Lab: Smart Array Design

AP Computer Science with Data Structures

Background: Arrays

We have looked at arrays of data and we have discovered a number of advantages as well as disadvantages to using arrays. In your notebook, discuss the advantages and disadvantages of arrays. You must include in your analysis the Big-O running time of the following operations:

- getting a data item from the array
- modifying a data item in an array
- adding data to an existing array
- removing a data item from an array

You must justify your run time estimates. Merely stating O(whatever) will earn you O (zero) points as will advantages/disadvantages that have no justification.

Background: Smart Arrays

In your notebook, write down the capabilities you think a Smart Array object should have. You must also specify the desired Big-O runtime of each capability you specify as well as your ideas on how you might achieve your goal.

The object you create will be called `MyArrayList`. It will implement the `MyList` interface which will be given to you. The methods specified in `MyList` and what they do is given here for your reference. Your `MyArrayList` must implement these methods as well as any other methods required to complete the capabilities you outlined above.

`Interface MyList<E>`

- `int size()` //returns the current `size` of the list
- `boolean add(E obj)` //appends `obj` to the end of the list; //returns true
- `void add(int index, E obj)` //inserts `obj` at position //`index`(0≤`index`≤`size`), moving elements //at position `index` and higher to the //right (adds one to their index) and //adjusts `size`
- `E get(int index)` // returns the element at position `index`
- `E set(int index, E obj)` //replaces the element at position `index` with `obj;` //returns the element formerly at the specified position
- `E remove(int index)` //removes element from position `index`, moving //elements at position `index+1` and higher to the left //(subtracts one from their index) and adjusts the size; //returns the element formerly at the specified position.
- `Iterator<E> iterator()`
- `MyListIterator<E> listIterator()`

We will look at iterators in another lab. For now, have the last two methods return the value `null.` This will allow your `MyArrayList` class to fully implement the interface. You need not provide any specifications for these two `Iterator` methods.

Background: Generics

Generics were added to Java in order to help maintain homogenous data structures. In order to generate an Abstract Data Type (ADT), such as a smart array, without generics you were required to store everything as type `Object`. This makes sense because `Object` is the super class of every class. As a result, code using the ADT was filled with type casts as items were taken from the ADT.

Generics help solve this problem by allowing us to specify that the ADT will contain some arbitrary type of data. When the ADT is instantiated, the type is specified and Java will ensure that only items of the specified type are put into the ADT.

A study of how to write generic ADT's would take us to far afield. Outlined here are simplified instructions for creating generic ADT's that will serve our needs.

- The placeholder for an unknown type will usually be `E`. The only exception is when we create the `Map` ADT.
- We will define a generic class as `public class MyGenericClass<E>`
- The type `E` can be used just as any other type within your ADT code. It can be a parameter type, a return type, or a local variable type.

- Because the actual type is unknown, you cannot declare an array of type `E`. If you use an internal array, it must be of type `Object` and you must do the necessary type casting whenever you return or use an element from the array.
- You may not use wild cards in your generic classes.

Exercise: `MyArrayList` Design Document

Create a design document that specifies the underlying data structure(s), instance fields, methods and their performance. You must justify your design – it is not sufficient, for example, to say that a method runs in constant time, you must provide enough information so that someone else could successfully implement a constant time solution. All of your design decisions must be communicated and justified in this document. A successful design document will allow anyone with a basic knowledge of Java to code your design. You must also specify a test plan that will verify the performance of your design.

Turn in your final design document to Athena2 and turn in one printed copy of your design document to your teacher.

Exercise: `MyArrayList` **Test Plan**

Create a test plan that specifies how you will verify that your `MyArrayList` class meets all of its specifications and works as intended. Your test plan should be done in your notebook and must be made available on demand. It must be completed prior to beginning the next lab.

Your test plan must be specific. Stating only that you will test the `add` method is not sufficient. You must outline exactly how you will test it, the required data, and the expected results. Your test plan must test all boundary cases, viz. adding to an empty list, removing an item from a list that has only one element, setting the first and last values, etc.

Note that often the specifics of a design and therefore the test plan will change as the design evolves. This is normal, and when changes are made to the design, they must be documented and the test plan must be updated.

Appendix: Design Document Format

Here is the format for a design document that specifies a single object (class). For a large project, you will need to document several objects and their interactions. Large project design documents will use this format for individual objects and include a document that describes the entire project.

As you complete the design document, keep in mind that your objective is to create a document that would allow someone else to implement your design without any other help. Your document will usually be one to two pages in length, depending on the number of services your object makes available.

Structure of the Design Document

<Name of Object> Design

Designer Name: ____________________ Period: ____________

Description

Describe what this object does. Tell the reader why they would want to use your object. This is usually a single, succinct paragraph. Spelling and grammar count!

Services

Describe the services (public methods) that this object exposes. Here you need to specify run time requirements, pre and post conditions you have identified in the process of the design, and any other considerations such as state variables affected. Some of the method documentation is likely to come directly from this section.

Internal Data Structures and State

This section describes any internal data structures such as arrays or other objects. You must describe their purpose, how they are initialized and how they are used. Additionally, any state variables needed by your object are described in this section.

Example Design Document

JavaTown LinkedList Design

Description

The `LinkedList` class defines a singly linked list of objects, with each object contained in a `ListNode` object. The `LinkedList` object maintains a pointer to the first element in the linked list. The size of the list is maintained as a state variable so that the size can be queried in O(1) time. All operations, with the exception of initialization and getting the size of the list are implemented recursively.

Services

The following services are available:

`LinkedList()` – constructs a `LinkedList` object and initializes the internal data structures and state. Upon construction, the size of the `LinkedList` is zero. No `ListNOde` objects are created.

`int size()` – returns the size of this list in O(1) time.

`void add(Object)` – adds a new `ListNode` to the beginning of the linked list and sets the data value of the added `ListNode` to the `Object` passed as a parameter in O(1) time. The size of the linked list is incremented by 1.

`Object remove(int index)` – removes the `ListNode` at the position in the linked list specified by the parameter. Note that 0 <= `index` < `size()`. The value contained within the removed `ListNode` is returned and the size of this `LinkedList` is decremented by one. The `remove` operation is O(n) where n is the list size.

`Object get(int index)` – returns the value contained within the `ListNode` at position `index`; 0<= `index` <= `size()`. The `get` service runs in O(n) time where n is the size of the list.

`void set(int index, Object value)` – sets the value in the `ListNode` at position `index`; 0 <= `index` < `size()`. The `set` service runs in O(n) time where n is the size of the list.

Internal Data Structures and State

The `LinkedList` class contains a pointer to a `ListNode` object. The linked list consists of a sequence of `ListNode` objects linked via pointers internal to the `ListNode`.

An `int` state variable is maintained in order to make the `size()` method run in O(1) time.

Example Test Plan – JavaTown LinkedList

The `LinkedList` class provides the following services:

> Default constructor `LinkedList()`
> The `size` method
> The `add(obj)` method
> The `remove(index)` method
> The `get(index)` method
> The `set(index, value)` method

Refer to the `LinkedList` design document for a complete description of these methods.

Testing the constructor

The constructor is tested using the JavaTown environment. A `LinkedList` object is created within the JavaTown environment and the resulting object is examined to ensure that the state variables are created and initialized. No additional data is required.

The expected result is the creation of a `LinkedList` object with two instance fields: a `list` field initialized to null and a `size` field initialized to zero. No other variables or data structures are created.

Testing the `add` method

Using the constructed `LinkedList` object, a list will be built by sending the add message to the `LinkedList` object. A list will be created using only numbers, a second list will be created using only `String` objects and a third list will be created using a mix of numbers and `String` objects. In each case, the operation of the `add` method will be observed using the JavaTown environment to verify that the new data is added to the front of the list. This test adds to an empty list and to a list containing one or more values, thus verifying the two possible cases. As items are added to the list, the `size` instance field will be examined to verify that it increments each time a value is added.

The `add` method is required to run in O(1) time and this will be verified by analysis using the runtime analysis methods developed in class.

Testing the `remove(index)` method

The `remove` method returns the object that is removed from the list and the precondition for `remove` specifies that the list must contain at least one element, and that the index specified is less than the size of the list. The following tests will be performed in order to verify the correctness of the `remove` method:

1. Calling `remove` on a list of size 1. Using the JavaTown environment, this test will verify that the correct value is returned and that the list becomes null and that the size instance field becomes 0.
2. Calling `remove` on a list of size 3 and removing the last value. Using the JavaTown environment, this test will verify that the last node is removed, the correct value is returned and that the size instance field becomes 2.
3. Calling `remove` on a list of size 3 and removing the first value. Using the JavaTown environment, this test will verify that the first node is removed, the first value returned and that the size instance field becomes 2.
4. Calling `remove` on a list of size 3 and removing the middle value. Using the JavaTown environment, this test will verify that the middle node is removed, the middle value returned and that the size instance field becomes 2.

The `remove` method is specified to run in O(n) time. The runtime performance will be verified by analysis using the analysis techniques developed in class.

Testing the `size` method

The `size` method is specified to run in constant time. An analysis of the algorithm will be presented proving the O(1) run time using the analysis procedures developed in class.
The correctness of the `size` method will be verified by sending the `size` method during each of the above `add` and `remove` tests and verifying that `size` returns the correct value and that the value returned matches the `size` instance field.

Testing the `set` method

The `set` method changes the value of the node at the specified index. Correct operation of the `set` method will be verified by the following tests:

1. A `LinkedList` containing 3 elements will be constructed.
2. Three `set` messages will be sent to the list specifying each node in turn and with a different data value. Using the JavaTown environment, the resulting values in each node will be verified. It is important to use a different value for each `set` message to ensure that only the desired node is affected.

The `set` method is specified to run in O(n) time. This will be verified by analysis using the techniques developed in class.

Testing the `get` method

The `set` method changes the value of the node at the specified index. Correct operation of the `set` method will be verified by the following tests:

1. A `LinkedList` containing 3 different elements will be constructed.
2. Three `get` messages will be sent to the list specifying each node in turn. Using the JavaTown environment, the resulting return values will be verified. It is important to have a different value for each node to ensure that correct values are returned.

The `get` method is specified to run in O(n) time. This will be verified by analysis using the techniques developed in class.