Shortcuts: Abstract "Pointers"

J. Marco
jmarco@lsi.upc.es

X. Franch
franch@lsi.upc.es

Dept. Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya (UPC)
Campus Nord
Jordi Girona Salgado, 1-3
08034 BARCELONA

Abstract

In this work we present the specification and the implementation of a new abstract data type (ADT) called *STORE*. This new ADT allows the storage of a given collection of elements offering an abstract mechanism that supplies a direct access to them, alternative to the method defined by the standard operations of usual ADTs. The interest of the new mechanism stems from the efficiency of pointers, while avoiding the loss of modularity that usually occurs when pointers are used. The implementation of the operations offered by the new ADT is done by derivation from the equational specification. The representation chosen for the implementation of the new ADT makes the efficiency previously mentioned possible, even when the representation strategy requires the movement of the elements.

Contents

1 Introduction **3**

2 The ADT *STORE* 4

2.1 Operations 5

2.1.1 Constructor Operations 5

2.1.2 Observer Operations 5

2.1.3 Modifier Operations 6

2.2 Equational specification 6

2.2.1 Equations between Constructor Operations 6

2.2.2 Equations of the Observer Operations 8

2.2.3 Equations of Modifier Operations 10

3 Implementation in Main Memory **14**

3.1 The Type Representation 15

3.2 Deriving code for the operations 19

3.2.1 Derivation of *put* 19

3.3 Chosing implementations for the components of the representation 25

4 An Example: The Tennis Ladder **26**

4.1 Aho, Hopcroft & Ullman's Solutions to the Problem 26

4.2 The solution with the ADT *STORE* 28

5 Codifications **29**

6 Conclusions **30**

1 Introduction

The present work deals with the methodology of modular program development by means of abstract data types (ADT). The modular methodology offers many important properties such as understanding, abstraction, reusability, etc., but, often, it implies a loss of efficiency due to the impossibility of accessing implementation of ADTs from other modules.

Often, while designing a data structure, we face the problem of reducing the space cost (i.e., minimising redundant data) or the temporal cost (i.e., accessing more faster to data). To overcome these problems we may need an implementation which permits the access to the data by means of pointers (see, e.g, [1, pag. 149]). Recall that the notion of *pointer* corresponds to the memory address of a given object (with some high-level facilities, which vary among the different programming languages). Unfortunately, the use of pointers causes in general a complete loss of modularity because:

- It is necessary to know the underlying data structure used in the implementation and therefore, information hiding is not accomplished.
- Correctness proofs and software maintenance and reusability are, in general, more complicated. Also, programs are more difficult to understand.
- An implementation using pointers needs to guarantee that data will not change their position in the structure; otherwise we would not access the desired information (unless all client modules keep track of changes and this may be impossible).
- When the implementation is done by means of pointers not only the ADT operations can be used but also we can access directly to its representation and manipulate it. Therefore, it is possible to manipulate ADTs without respecting the policy defined by their equations.

With the aim of avoiding these drawbacks, we propose the design of a new ADT, which not only permits to store data and to access them by means of a key, but also allows for direct access to data without knowing how they are stored. We should mention that the addition of this mechanism will be done by defining its formal semantics. Certainly, the idea to study this new ADT comes from the necessity of a compromise between modularity and efficiency.

The goal of the present work is to present the design of this new ADT which offers, in the one hand, functions of data storage and access by key to data structures, independently of their organization

and of the key type and, on the other hand, the possibility of a direct access to data by means of an address we call the *shortcut*. Essentially, we obtain an abstract mechanism inside the ADT which corresponds to the pointers in both concept and efficiency.

The rest of the paper is organized as follows. In section 2, we present the algebraic specification of the new ADT showing the sorts and the operations offered as well as the equations that define the behaviour of each of them. In section 3, we choose first the representation of the new data type, then we show the complete derivation of one of the operations and finally we indicate the cost of each operation, depending on the detailed implementation of the representation used. In section 4, we consider a classic example that requires the use of pointers for efficiency reasons, and where we can see that the use of the new ADT assures the same efficiency without losing modularity at all. In section 5, we include several conclusions regarding the codification of the new ADT in ADA, Modula-2 and C++. Finally, in section 6, we summarise the advantages of the new ADT and we present some open questions for further research. This paper is based on [11], where all the remaining discussions can be found, regarding the derivation of all the operations and details of the three implementations.

2 The ADT *STORE*

Since we want to obtain an ADT which permits to store any kind of information, we have designed an ADT that is "generic" ("parameterized"). The new ADT, called *STORE*, has the data to be stored as a parameter , decomposed into *key* and *information*. The parameter *key* is used to access the data. There are no operations required on *information*; the sort *key* must have comparison operations[1]. The ADT *STORE* offers two sorts:

- **store:** where the data is stored.
- **shortcut:** which supplies the direct access to the stored data.

[1] From now on, we use the operation symbol *cmp* to refer to the equality operation.

2.1 Operations

2.1.1 Constructor Operations

create: $\longrightarrow$ *store*

creates an empty store.

put: store, key, information $\longrightarrow \langle$ *store, shortcut* $\rangle$

returns a pair formed by the resulting store after adding the pair of key and information, and the shortcut which gives direct access to this pair.

Alternatively, we could decompose *put* into two operations, one for each component of the result. However, we have preferred the above mentioned structure for the sake of clearness of algebraic specification presented in Subsection 2.2.

2.1.2 Observer Operations

getInfKey: store, key $\longrightarrow$ *information*

returns the information associated to the key. Precondition: the key is in the store.

getInfSho: store, shortcut $\longrightarrow$ *information*

returns the information associated to the shortcut. Precondition: the shortcut has data associated inside the store.

getKey: store, shortcut $\longrightarrow$ *key*

returns the key associated to the shortcut. Precondition: the shortcut has data associated inside the store.

isIn?: store, key $\longrightarrow$ *bool*

returns a boolean value indicating whether the key is in the store.

getShortcut: store, key $\longrightarrow$ *shortcut*

returns the shortcut associated to the key. Precondition: the key is in the store.

isEmpty?: store $\longrightarrow$ *bool*

returns a boolean value indicating whether the store is empty.

2.1.3 Modifier Operations

remove: store, shortcut $\longrightarrow$ *store*

returns the resulting store after removing the data associated to the shortcut. Precondition: the shortcut has data associated inside the store.

modify: store, shortcut, information $\longrightarrow$ *store*

returns the resulting store after substituting the information associated to the shortcut, by the new information. Precondition: the shortcut has data associated inside the store.

2.2 Equational specification

Following the general method of algebraic specification with initial semantics, we give first the equations between constructor operations [4, 5, 6].

2.2.1 Equations between Constructor Operations

It seems that we could assume that the *stores* behave as sets, in that the ordering in which the data are stored is not important. Actually this happens in "table" like ADT's [6]. In order to assure this property we should have to guarantee that, for a given *key*, we obtain always the same *shortcut*, regardless of the moment in which the *key* and its *information* are stored. This would imply that different keys could not have the same *shortcut*; otherwise, storing an *information* and a *key* implies a risk of modifying the *information* of another key (of the same *shortcut*) which was stored previously. Therefore, there should be a unique *shortcut* corresponding to a *key*, and this, in turn, requires the number of keys and the keys themselves to be known before hand. Alternatively, we can do it by an injective function from the set of keys to the set of shortcuts. The last requirement is very restrictive, if we want the *store* to be independent from the *key*.

We overcome this restriction, by letting the *shortcut* to depend also on the *store*, and in this way different keys can have the same *shortcut*, if they are stored in different stores. Therefore, the order in which data are stored is important, and thus *stores* will not behave as sets.

Now, let us consider the relations between the two consecutive *put* operations, when we store some *information* with a *key* which was previously in the *store*. Since we can have only a unique *information* associated to each *key*, the result will be that of replacing the *information* associated to the *key* by the new *information*. The question that arises here is: which is the *shortcut* obtained?

Since there was a *shortcut* associated to the *key* (i.e., the *shortcut* obtained when stored the previous *information*) and since we want a unique *shortcut* corresponding to the *key*, then it is precisely the former *shortcut* the one we will obtain.

Let us see the equations which express these relations. Let us start by those related to stores; we will use the operation *isIn?* which tell us whether the key is in the store. This operation is used only when the store is not empty, because otherwise the store would not have this relation.

In the case when the *key* is therein, we consider the following two cases:

1. The key is the last one added to the store.
2. The key exists but it is not the last one.

In the first case, the equation is simple; it just expresses the fact that the obtained store will be the same as if the last time we had stored the new information.

$$\Big[\text{isIn?}(\text{put}(\text{st},\text{k}_1,\text{i}_1).\text{store},\text{k}_2) \wedge \text{cmp}(\text{k}_1,\text{k}_2)\Big] \Rightarrow$$
$$\text{put}(\text{put}(\text{st},\text{k}_1,\text{i}_1).\text{store},\text{k}_2,\text{i}_2).\text{store} = \text{put}(\text{st},\text{k}_2,\text{i}_2).\text{store}$$

In the second case, the equation expresses the swap between the two last keys obtaining this:

$$\Big[\text{isIn?}(\text{put}(\text{st},\text{k}_1,\text{i}_1).\text{store},\text{k}_2) \wedge \neg\ \text{cmp}(\text{k}_1,\text{k}_2)\Big] \Rightarrow$$
$$\text{put}(\text{put}(\text{st},\text{k}_1,\text{i}_1).\text{store},\text{k}_2,\text{i}_2).\text{store} = \text{put}(\text{put}(\text{st},\text{k}_2,\text{i}_2).\text{store},\text{k}_1,\text{i}_1).\text{store}$$

Notice that in this case there is no a substitution of the information associated to the key, but applying this equation successively, we will arrive to the first case since the key is therein. Therefore, we can apply the first equation and in this way to substitute the information associated to the key.

Let us see now the equations for shortcuts. As before, we distinguish two cases:

1. The key is the last one added to the store.
2. The key exists, but it is not the last one.

In the first case, we only have to express that the obtained *shortcut* will be the one obtained previously.

$$\Big[\text{isIn?}(\text{put}(\text{st},\text{k}_1,\text{i}_1).\text{store},\text{k}_2) \wedge \text{cmp}(\text{k}_1,\text{k}_2)\Big] \Rightarrow$$
$$\text{put}(\text{put}(\text{st},\text{k}_1,\text{i}_1).\text{store},\text{k}_2,\text{i}_2).\text{shortcut} = \text{put}(\text{st},\text{k}_1,\text{i}_1).\text{shortcut}$$

In the second case we have to just express that the obtained *shortcut* will be the same as the one that we would have obtained, if the new key and the new information had been stored in the previous *store*.

$$\Big[\text{isIn?}(\text{put}(\text{st},k_1,i_1).\text{store},k_2) \wedge \neg\ \text{cmp}(k_1,k_2)\Big] \Rightarrow$$
$$\text{put}(\text{put}(\text{st},k_1,i_1).\text{store},k_2,i_2).\text{shortcut} = \text{put}(\text{st},k_2,i_2).\text{shortcut}$$

This does not specify yet completely the behaviour of shortcuts, because we have not considered the case in which the key is not therein; in fact, in this case, *put* may return any value as shortcut but the access to the shortcut should supply the appropriate results. The equations of the operation *getInfSho* will describe this behaviour, as given next.

2.2.2 Equations of the Observer Operations

Let us study now the behaviour of the observer operations.

- The equations of *getInfSho*:
  - The first one will be an error equation, indicating that the operation is not defined for an empty *store*. Recall that this operation is partial.

    $$\text{getInfSho}(\text{create},\text{sc}) = \text{error}$$

  - We also include an equation to express, the behaviour when the *shortcut* whose *information* we want to consult is the same as the one returned in the last operation of *store*. The *information* that we will obtain will be the one of this last operation. In order to compare the *shortcuts* we will use a private operation, denoted *cmp*.

    $$\Big[\text{cmp}(\text{sc},\text{put}(\text{st},k,i).\text{shortcut})\Big] \Rightarrow \text{getInfSho}(\text{put}(\text{st},k,i).\text{store},\text{sc}) = i$$

  - The next equation indicates that, if the *shortcut* is not the same as the one obtained in the last operation of *put*, then the result of consulting the information associated to the *shortcut* will be the same as consulting the previous *store*.

    $$\Big[\neg\ \text{cmp}(\text{sc},\text{put}(\text{st},k,i).\text{shortcut})\Big] \Rightarrow$$
    $$\text{getInfSho}(\text{put}(\text{st},k,i).\text{store},\text{sc}) = \text{getInfSho}(\text{st},\text{sc})$$

Notice that, if there was no *information* associated to the *shortcut*, by applying successively this equation, we would obtain *st=create*, and therefore from the first equation this is an error.

- Equations of *getKey*, similar to those of *getInfSho*:

  $$\text{getKey(create,sc)} = \text{error}$$
  $$\left[\text{cmp(sc,put(st,k,i).shortcut)}\right] \Rightarrow \text{getKey(put(st,k,i).store,sc)} = \text{k}$$
  $$\left[\neg\ \text{cmp(sc,put(st,k,i).shortcut)}\right] \Rightarrow \text{getKey(put(st,k,i).store,sc)} = \text{getKey(st,sc)}$$

- The Equations of *getInfKey*:

  - The first equation states that, if the *key* to be consulted is the last one stored then we will obtain the last the *information* stored.

    $$\left[\text{cmp}(\text{k}_1,\text{k}_2)\right] \Rightarrow \text{getInfKey(put(st,}\text{k}_1\text{,i).store,}\text{k}_2\text{)} = \text{i}$$

  - If the condition of the first equation does not hold then we have a second equation, and the corresponding result will be that of applying *getInfKey* on the previous *store*.

    $$\left[\neg\ \text{cmp}(\text{k}_1,\text{k}_2)\right] \Rightarrow \text{getInfKey(put(st,}\text{k}_1\text{,i).store,}\text{k}_2\text{)=getInfKey(st,}\text{k}_2\text{)}$$

  - Since the operation in question is a partial one (because it is not defined when the *key* is not in the *store*) we must also add a last equation which indicates that consulting the *information* associated to a *key* in an empty *store* will be an error.

    $$\text{getInfKey(create,k)} = \text{error}$$

- Equations of *isIn?*:

  - The first equation is intuitively clear. If we apply *isIn?* to a *key* and to an empty *store*, then the result will be false.

    $$\text{isIn?(create,k)} = \text{false}$$

  - In the same way, the second equation comes naturally. If the *key* to which we apply the operation *isIn?* is the last one added to the *store*, then the result will be true; if the *key* is not the last one added, then the result will be that of applying *isIn?* to the previous *store*.

$$\text{isIn?}(\text{put}(\text{st},k_1,i_1).\text{store},k_2) = \text{cmp}(k_1,k_2) \vee \text{isIn?}(\text{st},k_2)$$

- The corresponding equations of *getShortcut*:
  - Applying *getShortcut* to a *key* and an empty *store* will give error.

    $$\text{getShortcut}(\text{create},k) = \text{error}$$

  - If the *key* to which we apply *getShortcut* is the last one added to the *store*, then we will obtain the *shortcut* of the last application of *put*.

    $$\left[\text{cmp}(k_1,k_2)\right] \Rightarrow \text{getShortcut}(\text{put}(\text{st},k_1,i_1).\text{store},k_2) = \text{put}(\text{st},k_1,i_1).\text{shortcut}$$

  - If the *key* to which we apply *getShortcut* is not the last one added to the *store*, then we will have the same *shortcut* as the one obtained from applying *getShortcut* to the previous *store*.

    $$\left[\neg\ \text{cmp}(k_1,k_2)\right] \Rightarrow \text{getShortcut}(\text{put}(\text{st},k_1,i_1).\text{store},k_2) = \text{getShortcut}(\text{st},k_2)$$

    When the *key* is not in the *store*, then eventually we would have *st=create*, and from the first equation, it will be an error.

- Equations of *isEmpty?*:
  - First, we express that applying *isEmpty?* to a *store* with nothing therein yields true.

    $$\text{isEmpty?}(\text{create}) = \text{true}$$

  - Next, applying *isEmpty?* to a *store* with information in it yields false.

    $$\text{isEmpty?}(\text{put}(\text{st},k,i).\text{store}) = \text{false}$$

2.2.3 Equations of Modifier Operations

Let us study now the behaviour of the modifier operations.

- The equations of *remove*:
  - If we try to *remove* the data associated to a *shortcut* of an empty *store* we will raise an error.

    $$\text{remove}(\text{create},\text{sc}) = \text{error}$$

– If the *store* is not empty we distinguish two cases: when the data to be removed is the one accessible by means of the *shortcut* obtained from the last application of *put*, and the opposite one.

In the first case, since we cannot have error as a basic case, i.e., we are not able to remove the data associated to a *shortcut* in an empty *store*, then we consider separately the following two subcases:

a) There is no *information* previously stored using the *key* of the last application. Therefore, the result of *remove* will be the same *store* as the one before applying this operation.

$$\Big[\text{cmp(sc,put(st,k,i).shortcut)} \wedge \neg\ \text{isIn?(st,k)}\Big] \Rightarrow \text{remove(put(st,k,i).store,sc)} = \text{st}$$

b) Otherwise, the operation *remove* will be applied again.

$$\Big[\text{cmp(sc,put(st,k,i).shortcut)} \wedge \text{isIn?(st,k)}\Big] \Rightarrow$$
$$\text{remove(put(st,k,i).store,sc)} = \text{remove(st,sc)}$$

When the *shortcut* is not the last one obtained, then we will have the same result as that of applying *put* to the last *key*, to the last *information* and to the *store* resulting of applying *remove* to the previous *store* and to the *shortcut*.

$$\Big[\neg\ \text{cmp(sc,put(st,k,i).shortcut)}\Big] \Rightarrow$$
$$\text{remove(put(st,k,i).store,sc)} = \text{put(remove(st,sc),k,i).store}$$

Notice that if there are no data associated to the *shortcut* then we would have *st=create* and, therefore, from the first equation it would be an error.

- Equations of *modify*:

– First, we express that trying to modify the *information* associated to a *shortcut* of an empty *store* produces an error.

$$\text{modify(create,sc,i)} = \text{error}$$

– Next, if the *shortcut* whose *information* is to be modified is the same as that obtained from the last application of *put*, the result will be the same as if the last *put* had stored the new *information*.

$$\Big[\text{cmp(sc,put(st,k,i}_1\text{).shortcut)}\Big] \Rightarrow$$
$$\text{modify(put(st,k,i}_1\text{).store,sc,i}_2\text{)} = \text{put(st,k,i}_2\text{).store}$$

– Finally, consider the case that the *shortcut* whose *information* is to be modified is different from the one obtained by the last application of *put*. Then, we will obtain the result of applying *put* to the last *key*, to the last *information* and to the *store* which is given by modifying the previous *store*. Notice that, again, if there were no *information* associated to the *shortcut*, then the previous *store* would be empty and from the first equation, if would be an error.

$$\left[\neg\ \text{cmp(sc,put(st,k,i}_1\text{).shortcut)}\right] \Rightarrow$$
$$\text{modify(put(st,k,i}_1\text{).store,sc,i}_2\text{)} = \text{put(modify(st,sc,i}_2\text{),k,i}_1\text{).store}$$

We summarize the results in the following parameterized ADT:

universe STORE (KEY, INFORMATION) **is**

type store, shortcut

imports BOOL

ops

create: $\longrightarrow$ store

put: store, key, information $\longrightarrow \langle$ store,shortcut $\rangle$

getInfKey: store, key $\longrightarrow$ information

getInfSho: store, shortcut $\longrightarrow$ information

getKey: store, shortcut $\longrightarrow$ key

isIn?: store, key $\longrightarrow$ bool

getShortcut: store, key $\longrightarrow$ shortcut

isEmpty?: store $\longrightarrow$ bool

remove: store, shortcut $\longrightarrow$ store

modify: store, shortcut, information $\longrightarrow$ store

private cmp: shortcut, shortcut $\longrightarrow$ bool

errors $\forall$ k $\in$ key; $\forall$ i $\in$ information; $\forall$ sc $\in$ shortcut

getInfKey(create,k) = error

getInfSho(create,sc) = error

getKey(create,sc) = error

getShortcut(create,k) = error

remove(create,sc) = error

modify(create,sc,i) = error

eqns $\forall$ st $\in$ store; $\forall$ $k,k_1,k_2 \in$ key; $\forall$ $i,i_1,i_2 \in$ information; $\forall$ sc $\in$ shortcut

$\left[\text{isIn?}(\text{put}(st,k_1,i_1).\text{store},k_2) \wedge \text{cmp}(k_1,k_2)\right] \Rightarrow \text{put}(\text{put}(st,k_1,i_1).\text{store},k_2,i_2).\text{store} = \text{put}(st,k_2,i_2).\text{store}$

$\left[\text{isIn?}(\text{put}(st,k_1,i_1).\text{store},k_2) \wedge \neg\, \text{cmp}(k_1,k_2)\right] \Rightarrow$

$\text{put}(\text{put}(st,k_1,i_1).\text{store},k_2,i_2).\text{store} = \text{put}(\text{put}(st,k_2,i_2).\text{store},k_1,i_1).\text{store}$

$\left[\text{isIn?}(\text{put}(st,k_1,i_1).\text{store},k_2) \wedge \text{cmp}(k_1,k_2)\right] \Rightarrow$

$\text{put}(\text{put}(st,k_1,i_1).\text{store},k_2,i_2).\text{shortcut} = \text{put}(st,k_1,i_1).\text{shortcut}$

$\left[\text{isIn?}(\text{put}(st,k_1,i_1).\text{store},k_2) \wedge \neg\, \text{cmp}(k_1,k_2)\right] \Rightarrow$

$\text{put}(\text{put}(st,k_1,i_1).\text{store},k_2,i_2).\text{shortcut} = \text{put}(st,k_2,i_2).\text{shortcut}$

$\left[\text{cmp}(k_1,k_2)\right] \Rightarrow \text{getInfKey}(\text{put}(st,k_1,i).\text{store},k_2) = i$

$\left[\neg\, \text{cmp}(k_1,k_2)\right] \Rightarrow \text{getInfKey}(\text{put}(st,k_1,i).\text{store},k_2) = \text{getInfKey}(st,k_2)$

$\left[\text{cmp}(sc,\text{put}(st,k,i).\text{shortcut})\right] \Rightarrow \text{getInfSho}(\text{put}(st,k,i).\text{store},sc) = i$

$\left[\neg\, \text{cmp}(sc,\text{put}(st,k,i).\text{shortcut})\right] \Rightarrow \text{getInfSho}(\text{put}(st,k,i).\text{store},sc) = \text{getInfSho}(st,sc)$

$\left[\text{cmp}(sc,\text{put}(st,k,i).\text{shortcut})\right] \Rightarrow \text{getKey}(\text{put}(st,k,i).\text{store},sc) = k$

$\left[\neg\, \text{cmp}(sc,\text{put}(st,k,i).\text{shortcut})\right] \Rightarrow \text{getKey}(\text{put}(st,k,i).\text{store},sc) = \text{getKey}(st,sc)$

isIn?(create,k) = false

$\text{isIn?}(\text{put}(st,k_1,i_1).\text{store},k_2) = \text{cmp}(k_1,k_2) \vee \text{isIn?}(st,k_2)$

$\left[\text{cmp}(k_1,k_2)\right] \Rightarrow \text{getShortcut}(\text{put}(st,k_1,i_1).\text{store},k_2) = \text{put}(st,k_1,i_1).\text{shortcut}$

$\left[\neg\, \text{cmp}(k_1,k_2)\right] \Rightarrow \text{getShortcut}(\text{put}(st,k_1,i_1).\text{store},k_2) = \text{getShortcut}(st,k_2)$

isEmpty?(create) = true

isEmpty?(put(st,k,i).store) = false

$\left[\text{cmp}(sc,\text{put}(st,k,i).\text{shortcut}) \wedge \text{isIn?}(st,k)\right] \Rightarrow \text{remove}(\text{put}(st,k,i).\text{store},sc) = \text{remove}(st,sc)$

$\left[\text{cmp}(sc,\text{put}(st,k,i).\text{shortcut}) \wedge \neg\, \text{isIn?}(st,k)\right] \Rightarrow \text{remove}(\text{put}(st,k,i).\text{store},sc) = st$

$\left[\neg\, \text{cmp}(sc,\text{put}(st,k,i).\text{shortcut})\right] \Rightarrow \text{remove}(\text{put}(st,k,i).\text{store},sc) = \text{put}(\text{remove}(st,sc),k,i).\text{store}$

$\left[\text{cmp}(sc,\text{put}(st,k,i_1).\text{shortcut})\right] \Rightarrow \text{modify}(\text{put}(st,k,i_1).\text{store},sc,i_2) = \text{put}(st,k,i_2).\text{store}$

$\left[\neg\, \text{cmp}(sc,\text{put}(st,k,i_1).\text{shortcut})\right] \Rightarrow \text{modify}(\text{put}(st,k,i_1).\text{store},sc,i_2) = \text{put}(\text{modify}(st,sc,i_2),k,i_1).\text{store}$

end universe

universe INFORMATION **characterise**

type information

end universe

universe KEY **characterise**

type key

imports bool

ops

cmp, _<_: key, key $\longrightarrow$ bool

eqns $\forall$ k, k_1, k_2, $k_3 \in$ key

cmp(k,k) = true

$\left[\text{cmp}(k_1,k_2)\right] \Rightarrow \text{cmp}(k_2,k_1) = \text{true}$

$\left[\text{cmp}(k_1,k_2) \wedge \text{cmp}(k_2,k_3)\right] \Rightarrow \text{cmp}(k_1,k_3) = \text{true}$

$k < k = \text{false}$

$\left[k_1 < k_2\right] \Rightarrow k_2 < k_1 = \text{false}$

$\left[k_1 < k_2 \wedge k_2 < k_3\right] \Rightarrow k_1 < k_3 = \text{true}$

end universe

We determine the model of ADT *STORE* interpreting the equations by means of initial semantics. Given the properties of *put* it can be derived that the model corresponding to the carrier sets of sort *store* consists of the pairs of functions $g : K \rightarrow U \times h : K' \leftrightarrow A'$ that satisfy: $K' = domain(g)$, $A' \subseteq A$, $K' \subseteq K$ where A, K and U are the carrier sets of the *shortcuts*, the *keys* and the *information* respectively. The bijection assures that to any *key* there is one and only one *shortcut* associated to it, while the condition on the domains assures that the *shortcuts* corresponds only to the *keys* already defined. The operations of the model are obtained by interpreting the operations of the ADT on the above functions; for example, *create* is interpreted as the functions g and h that satisfy $domain(g) = domain(h) = \emptyset$. The proof of the model is out of the scope of this paper, and it is omitted.

3 Implementation in Main Memory

In this section we choose a representation for the sorts of the new ADT and we show a derivation of one of the operations as example starting from its pre-post specification [3]. Also, we present different possibilities to implement the new ADT with the chosen representation, and we give the cost of each one of them.

3.1 The Type Representation

In order to have direct access to the data we may think of representing the *store* as an array having as components pairs of a *key* and an *information*, and the type of *shortcut* to be the index of the array (see Fig. 1). With this representation, accessing the *store* by means of the *shortcut* could be done with constant cost while accessing by means of the *key* would have a linear cost (in terms of the array size) due to the searching of the *key* in the array.

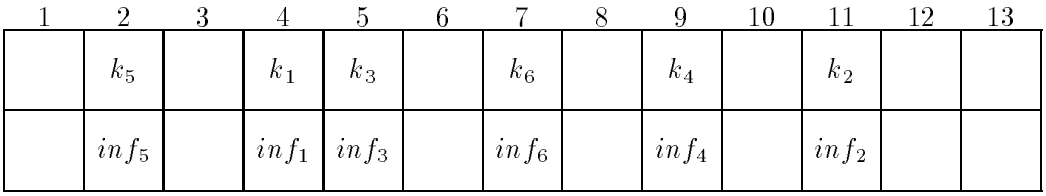

Figure 1: array

To avoid this cost we decided to add another data structure that, given a *key* allows us to know which is the *shortcut* associated to it, that means, the position on the array where the data is found. Thus, the type *store* would be a pair that consist of a table of pairs $\langle key, shortcut \rangle$ and the previous array (see Fig. 2). With this new structure the operations with the *key* will have a reasonable cost depending on the implementation of the table used.

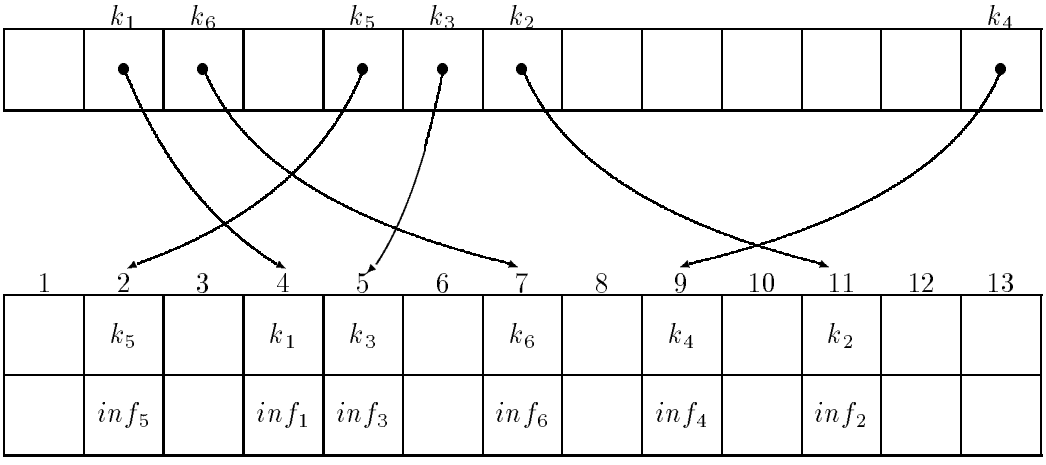

Figure 2: array and table

Using this representation of the type *store* we still have an operation of high cost, namely the *put* operation. When we store the data by means of a *key* which is already in the *store*, we have only to obtain the *shortcut* associated to the *key* and then using it to access to the array, assigning the value of the new data. In this case, the cost is that of consulting the table. However, when the *key* is not in the *store*, we have to search for free position in the array, and this, in the worst case, has linear cost. To avoid this cost we add another structure to find quickly a free position. This structure is a queue where the free positions of the array are stored (see Fig. 3). With this modification, the cost of finding a free position is constant (provided that the implementation of the queue is good enough, it corresponds just to obtaining the first element of the queue). The cost of putting a position as a free one, while removing, could be made also constant if it just corresponds to the cost of putting an element to the queue.

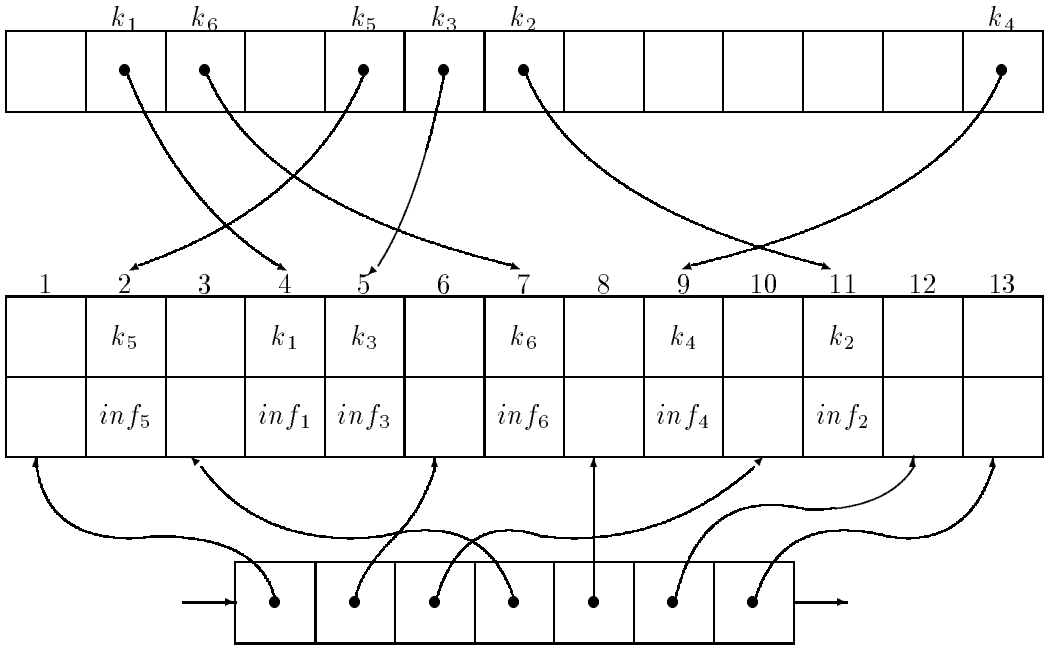

Figure 3: array, table and queue

The representation of the new sorts of data is:

- *store* is implemented by a record consisting of three fields:

  1. a: array of n components, where the valid indices are the natural numbers between 1 and n, for a certain natural n; the components of the array are pairs consisting of two fields: *inf* (of type *information*) and *key* (of type *key*).
  2. f: a queue of *shortcuts* i.e., a queue containing the indices corresponding to the free positions of the array.
  3. t: is a table of pairs $\langle key, shortcut \rangle$.

- The *shortcut* is implemented by a natural in the range $0 \ldots n$. The value 0 is used as an undefined value. The rest of the values will correspond to the indices of the array.

To verify that the implementation of the new ADT is correct with respect to its specification we follow the method presented in [8]. It consists of defining the abstraction function, which transforms a value of the implementation to a value of its model, denoted by a term which belongs to its associated equivalence class in the quotient-term algebra [4]. Since this function is partially defined, we must also define the invariant of the representation which establishes the condition that the values of the implementation must satisfy so to represent valid values of the ADT. Also, we need to indicate when two different values of the implementation correspond to the same value of ADT under implementation, in other words, we have to specify the redefinition of the equality.

Here is a short description of the array, queue and table operations, respectively, that we are going to use. $cons(a, p)$ returns the value in the position p of the array a; $ass(a, p, v)$ assigns the value v in the position p of the array a; $putFirst(q, v)$ puts the value v in the front of the queue q; $assign(t, k, v)$ associates the value v to the key k in the table t; $deassign(t, k)$ associates an undefined value to the key k in the table t; and $lookup(t, k)$ returns the value associated to the key k in the table t.

The Abstraction Function

$$convert(\langle a, f, t \rangle) = \begin{cases} create & \text{if } t\text{=TABLE.create} \\ \\ put(convert(\langle a, putFirst(f, sc), deassign(t, k) \rangle), k, cons(a, sc).inf).store & \text{if } t\text{=assign(t',k,sc)} \end{cases}$$

The Representation Invariant

The representation invariant must assure that every position of the array is free or occupied but not both of them at the same time. Therefore, it indicates that any position of the array, i.e., any natural between 1 and n must be either in the queue of free positions (only once) or as a value associated to a *key* in the table. In this last case, the *key* cannot be whatever: it must coincide with the value of the field of the array in this position. Since a *shortcut* cannot be associated to more than a *key*, at a certain moment, we must assure also that, given a position there is only one *key* in the table which it is associated to.

But the *shortcut* depends on the *store*, therefore the representation invariant must indicate which is the *shortcut* that will assign our representation when we store an information by means of a *key* that is not found in the *store*. Since the only condition that must satisfy the *shortcut*, is not to be associated to another *key*, any of the queue positions can be used, for instance its head.

We can now write down the invariant.

First we will define a predicate that indicates whether a *shortcut* belongs only once to the queue.

$\forall f \in queue; \forall sc, sc_1, sc_2 \in shortcut$

$$
\begin{array}{l|l}
unique(create, sc) = false & sc \in create = false \\
unique(enqueue(f, sc), sc) = \neg(sc \in f) & sc \in enqueue(f, sc) = true \\
[\neg ig(sc_1, sc_2)] \Rightarrow & [\neg ig(sc_1, sc_2)] \Rightarrow \\
\quad unique(enqueue(f, sc_1), sc_2) = unique(f, sc_2) & \quad sc_2 \in enqueue(f, sc_1) = sc_2 \in f
\end{array}
$$

$\text{Inv\_Rep}(\langle a, f, t\rangle) \equiv$

$\forall j : 1 \leq j \leq n : \Big(\Big(unique(f, j) \wedge \neg\exists k : lookup(t, k) = j\Big) \vee \Big(\neg(j \in f) \wedge \exists! k : lookup(t, k) = j\Big)\Big) \wedge$

$\forall k : lookup(t, k) \neq undefined : cons(a, lookup(t, k)).key = k \wedge$

$\forall k : \neg isIn?(convert(\langle a, f, t\rangle), k) : put(convert(\langle a, f, t\rangle), k, i).shortcut = head(f)$

The Equality Redefinition

The equality redefinition expresses that the values of the array in its free positions (those found in the queue) are not relevant to the abstract value. This means that the same *store* is obtained if any free position of the array were modified. Also, it should express that two *stores* are equal when two tables, constructed in different ways, contain the same set of pairs. But this has been contemplated in the equations of the type table, and therefore there is no need to include it in the equality redefinition.

$$\forall j : 1 \leq j \leq n \wedge (j \in f) : convert(\langle a, f, t\rangle) = convert(\langle ass(a, j, val), f, t\rangle)$$

3.2 Deriving code for the operations

In this subsection, we present the derivation of the implementation a particular operation, *put*, as a case study of the whole derivation process[2]. In order to simplify the demonstrations we work under the hypothesis that n is big enough to assure that the queue is never empty. Later, at the final implementation, we have considered this possible case of error.

3.2.1 Derivation of *put*

According to [8] the operation *put* must be implemented by a function satisfying the following pre-post specification:

function impl_put(⟨a:array;f:queue;t:table⟩; k:key;i:information)

return⟨⟨a_1:array;f_1:queue; t_1:table⟩; sc:shorcut⟩

{Pre: Inv_Rep(⟨a,f,t⟩) }

P

{Post: Inv_Rep(⟨a_1,f_1,t_1⟩) ∧ convert(⟨a_1,f_1,t_1⟩)=put(convert(⟨a,f,t⟩),k,i).store ∧

sc=put(convert(⟨a,f,t⟩),k,i).shortcut}

return ⟨⟨a_1,f_1,t_1⟩,sc⟩

The derivation of this function is crucial for the verification process not only because of its difficulty but also due to the fact that we have to prove, previously, a serie of lemmas necessary also for the derivation of the rest of the functions.

The Lemmas regarding the tables

Lemma 1 $[lookup(assign(t,k,sc),k_1)=undefined] \Rightarrow \neg\ ig(k,k_1)$

Lemma 2 $[lookup(t,k)=undefined] \Rightarrow deassign(t,k)=t$

Lemma 3 $[lookup(t,k)=undefined] \Rightarrow deassign(assign(t,k,sc),k)=t$

Lemma 4 $[lookup(t,k)\neq undefined] \Rightarrow t\neq TABLE.create$

[2]See [11] for the complete derivation.

Lemma 5 $[lookup(t,k) \neq undefined] \Rightarrow t = assign(t_1, k, lookup(t,k))$

Lemma 6 $lookup(deassign(t,k),k) = undefined$

Lemma 7 $[t = assign(t_1, k_1, sc_1)] \Rightarrow t = assign(t_2, k_1, sc_1) \wedge lookup(t_2, k_1) = undefined$

Lemma 8 $isIn?(convert(\langle a,f,t \rangle),k) = (lookup(t,k) \neq undefined)$

The Lemmas regarding the queues

Lemma 9 $putFirst(dequeue(q), head(q)) = q$

Lemma 10 $head(putFirst(q,v)) = v$

Using these lemmas let us see the derivation of the function *put*.

The result of the operation will depend on whether the *key* to be stored is found in the *store* or not. Therefore, we consider an alternative to the design of the function. Since, from Lemma 8, the result of $isIn?(convert(\langle a,f,t \rangle),k)$ is equivalent to $lookup(t,k) \neq undefined$ we will put this last one as a condition of the alternative.

function impl_put(⟨a:array;f:queue;t:table⟩; k:key;i:information)

return⟨ ⟨a_1:array;f_1:queue;t_1:table⟩; sc:shortcut⟩

{Pre: Inv_Rep(⟨a,f,t⟩) }

if lookup(t,k)=undefined **then**

{A_1: Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined }

P_1

else

{A_2: Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)≠undefined}

P_2

end if

{Post: Inv_Rep(⟨a_1,f_1,t_1⟩) ∧ convert(⟨a_1,f_1,t_1⟩)=put(convert(⟨a,f,t⟩),k,i).store ∧

sc=put(convert(⟨a,f,t⟩),k,i).shortcut}

return ⟨⟨a_1,f_1,t_1⟩,sc⟩

Let's see what assertion we can deduce from A_1.

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined

⇒ (Lemma 8)

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined ∧ ¬ isIn?(convert(⟨a,f,t⟩),k)

⇒ (Definition of Inv_Rep)

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(f)

⇒ (Equality Redefinition and head(f) ∈ f)

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(f) ∧

convert(⟨a,f,t⟩)= convert(⟨ass(a,head(f),⟨k,i⟩),f,t⟩)

⇒ (The introduction of the operation put in both sides of the equality)

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(f) ∧

put(convert(⟨a,f,t⟩),k,i).store= put(convert(⟨ass(a,head(f),⟨k,i⟩),f,t⟩),k,i).store

⇒ (Lemma 3)

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(f) ∧

put(convert(⟨a,f,t⟩),k,i).store= put(convert(⟨ass(a,head(f),⟨k,i⟩),f,deassign(assign(t,k,head(f)),k) ⟩),k,i).store

⇒ (Lemma 9)

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(f) ∧

put(convert(⟨a,f,t⟩),k,i).store=

put(convert(⟨ass(a,head(f),⟨k,i⟩),putFirst(dequeue(f),head(f)),deassign(assign(t,k,head(f)),k)⟩),k,i).store

⇒ (Eq. of array cons(ass(A,i,v),i)=v)

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(f) ∧

put(convert(⟨a,f,t⟩),k,i).store= put(convert(⟨ass(a,head(f),⟨k,i⟩),putFirst(dequeue(f),head(f)),

deassign(assign(t,k,head(f)),k)⟩),k,cons(ass(a,head(f),⟨k,i⟩), head(f)).inf).store

⇒ (Abstraction Function)

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(f) ∧

put(convert(⟨a,f,t⟩),k,i).store= convert(⟨ass(a,head(f),⟨k,i⟩),dequeue(f), assign(t,k,head(f))⟩)

⇒ ([Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined] ⇒ Inv_Rep(⟨ass(a,head(f),⟨k,i⟩),dequeue(f), assign(t,k,head(f))⟩))

Inv_Rep(⟨ass(a,head(f),⟨k,i⟩),dequeue(f), assign(t,k,head(f)) ⟩) ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(f) ∧

put(convert(⟨a,f,t⟩),k,i).store= convert(⟨ass(a,head(f),⟨k,i⟩),dequeue(f), assign(t,k,head(f))⟩)

Therefore, from A_1 we can deduce this last assertion, denoted A_3.
Let's see what assertion we can deduce from A_2.

Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)≠undefined

⇒ (Lemma 5)

Inv_Rep(⟨a,f,t⟩) ∧ t=assign(t_1,k,lookup(t,k)))

⇒ (Abstraction Function)

Inv_Rep(⟨a,f,t⟩) ∧

convert(⟨a,f,t⟩)= put(convert(⟨a,putFirst(f,lookup(t,k)),deassign(t,k)⟩),k, cons(a,lookup(t,k)).inf).store

⇒ (Equality Redefinition and ∀k:k ∈ putFirst(f,k))

Inv_Rep(⟨a,f,t⟩) ∧

convert(⟨a,f,t⟩)= put(convert(⟨ass(a,lookup(t,k),⟨cons(a,lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(a,lookup(t,k)).inf).store

⇒ (The introduction of the operation put in both sides of the equality)

Inv_Rep(⟨a,f,t⟩) ∧

put(convert(⟨a,f,t⟩),k,i).store= put(put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(a,lookup(t,k)).inf).store,k,i).store ∧

put(convert(⟨a,f,t⟩),k,i).shortcut= put(put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(a,lookup(t,k)).inf).store,k,i).shortcut

⇒ (Eq. of store put(put(st,k,i_1).store,k,i_2).store= put(st,k,i_2).store, and

put(put(st,k,i_1).store,k,i_2).shortcut=put(st,k,i_1).shortcut)

Inv_Rep(⟨a,f,t⟩) ∧

put(convert(⟨a,f,t⟩),k,i).store= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,i).store ∧

put(convert(⟨a,f,t⟩),k,i).shortcut= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(a,lookup(t,k)).inf).shortcut

⇒ (Eq. of array cons(ass(A,i,v),i)=v)

Inv_Rep(⟨a,f,t⟩) ∧

put(convert(⟨a,f,t⟩),k,i).store= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(ass(a,lookup(t,k), ⟨cons(a,lookup(t,k)).key,i⟩),lookup(t,k)).inf).store ∧

put(convert(⟨a,f,t⟩),k,i).shortcut= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(a,lookup(t,k)).inf).shortcut

⇒ (Lemma 6)

Inv_Rep(⟨a,f,t⟩) ∧

put(convert(⟨a,f,t⟩),k,i).store= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(ass(a,lookup(t,k), ⟨cons(a,lookup(t,k)).key,i⟩),lookup(t,k)).inf).store ∧

put(convert(⟨a,f,t⟩),k,i).shortcut= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(a,lookup(t,k)).inf).shortcut ∧

lookup(deassign(t,k),k)=undefined

⇒ (Lemma 8)

Inv_Rep(⟨a,f,t⟩) ∧

put(convert(⟨a,f,t⟩),k,i).store= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),

putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(ass(a,lookup(t,k), ⟨cons(a,lookup(t,k)).key,i⟩),lookup(t,k)).inf).store ∧

put(convert(⟨a,f,t⟩),k,i).shortcut= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),
putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(a,lookup(t,k)).inf).shortcut ∧
¬ isIn?(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩), putFirst(f,lookup(t,k)),deassign(t,k)⟩),k)
⇒ ([Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)≠undefined] ⇒
Inv_Rep(⟨ass(a,lookup(t,k),⟨cons(a,lookup(t,k)) .key,i⟩), putFirst(f,lookup(t,k)),deassign(t,k)⟩))
Inv_Rep(⟨a,f,t⟩) ∧ put(convert(⟨a,f,t⟩),k,i).shortcut=head(putFirst(f, lookup(t,k)) ∧
put(convert(⟨a,f,t⟩),k,i).store= put(convert(⟨ass(a,lookup(t,k),⟨cons(a, lookup(t,k)).key,i⟩),
putFirst(f,lookup(t,k)),deassign(t,k)⟩),k,cons(ass(a,lookup(t,k), ⟨ cons(a,lookup(t,k)).key,i⟩),lookup(t,k)).inf).store
⇒ (Abstraction Function and Lemma 10)
Inv_Rep(⟨a,f,t⟩) ∧ put(convert(⟨a,f,t⟩),k,i).store= convert(⟨ass(a,lookup(t,k),⟨cons(a,lookup(t,k)).key, i⟩),f,t⟩)∧
put(convert(⟨a,f,t⟩),k,i).shortcut=lookup(t,k)
⇒ ([Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)≠undefined] ⇒ Inv_Rep(⟨ass(a,lookup(t,k), ⟨cons(a,lookup(t,k)).key,i⟩),f,t⟩))
Inv_Rep(⟨ass(a,lookup(t,k),⟨cons(a,lookup(t,k)).key,i ⟩),f,t⟩) ∧
put(convert(⟨a,f,t⟩),k,i).store= convert(⟨ass(a,lookup(t,k),⟨cons(a,lookup(t,k)).key, i⟩),f,t⟩) ∧
put(convert(⟨a,f,t⟩),k,i).shortcut=lookup(t,k)

Therefore, from A_2 we can deduce this last assertion, denoted A_4.

By introducing these new assertions we have:

function impl_put(⟨a:array;f:queue;t:table⟩; k:key;i:information)
return⟨ ⟨a_1:array;f_1:queue;t_1:table⟩; sc:shortcut⟩
{Pre: Inv_Rep(⟨a,f,t⟩) }
if lookup(t,k)=undefined **then**
{A_1: Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined}
{A_3}
P_1
else
{A_2: Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)≠undefined}
{A_4}
P_2
end if
{Post: Inv_Rep(⟨a_1,f_1,t_1⟩) ∧ convert(⟨a_1,f_1,t_1⟩)=put(convert(⟨a,f,t⟩),k,i).store ∧
sc=put(convert(⟨a,f,t⟩),k,i).shortcut}
return ⟨⟨a_1,f_1,t_1⟩,sc⟩

Having these two new assertions it is easy to identify, by means of the assignment rule, which expressions should be assigned to a_1, f_1, t_1, and *sc* in each case. By doing these assignments we obtain the following implementation of the *put* function:

function impl_put(⟨a:array;f:queue;t:table⟩; k:key;i:information)

return⟨ ⟨a_1:array;f_1:queue;t_1:table⟩; sc:shortcut⟩

{Pre: Inv_Rep(⟨a,f,t⟩) }

if lookup(t,k)=undefined **then**

{A_1: Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)=undefined}

{A_3}

a_1:= ass(a,head(f),⟨k,i⟩)

f_1:= dequeue(f)

t_1:= assign(t,k,head(f))

sc:= head(f)

else

{A_2: Inv_Rep(⟨a,f,t⟩) ∧ lookup(t,k)≠undefined}

{A_4}

a_1:= ass(a,lookup(t,k),⟨cons(a,lookup(t,k)).key,i⟩)

f_1:= f

t_1:= t

sc:= lookup(t,k)

end if

{Post: Inv_Rep(⟨a_1,f_1,t_1⟩) ∧ convert(⟨a_1,f_1,t_1⟩)=put(convert(⟨a,f,t⟩),k,i).store ∧

sc=put(convert(⟨a,f,t⟩),k,i).shortcut}

return ⟨⟨a_1,f_1,t_1⟩,sc⟩

3.3 Chosing implementations for the components of the representation

Once we have derived the code of the operations, we study two particular implementations of the new ADT[3]. In both of them we use the dynamic memory zone as an array and the queue of the system as that of the free shortcuts. These two implementations differ in the implementation of the table. In the first one, we have used a hashing table with chaining. Using a hashing table requires the parameter *KEY* to include also a hashing function. In the second implementation we have implemented the table with an AVL[4] [1].

We give next the cost of the operations of the ADT depending on the implementation used for table (see Table 1).

Table 1: The cost of the functions

FUNTIONS	COST	
	AVL	HASHING TABLE
create	$O(1)$	$O(r)$
put	$O(\log n)$	$O(1)$
getInfKey	$O(\log n)$	$O(1)$
getInfSho	$O(1)$	$O(1)$
getKey	$O(1)$	$O(1)$
isIn?	$O(\log n)$	$O(1)$
getShortcut	$O(\log n)$	$O(1)$
isEmpty?	$O(1)$	$O(1)$
remove	$O(\log n)$	$O(1)$
modify	$O(1)$	$O(1)$

As it can be seen, the cost of the operations *getKey*, *getInfSho* and *modify* is constant in both implementations because the table is not accessed, in this case. The cost of the remaining operations

[3]See [11] for more details.

[4]An AVL is a binary search tree where the diference between the height of its subtrees is less or equal than 1 and the subtrees are AVL in turn.

depends on the cost of the operations with the table. Thus, the operation *create* has a cost $O(1)$ if an AVL is used, because the cost of creating an AVL is constant; on the other hand, if the hashing table is used, the cost is $O(r)$, where r is the number of hashing values.

The operations *put*, *getInfKey*, *isIn?*, *getShortcut* and *remove* have cost $O(\log n)$, if an AVL is used, and this is due to the fact that the operations of consulting, assignment and removing in an AVL have cost logarithmic in the height of the tree [9, 13], while for the hashing table these operation has an average cost $O(1)$ [10, 7] therefore the operations *put*, *getInfKey*, *isIn?*, *getShortcut* and *remove* will have a cost that is $O(1)$ in average. We notice that, if the number of the elements is far superior to the number of hashing values or, if the hashing function is not good enough, the cost of this operations may well be linear.

4 An Example: The Tennis Ladder

In this section we present a simple example, taken from [1], of an application that requires the use of pointers for efficiency. We will show a modular solution, another with pointers and one using the new ADT *STORE* we have designed. Then, we will compare the results.

Aho, Hopcroft and Ullman [1] presented this example to justify the use of pointers to achieve efficiency. We treat again this example in order to see the effect of using the new ADT.

Suppose we wish to maintain a "tennis ladder", in which each player is on a unique "rung". New players are added to the bottom, that is, the highest-numbered rung. A player can challenge the player on the rung above, and if the player below wins the match, they trade rungs. We can represent this situation as an abstract data type, where the underlying model is a mapping from names (character strings) to rungs (integers 1, 2,...). The three operations we perform are

ADD(LAD,*name*) adds the named person at the highest-numbered rung.

CHALLENGE(LAD,*name*) is a function that returns the name of the person on rung $i-1$ if the named person is on rung i, $i > 1$.

EXCHANGE(LAD,i) swaps the names of the players on rungs i and $i-1$, $i > 1$.

4.1 Aho, Hopcroft & Ullman's Solutions to the Problem

The first solution uses an array $LADDER$, where $LADDER[i]$ is the name of the person on rung i. If we also keep a count of the number of players, we can add a player to the first unoccupied rung can

in some small constant number of steps.

The operation EXCHANGE is also easy, as we simply swap two elements of the array. However, CHALLENGE(LAD,$name$) requires that we examine the entire array in search of the name, which takes $O(n)$ time, if n is the number of players on the ladder.

On the other hand, as a second solution, we might consider a hash table to represent the mapping from names to rungs. Under the assumption that we can keep the number of buckets roughly proportional to the number of players, ADD takes $O(1)$ time on the average. Challenging takes $O(1)$ time on average to look up the given name, but $O(n)$ to find the name on the next lower-numbered rung, since the entire hash table may have to be searched. Exchanging requires $O(n)$ time to find the players on rungs i and $i-1$.

Suppose, however, that we combine the two structures. The cells of the hash table will contain pairs consisting of a name and a rung, while the array will have in $LADDER[i]$ a pointer to the cell for the player on rung i. In this way we can add a name by inserting into the hash table in $O(1)$ time on the average, and also placing a pointer to the newly created cell into the array $LADDER$ at the position marked by the cursor $nextrung$ (this is used to know the position the new player enters in). To challenge, we look up the name in the hash table, taking $O(1)$ time on the average, get the rung i for the given player, and follow the pointer in $LADDER[i-1]$ to the cell of the hash table for the player to be challenged. Consulting $LADDER[i-1]$ takes constant time in the worst case, and the lookup in the hash table takes $O(1)$ time on the average, so CHALLENGE is $O(1)$ in the average case.

EXCHANGE(LAD,i) takes $O(1)$ time to find the cells for the players on rungs i and $i-1$, swap the rung numbers in those cells, and swap the pointers to the two cells in LADDER. Thus EXCHANGE requires constant time even in the worst case. Clearly, this solution is best in terms of efficiency; but it has the problems mentioned in the introduction:

- The implementation of the table must be known.
- The function of inserting to the table must be modified, in order to obtain a pointer to the cell.
- We must assure that the functions of the table maintain the data in the same physical place.

These aspects do not fit well into the modular design, since they do not respect its main properties, like abstraction and reusability.

4.2 The solution with the ADT *STORE*

We start from the same idea. We maintain two structures: a *store* that corresponds to the hash table of the previous solution and the array *LADDER*. In the *store*, we will maintain the names of tennis players together with its rung, and in the array we will have the *shortcuts* to access to the name of tennis players.

Using this solution, we can add a name to the *store* by means of the *put* operation and then we assign the *shortcut* returned by this operation to the corresponding position into the array *LADDER*. Thus, ADD(*LAD*,*name*) takes $O(1)$ time to assign the *shortcut* to the corresponding position in the array and $O(1)$ time on the average, the operation *put*, supposing the *store* is implemented with a hash table or $O(\log n)$ time in the number n of tennis players, even in the worst case when the *store* is implemented by an AVL.

Therefore, ADD(*LAD*,*name*) takes $O(1)$ time on the average or $O(\log n)$ worst-case depending on the implementation used for the *store*.

For challenging we search the name in the *store*, obtaining its classification i; then, using the *shortcut* $LADDER[i-1]$, we access to the wanted player. Accessing to the data using the *shortcut* $LADDER[i-1]$ takes a constant time, in the worst case, and accessing the *store* by the name needs again $O(1)$ time on the average or $O(\log n)$ time in the worst case. Consequently, that is the time for CHALLENGE(*LAD*,*i*).

The operation EXCHANGE(*LAD*,*i*) needs time $O(1)$ to update the ladder of players i and $i-1$ in the *store* by using the *shortcuts* and interchanging the them in the *LADDER*.

To summarize, the efficiency using the ADT *STORE* is the same as the best of those previously proposed, but without the drawbacks arising in this case. If we use the ADT *STORE* implemented by a hash table, the cost of all the operations is exactly the same as in the example using the pointers. If we use the ADT *STORE* implemented by an AVL the cost of some operations grows up to $O(\log n)$ instead of $O(1)$ on the average, but this has nothing to do with the use of *shortcuts* (it comes from the AVL itself).

By using the *STORE* we do not need to know the implementation of the table, neither to modify the insertion operation, nor to assume that the data occupies the same place in the structure. Concluding, we obtain a solution that is completely modular without penalising the efficiency (see Fig. 4). Last but not least, at this point all the ADT's used in the solution offer a full equational specification,

which allows one to either formally derive or, a posterior, formally verify, if necessary, the programs implementing the LADDER operations [3].

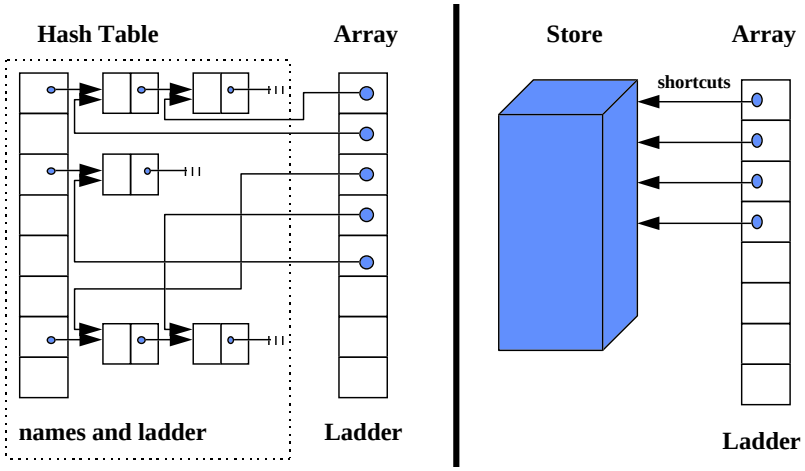

Figure 4: Solutions to the ladder problem without *shortcuts* (left side of picture) and using *shortcuts* (right side).

5 Codifications

The ADT *STORE* has been codified in the following three programming languages[5]: ADA [2], Modula-2 [14] and C++ [12]. The details are found in [11], here we describe only some of the most relevants aspects.

Modula-2 is not able to codify a modular design; it does not offer a mechanism for creating a generic or parameterized ADT. In order to overcome the lack of this mechanism, we have constructed a definition module where the parameters are defined. This solution has the inconvenience that it does not permit more than one instance of the ADT. Another problem arises while encapsulating the new data type. Since the encapsulating needs the data type be implemented by pointers another level of indirection in the operations is produced.

ADA, in its whole, offers good mechanisms to program by using the modular methodology. In

[5]These three codifications are available at http://www-lsi.upc.es/~jmarco/

particular, it gives a good mechanism for encapsulating and genericity. However, notice that since the representation must be in the package definition it is visible to the user, although it is not accessible. The fact that ADA uses garbage-collection technique enables us to have a greater control on the errors of the operations with the shortcuts.

C++ offers many advantages coming from the fact of being a Object Oriented language. Since it is possible to define virtual classes, it allows us to do a unique implementation of the new ADT with an additional parameter which indicates the implementation of the table to be chosen. The constructive and destructive methods of the classes enable us to have our own garbage-collection on the shortcuts.

6 Conclusions

We have designed and implemented a new abstract data type (ADT), that we call *STORE*. Our motivation was to obtain an abstract mechanism which provides direct access to the data without losing the modularity (in fact, it guarantees full modularity), and also obtaining the same efficiency as with pointers. The ADT *STORE* offers such a mechanism, referred to as *shortcut*, which behaves naturally as pointers. The difference between both of them is that, using shortcuts, the access to the data is done without knowing how is stored the data in the structure, and therefore there is no loss of modularity at all.

Also, we have done two implementations of the new ADT, based in a certain representation obtained by usual methods of modular programming. It is interesting to notice that, given the complete modularity of the new ADT, our implementations can be used interchangeably in the same contexts.

The future research will consist in finding a method for an automatic definition of shortcuts within the paradigm of Object Oriented Programming. More precisely, the idea is to define a new class, i.e. a *shortcut*, that is independent from the data structure used to store the data. This, in turn, would have two aspects: the *shortcut* will depend only on the data and the inheritance mechanism will be used. Other lines of interest for further research would be the study of more adequate formal institutions for our specification (e. g. behavioural or loose semantics) and finding a more realistic model for the dynamic memory in order to rule out the supposition that it behaves like an array.

Acknowledgment

We want to thank J.L. Bálcazar, for his direction of the project from which originated the main idea of this work, and also for his suggestions and for the amount of time he spent in reviewing carefully this paper.

References

[1] **Aho, A.V.; Hopcroft, J.E. and Ullman, J.D.** *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] **Barnes, J.G.P.** *Programming in ADA.* Addison-Wesley, 1984.

[3] **Dijkstra, W.** *Discipline of Programming.* Prentice-Hall, 1976.

[4] **Ehrig, H. and Mahr, B.** *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[5] **Ehrig, H. and Mahr, B.** *Fundamentals of Algebraic Specification 2.* Springer-Verlag, 1990.

[6] **Franch, X.** *Estructuras de datos: Especificación, Diseño e Implementación.* Edicions UPC, colección Politext n. 30, 1994.

[7] **Gonnet, G.H. and Baeza-Yates, R.** *Handbook of Algorithms and Data Structures.* Addison-Wesley, 2nd. edition, 1991.

[8] **Hoare, C.A.R.** *Proofs of Correctness of Data Representation.* Acta Informatica, 1972.

[9] **Horowitz, E. and Sahni, S.** *Fundamentals of Data Structures in Pascal.* Computer Science Press, 4th. edition, 1994.

[10] **Knuth, D.E.** *The Art of Computer Programming.* Vol. 3, Addison-Wesley, 1973.

[11] **Marco, J.** *Dreceres: "pointers" abstractes.* Projecte Final de Carrera, Facultat d'Informàtica de Barcelona, 1996. Directed by J.L. Balcázar (written in Catalan).

[12] **Robert, S.** *C++ Components and Algorithms.* Prentice-Hall, 1992.

[13] **Wirth, N.** *Algorithms and Data Structures.* Prentice-Hall, 1986.

[14] **Wirth, N.** *Programming in Modula-2.* Springer-Verlag, 3rd. edition, 1988.