# Lab8: Data Structures I

## CSSSKL162: Programming Methodology Skills,

## Summary

The purpose of this lab is to practice what we've covered in class and in the readings (Chapter 15) by building a Node, a Stack, and a Queue using links. We'll start by examining each of the "contracts" or ***interfaces*** for each class.

## The Stack (LIFO) Interface

- void push(Object)
- Object pop()
- int size()
- String toString()
- boolean isEmpty()
- boolean equals(Object);

## The Queue (FIFO) Interface

- void enqueue(Object)
- Object dequeue()
- int size()
- String toString()
- boolean isEmpty()
- boolean equals(Object);

## ADT/Data Structure Background

A Data Structure is a composite data storage tool that organizes elements of a set and offers operations over those elements in the set. Frequently called a structure, class or ADT, these tools are used to provide order and operations to a collection of elements. Data structures vary in how they are built internally, how they organize data (sorted or unsorted, for example), and the operations provided. Usually these operations are compared to one another with respect to efficiency using the"Big Oh" notation. One structure that is useful for quick searches (say for an airlines company searching for connections) may have disadvantages in other areas, such as in memory consumed – or trade one fast operation that will be used frequently for a slow operation that will be rarely used.

## Static Data Structures

Array-based data structures are sized at compile-time, since this is a ***static*** structure. This implies that your structure cannot grow (or shrink) at runtime depending on the applications needs. We built these on top of arrays, which are mapped contiguously in memory for fast (or constant, meaning a BigO(1)) access to any element. Using arrays to build more complex data structures serves as a good introduction to the behaviors and mechanics of Stacks, Queues, and (more generally) Lists, and we'll build off our understanding here to construct the new structures with the same interface (such as push() or pop()) but with dramatically different implementation details (using Nodes and links rather than arrays).

## Dynamic Data Structures

Today we'll focus on building structures that can grow or shrink arbitrarily at runtime. These structures can allocate and de-allocate memory at runtime depending on the requirements of the client application. These are ***dynamic*** structures in that their memory footprint may change over the duration of the program's execution. To build such structures, it would be a needless limitation to impose contiguous storage, requiring n back-to-back blocks of memory to hold a list of n items. With such a requirement the total memory required may be available, but if we insist on a linear mapping we may be unable to make use of the memory due to fragmentation (what about **coalescing holes**?). Our dynamic structures will instead allocate only the memory they need wherever there is RAM available, and we will "stitch together" lists from these individual elements or nodes. This will be our new Stack and Queue implementation, built using Nodes and links.

## Getting Started With the Stack Class

Start by and reading the driver code, included for both Stack and Queue. Build two new empty classes called Stack and Queue, and copy the corresponding main driver code into each class. Next, we'll add an identical Node class to both Stack and Queue.

- Create two classes: Stack.java and Queue.java
- Copy the driver code provided below into each class (You need to modify the driver to work with your Queue class interface) . What imports do you need?
- Execute the driver code and observe the results for both Stack and Queue class

## Stack Driver

```
public static void main(String[] args) {

      Stack<Character> a = new Stack<Character>();
     // Queue<Character> q = new LinkedList<Character>();
```

```
        a.push('R');
        a.push('a');
        a.push('c');
        a.push('e');
        //a.push('c');
        //a.push('a');
        a.push('r');

        System.out.println("Size : " + a.size());

        while(!a.isEmpty()) {
              System.out.println(a.pop());
        }
}
```

## The Node Class

Our Stack and Queue structures are linked lists of Node objects. A Node defines a single data item (frequently of type Object) and also defines a pointer/reference to the next node in the chain (or null if at the end of the chain). We'll start by building our Node class first, which will be only a handful of lines of code.

- Define a new class (called Node) inside your Stack class
  - Once this is complete, copy this inner class to your Queue as well
- Add two data items to the Node class
  - Object data=null;
  - Node next = null;
- Add a constructor to the Node class:

```
public Node(Object d, Node n) {

        data = d;          //shallow copy on purpose for ADTs

        next = n;

}
```

## The Stack Using Nodes & Links

Download the **LLStack .java** (as in LinkedListStack) template provided in Lab8. In this section we will build our own stack data structure, using Nodes, utilizing the concept of linked Nodes as a data structure. When implemented successfully, this data structure will provide you the functionality of a stack and should produce the same result as your built-in Stack. You will also get to implement the Node class above as a private inner class in your code.

- Follow the instructions embedded in the file and fully implement methods that are incomplete.
- Compile and run your software with the provided driver in the same file.
- Test your stack-does it reverse the output it is given when elements are pushed?
- Compare your output with output observed when using Java's built-in Stack.
- Explain specifically which method is responsible for making sure that LIFO behavior (a characteristic of Stack) is correctly implemented in your software.