Concatenating Approach: Improving the Performance of Data Structure Implementation

Davud Mohammadpur
Faculty of Electrical and Computer Engineering, Malek Ashtar University of Technology
dmp@znu.ac.ir
Ali Mahjur*
Faculty of Electrical and Computer Engineering, Malek Ashtar University of Technology
mahjur@gmail.com

Received: 24/Sep/2017 Revised: 19/Jan/2018 Accepted: 08/Apr/2018

Abstract

Data structures are important parts of the programs. Most programs use a variety of data structures and quality of data structures excessively affects the quality of the applications. In current programming languages, they are defined by storing a reference to the data element in the data structure node. Some shortcomings of the current approach are limits in the performance of a data structure and poor mechanisms to handle key and hash attributes. These issues can be observed in the Java programming language which that dictates the programmer to use references to data element from the node. Clearly it is not an implementation mistake. It is a consequence of the Java paradigm which is common in almost all object-oriented programming languages. This paper introduces a new mechanism called access method, to implement a data structure efficiently which is based on the concatenating approach to data structure handling. In the concatenating approach, one memory block stores both the data element and the data structure node. According to the obtained results, the number of lines in the access method is reduced and reusability is increased. It builds data structure efficiently. Also it provides suitable mechanisms to handle key and hash attributes. Performance, simplicity, reusability and flexibility are the major features of the proposed approach.

Keywords: Programming Language; Data Structure Handling; High-Level Abstraction; Concatenating.

1. Introduction

Data structures are important parts of programs. They are the building blocks of any program, and provide useful mechanisms to store and retrieve data [1]. Most programs use a variety of data structures. They often use simple variations or compositions of basic data structures such as linked lists, queues, stacks and tree types [2].

To illustrate some pervasive and serious problems in data structure management, we investigated data structures in many applications. For example, Hadoop, a distributed processing framework for large data sets, uses many Java data structures such as List, LinkedList, Queue, Set, TreeSet, LinkedHashSet, HashSet and HashMap. It can be concluded that, quality of data structures excessively affects the quality of the applications [3], [4].

Unfortunately, the usual approach to apply a data structure on a set of data elements is to store a reference to the data element in the data structure node (Fig. 1a) [5]. We call it referencing approach. In this approach, the data element and data structure node are allocated separately and the address of the data element is stored in the node. These references provide paths from structure nodes to data elements.

The referencing approach has two issues. First, it breaks an object into multiple parts (data element and data structure nodes). As stated in [6], breaking an object into multiple parts causes performance and memory penalties: 'It incurs allocation and garbage collection overhead. Moreover, the fact that objects are accessed by reference introduces extra pointer dereferences. Finally, it incurs memory overhead: at a minimum, a pointer to the object and some memory for allocation administration is required'.

Second, there is no path from the data element to the corresponding data structure node. So, to reach the data structure node from the data element, the programmer has to scan the data structure. This increases the operations time, and limits performance on data structures.

a- Referencing Approach b- Concatenating Approach

Fig. 1. Data structure implementations

The better approach to apply a data structure on a set of data elements is to concatenate the data structure node to the data element (Fig. 1b). We call it concatenating approach. In this approach, one memory block stores both the data element and the data structure node.

This paper introduces a new mechanism called access method, to implement a data structure efficiently which is based on the concatenating approach. In this mechanism, a data structure is implemented independently. Later, programmers can apply data structures on data elements based on the concatenating approach.

* Corresponding Author

An important portion of data structures is the keys. In the access method mechanism, we define a special way to handle them. It allows a programmer to set a field(s) of the data element as a key.

2. Referencing Approach

In current programming languages, to apply a data structure on a set of data elements, data elements are not stored in the data structure, but only references to the data elements are stored [7]. We call it referencing approach. As the Fig. 1a shows, the data element and data structure node are separated from each other, and the address of the data element is stored in the node. Therefore, two memory blocks are allocated per data element; one to store the data element and another one to store the data structure node [8].

It increases the memory footprint and reduces the performance of the code. The memory footprint is increased in two ways. One, as shown in Fig. 1a, some storage is used to store additional references in the data structure nodes. Two, dynamic memory management uses some extra storage to store its information. As this information is stored per block, increasing the number of blocks increases this overhead too.

The performance of the code is reduced due to the following reasons. One, two memory blocks should be allocated and freed, which increases the memory management time [9]. Second, it is not possible to reach a data structure node from its corresponding data element (Fig. 1a). The references only provide a path from the data structure node to the data elements. To find the corresponding data structure's node, the structure should be traversed which needs extra time [10].

As an example Fig. 2 shows an implementation of the doubly linked list in the current approach. It has two pointers: *head* and *tail*. *head* points to the first node of the list and *tail* points to the last node of the list. The node of the doubly linked list has two references: *next*, *prev* that point to other nodes. As is shown in Fig. 3 code snippet, removing a node from it needs O(1) time.

Even though removing a node from the linked list needs O(1) time, removing a data element from it needs O(n) time. To remove an arbitrary data element from the linked list, the programmer has to iterate over the linked list nodes to find the corresponding node and remove it [9]. Therefore, removing a data element from the linked list needs O(n) time.

```
public class LinkedList<E> {
    Node<E> head,tail;

    public LinkedList() {
    }
    private static class Node<e>{
        E item;
        Node<E> next;
        Node<E> prev;
        /*rest of the class */
    };
}
```

Fig. 2. A Linked List

This issue can be observed in the Java *LinkedList*. The Java programming language dictates the programmer to use references to data element from the node. The following code snippet shows the node of the Java *LinkedList*. It has a field named *item* which points to the data element.

```
private static class Node<E>{
        E item;
        Node<E> next;
        Node<E> prev;
};
```

```
boolean remove(Object e) {
        if (head == e) {
                head = head.next;
                if (head)
                        head.prev = null;
        }
        else
                e.prev.next = e.next;
        if (tail == e){
                tail = tail.prev;
                if (tail)
                        tail.next = null;
        }
        else
                e.next.prev = e.prev;
        return true;
}
```

Fig. 3. Removing A Node

To remove a node, it has *unlink()* method and to remove a data element it defines *remove()* method. *unlink()* needs O(1) time while *remove()* needs O(n) time. *remove()* is shown in the Fig. 4 code snippet. It traverse to locate the data element which needs O(n) time. After determining the corresponding node, the *unlink()* method is used to remove it.

```
public boolean remove(Object e){
   if (e == null) {
      for (Node<E> x = first; x != null; x = x.next)
         if (x.item == null){
            unlink(x);
            return true;}
   } else {
      for (Node<E> x = first; x != null; x = x.next)
         if (e.equals(x.item)){
            unlink(x);
            return true;}
   }
   return false;
}
```

Fig. 4. Removing A Data Element

Clearly it is not an implementation mistake. It is a consequence of the Java paradigm which is common in almost all object-oriented programming languages.

Of-course C++ and non-object-oriented programming languages such as C lets programmer store the data element in the data structure's node, and thereby they are capable to alleviate the above issue. However in those languages to implement a data structure generally the programmer has to store a reference to the data element in the data structure's node. So, they have the same problem too.

Most data structures use a key to organize and retrieve data elements. The current approach to handle key is the key/value pair method [11]. As an example, Fig.5 is the

Java *TreeMap*. In this implementation, a parameter named *K*, is used as the data type of the key. In the *Entry* class, a new attribute, named *key*, is defined for internal storage of the key value. Also, a parameter named *V*, is used as the data type of the data element, and in the *Entry* class a new attribute, named *value*, is defined for internal storage of the data element. It should be considered that the key is a field(s) of the data element and can be extracted from it.

The mechanism has using additional storage for the key issue. Since the value of the key can be extracted from the data element, there is no need to store it. Moreover, the managing changes of the key value can lead to the redundancy.

```
public class TreeMap<K, V> {
   static final class Entry<K, V> {
      K key;
      V value;
      Entry<K, V> left;
      Entry<K, V> right;
      Entry<K, V> parent;
      /*rest of the class */
   };
   /*rest of the class */
}
```

Fig. 5. Java TreeMap

3. Access Method

Examining data structures shows that their constructions follow the same framework. This framework has two segments. First, it has a segment, called node, which is responsible for keeping the main data. The node segment includes reference or references to other nodes along with the main data. The number and type of the references depend on the type of the data structure.

Second, for each data structure, a second segment, called root, is defined in which the general information of the structure is stored in. It is known as the input point to the structure. The root segment includes reference or references to some of the nodes of the structure. The management of the structure is implemented in the different operations in the root segment. The node segment usually does not perform separate operations except for providing the data [5]. We have presented the main idea called access method based on this common framework.

The access method is an abstraction for defining data structures. In this abstraction, the data structure is defined along with operations. For instance, the access method definition of a linked list is presented in Fig. 6.

In the implementation of the access method a section called element is used. The element points to the data structure of the node in the access method.

The element is the type too and points to the class of the node as a hypothetical data type. In this case, the element could be used as a data type for defining variables or in the definition of parameters. However, defined variables could not be allocated in any part of the access method. In fact, no part of the access method could get an independent memory. It is only allowed to point to the input memories.

```
access LinkedList(){
  element{
    element prev ;
    element next ;
  }
  element head , tail;
  LinkedList() {
    head = NULL;
    tail = NULL;
  }
  void remove(element e){
    if(head == e)
      head = head.next;
    if(tail == e)
      tail = tail.prev;
    if(e.prev != null)
      e.prev.next = e.next;
    if(e.next != null)
      e.next.prev = e.prev;
  }
  /* rest of the access */
}
```

Fig. 6. LinkedList Access Method

As shown in the code, the element section of the *LinkedList* has two attributes: *next* and prev. The defined access method for *LinkedList* includes one operator: *remove*. This operator acts on a variable of type *element*.

In the usage step, access method should be applied to a data element. By applying the access method on the data element, a new object is created, and the defined operations in the access method are provided along with the attributes and methods of the data element. The access methods could not be instantiated directly unlike conventional data structures. When an access method is applied, the created structure will include two segments: node and root. When an access method is applied to data, the element section is concatenated to the data, and the node segment is formed. The root segment consists of other attributes and operations, defined in the access method.

As an example, if class *Person* is defined as follows:

class Person{
int id;
string first_name;
string last_name;
string father_name;};

We could apply the *LinkedList* as shown below on the class *Person*. Thus, *people* will be a *LinkedList* of class *Person*.

Person[LinkedList()] people;

In the above example, the node object which is created for people by the compiler, includes two parts. The first part includes the defined items for class Person and the second part includes the defined items in the element section from the access method.

3.1 Key

An important characteristic of data structures is key values. To support key values, the access method has a special mechanism: hypothetical key type. If an access method has a key, it should define a key type. Inside an access method, the key type is like a usual data type. It

can be used to declare variables and arguments. The only attribute of a key type is that it defines a linear order on the elements of the data set. Therefore, it is possible to compare two key values by their key.

Often it is required to extract the key of a data element. Assume that *e* is a data element that the key *k* is defined on it, *e.k* extracts it. As an example of key type, Fig. 7 code snippet has the definition of the binary search tree access method (*Tree*). It shows that the *Tree* access method has a key type named *k*. The *lookup* operation has an argument of type *k* and finds an element having that key, i.e. *e.k == ka*. Also, in the body of *insert* operation *k* is used to compare two key values, *e1.k < e2.k*.

An access method can have more than one key. As shown in the following code snippet, *k1* and *k2* are defined as two key types of *X*.

```
access X (key k1, key k2){
    /* rest of the access method */
}
```

When an access method is instantiated, its abstract key types should be assigned values. The value of a key is a sequence of expressions composed of the data element attributes and literals. The definition of an expression of a key type is embraced in a <> pair. Some examples of key definitions are followed (Person is the base type):

```
<id>
<lname, fname>
```

The first expression consists of one attribute and the second one consists of two attributes. For instance, applying the Tree access method can be done as follows:

```
Person[Tree(<id>)] people;
```

```
access Tree(key k){
   element{
     element left, right;
   }
   element root;
   element lookup(k ka){
      element e;
      for (e = root; e; ) {
        if(e.k == ka) {
            return e;
        }
        if(e.k < ka){
          e = e.left;
        } else {
          e = e.right;
        }
      }
      return null;
   }
   element insert(element e1){
      e1.left = null;
      e1.right = null;
      if (root == null) {
        root = e1;
        return e1;
      }
      element e2,e3;
      for(e2 = root;e2;) {
        if(e1.k < e2.k){
          e3 = e2;
          e2 = e2.left;
        } else {
          e3 = e2;
          e2 = e2.right;
        }
      }
      if(e1.k < e3.k) {
        e3.left = e1;
      } else {
        e3.right = e1;
      }
      return e1;
   }
   /* rest of the access */
}
```

Fig. 7. Tree Access Method

4. Translation into Java

The access method was implemented as an extension to the Java programming language. The compiler gets a code in the extended language and produces output in the Java language. The output can be compiled using any Java compiler to produce byte code. The compiler is implemented as a multi-pass translation in Java. The translation process is implemented by means of common tools such as JFlex and Cup. It includes three phases: lexical analysis, parsing and code generation (Fig. 8).

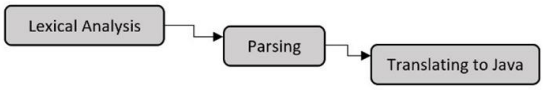

Fig. 8. Translation from the access method to Java

As specified in Fig. 8, in the first phase of lexical analysis and parsing, we perform syntactic checks like multiple declarations of the same named access methods, or declaration of element sections and operations. Next, if access method declaration and usage are matched, then the next step is the translation into Java. When we compile the back-end generated Java for execution, Type checking is handled in Java. The translation into Java is the most demanding step. During this phase, structural translation rules are followed to translate each class and access method into one or multiple classes. The resulting classes are then composed to build the complete Java representation of the source.

5. Results

As it was mentioned in the introduction, current programming languages use the referencing approach to apply a data structure on a set of data elements. The referencing approach has some issues. First, it increases the memory footprint, and second, it reduces the performance of the code. Now, the access method is implemented based on the concatenating approach, and it solves the issues of referencing approach.

To evaluate the access method, in this section it is compared with the Java and hand-coded implementations. In the Java implementation *LinkedList* and *TreeMap* is used from Java SE 10. As the time complexity of the Java approach is not satisfactory, the proposed data structures is implemented in hand-coded. In hand-coded implementation, data structures are implemented from scratch. This make more lines of codes than the access method implementation.

We perform testing for a variety of list sizes from 1000 items to 100M items. We use the Java Microbenchmark Harness (JMH) [12] test to conduct the test on a four core machine. The results are presented below subsections.

5.1 LinkedList

Assume that the *LinkedList* access method is implemented as is presented in Fig. 6. Consider the

following code snippet, *LinkedList* access method is applied on class *Person*.

Person[LinkedList ()] people;
…
people.remove (p);

Fig. 9 shows the produced code for the above code snippet.

```
class Person{
  // Person fields
  int id;
  String first_name;
  String last_name;
  String father_name;
  // injected LinkedList element
  Person prev;
  Person next;
}
class LinkedList_people{
  Person head, tail;
  LinkedList_people() {
    head = NULL;
    tail = NULL;
  }
  void remove(Person e){
    if (head == e)
      head = head.next;
    if (tail == e)
      tail = tail.prev;
    if (e.prev != null)
      e.prev.next = e.next;
    if (e.next != null)
      e.next.prev = e.prev;
  }
  /* rest of the class */
}
…
LinkedList_people people;
```

Fig. 9. Produced Code for Applied Access Method

As noted before, element part of *LinkedList* is concatenated to class *Person* as data element. So, there is no need to additional references to operate on data structures. References to class *Person* are added to class *LinkedList_people* as root of data structure. Also *remove* method is customized and added to class *LinkedList_people* based on class *Person* as data element. As the data element and the node of data structure is concatenated together, so there is no need to scan data structure, and its *remove* operation be in O(1) time.

As mentioned, to evaluate the access method, we perform testing for the linked list. This test measures the performance of creating the linked list and populating the linked list for a specified number of items in the access method, Java, and hand-coded implementations. The test code is shown below. A specified number of integers is created using the Random class and collecting them into the linked list.

```
@State(Scope.Thread)
static public class MyState {
 @Param("1000")
 public int NSIZE;
}

@Benchmark
public void test_createLinkedList(MyState state) {
 Random random = new Random();
 LinkedList< Integer > list = random
  .ints(state.NSIZE)
  .collect(LinkedList::new, List::add, List::addAll);
}
```

The performance of the insert operation of the linked list is shown in Fig. 10 in the access method, the Java and hand-coded implementations. We tested from 1000 through 100M items as shown on the X-axis. The Y-axis is nanoseconds of an operation, and is shown in log scale since there is a slope up as the size increases in the java implementation.

In the next, the performance of the remove operation of the linked list is shown in Fig. 11 too.

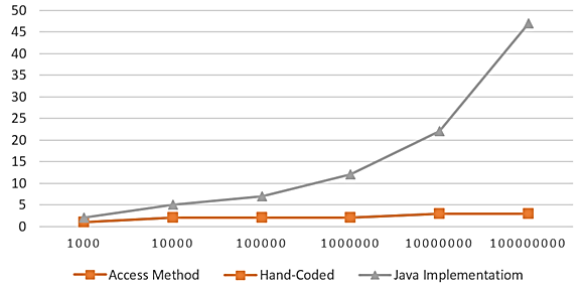

Fig. 10. Performance of the linked list insert operation in the access method, the Java and hand-coded implementations

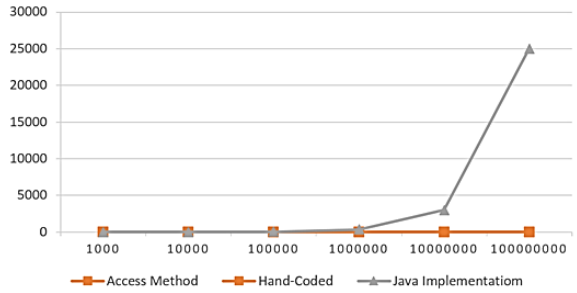

Fig. 11. Performance of the linked list remove operation in the access method, the Java and hand-coded implementations

5.2 Tree

The second test measures the performance of creating the tree and populating the tree for a specified number of items in the access method, the Java and hand-coded implementations. The test code is shown below. A specified number of integers is created using the Random class and collecting them into a particular type of tree in the access method, the Java and hand-coded implementations.

```
@State(Scope.Thread)
static public class MyState {
 @Param("1000")
 public int NSIZE;
}

@Benchmark
public void test_createTree(MyState state) {
 Random random = new Random();
 TreeMap < Integer, Integer > tree = random
  .ints(state.NSIZE)
  .collect(TreeMap::new, tree::add, tree::addAll);
}
```

The performance of the insert operation in the tree is shown in Fig. 12. As mentioned before, we tested from 1000 through 100M items as shown on the X-axis. The Y-axis is nanoseconds of an operation and is shown in

log scale since there is a slope up as the size increases in the Java implementation.

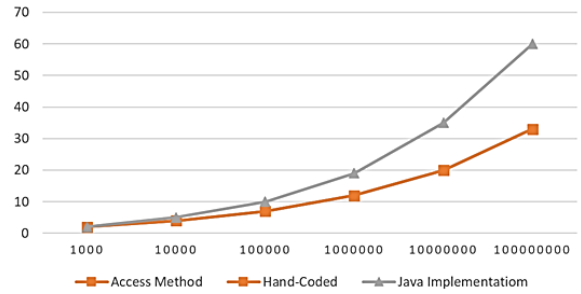

Fig. 12. Performance of the tree insert operation in the access method, the Java and hand-coded implementations

In the next, the performance of the remove operation in the tree is shown in Fig. 13.

According to the obtained results, the number of lines in the hand-coded implementation is high. Large volume of codes in the hand-coded approach makes it difficult to change and maintenance, and increase the complexity and cost of production. It's important to remember that hand-coded implementations are not reusable.

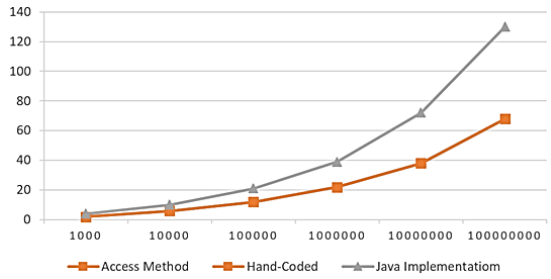

Fig. 13. Performance of the tree remove operation in the access method, the Java and hand-coded implementations

5.3 Discussion

The access method is similar to the hand-coded in time complexity. There's no perceptible difference between the access method and the hand-coded operations time. But, the number of lines in the access method implementations are low, and easy to reuse as the Java implementations. Since, the Java general data structures have high time complexity, and results show that as the number of items increases, they becomes slower which leads to lower efficiency compared to others.

6. Related Works

Several programming models attempted to provide high-level programming abstraction or interface in data structures. High-level programming models are in high-demand as they reduce the burdens of programmers [13]. However, the issue of the right high-level programming interface, especially in data structures, is not settled yet [14].

Rosenschein et al. [15] describe a language for specifying the requirements of a data structure. Then, the programming language selects the suitable data structure based on the specified requirements. Katz et al. [16] describe an expert system on data structures. The system is consulted by programmers during the design stage of their programs.

Schonberg et al. [17],[18] describe a technique for automatic selection of appropriate data representations during compile-time, and present a data structure selection algorithm in the SETL language.

Low [19] suggests that the data structures are represented as the abstract data types. For each abstract data type, some representations are provided, and the compiler chooses the best implementation.

7. Conclusions and Future Works

This paper introduced a new approach to implement data structures. The approach is based on four features: performance, simplicity, flexibility and not making any decision on behalf of the programmer. The approach consists of a new abstraction, the access method to define a data structure, and a new type for defining key. The provided samples show that the approach effectively reduces the cost of data structures operations and the approach creates a program-independent way to data structures define and manipulation.

The key direction for future work is extending the access method abstraction to support data structures compositions to provide the ability that an access method can make using other access methods.

References

[1] J. H. Drew, D. L. Evans, A. G. Glen, and L. M. Leemis, “Data Structures and Simple Algorithms,” in Computational Probability, Springer, 2017, pp. 89–109.

[2] I. Haller, A. Slowinska, and H. Bos, “Scalable data structure detection and classification for C/C++ binaries,” Empir. Softw. Eng., vol. 21, no. 3, pp. 778–810, 2016.

[3] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr, “Optimising Darwinian Data Structures on Google Guava,” in International Symposium on Search Based Software Engineering, 2017, pp. 161–167.

[4] M. Basios, L. Li, F. Wu, L. Kanthan, D. Lawrence, and E. Barr, “Darwinian Data Structure Selection,” arXiv Prepr. arXiv1706.03232, 2017.

[5] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, and C. STEIN, Introduction to Algorithms 3rd Edition. MIT press, 2009.

[6] C. Van Reeuwijk and H. J. Sips, “Adding tuples to Java: A study in lightweight data structures,” in Proceedings of the Concurrency Computation Practice and Experience, 2005, vol. 17, no. 5–6 SPEC. ISS., pp. 423–438.

[7] M. Sakkinen, "Disciplined Inheritance," in ECOOP 1989: European Conference on Object-Oriented Programming, 1989, pp. 39–57.
[8] Y. Zhang, M. C. Loring, G. Salvaneschi, B. Liskov, and A. C. Myers, "Lightweight, flexible object-oriented generics," in ACM SIGPLAN Notices, 2015, vol. 50, no. 6, pp. 436–445.
[9] S. Lindell, "A normal form for first-order logic over doubly-linked data structures," Int. J. Found. Comput. Sci., vol. 19, no. 1, pp. 205–217, 2008.
[10] C. Loncaric, E. Torlak, and M. D. Ernst, "Fast synthesis of fast collections," ACM SIGPLAN Not., vol. 51, no. 6, pp. 355–368, 2016.
[11] Y. Smaragdakis and D. S. Batory, "DiSTiL: A transformation library for data structures," in Proceedings of USENIX Conference on Domain-Specific Languages, 1997, no. October, p. 257270.
[12] Java.net, "JMH Test," 2017. [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh/.
[13] N. Khammassi and J.-C. Le Lann, "A high-level programming model to ease pipeline parallelism expression on shared memory multicore architectures," Simul. Ser., vol. 46, no. 5, pp. 63–70, 2014.
[14] Y. Smaragdakis, "Technical Perspective High-Level Data Structures," Commun. ACM, vol. 55, no. 12, p. 2380656, 2012.
[15] S. J. Rosenschein and S. M. Katz, "Selection of representations for data structures," in Proceedings of the 1977 symposium on Artificial intelligence and programming languages., 1977, pp. 147–154.
[16] S. Katz and R. Zimmerman, "An advisory system for developing data representations," in Proceedings of the 7th international joint conference on Artificial intelligence, 1981, pp. 1030–1036.
[17] E. Schonberg, J. T. Schwartz, and M. Sharir., "An automatic technique for selection of data representations in setl programs," ACM Trans. Program. Lang. Syst., vol. 3, no. 2, pp. 126–143, 1981.
[18] E. Schonberg, J. T. Schwartz, and M. Sharir., "Automatic data structure selection in setl.," in Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1979, pp. 197–210.
[19] J. R. Low, "Automatic data structure selection: an example and overview," Commun. ACM, vol. 21, no. 5, pp. 376–385, 1978.

Davud Mohammadpur received his M.Sc. degree in Software Engineering from Iran University of Science and Technology. Currently he is a faculty member of University of Zanjan and a Ph.D. candidate at Malek-Ashtar University of Technology. His research interests are programming languages and information systems.

Ali Mahjur received his B.Sc., M.Sc. and Ph.D. from Sharif University of Technology. Currently he is a faculty member of Malek Ashtar University of Technology. His research interests are programming languages, operating systems and processor microarchitecture.