separation of the implementation from the functionality definition that is typical for the ADT. The underlying implementation is actually a familiar object, namely the linked list. The functionality of the queue is provided by the two member functions push() and pop() that implement the FIFO policy of element management in the data structure. This example also shows how from the point of view of the linked list as a notion, step 9 of Usova is used: application of the notion in more complex examples. The same step latter is introduced for stacks and queues, when they are applied in the implementation of graph algorithms (see also (Cormen, 2009), (Sedgewick, 2001), (Goodrich, 2011) and (Skiena, 2003)).

The implementation of the stack data structure is presented in analogy to the queue, by pointing out both similarities, and differences between them.

Task 18. Based on the previous example for queue implementation, implement a class `Node`, and a class `Stack`. Do not store the location of the tail. All members without the function push() are nearly the same. Instead of function `front`(), implement a function top() that returns the value at the top of the stack, that is stored at the head node.

Defining the notion ADT based on the examples of queues and stacks

After fully defining the notions queue and stack, we are ready do give the formal definition of ADT, based on these two typical examples.

Definition 1. Abstract Data Type (commonly abbreviated ADT) is a concept that defines a data type by specifying the operations on it independently on the concrete implementation.

Both queues and stacks are ADTs.

What remains is to show how the notion ADT can be enriched, and this can be achieved by asking the learner to provide a completely different intrinsic implementation to both queue and stack.

Task 19. Implement both queue and stack, but this time based on arrays, instead of linked list. The interfaces of two data types must remain completely the same.

Task 19 helps the learner to understand the independence of ADT intrinsic implementation from its functionality that is provided by the public member functions of the classes. Something more, now it is also obvious that in different situations, different implementations can show better results, but this does not change the way the data type is used. For example, when the size of the expected data is large, and impracticable, the implementation based on linked list is more appropriate. On contrary, when the size of the data is not that huge, and can be estimated, implementation based on arrays can give better results.

Conclusion

The presented approach in formation of the notions linked list, queue, stack, and abstract data type (ADT) develops more in-depth, conscious and enduring knowledge. It is based on the greater activity of the students and as a methodology it is implemented as a specially developed system of tasks. The system of tasks provides the whole process of knowledge acquisition: introduction, consolidation and application. Its creation requires more effort on the part of the lecturer and more time resources for its application in the learning process. For this reason, it is not possible to use it fully during each lesson, but it gives good results for the use of its individual elements in certain lectures, as well as full use on those topics that are more complex for students.

The system of tasks, or a system that is derived from this one, is applicable in high-school course of computer programming for advanced students, or in an introductory course on data structures in the university curriculum. In methodological terms, when developing a system of tasks, we follow the work of (Asenova, 1990) and (Assenova & Marinov 2019).

REFERENCES

Abelson H., G. J. Sussman, with J. Sussman (1996). *Structure and Interpretation of Computer Programs* (2nd ed.), MIT Electrical Endeepgineering and Computer Science, ISBN-13: 978-0262510875, ISBN-10: 0262510871.

Albertovna B. I., Z. N. Ivanovna, K. I. Evgenievna & P. U. Valerievna. (2017) *Levels of formation of notions in high-school course of Informatics*, KGU, Pedagogy. Psychology. Sociocinetics 3, 191 – 193. [In Russian]. Original title: Б. И. Альбертовна, Ивановна З. Н., Евгеньевна К. И., Валерьевна П. У. *Уровни формирования понятий школьного курса информатики*, Вестник КГУ 2017, Педагогика. Психология. Социокинетика № 3, с 191 – 193.

Aleksandrov A. D., A. N. Kolmogorov & M. A. Lavrent'ev. (1999) Mathematics: *Its Content, Methods and Meaning*, Dover Books on Mathematics Series, Courier Corporation, ISBN 0486409163.

Asenova, P. (1990). *Design and implementation of a system of tasks in teaching algorithms in Bulgarian schools*. PhD thesis. Moscow. RAE. [In Russian]. Original title: Асенова, П. (1990). *Построение и использование системы задач для обучения алгоритмизации в курсе информатики болгарской школы*. Дисс. канд. пед. наук. Москва. РАО.

Asenova, P., M. Marinov. (2019). *System of tasks in mathematics education*. Mathematics and Informatics, 62 (1), 52 – 70.

Asenova, P. & M. Marinov. (2018). *Teaching mathematics with computer systems*, Math. And Education in Math., 47, 213 – 121.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein, (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press, ISBN-13: 978-0262033848, ISBN-10: 9780262033848.

Davydov, V. V. (1996), *The Theory of Developmental Education*, INTOR Moskow. [In Russian].

Dijkstra, E. W. (1972) *Notes on structured programming. In Structured Programming*, Academic Press, New York.

Goodrich M. T., R. Tamassia & D. M. Mount (2011). *Data Structures and Algorithms in* C++, (2nd ed.), Wiley, ISBN: 978-0-470-38327-8.

Guttag, J. V. (1977) *Abstract Data Type and the Development of Data Structures*. *Commun. ACM* 20(6): 396 – 404.

Horsmann, C. & T. A. Budd (2008). *Big* C++ (2nd ed.), Wiley, ISBN: 978-0-470-38328-5.

Rubinstein, S.L. (1946) *Fundamentals of general psychology,* State Study-Pedagogical Publishing House of the Ministry of Education of Russian Soviet Federative Socialist Republic, Moscow, USSR, 704. [In Russian].

Sedgewick, R. (1997). *Algorithms in C: Parts 1 – 4, Fundamentals, Data Structures, Sorting, and Searching* (3rd ed.). Addison-Wesley Longman Publishing Co., Inc., USA, ISBN-13: 978-0201314526, ISBN-10: 0201314525.

Sedgewick, R. (2001). *Algorithms in C: Part 5, Graph algorithms* (3rd ed.). Addison-Wesley Professional, ISBN-13: 978-0201316636, ISBN-10: 0201316633.

Skiena S. S. & M. A. Revilla (2003). *Programming Challenges: The Programming Contest Training Manual* (Texts in Computer Science), Springer, ISBN 0387001638.

Vygotsky, L. S. (1987). *Thinking and speech*. In The collected works of L.S. Vygotsky. Problems of general psychology (Vol. 1, pp. 37 – 285) (translated by Norris Minick). New York, London: Plenum Press.

Usova, A. V. (2011). *Some methodological aspects of the problem of scientific notions formation in learners in schools and students in universities*, Mir nauki, kulturi, obrazovania, No 4(29). [in Russian]. Original title: А.В. Усова. *Некоторые методические аспекты проблемы формирования научных понятий у учащихся школ и студентов вузов*. Мир науки, культуры, образования. № 4 (29) 2011.

✉ **Dr. Lasko M. Laskov, Assoc. Prof.**
ORCID: 0000-0003-1833-8184
Department of Informatics
New Bulgarian University
21, Montevideo Blvd.
1618 Sofia, Bulgaria
E-mail: llaskov@nbu.bg