Multi linked lists: an object-oriented approach

Đorđe Stojisavljević*, Eleonora Brtka**, Vladimir Brtka**, Ivana Berković**
* University of Banja Luka/Faculty of law, Banja Luka, Republika Srpska
** University of Novi Sad/Technical faculty “Mihajlo Pupin”, Zrenjanin, Serbia
djordje.pfm@gmail.com, eleonorabrtka@gmail.com, brtkav@gmail.com, berkovic@tfzr.uns.ac.rs

***Abstract* — The paper deals with the approach to multi linked lists while teaching. Object oriented paradigm is used, so that multi linked lists are implemented in C++. Simple example presented in this paper cover the usage of structures inside classes and template classes. Basic concepts of understanding object oriented multi linked lists are defined, and five groups of students are singled out. The main contribution of this research is in the domain of education: the specification of student's understanding of these concepts is given, as well as guidelines how to recover to full understanding of multi linked lists.**

I. INTRODUCTION

Linked lists are widely known and exhaustively described in literature. An elaborate discussion of linked lists can be found in e.g. [1], while more detailed discussion about multi linked lists and their implementation in C language can be found in e.g. [2].

Each node of a multi linked list (Fig. 1) has a complex structure; it contains:

- Data field – represents the useful data, usually realized in structure form;
- One link field that points to the next node in the multi linked list (like in singly linked list). The last node points to NULL; and
- Two or more link fields who are pointing to another lists called *sublists*. Sublist has a structure like singly linked list.

The entry point into a linked list is called the *head* of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a NULL reference.

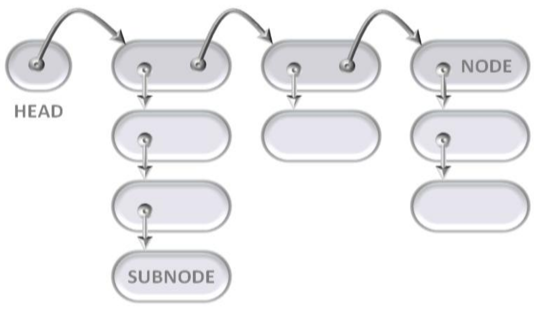

Figure 1. Multi linked list in traditional (structured) form

As we can see from Fig. 1 there are two structures:

- *subnode* which contains data field and link to the next subnode, and
- *node* which contains data field, link to the head of sublist and link to the next node.

These concepts are often hard to understand and implement in practice. Object Oriented (OO) programming paradigm is most common contemporary programming paradigm, so it is necessary to implement multi linked list in OO manner. The OO approach to multi linked list is even more confusing if not presented properly to the students. So, main questions are

- How to deal with OO C++ multi-linked list while teaching?
- What are the basic concepts?
- What to do with students who do not understand some of basic concepts?

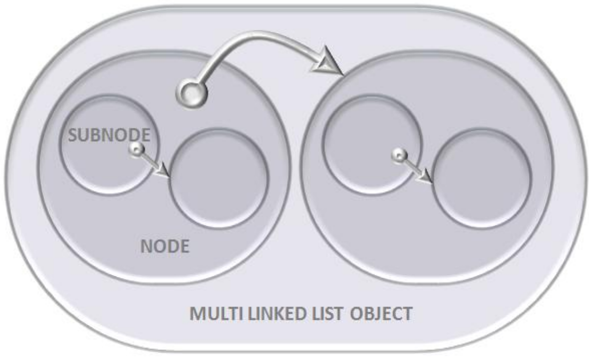

Figure 2. Multi linked list in object-oriented form

This paper is organized as follows: Section II deals with OO approach to multi linked lists. Simple example in C++ programming language is used to declare multi linked list, as well as constructor method, destructor method and some operations on multi linked lists. Section III presents the methodological approach to multi linked lists; six basic concepts were defined, while students were classified to five distinctive groups according to their understanding of six basic multi linked list concepts. There is no point to consider students who understand all basic concept in full extents, so neither of these five group covers them. Section IV is the conclusion of this research where some guidelines on how to recover to fully understanding of OO multi linked lists are given, for each group of students.

II. OBJECT – ORIENTED APPROACH

A definition of object-orientation is that an entity of whatever complexity and structure can be represented by exactly one object [3]. If we apply this definition on multi linked list we get an object-oriented multi linked list (Fig. 2). In object-oriented programming we treat a multi linked list as an object. That means that we will view a multi linked list as an object that stores data as a list, that allows the list to be manipulated using a set of methods provided by the multi linked list interface. An important element to good object-oriented programming is good object-oriented design. This means that we need to design a good interface for a multi linked list object that provides the operations that a programmer wants. Design of a multi linked list is shown on UML class diagram in Fig. 3.

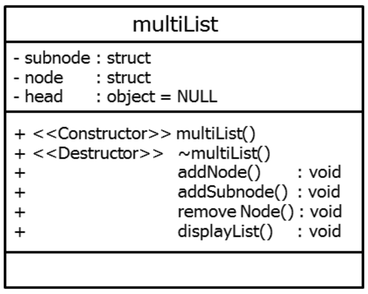

Figure 3. Multi linked list class diagram

As we can see, internal structure of a list is hidden. List can be manipulated only through the interface.

A. Multi linked list class

Implementation of a multi linked list will be hidden so that it can be modified without affecting the programs that use it. In particular, we will not let the programs that use it, to have access to its internal representation. Therefore, we will declare a multi linked list in C++ as shown in Listing 1.

```
template <class T>
class multiList{

private:
  struct subnode{
    T value;
    subnode *next_sub;
    subnode(T value1,subnode
*next_sub1=NULL){
       value=value1;
       next_sub=next_sub1; } };
  struct node{
    T value;
    subnode *head_sub;
    node *next;
    node(T value1, subnode *head_sub1=NULL,
         node *next1 = NULL){
           value = value1;
           head_sub=head_sub1;
           next = next1; } };
node *head;

public:
  multiList() { head = NULL; }
  ~multiList();
  void addNode(T value);
  void addSubnode(T n, T value);
  void removeNode(T value);
  void displayList();
  };
```

Listing 1. Declaration of a multi linked list in C++

As we can see from Listing 1. multiList class is realized as template class, because the class should be able to handle different types of data fields. While using a multi linked list and operating on a particular data type, only the data type needs to be specified when the template class object is defined or declared, e.g. `multiList<int> mList`.

B. Constructor and destructor

MultiList class has one constructor and destructor. Constructor *multiList()* has no arguments. Its function is to initialize multiList object by setting head to NULL.

Destructor gets called when multiList object needs to be deleted. Through while loop destructor runs-cross each node of a multiList object and deletes it by calling *removeNode()* method.

C. Methods

In order to make the multiList as universally usable as possible, we want to define a set of essential, primitive operations that programmers can use to assemble more complex operations. In other words, rather than trying to imagine every conceivable use for a multiList and placing an operation in the multiList's interface that supports that use, we try to envision a set of basic building block operations that can be used to create these more complicated operations.

To add a new node in the list, first thing to do is allocate memory space for the node, then we assign data to data field [4]. When node is created, it has no subnodes; therefore head of sublist is NULL. Next, we concatenate the new node with the original list and make the newly created node the first one of the list (Fig. 4).

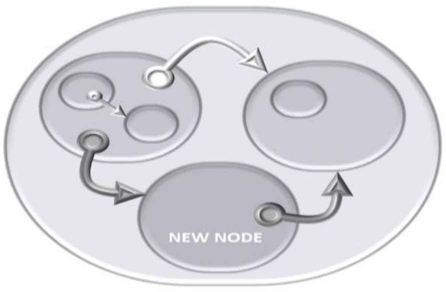

Figure 4. Adding a new node into the multiList object

Method for adding a subnode is given in Listing 2.

```
template <class T>
void multiList<T>::addSubnode(T n, T value){
   node *nodePtr=head;
        // check if node exists
        // if exists then add subnode
   if(nodePtr->head_sub==NULL)
   nodePtr->head_sub=new subnode(value);
   else {
      subnode *subnodePtr=nodePtr->head_sub;
      while(subnodePtr->next_sub!=NULL)
      subnodePtr=subnodePtr->next_sub;
      subnodePtr->next_sub=new subnode(value);
          }
}
```

Listing 2. Method that adds a new subnode into the multiList object

Removing a node from a multiList means to modify the list in such a way that the node is no longer connected to its predecessor and successor, while bridging the removed node to maintain the connection of the other nodes. After bridging, programmer explicitly frees the memory occupied by those nodes that are not needed anymore [2].

Method that removes the first element of a list is given in Listing 3.

```
template <class T>
void multiList<T>::removeNode(T value){
   node * nodePtr; node *previousNodePtr;
   if (!head)  return;
   // find the node that needs to be removed
   while(subnodePtr!=NULL){
   // remove his subnodes
       }
   head = head->next;
   delete nodePtr; }
   else {
       nodePtr = head;
       // update links
        }
   if (nodePtr){
         previousNodePtr->next=nodePtr->next;
          delete nodePtr; }
}
```

Listing 3. Method that removes a node from the multiList object

To perform an operation on all nodes of a list, we have to reach each node starting from the first one, by following the *next* references. The simplest way of doing this is through iteration [5].

In Listing 4. we give an example of using object-oriented multi linked list. Because the multiList class is a template class, in our example we will define it as a string.

```
int main(){
   multiList<string> list;
    string name;
    string value;

    cout << "Add 3 names to the List:\n";
    for (int i = 0; i < 3; i++){
       cout << "Name #" << i + 1 << ": ";
       getline(cin, name);
       list.addNode(name);
    }

   cout<<"Add subnode to name: ";
    getline(cin,name);
    cout<<"\nEnter value: ";
    getline(cin,value);
    list.addSubnode(name,value);
    list.displayList();
   cout<<"\nEnter a name to delete: ";
    getline(cin, name);
    list.removeNode(name);
    list.displayList();
   list.~multiList();
  return 0;
}
```

Listing 4. Test example of using multiList class

III. METHODOLOGICAL APPROACH

Methodological approach used to explain multi linked lists in the case when object oriented programming is applied is based on six concepts. These are basic concepts, arguably minimal number of basic concepts needed to practically understand object oriented multi linked lists. Basic concepts are:

1. Pointers and memory allocation.
2. Object-oriented paradigm.
3. Linked list basics.
4. Creating nodes.
5. Creating sub-nodes.
6. List operations.

The importance level of basic concepts is crucial for students to understand multi linked lists.

(The order of basic concepts is crucial for students to understand multi linked lists. These concepts were chosen after extensive research of literature references. In [6] was presented a "pointer-safe" object oriented paradigm including physical addresses, placements of objects, etc. in addition, in [7] the context-insensitive pointer analysis was described; this is based on applying cycle elimination to context-sensitive pointer analysis and refers to some advanced techniques. Object oriented paradigm is widely used in practice, so that there is no lack of literature references to this concept; in [3] this paradigm is described appropriately for this research. The linked lists concepts including basics, creating nodes and sub-nodes, as well as operations on lists are presented in [1, 4, 5, 8], and there is no lack of literature on this matter as well.

In contrast to a large number of literature references dealing with these concepts in the domain of software engineering, there is a lack of information about implementing these concepts in teaching. It is hard to assess the understanding of some concept, so we are not particularly sure if student understand these concept, and even less are we able to objectively assess the extent to which student understands a particular concept. The application of the scale (e.g. from 5 to 10 or from 1 to 10) is often used, as well as the measure of understanding in percents, but arguably more "rough" scale is better, so we are using just three values in this investigation: low (0), medium (1) and high (2). Instead of three-point scale, five or seven point scale is often used.

Still, there are some disagreements about basic concepts, so we had applied Fuzzy Screening method (R. Yager) and the Rough Sets Theory (Z. Pawlak). In both cases, we needed a data sample.

In this particular investigation we used data sample collected from multiple sources in mid-term exams. We have students, and each of them have a certain number of points ranging from 55 to 100 for each of this six attributes. Having in mind that this data sample is small and gathered from multiple sources we have discretized our data so that we have three values: low, medium and high: from 55 points to 70 points is low, from 76 to 85 is medium and from 86 points up to one hundred points is high. After discretization step, each row represents one or more students, Table I.

TABLE I
DATA SAMPLE

1. Pointers and memory allocation.	2. Object-oriented paradigm.	3. Linked list basics.	4. Creating nodes.	5. Creating sub-nodes.	6. List operations.
High (2)	High (2)	High (2)	High (2)	High (2)	High (2)
High (2)	High (2)	High (2)	High (2)	High (2)	High (2)
High (2)	High (2)	Medium (1)	Medium (1)	High (2)	High (2)
Medium (1)	High (2)	High (2)	Medium (1)	High (2)	High (2)
Medium (1)	High (2)	High (2)	High (2)	Medium (1)	High (2)
…	…	…	…	…	…
High (2)	Medium (1)	High (2)	Medium (1)	Medium (1)	Medium (1)
High (2)	Low (0)	Medium (1)	High (2)	High (2)	Medium (1)
Medium (1)	Medium (1)	Medium (1)	High (2)	Medium (1)	High (2)
Medium (1)	High (2)	Medium (1)	High (2)	Medium (1)	low (0)
High (2)	Medium (1)	Medium (1)	Medium (1)	Low (0)	Medium (1)
High (2)	Medium (1)	Medium (1)	Low (0)	Low (0)	Medium (1)
Medium (1)	Low (0)	Low (0)	Medium (1)	High (2)	Medium (1)
High (2)	Medium (1)	Low (0)	Medium (1)	Low (0)	Medium (1)
Low (0)	Medium (1)	High (2)	Low (0)	Medium (1)	Medium (1)
High (2)	High (2)	Low (0)	Low (0)	Low (0)	Medium (1)
Medium (1)	High (2)	Medium (1)	Low (0)	Low (0)	Medium (1)
Low (0)	High (2)	Medium (1)	Low (0)	Medium (1)	Low (0)
…	…	…	…	…	…

After Table I was obtained, we are able to calculate the score by (1) and sort table rows in descending order by score value.

$$score = \sum_{i=1}^{n} \alpha_i p_i, \alpha_i \in [0,1], p_i = \{0,1,2\} \qquad (1)$$

For $n = 6$ and $6 \le score \le 8$, we have a "window" marked in Table I, by thick rectangle. By changing the values that constrain the score, we are able to slide the window up or down. We have three groups of students: the group above the window are students that are almost there and, usually they are able to figure it out by themselves, while the group of student below the window are students who need to study harder. So, desired position was to find a group that needs a "little push" to understand these concepts.

Except for the case when the student knows concepts to the maximum extent, by "window" we singled out five cases of cumulative understanding of these six concepts. These five cases are presented in form of "radar maps" that are easy to understand and read. The concepts are arranged in a clockwise direction, which corresponds to their order. Fig. 5 presents the case when students understand the concept of Pointer and memory allocation in maximal extend (high), while they are not able to understand how to create sub-nodes (low), while there is a lack of maximal understanding of all other concepts (medium).

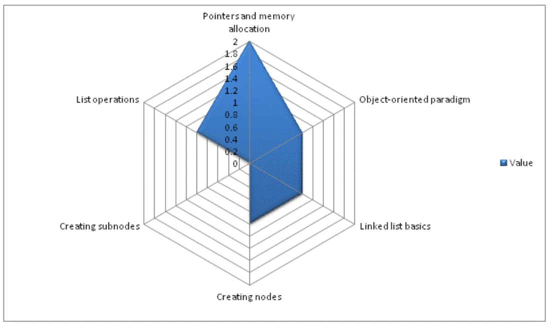

Figure 5: Pointer and memory allocation

Fig 6. represents students that are able to understand OO paradigm, as well as Creating nodes in the maximal extent, but understanding of the List operations is lacking (low).

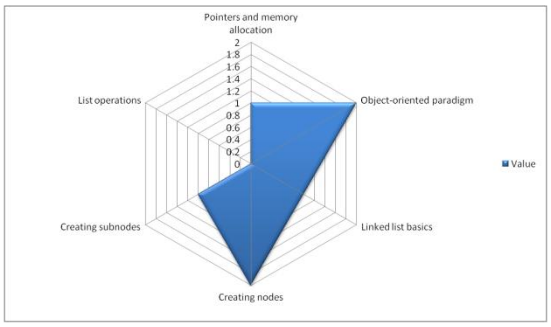

Figure 6: OO paradigm and creating nodes

Fig. 7 represents students who understand Pointers and memory allocation and Linked lists basics, however the understanding of other concepts is not maximal. There is no total lack of understanding of any concept.

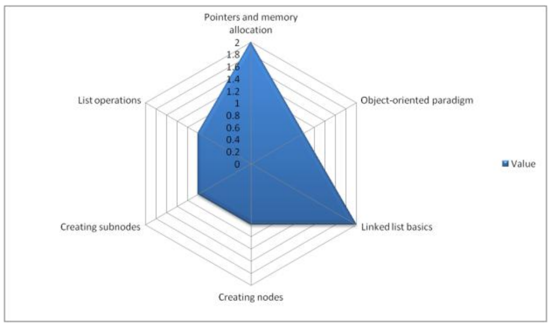

Figure 7:Pointers and memory allocation and List basics

Fig. 8 represents the group of students who understand three concepts in maximal extent, but they are not able to cope with OO paradigm.

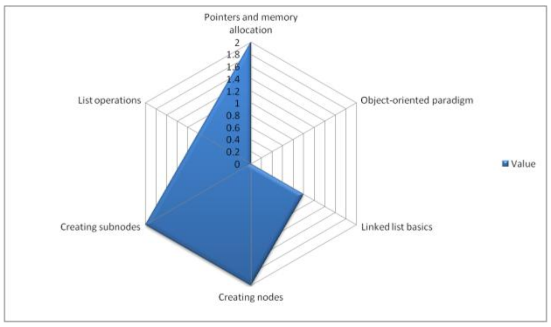

Figure 8: Pointers and memory allocation and Nodes

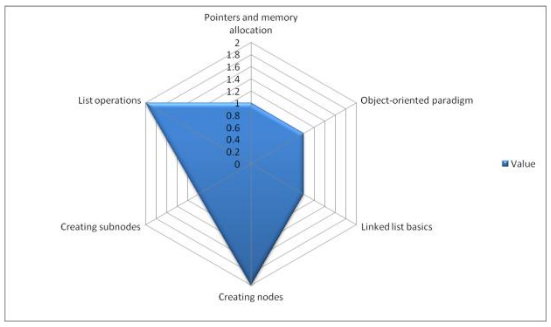

Figure 9: List operations and Creating nodes

Finally in Fig. 9 is represented a group of students who understand how to Create nodes and List operators, while other knowledge is lacking. There is no total lack of understanding of any concept.

IV. CONCLUSION

In this paper we show how object-oriented design can be applied to the implementation of a multi linked lists with a mixture of explanations, figures and sample codes. Linked lists are useful to study for two reasons. Most obviously, linked lists are a data structure which you may want to use in real programs. Somewhat less obviously, linked lists are great way to learn about pointers. Linked list problems are a nice combination of algorithms and pointer manipulation. Traditionally, linked lists have been the domain where beginning programmers get the practice to really understand pointers.

This paper is useful if you want to understand linked lists or if you want to see a realistic, applied example that uses structures inside classes and template classes.

We propose the exact way to estimate and visualize the extent of student's understanding of multi linked list in Object oriented C++ programming. This is a good starting point for further analysis of how students understand the basic concepts related to understanding of C++ linked lists. Six relevant basic concepts which are necessary for the understanding of C++ linked lists were defined by extensive literature review, while five group of students was formed in exact manner from empirical data. Six basic concepts are: Pointers and memory allocation, Object-oriented paradigm, Linked list basics, Creating nodes, Creating sub-nodes and List operations. The student's knowledge of basic concepts is rated as high (2), medium (1) or low (0).

First group of students is characterized by maximal understanding of Pointer and memory allocation, while they do not know how to create Sub-nodes of a C++ list. According to practical experience, they are able to recover through understanding of List operations. Second group of students is good in OO programming and Crating nodes, while they do not know how to implement List operations. These students are able to recover thanks to understanding of Node and Sub-node creation. Third group of students consists of students who understand each concept, although not with the maximal measure. Some backtracking to previous concepts is needed in order to fully understand C++ list implementation. Fourth group are students who lack in understanding of OO paradigm, but they are able to understand the implementation of C++ list, so their recover is possible by backtracking to OO paradigm. Finally, fifth group of students are those who understand each concept, but not with the maximal measure, so that backtracking to previous concepts is needed. According to our experience, the student who belongs to any of these five groups will be able to recover to maximal extent of C++ list understanding.

Some statistical analysis of presented approach is in progress so, future work will include more exact methods for student's knowledge assessment.

ACKNOWLEDGMENT

Ministry of Science and Technological Development, Republic of Serbia financially support this research, under the project number TR32044 ”The development of software tools for business process analysis and improvement”.

REFERENCES

[1] Parlante, N. “Linked list basics”, Document #103, Standford CS Education Library, 2001.

[2] Stojisavljević, Đ. Brtka E. “Application of multi linked lists technique for the enhancement of traditional access to the data”, Proceedings of the International Conference on Applied Internet and Information Technologies, pp 403-407, Zrenjanin, Serbia, 2013.

[3] Dittrich, K. “Object-Oriented Systems – the notation and the issues”, International Workshop in Object-Oriented Database Systems, Pacific Grove, CA, 1986.

[4] Parlante, N. “Linked list problems”, Document #105, Standford CS Education Library, 2001.

[5] Tanenbaum A. Augenstein M. Langsam Y. “Data structures using C and C++”, PHI Learning, 2009.

[6] Della Penna G. “A type system for static and dynamic checking of C++ pointers”, Computer Languages, Systems & Structures 31, pp. 71–101, 2005.

[7] Woongsik C. and Kwang-Moo C. “Cycle elimination for invocation graph-based context-sensitive pointer analysis”, Information and Software Technology 53, pp. 818–833, 2011.

[8] Tüzün E., Tekinerdogan B., Kalender M. E. and Bilgen S. “Empirical evaluation of a decision support model for adopting software product line engineering”, Information and Software Technology 60, pp. 77–101, 2015.