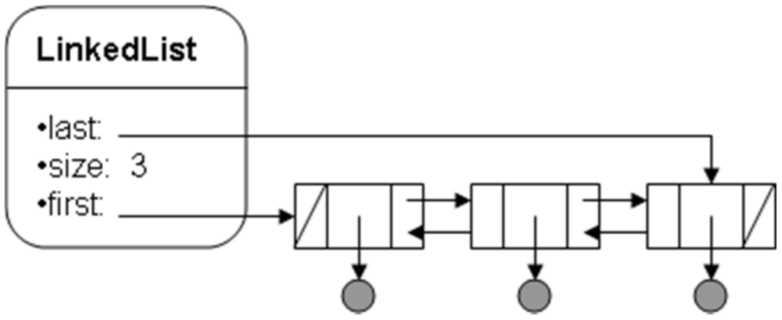
Lab: Linked List Design

AP Computer Science with Data Structures

Background: LinkedList

We used JavaTown to implement and study the properties of a singly-linked list consisting of `ListNode` objects. In a singly-linked list, each node has a pointer to its successor, which we called `next`.

Now we are going to examine the properties of a doubly-linked list. In a doubly-linked list, each node maintains a pointer to the previous node in the list as well as the next node in the list. A `LinkedList` is shown below. Notice that the state of the `LinkedList` consists of two pointers, one to the first node in the list and one to the last node in the list, and an `int` that holds the size. Your linked list should not have any additional state variables.



The actual list is made up of objects that are similar to the `ListNode` object you used before. You will need to design these objects.

Before going on, investigate the advantages and disadvantages of the `LinkedList` object compared to our previous linked list and compared to `MyArrayList.` Write your findings in your notebook.

Exercise: Double Jeopardy

The `ListNode` class can only be used to construct singly-linked lists. Since we'll need to make a doubly-linked list for the `MyLinkedList<E>` class, you'll first need to design the `DoubleNode` class. A `DoubleNode` is just like a `ListNode`, but it also stores a pointer to the previous node in the list. In your notebook, write down the capabilities you think a `DoubleNode` 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.

Exercise: Linked List Capabilities

In your notebook, write down the capabilities you think a Linked List 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 `MyLinkedList`. 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 `MyLinkedList` 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 `MyLinkedList` class to fully implement the interface. You need not provide any specifications for these two `Iterator` methods.

As you complete your design, observe that several of the methods require that you walk the list to find the desired node. It would be silly to walk the list from the front if the desired node was at the end. Similarly, it would be silly to walk the list from the end toward the front if the desired node was in the front of the list. Think about a private helper method that returns a reference to the node at a desired index and how you might take advantage of the relative location of the node using two other helpers that walk the list either from the front toward the back or from the back toward the front. Since your linked list is a recursive data structure, a recursive solution is appropriate and required.

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: `MyLinkedList` Design Document

Create a design document that specifies the underlying data structure(s), instance fields, methods and their performance. Your document must also contain the design for your `DoubleNode` object. 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.

The `MyLinkedList` data structure requires a significant amount of pointer manipulation, especially for adding and removing elements. Your design document must include diagrams that show explicitly how you will manipulate the pointers for all methods that add an element to the list or remove an element from the list.

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

Exercise: `MyLinkedList` **Test Plan**

Create a test plan that specifies how you will verify that your `MyLinkedList` 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 this project, you will need to document two objects and their interactions. 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

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.