Niema Widaha

MAC 286: Data Structure Summary – Stack

Describing and implementing a stack:

a) Define a stack in your own words
   a. What are they?
      i. A stack is a method to manage and store data in order to perform operations on elements that are not all readily available to the user. This is because a stack only allows you to access the last item inserted and is why the stack is known as the LIFO structure (Last-In-First-Out).
   b. What are their key features?
      i. Some key features of a stack are as follows:
         1. Push: This feature allows you to add an item of data to the top of the stack
         2. Pop: This feature allows you to remove an item of data that was the last item added to the sack
         3. Peek: This allows you to view the item that is at the top of the stack.

b) Modifying the stack code, we wrote and used in class, write your own stack class based on an array of chars. Include methods to pop and push, as well as any others you find helpful:

```
class MyStack{

   // private members:
   private int     maxSize;          // max size of the stack
   private char[]   charStackArray;  // array where we store the stack items
   private int     top;              // tells us where the top of the stack is:

   // constructor:
   //--------------------------------------------------------------
   public MyStack(int size){
       maxSize = size;                     // max size of stack
       charStackArray = new char[maxSize]; // initialize stackArray
       top = -1;                           // currently nothing is in the stack
   }

   // methods:
   //--------------------------------------------------------------
   // push()
   public void push(char newValue) throws FullStackException {

       if(isFull()){
           throw new FullStackException("Can't push to the stack");
       }

       // add an item to the top of the stack
       top += 1;
       charStackArray[top] = newValue;

   } // ends push()

   //--------------------------------------------------------------
   // pop():
   public char pop() throws EmptyStackException {
       if(isEmpty()){
           throw new EmptyStackException();
       }

       // want to remove the item from the top of the stack
```

```
        // also want to return that item:
        char popValue = charStackArray[top];

        top -= 1;

        return popValue;
    }

    //-----------------------------------------------------------------
    // isEmpty():
    public boolean isEmpty(){
        if(top == -1){
            return true;  // stack is empty
        } else
            return false; // stack isnt empty
    } // ends isEmpty()

    //-----------------------------------------------------------------
    // isFull():
    // returns true if the stack is full
    // returns false otherwise
    public boolean isFull(){

        if(top == maxSize - 1){
            return true; // stack is full bc maxsize -1 is the last index
        } else {
            return false;
        }
    } // ends isFull()

    //-----------------------------------------------------------------
    // display():
    public void display(){

        // want to print the stack from top to bottom:
        System.out.println("The stack from top to bottom is: ");

        for(int i = top; i >= 0; i--){
            System.out.println(charStackArray[i] + " ");
        }

        System.out.println("");
    } // ends display()

}  // ends MyStack class
```

c) Write a main method to test how your stack class works:

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

    char[] helper = new char[]{'a','b','c','d','e'};

    MyStack charStack = new MyStack(5);

    for(int i = 0; i < 5; i++ ){
        charStack.push(helper[i]);
    }

    charStack.display();

}
```

a. Why did you choose these test cases?

i. I chose these test cases, because I would like to see if the MyStack class can handle characters being inputted into the stack.

b. What results did you expect and why?

i. I expected to see a string of words, a b c d e, because I have a display() method that prints the contents of the stack from the top to the bottom.

c. Did the code result match what you expected?

i. The code results did not match what I initially expected because I forgot that stacks contain a LIFO structure, so the result wasn't -> (a b c d e) but -> (e d c b a).

d) What are the time complexities of your pop and push methods?

a. How do you know?

i. I know this because in case we need to insert or push an element in the stack, we add one element at the top of the stack so that is just one step, so it takes constant time. The same goes for the pop operation, because when you remove one element from the top of the stack, you just use one step, so it takes constant time and takes O(1)

e) Add in a method (in the class where you main () method lives, not the stack class) to take in a String and reverse it using your stack class. For example, if the input is "happy" this method should return "yppah". To get from a string to the chars for your stack, you might find the charAt() method for Strings helpful.

```
//-----------------------------------------------------------------
// reverseString():
public static String reverseString(String stringToReverse){
    int stackSize = stringToReverse.length(); // stores the size of the new stack

    MyStack reverseStack = new MyStack(stackSize); // makes the stack

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

        char ch = stringToReverse.charAt(i);
        reverseStack.push(ch);
    }

    String reversedString = ""; // to return the new string in reverse

    // add the items from the top of the stack to the bottom
    while(!reverseStack.isEmpty()){

        char reverseChar = reverseStack.pop();
        reversedString += reverseChar; // append to output
    }

    return reversedString;
} // ends reverseString()
```

f) Add to your main() method to test how your String reversal method works.

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

    // create string:
    String name = "Niema";

    // pass string to reverse string:
    String reverseName = reverseString(name);
    System.out.println("Unreversed: " + name + "\nReversed: " + reverseName);
}
```

a. Why did you choose these test cases?

i. I chose these test cases, because I know what my name is reversed and what I should expect and I also wanted to see whether the method would work with just passing in a string into the method.

b. What results did you expect and why? Did you get what you expected?

i. I expected the results I found because within my reverseString() method I'm creating a MyStack object and performing all the needed operations within it. I know that the item at the top of the stack is the last item in my string so I added the top of the stack (which is the last element in my name) to the beginning of the reverse string and further on till my stack is empty. I got what I expected.

Using Stacks: *(Code is below after the tests)

Test your code with some different start/end cities using the map in the image below or come up with your own maps! Make sure to explain why you chose your test cases, what you expected to happen, and whether the results matched your expectations.

1. Test 1: stack.pathBetween(3, 0 ,cityPath);
    1. Why I chose this test case: Because they're one city away and in between from each other.
    2. What I expected to happen: I expected there to be a path between these two cities based on the diagram given.
    3. Whether the results matched my expectations: Yes.

Main.java

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

    MyStack firstStack = new MyStack(10);

    int[][] cityPath = {{1,2},{0,2,3},{0,1,4},{1},{2},{6},{5}};

    System.out.println(firstStack.pathBetween(3, 0, cityPath));

  }
}
```

```
> javac -classpath .:/run_dir/junit-4.12.jar:target/dependency/* -d . EmptyStackException.java FullStackException.java Main.java
> java -classpath .:/run_dir/junit-4.12.jar:target/dependency/* Main
current: 3
There's a path between these two cities.
true
>
```

2. Test 2: stack.pathBetween(5, 6, cityPath);
    1. Why I chose this test case: Because they're only connected to each other and no other city.
    2. What I expected to happen: There to be a path between these two cities.
    3. Whether the results matched my expectations: Yes.

Main.java

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

    MyStack firstStack = new MyStack(10);

    int[][] cityPath = {{1,2},{0,2,3},{0,1,4},{1},{2},{6},{5}};

    System.out.println(firstStack.pathBetween(5, 6, cityPath));

  }
}
```

```
> javac -classpath .:/run_dir/junit-4.12.jar:target/dependency/* -d . EmptyStackException.java FullStackException.java Main.java
> java -classpath .:/run_dir/junit-4.12.jar:target/dependency/* Main
current: 5
The stack from top to bottom is: 6
current: 6
There's a path between these two cities.
true
>
```

3. Test 3: stack.pathBetween(1, 3, cityPath);
    1. Why I chose this test case: Because these are connected to each other in the diagram
    2. What I expected to happen: I know based on the given diagram, that these cities are connected.
    3. Whether the results matched my expectations: Yes.

Main.java

```
1  class Main {
2    public static void main(String[] args) {
3
4      MyStack firstStack = new MyStack(10);
5
6      int[][] cityPath = {{1,2},{0,2,3},{0,1,4},{1},{2},{6},{5}};
7
8      System.out.println(firstStack.pathBetween(1, 3, cityPath));
9
10   }
11 }
12
```

```
> javac -classpath .:/run_dir/junit-4.12.jar:target/dependency/* -d . EmptyStackException.java FullStackException.java Main.java
> java -classpath .:/run_dir/junit-4.12.jar:target/dependency/* Main
current: 1
The stack from top to bottom is: 0
The stack from top to bottom is: 2 0
The stack from top to bottom is: 3 2 0
current: 3
There's a path between these two cities.
true
>
```

4. Test 4: stack.pathBetween(0, 3, cityPath);
    1. Why I chose this test case: Because I tested the reverse above, and I wanted to see if the method functions when given the reverse case.
    2. What I expected to happen: I expected there to be a path between cities.
    3. Whether the results matched my expectations: It does not work as expected because there's a path between 0 and 3 and it is 1. But above in Test 1, when I'm testing the opposite of these conditions, with the start city being 3 and end city being 0, It does give me the correct results. I have been trying to develop the solution to getting the reverse functionality of this operational.

Main.java

```
1  class Main {
2    public static void main(String[] args) {
3
4      MyStack firstStack = new MyStack(10);
5
6      int[][] cityPath = {{1,2},{0,2,3},{0,1,4},{1},{2},{6},{5}};
7
8      System.out.println(firstStack.pathBetween(0, 2, cityPath));
9
10   }
11 }
```

```
> javac -classpath .:/run_dir/junit-4.12.jar:target/dependency/* -d . EmptyStackException.java FullStackException.java Main.java
> java -classpath .:/run_dir/junit-4.12.jar:target/dependency/* Main
current: 0
There's no path between these two cities.
false
>
```

Code: link to Repl.it with all tests above -> https://repl.it/@NiemaWidaha/Stacks-during-class-version#Main.java

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

    MyStack firstStack = new MyStack(10);

    int[][] cityPath = {{1,2},{0,2,3},{0,1,4},{1},{2},{6},{5}};


    System.out.println(firstStack.pathBetween(3, 0, cityPath));
    System.out.println(firstStack.pathBetween(5, 6, cityPath));
    System.out.println(firstStack.pathBetween(1, 3, cityPath));
    System.out.println(firstStack.pathBetween(0, 3, cityPath));

  }
}

class MyStack {

  private int maxSize; //maximum size of the stack
  private int[] stackArray; //array where we store the stack items
  private int top; //tells us where the top of the stack is

  public MyStack(int max) {
```

```
  maxSize = max; //maximum size of the stack
  stackArray = new int[maxSize]; //initializing the stack
  top = -1; //currently nothing is in it
}

public boolean pathBetween(int start, int end, int[][] neighbors){

    int[] visited = new int[7];
    int totalVisited = 0; // how many cities we have visited
    MyStack toSee = new MyStack(25); // bigger than cities for safety


    int visitedValue = 0;

    // push your start city to the stack
    toSee.push(start);

    // somewhere in this loop, you'll need to check if where you are == end
    while(!toSee.isEmpty()){

      // pop from the stack to get current city
      int current = toSee.pop();
      System.out.println("current: " + current);

      // figure out if you've visited the current city
      // if the item hasnt been here
      for(int i = 0; i < visited.length; i++){

        if(visited[i] != current){

          //System.out.println("This hasn't been visited before");
          visitedValue = 0;

        } else {
            // System.out.println("This has been visited before");
            visitedValue = 1;
        } // ends if else
      } //  ends for loop

      // if you've visited it, them move on to the next iteration of the while loop
      // if you haven't visited it, add it to the visited & push its neighbors to the stack
      if(visitedValue == 0){

      visited[totalVisited] = current; // adding it to visited

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

        // System.out.println("Visited: " + visited[i]);
        if(visited[i] == end){
          System.out.println("There's a path between these two cities.");
          return true;
        }
      }

      totalVisited+=1; // to keep count of visited cities

      for(int j = 0; j < neighbors[current].length; j++){
             toSee.push(neighbors[current][j]);
            toSee.display();
      }  // ends for
      } else if (visitedValue == 1){
        continue;
      }
    } // ends while

    System.out.println("There's no path between these two cities.");
    return false;
} // ends pathBetween
```

```
  public void push(int newVal) throws FullStackException{
    if (isFull()) {
      throw new FullStackException("Can't push to a full stack!");
    }
    //want to add an item to the top of the stack
    top += 1;
    stackArray[top] = newVal;
  }

  public int pop() throws EmptyStackException{
    if (isEmpty()) {
      throw new EmptyStackException("Can't pop from an empty stack!");
    }

    //want to remove the item from the top of the stack
    //also want to return that item
    int popVal = stackArray[top];
    top -= 1;
    return popVal;
  }

  public boolean isEmpty() {
    //return true if the stack is empty
    //return false otherwise

    if (top == -1) {
      return true; //stack is empty
    } else {
      return false; //stack is not empty
    }
  }

  public boolean isFull() {
    //return true if the stack is full
    //return false otherwise

    if (top == maxSize-1) {
      return true; //stack is full bc maxsize-1 is the last inex
    } else {
      return false; //stack is not full
    }
  }

  public static void transfer(MyStack S, MyStack T) {
    //transfers items from stack S to stack T until S is empty or T is full

    while(!S.isEmpty() && !T.isFull()) {
      int popVal = S.pop();
      T.push(popVal);
    }
  }

  public void display() {
    //want to print the stack from top to bottom
    System.out.print("The stack from top to bottom is: ");
    for (int i = top; i >= 0; i -= 1) {
      System.out.print(stackArray[i]+" ");
    }
    System.out.println("");
  }
}
```