DYNAMIC DATA STRUCTURES

A. Introduction

A data structure is used to store, organize, and retrieve collections of related values. A static data structure has a fixed size, whereas a dynamic data structure will grow and shrink according to the storage needs of the program.

In this tutorial, we will discuss a few common dynamic data structures:

1. ArrayList
2. LinkedList
3. Stack
4. Queue

An Abstract Data Type (ADT) is a data type where only the behavior is specified, not the implementation. For example, a stack is a data type. A stack can be implemented a number of different ways (e.g. using an array). The individual using the stack in their program is not interested in how the stack is implemented. The user only cares about the behavior of the stack. A stack needs to have certain behaviors, such as the ability to push and pop items on/off the stack. Therefore, regardless of how a stack is implemented, it will need to contain the methods push() and pop(). Interfaces are often used to define the behavior of a class without specifying the implementation.

B. Tutorial

1. Array

We discussed arrays at length in an earlier tutorial. To refresh your memory, arrays are a sequence of related values stored in contiguous locations in memory. Arrays are a static data structure, which means that they are fixed size. When you create an array you declare its size, and the size remains the same throughout the program – it does not grow or shrink.

2. ArrayList

There are times when you do not know in advance how many elements you will need to store, and therefore, you need a dynamic data structure, one that will grow and shrink.

You have also been introduced to the ArrayList class. An ArrayList is a dynamic data structure that grows and shrinks as needed. You’ve learned some useful class methods to help you manipulate an ArrayList, such as get(), set(), add(), and remove().

You probably weren’t interested in how the ArrayList was implemented, you just needed it to have certain behaviors - get(), set(), add(), remove(), etc. An ArrayList can be implemented using a static array. An array can add elements until it is full, at which time a new array (double the size) must be allocated and the elements copied over. By using a static array in this manner, the implementation provides the illusion of a dynamic data structure.

When you implement an ArrayList using static arrays, you must keep track of the size (total number of elements) and the capacity of the array. If you exceed the capacity, you must allocate a new array that is twice the capacity of the original.
This implementation of an ArrayList allows for fast access to elements using an index, just like an array. However, when you insert or delete an element, you have to shift the other elements forward or backward accordingly. As well, doubling the size and copying elements over to the new array can slow down execution. Efficient memory utilization can also be an issue if the underlying array becomes large and the memory is not being used.

These are good examples of the advantages and disadvantages of a particular data structure. When deciding which data structure to use in your program, you will consider all of the benefits and drawbacks. Is efficient random access a priority? Is efficient use of memory a consideration? You will study these types of considerations in detail in a later course.

3. Linked List

Now we will discuss a new data structure. A linked list is a collection of objects that are linked together with pointers (references). A linked list is a dynamic data structure. A linked list can efficiently grow and shrink without the added expense of resizing, copying, and wasting memory. However, a drawback is that accessing elements can be more time consuming.

The objects that make up the linked list are referred to as nodes. Nodes have a field for data, such as a String value, and another field for a link (reference) to the next node in the list. The link may be null, which would indicate that there are no more nodes in the list.

The following figure is a visual representation of a linked list:

![Linked List Diagram](image)

**FIG. 1 Linked List**

There is a variable head, which is a reference to the very first node in the list. If there are no nodes in the list, the head is null. There could optionally be a variable tail, which is a reference to the last node in the list. You might include a reference to the last node in the list if you wish to add data to the end of the list efficiently.

As you can see from Figure 1, nodes consist of a data portion and a reference to the next node in the list. The last node’s reference is null, which can be used as an indicator that you have reached the end of the list.

When you add and remove nodes, you must maintain the correct relationships among the other nodes. For example, if you add a node to the beginning of the list, you must first create a new node, make that node point to what the head currently points to (the current first object in the list), and then point the head to the new node.
**FIG. 2 Creating and Adding New Node to Beginning of Linked List**

**Tips and Tricks:** You cannot create a node and then point the head to the newly created node first. If you do, you will lose your reference to the rest of the linked list.

Accessing elements at the front or back of the list is easy through the use of the head and tail pointers. However, if you wish to access an element in the middle, you must traverse the linked list, stepping over every element in the list. You know you’ve reached the end of the linked list when you hit a null pointer.
Let’s look at the basic structure of a linked list implementation, which does not currently include methods to add/remove/traverse the linked list.

```java
public class MyLinkedList {
    private Node head;
    private int size;

    private class Node {
        String value;
        Node next;

        public Node(String value) {
            this.value = value;
            this.next = null;
            size++;
        }
    }

    public MyLinkedList() {
        head = null;
        size = 0;
    }
}
```

Line 2 declares a variable head that references the first node in the list. Line 3 declares a variable size to maintain the total number of nodes in the linked list.

An inner class Node is implemented on Lines 5-14. This class is used to create the Node objects that make up the linked list. Each Node object has a data field and reference field. We will use a String for our data field, called value. As well, each Node object contains a field that is a reference (pointer) to another Node object in the list, called next.

The Node class constructor is found on lines 9-13. When a new Node object is created, the String value is initialized. The reference field is set to null because when a Node object is initially created, it is not connected to the linked list yet, as shown in Figure 2.

Lastly, the outer MyLinkedList class has a constructor on Lines 16-19 which initializes the head to null and size to zero. In a more complete implementation, the outer class MyLinkedList would have accessors and mutators to access the fields.
Next, let’s write a method to add a node to the linked list. Remember, the first step is to create a new Node object. The next step will be to link the newly created Node object to the beginning of the list. Our add() method will have a parameter String value to be passed to the Node constructor.

```java
public class MyLinkedList {
    private Node head;
    private int size;

    private class Node {
        String value;
        Node next;

        public Node(String value) {
            this.value = value;
            this.next = null;
            size++;
        }
    }

    public MyLinkedList() {
        head = null;
        size = 0;
    }

    public void add(String value) {
        Node temp = new Node(value);
        temp.next = head;
        head = temp;
    }
}
```

Lines 21-25 contain the add() method implementation. Line 22 creates a new Node object called temp by calling the Node constructor and passing the String value as an argument.

Lines 22-24 may seem confusing at first and require a bit of practice. Recall, the first step for linking a Node object into the linked list is to set the next reference of the new Node to the object pointed to by head. In other words, we are setting temp’s reference, next, to be equal to what head is referencing, i.e. the current first node in the list. Then, we have to change the head reference to point to temp. Now, the newly created Node object is linked to the list! Figure 2 is a visual representation of these steps.

You will practice traversing a linked list and removing nodes from a linked list in the Practice and Challenge sections below. Solutions are provided, but it is recommended that you give it a go before checking the solution. The best possible practice is to work your way through the linking and unlinking process in order to train your brain how to visualize these steps. It is an especially good idea to draw figures of the steps to help visualize each step. The lectures slides and textbook provide a detailed explanation of the steps for reference!
4. Stack

A stack is a dynamic data structure that accesses and stores items according to the LIFO (last in, first out) principle.

Think of Figure 2 as a stack of boxes. When you add a box to the stack, you place it on top. When you remove a box from the stack, you pull from the top. The most recent (last) box that was placed on the stack is the first box to be removed from the stack. The stack data structure works exactly the same way.

You can add items to and remove items from a stack using the `push()` and `pop()` methods. As you saw in Figure 3, adding (pushing) and removing (popping) elements always takes place at the top of the stack.

The following are some of the standard methods found within a stack implementation:

- `push()` adds an item to the top of the stack.
- `pop()` removes an item from the top of the stack and returns it to the caller.
- `isEmpty()` returns `true` if the stack is empty and `false` if the stack is not empty.
- `top()` returns the item on the top of the stack, but does not pop it off.

Recall, a stack is an Abstract Data Type (ADT). The user of the stack is not concerned with its implementation. For example, it may be implemented using an array or linked list, but the user only cares that the stack data structure provides these basic behaviors.

Let’s look at the basic structure of a stack implementation. In lecture, you learned how to implement a stack using a linked list. Now, let’s look at how to implement a stack using an array. Keep in mind the following class is not a complete implementation of a stack. For the purposes of this tutorial, we will only implement the `push()` method.
import java.util.Arrays;

public class MyStack {
    private String[] strings;
    private int size;

    public MyStack() {
        this.strings = new String[10];
        this.size = 0;
    }

    public void push(String str) {
        if (size == strings.length) {
            strings = Arrays.copyOf(strings, strings.length*2);
        }
        strings[size++] = str;
    }

    public String pop() {
        // TODO
    }

    public String top() {
        // TODO
    }

    public boolean isEmpty() {
        // TODO
    }

    public void printStack() {
        // TODO
    }
}

Line 8 creates an array with space for 10 String objects. This array represents the stack. Line 9 initializes the size of the stack to be zero. size represents how many String objects are on the stack.

Lines 12-17 provide an implementation for push(). push() has a parameter String str, which is the String object to be added to the stack.

Line 13 checks if size is equal to the length of the array strings. If size (the number of String objects on the stack) is equal to the length of the array (the capacity of the stack), the array has to be resized.

Tips and Tricks: The Array class has a method copyOf() which takes the original array and the new size. copyOf() copies all of the elements in the original array to a new array of the specified size. Notice that we specify that the new size should be twice the length of the current array. This is standard practice for resizing. An alternate and perhaps more familiar approach would be to create a new array, manually copy over each element using a for loop, and then point to the new array. Regardless of your approach, an implementation that uses a static array must account for when to resize the array so that it behaves the way a dynamic stack does, growing and shrinking as needed.

Line 16 adds str to the top of the stack and increments size (total number of String objects).
Keep in mind, a full stack implementation would need methods such as `pop()`, `top()`, `isEmpty()`, etc. However, they are not included in this implementation because they are excellent practice! Try your hand at implementing these stack methods using an array. The lectures slides, textbook, and JavaDocs are excellent resources!

5. Queue

A queue is a data structure that accesses and stores elements according to the FIFO (first in, first out) principle.

Consider a line at the grocery store. The first person in line is the first person checked out. You can add items to and remove items from a queue using the `enqueue()` and `dequeue()` methods. As shown in Figure 4, you add (enqueue) elements to the back of the list and remove (dequeue) elements from the front of the list.

The following are standard methods found within a queue implementation:

- `enqueue()` adds an item to the back of the queue.
- `dequeue()` removes item from the front of the queue.
- `peek()` retrieves, but does not remove, the element at the front of the queue or returns `null` if this queue is empty.
Let’s look at the basic structure of a queue implementation. We will use a linked list to implement our queue. Keep in mind, the following class is not a complete implementation. For the purposes of this tutorial, we will only implement the enqueue() method.

```java
public class MyQueue {
    private Node head;
    private Node tail;

    private class Node {
        String value;
        Node next;

        Node(String value) {
            this.value = value;
            this.next = null;
        }
    }

    public MyQueue() {
        this.head = null;
        this.tail = null;
    }

    public void enqueue(String value) {
        Node temp = new Node(value);
        if (head == null) {
            head = temp;
            tail = temp;
        } else {
            tail.next = temp;
            tail = temp;
        }
    }

    public String dequeue() {
        // TODO
    }

    public String peek() {
        // TODO
    }
}
```

Lines 2-3 create Node references to the head and tail of the queue. If it’s easier, you could refer to these references as the front and back of the queue respectively. They are initialized to null (lines 16-17) because there are no elements in the queue.

The inner class Node on Line 5-13 should look familiar from our discussion above about linked lists. This class is used to create the Node objects that make up the linked list that represents our queue.
Lines 20-31 provide an implementation of `enqueue()`, which will add a Node object to the linked list that represents our queue. `enqueue()` has a parameter String value which will be passed to the Node constructor to initialize the Node object.

Line 23 checks if the head reference is null. Remember, we are implementing our queue using a linked list. What does it indicate if the head points to null? It means the list is empty. It means there is nothing in the queue. Note, we add Node objects to the tail\lstinline{i.e.} the back of our queue, just like if people were getting in line at the grocery store.

So if head is null (line 23), we are adding the first Node in the queue. Because there is only one Node in the queue, both head and tail will point to this object (lines 24-25).

Line 27 is the else statement, i.e. the head was not null, indicating there is already a Node object(s) in the queue. In this case, the new Node will be added to the tail of the linked list (line 28-29).

Keep in mind, a full queue implementation would need methods such as dequeue(), peek(), etc. However, they are not included in this implementation because they are excellent practice! Try your hand at implementing these queue methods using a linked list. The lectures slides, textbook, and JavaDocs are excellent resources!

C. Tips and Tricks

1. Java provides Array, ArrayList, LinkedList, Stack, and Queue classes. The JavaDocs is an excellent resource to learn more about the fields and methods of these classes.

   https://docs.oracle.com/javase/8/docs/api/

2. Don’t forget that a tail pointer can be useful for efficiently inserting at the end of a linked list.

3. There are a number of situations that can generate NullPointerExceptions when working with these data structures, (i.e. attempting to remove from an empty list). Be cautious! Sometimes it helps to draw a diagram, labeling the references (pointers).

D. Practice

Please review the following method for the MyLinkedList class discussed in part 3 of the Tutorial section above:

```java
void printList() {
    Node current = head;
    System.out.print("Linked List (size " + getSize() + ": ");
    while ( /* TODO: Fill in the loop condition */ ) {
        System.out.print(current.value + " -> ");
    } // TODO: Update current reference
    System.out.println("null");
}
```

`printList()` creates a reference to the first node in the list, called current. Complete the "TODO" tasks to iterate over every node in the linked list.
E. Challenge

The `MyLinkedList` class in part 3 of the `Tutorial` section was incomplete. Write an implementation for `remove()` that removes the first node in the list and returns the `String` value. Use your completed `printList()` method from the `Practice` section for debugging purposes.

The following `main()` method may be helpful for testing:

```java
public static void main(String args[]) {
    MyLinkedList list = new MyLinkedList();

    list.add("Jane");
    list.add("is");
    list.add("Name");
    list.add("My");
    list.add("Hello");

    list.printList();
    String temp = list.remove();
    list.printList();
    temp = list.remove();
    list.printList();
    temp = list.remove();
    list.printList();
    temp = list.remove();
    list.printList();
    temp = list.remove();
    list.printList();
    temp = list.remove();
    list.printList();
}
```

The expected output of this program is:

```
Linked List (size 5): Hello -> My -> Name -> is -> Jane -> null
Linked List (size 4): My -> Name -> is -> Jane -> null
Linked List (size 3): Name -> is -> Jane -> null
Linked List (size 2): is -> Jane -> null
Linked List (size 1): Jane -> null
Linked List (size 0): null
```