The Abstract Data Type List

Definition

The **ADT List** is a linear sequence of an arbitrary number of items, together with the following access procedures:

- **createList()**
  
  // post: Create an empty list

- **add(index, item)**
  
  // post: Insert item at position index of a list
  // if 1 <= index <= size()+1. If index <= size(), items
  // at position index onwards are shifted one position
  // to the right

- **isEmpty()**
  
  // post: Determine if a list is empty

- **get(index)**
  
  // post: Returns item at position index of
  // a list if 1 <= index <= size()

- **remove(index)**
  
  // post: Remove item at position index of a list
  // if 1 <= index <= size(). Items at position
  // index+1 onwards are shifted one position to the left

- **size()**
  
  // post: Returns number of items in a list

The items in the list must be of the same type

To elaborate on the notion of an ADT, we consider the notion of a list. Think about a list that you might encounter, such as a list of important dates, a list of addresses or a grocery list.

What are the main features of a list? Assuming that it is a neat one-column list, the items on the list appear in a sequence. The list has one first element (the head) and one last element. Except for the first and the last, each element has one unique predecessor and one unique successor. So we can say that a list is a **linear sequence of a number of items**. The items in a list have also to be of the same type.

What are the operations that we can perform on the items of a list? First of all, we might ask ourselves, where do we put new items? To keep the list neat, we could just as well add the new item at the beginning of the list, or add it at a certain position (perhaps to keep the list sorted, if it is already). Other operations can be counting the items to determine the length of the list, removing an item from the list, checking whether the list is empty, looking at (retrieving) an item. These operations are examples of the access procedures which define what an Abstract Data Type list is.

In this slide, we define the header and post-conditions of the access procedures, without including the cases for exceptions (see slide 6 for a full specification of the access procedures). Given these procedures, we can say that “the ADT list is a linear sequence of an arbitrary number of items together with these access procedures”. Note that the elements of a list do not need to be in any specific order: the fact that a list is a linear sequence of items means simply that a list is a collection of items that we can reference by position number.
An example

Consider the list:  *milk, eggs, butter, apples, bread, chicken*

1. How can we construct this list?
   - Create an empty list
   - Use series of *add* operations
   
   ```
   myList.createList()
   myList.add(1, milk)
   myList.add(2, eggs)
   myList.add(3, butter)
   . . .
   ```

   *milk, eggs, butter, apples, bread, chicken*

2. How do we insert a new item into a certain position?
   - Use *add* operation
   
   ```
   myList.add(4, nuts)
   ```

   *milk, eggs, butter, nuts, apples, bread, chicken*

3. How do we delete an item from a certain position?
   - Use *remove* operation
   
   ```
   myList.remove(5)
   ```

   *milk, eggs, butter, nuts, bread, chicken*

To get an idea of how the operations work, let’s consider the grocery list

*milk, eggs, butter, apples, bread, chicken*

where *milk* is the first item on the list and so on. To begin, we can consider how we would construct such a list, using the operations given in the previous slide. One way is first to create an empty list `myList` and then use series of *add* operations to append successively the items to the list, as described in the right hand side of the slide. Note that `myList.O` indicates that the operation `O` applies to the list `myList`.

The operation *add* can insert a new item also into any position of the list, not just at the front or at the end. According to the specification of this operation given in the previous slide, if a new item is inserted into position `i`, then the position of each item that was at position `i` or greater is increased by 1. So, for example, if we start with the list given at the top of this slide, and we perform the operation `myList.add(4, nuts)`, then the list `myList` becomes *milk, eggs, butter, nuts, apples, bread, chicken*. So all items that had position numbers greater than or equal to 4 before the insertion, have position numbers increased by 1 after the insertion. This is shown in the right hand side of the slide.

Similarly, the deletion operation (*remove*) specifies that if an item is deleted from position `i`, then the position of each item that was at a position greater than `i` is decreased by 1. For example, if `myList` is *milk, eggs, butter, nuts, apples, bread, chicken*, and we perform the operation `myList.remove(5)`, then the list `myList` becomes *milk, eggs, butter, nuts, bread, chicken*. So all items that had position numbers greater than 5 before the deletion have their position numbers decreased by 1 after the deletion.

These examples show that an ADT can specify the effects of its operations without having to indicate how to store the data. The definition of the operations tells you what you can do to the list, and not how to perform the operations.
The ADT List Slide Number 3

Course: Programming II - Abstract Data Types

Access Procedures

Can be grouped into three broad categories:

**Constructor operations** to construct Lists

i. createList to create an empty List
ii. add operation to add data to a List

**Predicate operations** to test Lists

i. isEmpty to test whether the List is empty

**Selector operations** to select items of a List

i. get to return one item at a particular position.

ii. remove operation to delete a data item from a List

With these procedures we can manipulate lists without knowing anything about how they are implemented, and independently of the item type.

It seems that the list operations seen in the previous slide, with the exception of size(), fall into three broad categories: the operations for constructing a list, which also include the add operation; the operations for testing a list, and the operations for selecting a part of the list, which include the remove operation.

Once these procedures have been specified, as for instance we have done in slide 1, then the behavior of the ADT List has been specified, and application programs can be designed to access and manipulate the ADT’s data. Any such program will depend solely on the behavior of the operations without regard for their implementation.
Using a List: an example

A method `displayList` that displays the items in a list:

The pseudocode:

```java
displayList(aList)
  // post: displays the items in the list aList;
  for (index =1;  through aList.size())
    { dataItem = aList.get(index);
      Display dataItem;
    }
```

Once the operations of an ADT have been satisfactorily specified, applications can be designed that access and manipulate the ADT’s data, by using only the access procedures without regard for its implementation. A little example is considered here: suppose that we want to display the items in a list. The “wall” between the implementation of the ADT and the rest of the program prevents the program from knowing how the list is being stored, i.e. which data structure has been used to implement a List. The program can, however, still access the items of a list by means of the access procedures. In this case the method `displayList`, can use the operation `aList.get(index)` to access the item at position index in the list, and the operation `aList.get(index)` returns such item to the application program. This is diagrammatically shown in the following picture:
The ADT List Slide Number 5

Course: Programming II - Abstract Data Types

Axioms of an ADT

**Syntax:** the definition of Access Procedures (as in slide 1);

**Semantics:** axioms defining the behaviour of the Access Procedures:

1. (aList.createList()).size() = 0
2. (aList.add(i,item)).size() = aList.size() + 1
3. (aList.remove(i)).size() = aList.size() - 1
4. (aList.createList()).isEmpty() = TRUE
5. (aList.add(i,item)).isEmpty() = FALSE
6. (aList.createList()).remove(i) = ERROR
7. (aList.add(i,item)).remove(i) = aList
8. (aList.createList()).get(i) = ERROR
9. (aList.add(i,item)).get(i) = item
10. aList.get(i) = (aList.add(i,item)).get(i+1)
11. aList.get(i+1) = (aList.remove(i)).get(i)

The implementation of the ADT List must

1. include the defined syntax (access procedures with the parameters and results as defined).
2. Satisfy the axioms in the way they operate.

The specification for the ADT operations given in slide 1 is rather informal and relies on the reader’s intuition. However, some abstract data types are much more complex and less intuitive than a list. Therefore to specify an ADT we can make use of a more rigorous method for defining the behaviour of each access procedure. This consists of defining a set of mathematical rules, called AXIOMS, that precisely specify the behaviour of each access procedure, for all lists.

For example, the statement “A newly created list is empty” is an axiom since it is true for all newly created lists. We can state this axiom succinctly in terms of the operations of the ADT list as follows: (aList.createList()).isEmpty() is true, which means that the list aList is empty (refer to slide 2 for an explanation of the notation used here). The full set of axioms for the ADT list is given in this slide. Note that the brackets denote the list obtained after the execution of the access procedure specified within the brackets.

The implementation of the ADT List must then take into account both the syntactic and semantic specification of the ADT. The implementation will include the operations as they are syntactically defined, i.e. with their respective parameters and result types. It will also implement the operations so that their behaviours satisfy the axioms for any possible list under consideration.
A set of axioms is not complete if the behaviour of the ADT is not defined in all possible situations in which ADT operations can be used. We have given here an example. Possible solutions are (i) to define special pre- and post-conditions for each operation, which exclude the cases not covered by the axioms; (ii) define the implementation of the ADT operations in such a way that these cases are covered and the behaviour of the ADT defined. An example of this second approach is given in this slide.

Similar examples of incompleteness can be identified for the operations “remove” and “get”. Their new specification is given below:

remove(index)
//post: Removes the item at position index of a list if 1<=index<= size().
// If index < size(), items at position index+1 onwards are shifted one position to the left
// Throws an exception when index is out of range, or if list is empty.

get(index)
//post: Returns the item at position index of a list, if 1<=index<=size(). The list is left unchanged
// by this operation. Throws an exception if index is out of range.

The post-conditions given here for the access procedures add, remove and get provide the full specifications of these procedures including also exceptions.
The axioms, together with the specification of the operations, should define the outcome of all permissible operations on data types. So, for instance, they should be able to define what a list looks like after a concatenation of operations of the ADT list. In the example given above, if we are performing the above sequence of two “add” operations over a given aList, are we able to show that the element at position index1 is item1 and that the element at position index2 is item2?

Let’s start from \(((\text{aList.add}(1,\text{item1})).\text{add}(1,\text{item2})).\text{get}(1)\). We can assume \text{tempList} to represent \text{aList.add}(1,\text{item1}) and write the given sequence of operations as

\[(\text{tempList.add}(1,\text{item2}))).\text{get}(1) = \text{item2}\] by axiom 9.

Now consider \(((\text{aList.add}(1,\text{item1})).\text{add}(1,\text{item2})).\text{get}(2)\). Again this can be written as

\[
(\text{tempList.add}(1,\text{item2}))).\text{get}(2) = \\
\text{tempList.get}(1) = \\
(\text{aList.add}(1,\text{item1})).\text{get}(1) = \\
\text{item1}
\]

by axiom 10

by definition of \text{tempList}

by axiom 9
Completeness (continued)

Given the new specifications of the operations, the behaviour of the access procedures is defined for each type of given list:

- Size( ) by axioms 1, 2, 3
- isEmpty( ) by axioms 4, 5
- Remove(i) by axioms 6, 7
- Get(i) by axioms 8, 9, 10, 11

Hence, our set of axioms is complete.

Consistency of the ADT axioms

If any two axioms are contradictory then the specification is said to be inconsistent.

In the general case, it is not possible to determine formally whether any set of axioms is consistent (i.e. it is not decidable)

In practice, for small sets of axioms, inspection may generate confidence in their consistency.
Access procedures can be used to construct higher level procedures for manipulating the ADTs. In the case of an ADT List, an example of an higher level procedure is “Replace”, that replaces the item in position $i$ with a new item. If the $ith$ item exists, Replace deletes the item and inserts the new item at position $i$, as shown in this slide.
Higher level procedures are written using only the access procedures to manipulate the ADT.

This means that higher level procedures, and other application procedures which use them, are independent of the way the ADT is implemented.

There are many possible implementations for a single ADT. We shall study examples of two main types of implementations:
- static
- dynamic
Principles for implementing ADTs

ADT operations as “walls” between programs and data structures

- ADT operations are declared using an *interface* in Java.
- Data structure is defined using a *class* that implements the interface.
- Variables for storing ADT items are *private data field* of this class.
- ADT operations are implemented as *public methods* of this class.
- Auxiliary methods are *private methods* of this class.

At the beginning of the lecture, we have said that ADT operations are like “walls” between any client program that uses an ADT and the actual data structure that implements the ADT. This general principle is reflected in the design of an ADT Java implementation. To implement an ADT, we can define an interface in Java that provides just the declaration of the operations or access procedures of our ADT, and construct a class for implementing the data structure. This class may use one or more other classes needed to define the type of the data.

Each ADT operation is implemented as a *public method* of this class. For instance, let us assume for the sake of argument, that we are using an array as data structure of our ADT. The operation of the ADT will require access to the array items. So the array can be declared a data field of the ADT class. However, in order to support the principle of building a wall around the data structure, we need to keep it hidden from the external program. This is done by declaring the array to be *private* to the class that implements the ADT.

In addition, the implementation of the methods for the access procedures may require auxiliary methods. These have to be declared as *private methods* of the class. An example is the private method “translate(position)” given in the next lecture.
The ADT operations can be declared using a Java interface. In this slide we have defined an interface for the ADT list, called ListInterface. We said at the beginning of this lecture that all items in a list have to be of the same type. The notion of a list and its operations has to be independent of the type of the items that are stored in the list. We have therefore used class `Object` as the type for the list’s elements. This is possible because every class in Java is ultimately derived from the class `Object` through inheritance. This means that any class created in Java can be used as a type for the items in the list. This will appear clear when we see the class that implements the array-based data structure for our ADT list.

**Note:** In this interface we haven’t included the ADT operation `createList()`. This is because we are going to implement it as constructor for the class that implements the ADT list.

**Note:** The interface `ListInterface` includes also the two methods given in the next slide, which define the list operations “add” and “remove” respectively.
public void add(int index, Object item) throws ListIndexOutOfBoundsException, ListException;  
//Pre: index indicates the position at which the item should be inserted in the list  
//Post: If insertion is successful, item is at position index, and other items are  
//       renumbered accordingly.  
//Throws: ListIndexOutOfBoundsException when index <1 or index > size()+1  
//Throws: ListException if item cannot be placed on the list.

public void remove(int index)  
//Pre: index indicates the position in the list of the item to be removed  
//Post: if 1<= index <= size(), the item at position index is deleted, and other items  
//       are renumbered accordingly.  
//Throws: ListIndexOutOfBoundsException if index <1 or index > size().

}  // end of ListInterface
Some of the list operations have as parameter an index value which allows reference to the data item at a particular position in the list. In the specifications given in the first slide of this lecture, we haven’t considered run time exceptions. The full specifications for the ADT operations add, remove and get are as follows:

add(index, item)
//post: Insert item at position index of a list if 1<=index<= size()+1.
//If index <= size(), items at position index onwards are shifted one position
//to the right. Throws exception when item is out of range, or list full.

get(index)
//Returns item at position index of a list, if 1<=index<=size(). The list is left unchanged.
//Throw exception when index is out of range

remove(index)
//Remove item at position index of a list if 1<=index<= size().
//If index < size( ),Items at position index onwards are shifted one position to the left.
//Throws exception when item is out of range, or list is empty.

Each of the above three operations might be provided an index value that is out of range. In this case the operation has to throw an exception. We have defined in this slide a class that implements the out-of-range index exception for lists extending the more general IndexOutOfBoundException given by the Java API.

The second exception here, is useful mainly in the case of static implementation of a list, defined in the next lecture. A static implementation, uses an array as the underlying data structure, and therefore we have to guess a priori a maximum size of the list for the Java runtime environment to reserve a specific number of memory references for the elements of the array. So we also need a list exception occurring when the array storing the list becomes full. We have defined this exception with the class ListException, which extends the general RuntimeException of Java.
A static implementation:  
- uses an array of a specific maximum length, and all storage is allocated before run-time.  
- orders the items in the array with the index related to the position of the item in the list.

We need:  
- a variable to keep track of array elements assigned to the list, i.e. the current number of items in the list, or current size of the list.  
- a variable to record the maximum length of the array, and therefore of the list

Data Structure:  
```java
private final int MAX_LIST = 100; //max length of list
private int items[MAX_LIST]; //array of list items
private int numItems; //current size of list
```

A static implementation of a list is an array-based implementation. This is a natural choice since both an array and a list identify their items by numbers. (However, the ADT list has operations such as `removeAll` that an array does not.) We call this array “items”. But, how much of the array will the list occupy? Possibly all of it, but probably not. Therefore, we need to keep track of the array elements that are assigned to the list and those that are available for use. We define MAX_LIST to be a pre-fixed value representing the physical size of the array. This value gives the maximum length of the array and thus the maximum length of our list. In addition, we need to keep track of the current number of items in the list, that is the list’s logical size. We do this by using a variable called `numItems`. The value of numItems gives also the current number of elements of the array assigned to the list, and at the same time the index of the next available space in the array, since array indices start from 0.

We said above that both arrays and lists identify their items by numbers (i.e. the position in the list and the index in the array, respectively). Note, however, that whereas the positions of items in a list starts from 1 (the first item in a list is at position 1), in the array the index of the first item is 0. So, we can think of storing the list’s kth item in items[k-1].

In the next lecture we’ll show a private method “translate(index)” for translating the position of a item in the list into its respective array index value.
Array-based implementation of Lists

This slide gives a diagrammatical representation of an array-based implementation of the ADT list in terms of storing method, variable declarations and their relationships. We are considering here an example of a list of integers. But in general the type of the element of a list can be any object or a basic type of Java.
Example of Operations

- **Add 44 at position 3 in the list - add(3,44)**

  Array indexes: 0 1 2 3 4 k
  New item: numItems

  items: 12 3 44 19 90 ........... 5 10 ? ........ ?
  k+1
  MAX_LIST-1

- **Delete item at position 4 from the list - remove(4)**

  Array indexes: 0 1 2 3 k-1
  numItems: MAX_LIST-1

  items: 12 3 44 90 ........... 5 10 ? ? ........ ?
  k
  MAX_LIST

To insert a new item at a given position in the array, we must shift to the right the items from this position on, and insert the new item in the newly created opening. This is illustrated diagrammatically in the first array picture above.

Let’s see instead how to delete an item at a particular position in the list. We could in principle just blank it out, but this strategy leads to gaps in the array. An array that is full of gaps will give the following problems:

1. numItems – 1 is no longer the index of the last item in the array, since numItems will appropriately be decreased because of the delete operation, but the index of the last element used for the list will not necessarily be decreased. We would therefore need to use an additional variable to keep track of the last position used in the array.

2. Because the items are spread out, the method get might have to look at every cell of the array.

3. When items[MAX_LIST-1] is occupied, the list could appear full, even when fewer than MAX_LIST items are present.

Thus, what we need to do is to shift the elements of the array to fill the gap left by the deleted item, as shown in the second diagram above.
In this lecture we have considered in detail the definition of an ADT list. We have seen that to define an ADT we need to define a set of access procedures which will be the only procedures to have access to and manipulate the data structure that implements the ADT. We have seen how to define an ADT in terms of specification of its access procedures using a pseudocode and/or in terms of a set of axioms. These axioms define the behaviour of the access procedures formally and are independent of the method of implementation of the ADT. We have also briefly introduced the concepts of consistency and completeness of the axioms that specify an ADT and we also illustrated how they can be used to define the output in terms of the manipulated ADT after the execution of any given sequence of ADT operations, without having to look at the ADT implementation.

In the next lecture we will see in detail examples of static implementation of the access procedures for a list, and the dynamic implementation of lists.