# Course Description

**Weekly Overview**

This week focuses on the Stack data structure, but also introduces the notion of Nodes. Students will have been exposed to the idea during the Tree lab, but there they will have just bee given the and not asked to work with them. In writing the Stack this way, this also gives students a way to see two different implementations of the same interface – they first being the array-based Stack that they saw during the semester exam. Finally, students will use the Stack to solve a programming problem.

# Institutional Learning Outcomes

**Main Objectives**

* Implement and work with linked nodes.
* Implement a node-based Stack.
* Use Stacks to solve problems.

# Understand recursion, both mathematic

# Discipline Specific Outcomes

# Student Readings

None

**Daily Outline**

Day 1: Define Stacks and Linked Nodes

Day 2: Using Stacks

Day 3: Stack Lab

Day 4: Stack Lab (continued)

Day 5: Flexible

**Included Resources**

Lecture Notes: Stacks and Linked Nodes

Lecture/Lab Notes: Circuit Routing

Lab Assignment: Stacks and Circuit Routing

Node.java

Stack.java (solution)

Circuit.java (solution)

**Lecture Notes: Stacks and Linked Nodes**

**Bell Work (5 minutes)**

Write down what you can remember about a Stack from the semester exam.

***Answer:*** Answers will vary.

**Main Lecture: Part 1 – The Stack (10 minutes)**

Recall with students the basic interface of a Stack:

public class Stack<E>{

public Stack()

public void push(E x)

public E pop()

public E peek()

public int size()

}

Take students through an example of a Stack after a variety of calls, e.g.

Stack<String> s = new Stack<String>();

s.push(“Jake”);

s.push(“Christina”);

s.push(“Tatiana”);

int j = s.size(); //what is j?

String one = s.pop(); // what is one?

String two = s.pop(); // what is two?

s.push(“Kateri”);

s.push(“Jude”);

s.push(“Benedict”);

j = s.size(); //what is j?

String three = s.pop(); //what is three?

s.push(“Sebastian”);

s.push(“Thomas”);

s.push(“Edmund”);

s.push(“Marie”);

String stuff = s.toString(); // what is stuff?

s.pop();

s.pop();

j = s.size(); //what is j?

stuff = s.toString(); // what is stuff?

A Stack is normally described as “LIFO”, or “Last In – First Out”. In other words, it is the opposite of a “line” (which in computer science is called a Queue). In a line, the first person in line gets priority to be the first one “called”, or the first one out of the line. In a Stack, it is the other way around. The most recent addition to the Stack will be the first one “popped off”. Think of it like a stack of papers: the most recent addition (the “top”) is the first one accessible in the Stack. The very first one added (or “pushed”), which is on the “bottom”, will the last one accessible. You can only access one element at a time, and you cannot access anything in the middle. It is a very simple data structure. Because of its simplicity, we can achieve some pretty remarkable time and memory efficiencies.

Note also the peek() method, which allows us to look at the top element without having to pop it off.

**Main Lecture: Part 2 –Linked Nodes (20 minutes)**

We have already seen how to implement the Stack using an array in the semester exam, but this has the tremendous disadvantage of having to resize and copy when the underlying array gets full. To implement a Stack in a different way, we are going to define a new class called a Node.

public class Node<E>{

private E data;

private Node<E> next;

public Node(){

this(null, null);

}

public Node(E d){

this(d, null);

}

public Node(E d, Node<E> n){

data = d;

next = n;

}

public void setData(E d){

data = d;

}

public void setNext(Node<E> n){

next = n;

}

public E getData(){

return data;

}

public Node<E> getNext(){

return next;

}

}

There are several talking points:

1. The “null” reference needs to be reviewed. Remember that a reference stores an address. A “null” reference is a special addressed reserved in the computer that is used to indicate “no address.”
2. The generic type needs to be either reviewed or covered. It is a placeholder for a type that will come at declaration type.
3. The definition itself is recursive. A Node has as one attribute a Node. This allowed us to “chain” these Nodes in an elegant way.

Take students through an example using Nodes.

Node<String> myNode1 = new Node<String>(“Jake”);

Node<String> myNode2 = new Node<String>(“Christina”, myNode1);

Node<String> myNode3 = new Node<String>(“Tatiana”, myNode2);

Node<String> myNode4 = new Node<String>(“Kateri”, myNode3);

The state of the Node is as follows:

myNode4 myNode3 myNode2 myNode1

data: Kateri

next:

data: Tatiana

next:

data: Christina

next:

data: Jake

next: NULL

The “next” for each Node is a Node itself. It is an ingenious way to string together instances of a datat type.

Notice that the last Node has a next that is still “NULL” indicating the end of the linked structure.

Now, what is the output of the following:

System.out.println(myNode4.getNext().getData());

//Answer: Tatiana

System.out.println(myNode3.getNext().getNext().getData());

//Answer: Jake

System.out.println(myNode2.getNext());

//Answer: an address, something like Node@43249c66

System.out.println(myNode2.getNext().getNext());

//Answer: null

System.out.println(myNode2.getNext().getNext().getData());

//Answer: nothing, this throws a null pointer exception

Now, what happens when I execute the following:

myNode2.setNext(new Node<String>(“Jude”));

System.out.println(myNode2.getNext().getData());

//Answer: Jude

Notice that the old Node containing “Jake” is still here, but not part of the chain. The diagram looks like this:

myNode4 myNode3 myNode2 [No Name]

data: Kateri

next:

data: Tatiana

next:

data: Christina

next:

data: Jude

next: NULL

myNode1

data: Jake

next: NULL

What is interesting now is if we execute some similar code:

myNode2.setNext(new Node<String>(“Benedict”));

System.out.println(myNode2.getNext().getData());

//Answer: Benedict

The interesting thing is that the old “Jude” Node no longer has a reference to it:

myNode4 myNode3 myNode2 [No Name]

data: Kateri

next:

data: Tatiana

next:

data: Christina

next:

data: Benedict

next: NULL

myNode1 [No Name]

data: Jude

next: NULL

data: Jake

next: NULL

In practical terms, this means you have lost the data. You have no way to access it through code because it does not have a “named” reference pointing to it. It is what we call an orphaned record. It other languages, this can become a problem because it takes up memory, but you cannot deleted it (because there is no reference to it). But in Java, we have “automatic garbage collection.” The system periodically goes through and finds these orphaned records and deletes them. “But what if I still want it and the system deleted it?” *Answer:* You cannot access it anyway. If you wanted it, you should have kept a reference to it.

Have students next think through the following:

myNode3.setNext(myNode1);

What is the picture?

myNode4 myNode3 myNode1

data: Kateri

next:

data: Tatiana

next:

data: Jake

next: NULL

myNode2 [No Name]

data: Christina

next: NULL

data: Benedict

next: NULL

We effectively have two chains now.

**Main Lecture: Part 2 –Linked Nodes (10 minutes)**

We are now ready to implement the Stack. Lead the class through as much of the following as possible. Whatever is not completed in class should be completed for homework. When talking them through it, pay attention to border cases (empty Stacks and one-element Stacks).

The “answer” is found as an attachment on the next page.

**Homework:** Complete the rest of the Stack class.

//Stack.java

public class Stack<E>{

private Node<E> top;

private int size;

public Stack(){

top = null;

size = 0;

}

public void push(E x){

Node<E> newTop = new Node<E>(x); // construct the new top Node

newTop.setNext(top); //link the new top Node to the Stack

top = newTop; //reassign the top reference

//if the last two lines are switched, we orphan the entire Stack

size++; //adjust size

//Note that in the case of an empty Stack (pushing the first)

//element, the code still works.

}

public E pop(){

E retObj = top.getData(); //save object to return

top = top.getNext(); //trickiest part, assign top to the next

//this is how we delete – we let the old top Node get orphaned

//(while saving the data to return)

size--; //adjust size

return retObj;

//Note that in the case of an empty Stack, the first line

//throws a null pointer exception, which is what we want.

//Note also in the case of a one-element Stack, top is set

//to null, which is what we want.

}

public E peek(){

return top.getData();

}

public int size(){

return size;

}

}

**Lecture Notes: Circuit Routing**

**Bell Work (0 minutes)**

None.

**Main Lecture: (15 minutes)**

This should be a short lecture that is an introduction to the lab. Pass out the lab sheet, and talk through the circuit routing problem. Students will download Node.java and Stack.java (both implementations that were done in class) from GitHub and use them to think through a problem. What will make this challenging is that we are not giving students the algorithm. They should work in pairs on paper first to decide how to actually accomplish the problem.

Note that the code provides them with how to achieve console input.

The algorithm solution is provided as an attachment, but the basic idea is to go through the pins in order. If the top of the stack is a matching pair (a wire), we pop off that number. Otherwise (and including if the Stack is empty), we push the number on the Stack. If the Stack is not empty in the end, then the box is not routable. If the Stack is empty in the end, the box is routable.

For example, consider the example in the lab sheet: (0,3), (1,2), (4,5), (6,7)

1. The Stack is empty, so we push “0”, so the Stack is “0”.
2. Next we deal with “1”. 1 is not a pair with the top of the Stack (0), so it gets pushed. The Stack is now: 0, 1.
3. Next we deal with 2, which *is* a pair with the top of the Stack (1), so we pop the Stack. The Stack is now: 0.
4. Next we deal with 3, which is a pair with the top of the Stack (0), so we pop the Stack. The Stack is now empty.
5. Next we deal with 4. The Stack is empty, so we push the 4. The Stack is now: 4.
6. Next we deal with 5, which is a pair with the top of the Stack (4), so we pop the Stack. The Stack is now empty.
7. Next we deal with 6. The Stack is empty, so we push the 6. The Stack is now: 6.
8. Next we deal with 7, which is a pair with the top of the Stack (6), so we pop the Stack. The Stack is now empty.

This is the end of the wires, and the Stack is empty, so the box is routable.

For a counter example, try (0,4), (1,2), (3,6), (5,7).

1. The Stack is empty, so we push “0”, so the Stack is “0”.
2. Next we deal with “1”. 1 is not a pair with the top of the Stack (0), so it gets pushed. The Stack is now: 0, 1.
3. Next we deal with 2, which *is* a pair with the top of the Stack (1), so we pop the Stack. The Stack is now: 0.
4. Next we deal with 3, which is not a pair with the top of the Stack (0), so we push 3 onto the Stack. The Stack is now: 0, 3.
5. Next we deal with 4, which is not a pair with the top of the Stack (3), so we push 4 onto the Stack. The Stack is now: 0, 3, 4.
6. Next we deal with 5, which is not a pair with the top of the Stack (4), so we push 5 onto the Stack. The Stack is now: 0, 3, 4, 5.
7. Next we deal with 6, which is not a pair with the top of the Stack (5), so we push 6 onto the Stack. The Stack is now: 0, 3, 4, 5, 6.
8. Next we deal with 7, which is not a pair with the top of the Stack (6), so we push 7 onto the Stack. The Stack is now: 0, 3, 4, 5, 6, 7.

We are through the pins, and the Stack is not empty. Therefore, the box is not routable.

It is worth noting that the problem occurred with the (0, 4) and (3, 6). These were the wires that crossed. The 4 should have popped the 0, but the 3 was pushed on first, preventing the 0 from being on top.

Do *not* lead students through this. Help struggling students by giving some hints. The challenge of this lab is that students need to think about the nature of a Stack and how it can help solve this problem.

**Homework:** None

**Lab Assignment: Circuit Box Routing**

In the switch box routing problem, we are given a rectangular region with pins at the periphery. The pins are numbered consecutively from 0 to *n* in a clockwise fashion. Pairs of pins are to be connected together by laying metal path between the two pins. The path, conveniently, are called “wires”. Each pair of pins (to be connected by a wire) will be represented by an ordered pair. Thus, a switch box can be described by a series of these wires, e.g. (0,3), (1,2), (4,5), (6,7).

When wiring a switch box, we cannot have intersecting wires, otherwise an electrical short will occur. You are to determine if a given switch box is **routable**, that is if we can connect the ordered pairs with no intersections. The figure below shows that the switch box above is indeed routable.

0 1

7 2

6 3

5

5

4

It will not take you long to figure out that only the *order* of the pins is important for determining “routability.” In other words, the “1” pin could be above the 2 on the right side of the box, and the box remains routable. In fact the basic “solution” for routability does not change.

However, some boxes, such as the one given by (0,4), (1,2), (3,6), (5,7), are **not** routable. The (0,4) wire must intersect the (3,6) wire. (Try it, and notice again that, so long as the order of pins is the same, it does not matter which pins are on which side of the box. In fact, they could all be on one side of the box … hint, hint.)

For our implementation, we will assume that the number of pins is even and that each pin has exactly one partner pin.

The user should be able to enter the switch boxes as follows:

Please enter the switch box configuration: (0,3) (1,2) (4,5) (6,7)

This box is routable.

You will be provided with a file that takes care of the user input as well as a TestFile.txt which has a collection of routable and non-routable boxes for you to test. Feel free to add to this if you see fit.

There is one requirement for this project. **You must use a Stack to decide whether or not the box is routable.** Therefore, the process is in three steps:

1. Download Node.java and Stack.java
2. Download CircuitRouting.java and CircuitBoxes.txt which has the shell plus the code for input and some file testing.
3. Write the isRoutable method that accomplishes the required task.

//CircuitRouting.java

//Full solution – do not distribute to students

import java.io.\*;

import javax.swing.JOptionPane;

import java.util.Scanner;

public class Circuit

{

/\*

\* The basic way to store wires, which are ordered pairs

\* is to use an array. The index is the first element,

\* and the value is the second element. Therefore, the circuit

\* pattern (0,1), (2,5), (3,4), (6,9), (7,8) would be stored

\* as follows: wires = [1, 0, 5, 4, 3, 2, 9, 8, 7, 6]

\* So, for example, wires[3] = 4 because (3,4) is a pair;

\* note toot aht wires[4] = 3 becuase (4,3) represents the same

\* wire. This allows us to check if (a,b) is a wire by checking

\* either wires[a]==b or wires[b]==a.

\*

\* For this lab, we presume errorless input. In other words,

\* the data will not have (2,3) and (5,3) in it, which would

\* be a contradiction in wiring.

\*/

public static boolean checkBox(int [] wires)

{

Stack<Integer> myStack = new Stack<Integer>();

for(int i=0; i<wires.length; i++){

if(myStack.size() == 0)

myStack.push(i); //for simplicity, takes advantage of autocasting

else{

if(myStack.peek().equals(wires[i]))

myStack.pop();

else

myStack.push(i);

}

}

if(myStack.size() == 0)

return true;

return false;

}

public static boolean isNumeric(char c){

if(c=='0'||c=='1'||c=='2'||c=='3'||c=='4'||c=='5'||c=='6'||c=='7'||c=='8'||c=='9')

return true;

return false;

}

public static int[] processInput(String s){

int commas = 0; //commas are the number of pairs

for(int i=0; i<s.length(); i++){

if(s.charAt(i) == ',')

commas++;

}

String[] numbers = new String[commas \* 2];

Scanner in = new Scanner(s).useDelimiter("\\) \\(|\\(|,|\\)");

int[] wires = new int[commas \* 2];

for(int i=0; i<commas; i++){

int a = Integer.parseInt(in.next());

int b = Integer.parseInt(in.next());

wires[a] = b;

wires[b] = a;

}

return wires;

}

public static void main(String [] args) throws IOException

{

//Code for console I/O - comment out if not using

/\*String boxConfig = JOptionPane.showInputDialog("Please enter the box configuration");

int[] wires = processInput(boxConfig);

String n = "not ";

if(checkBox(wires) == true){

n = "";

}

JOptionPane.showMessageDialog(null, "The box is " + n + "routable.");\*/

//Code for file I/O - comment out if not using

FileReader reader = new FileReader("CircuitBoxes.txt");

BufferedReader in = new BufferedReader(reader);

String circuitLine = in.readLine();

String answerLine = in.readLine();

while(circuitLine != null){

boolean routable = checkBox(processInput(circuitLine));

System.out.println(circuitLine);

System.out.println("Answer: " + answerLine);

System.out.println("Your answer: " + routable);

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

circuitLine = in.readLine();

answerLine = in.readLine();

}

reader.close();

}

}

CircuitBoxes.txt

(0,3) (1,2) (4,5) (6,7)

true

(0,4) (1,2) (3,6) (5,7)

false

(0,1) (2,5) (3,4) (6,9) (7,8)

true

(0,11) (1,10) (2,3) (4,9) (5,8) (6,7)

true

(0,1) (2,7) (3,9) (4,8) (5,6)

false

(0,11) (1,10) (2,9) (3,8) (4,7) (5,6)

true

(0,1)

true

(0,2) (1,3)

false