COMP 352: Data Structures and Algorithms

Assignment 2 on Stacks

Summer 2020, sections AA

May 27, 2020

Shadi Jiha #40131284

Question 1:

1. No it is not possible for all 3, one of them must have an O(n) complexity **sometimes**. for the max method you must keep track of the max value inside a variable in the stack class (to avoid looping through the stack each time). Something similar to the size(). Here’s an implementation in java:

public class StackWithMax {  
  
 private int[] data;  
 int pointer;  
 int size;  
 int max; // <---- This variable is important  
  
 /\* Unnecessary Implementation details hidden \*/  
  
 public void push(int number) {  
 data[pointer++] = number;  
 size++;  
  
 if (number > max) // <---- This part is really important for max()  
 max = number;  
 }

public void pop() {  
 int to\_remove = data[pointer];  
 data[pointer--] = 0;  
 size--;  
  
 // If the element to remove is also the max  
 // Then we have to recalculate the max  
 int current\_max = Integer.MIN\_VALUE;  
 if (to\_remove == max) {  
  
 for (int temp : data) {  
 if (temp > current\_max)  
 current\_max = temp;  
 }  
  
 max = current\_max;  
 }  
  
 }  
  
 public int max() {  
 return max;  
 }  
}

Question 2:

Note: My implementation works for both cases (all we have to do is change a Flag variable to swap cases)

1. Here’s the general idea: Instead of that array storing the data type directly (e.g. an integer), it will instead store a **Node** while will have a value, next, previous and index. So, this Node knows its position in the array, the position of the element after it and element before it. So, it is close to a Linked list but not quite there.

For case I, all we need to do is set a variable that holds the maximum slots a stack has and then when we push to the stack we check if there’s still place

For case I, we continue adding to the array as long as there are **null** elements in the array (which mean that there’s space)

1. Note: This implantation works for BOTH cases (all we need to do is change 1 variable and change from case I to case II). For a full Java implementation please check ANNEX I.

**class** **Node**

value: **int**

index: **int**

next : **int**

previous: **int**

constructor

**Input:** value, index, previous, next

**class** **SharedArrayStack**

N: **static** **const** **int** = 10

array: **static** SharedArrayStackNode array with size of N

NO\_MAX: **static const** **int** = -1

HALF\_SHARED\_ARRAY: **static const** **int** = N / 2

MAX\_ALLOCATION\_PER\_STACK: **static const** **int** = NO\_MAX // <-- This variable can be changed between NO\_MAX and HALF\_SHARED\_ARRAY to switch between case II and I

last: SharedArrayStackNode = **null**

**algorithm** **push**

**Input:** number to add to stack

**Output:** void

**throws:** throws and exception **if** the stack is full

// If the stack is full abort

**if** isFull() **then**

**throw** Exception

// First element in the stack

**if** last == **null** **then**

// Find a empty spot to put the new element

index\_to\_add = 0 // This variable holds the index of the spot

**while** array[index\_to\_add] != **null** **do**

index\_to\_add++

// When a spot is found, add a new node containing that value with a next of null and a previous of null

array[index\_to\_add] = new SharedArrayStack(number, index\_to\_add, **null**, **null**)

**else**

// Find a empty spot to put the new element

index\_to\_add = last.index;

**while** array[index\_to\_add] != **null** **do**

index\_to\_add++

array[last.index].next = index\_to\_add;

array[index\_to\_add] = new SharedArrayStackNode(number, index\_to\_add, last.index, **null**);

last = array[index\_to\_add];

**algorithm** **pop**

**Input:** void

**Output:** returns the element that was removed

**throws:** an exception if the stack is empty

**if** last == **null** **then**

**throw** Exception

temp\_last = last

array[last.index] = **null**

**if** temp\_last.previous == **null** **then**

// If the element removed is the last one of the stack

last = **null**

**else**

// Otherwise the last element is the previous of the last

last = array[temp\_last.previous]

**return** temp\_last.value

**algorithm** **size**

**Input:** void

**Output:** The size of the calling stack

count = 0

index = last.index;

// While there are still elements in the stack keep counting

**while** index != **null** **do**

count++

index = array[index].previous

**return** count

**algorithm** **isEmpty**

**Input:** void

**Output:** **return**s true **if** the stack is empty

**return** size() == 0

**algorithm** **isFull**

**Input:** void

**Output:** **return**s true **if** the stack **if** full

// If we don't have FAIRNESS (Case II)

**if** MAX\_ALLOCATION\_PER\_STACK == NO\_MAX **then**

// Loop through the array and see if there are any null elements

// If there's a single null element that means that the stack is not full

**foreach** element **in** array **do**

**if** element == **null** **then**

**return** false

**return** true;

// If we want FAIRNESS (Case I)

**else** **if** MAX\_ALLOCATION\_PER\_STACK == HALF\_SHARED\_ARRAY **then**

**if** size() == MAX\_ALLOCATION\_PER\_STACK **then**

**return** true

**return** false

**else**

**return** false

1. Complexity:
   1. Push() 🡪 O(n) because we have to keep going until we find an empty spot in the array. Worst case scenario, the empty spot is at the end of the array (N)
   2. Pop() 🡪 O(1) no loops are required
   3. Size() 🡪 O(n) we have to loop through the whole stack to find its size. Worst case scenario the stack has a size of N (the full array)
   4. isEmpty() 🡪 O(n) This function calls the size() function
   5. isFull() 🡪 O(n) This function either calls the size() function or loops though the whole array of N elements
2. Complexity:
   1. Push() 🡪 (1) best case scenario the empty spot is the beginning of the array
   2. Pop() 🡪(1) no loops are required
   3. Size() 🡪 (n) we have to loop through the whole stack to find its size. Worst case scenario the stack has a size of N (the full array)
   4. isEmpty() 🡪(n) This function calls the size() function
   5. isFull 🡪 (1) Best case scenario, the first element of the array if null so the function returns immediately that the stack is not full.
3. Yes it is possible, my algorithm is Stacks as we want but less than N. They will all store data in the same array. And both cases can be applied to all stacks.

Question 3:

Question 4:

**algorithm** **remove\_duplicates**

**Input**: array of integers

**Output**: array of integers without any duplicated values

stack = stack of integers

// Add element to the stack only if they are unique

**foreach** e **in** input **do**

add\_to\_stack(stack, e)

result: int[stack.size()]

// Covert the stack to an array in reverse order

**for** i = stack.size **to** i = 0 **do**

result[i] = stack.pop()

return result

**algorithm** **add\_to\_stack**

**Input**: stack of integers

**Input**: e: int

**Output**: void

**foreach** temp **in** stack **do**

**if** temp == e **then**

**return**

stack.push(e)

**return**

1. O(n) because add\_to\_stack() loops through the stack. Worst case scenario the stack has no duplicates and thus the stack has a size of N (same size as the array)
2. (n) because remove\_duplicates() still has to loop through the whole array which has a size of N

ANNEXE I

/\*\*  
 \*  
 \*/  
  
package driver;  
  
import java.util.Arrays;  
import java.util.EmptyStackException;  
  
public class SharedArrayStack {  
  
 private static final int N = 10;  
 private static final SharedArrayStackNode[] array = new SharedArrayStackNode[N];  
  
  
 public static final int NO\_MAX = -1;  
 public static final int HALF\_SHARED\_ARRAY = array.length / 2;  
  
 public static final int MAX\_ALLOCATION\_PER\_STACK = NO\_MAX;  
  
 // Member variables  
 private SharedArrayStackNode last;  
  
 public SharedArrayStack() {  
 last = null;  
 }  
  
 public void push(int number) throws RuntimeException {  
  
 // If the stack is full abort  
 if (isFull())  
 throw new RuntimeException("Cannot push an element to a full stack!");  
  
 // First element in the stack  
 if (last == null) {  
  
 int index\_to\_add = 0;  
 while (array[index\_to\_add] != null)  
 index\_to\_add++;  
  
 array[index\_to\_add] = new SharedArrayStackNode(number, index\_to\_add, null, null);  
 last = array[index\_to\_add];  
  
 } else {  
  
 int index\_to\_add = last.index();  
 while (array[index\_to\_add] != null)  
 index\_to\_add++;  
  
 array[last.index()].setNext(index\_to\_add);  
 array[index\_to\_add] = new SharedArrayStackNode(number, index\_to\_add, last.index(), null);  
  
 last = array[index\_to\_add];  
 }  
 }  
  
 public int pop() throws EmptyStackException {  
  
 // if stack is empty  
 if (last == null)  
 throw new EmptyStackException();  
  
 var temp\_last = new SharedArrayStackNode(last);  
 array[last.index()] = null;  
  
 if (temp\_last.previous() == null)  
 last = null;  
 else  
 last = array[temp\_last.previous()];  
  
 return temp\_last.value();  
 }  
  
 public int size() {  
  
 int count = 0;  
  
 Integer index = last.index();  
  
 while (index != null) {  
 count++;  
 index = array[index].previous();  
 }  
  
 return count;  
 }  
  
 public int top() throws EmptyStackException {  
  
 if (last == null)  
 throw new EmptyStackException();  
  
 return array[last.index()].value();  
 }  
  
 public boolean isEmpty() {  
 return size() == 0;  
 }  
  
 public boolean isFull() {  
  
 // If we don't have FAIRNESS (Case II)  
 if (MAX\_ALLOCATION\_PER\_STACK == NO\_MAX) {  
  
 // Loop through the array and see if there are any null elements  
 // If there's a single null element that means that the stack is not full  
 for (var e : array)  
 if (e == null)  
 return false;  
  
 return true;  
 }  
  
 // If we want FAIRNESS (Case I)  
 else if (MAX\_ALLOCATION\_PER\_STACK == HALF\_SHARED\_ARRAY) {  
 if (size() == MAX\_ALLOCATION\_PER\_STACK)  
 return true;  
 return false;  
 } else {  
 return false;  
 }  
 }  
  
 public String toString() {  
  
 StringBuilder builder = new StringBuilder();  
 builder.append("[");  
  
 Integer index = last.index();  
  
 while (index != null) {  
  
 var element = array[index];  
  
 builder.append(element.value());  
  
 // Don't add ", " if it is the last element  
 if (element.previous() != null)  
 builder.append(", ");  
  
 index = element.previous();  
 }  
  
 builder.append("]");  
  
 return builder.toString();  
 }  
  
 public static void printArray() {  
 System.out.println(Arrays.toString(SharedArrayStack.array));  
 }  
  
 private static class SharedArrayStackNode {  
  
 private int value;  
 private int index;  
 private Integer next;  
 private Integer previous;  
  
 private SharedArrayStackNode(int value, int index, Integer previous\_index, Integer next\_index) {  
 this.value = value;  
 this.index = index;  
 this.previous = previous\_index;  
 this.next = next\_index;  
 }  
  
 private SharedArrayStackNode(final SharedArrayStackNode other) {  
 value = other.value;  
 index = other.index;  
 previous = other.previous;  
 next = other.next;  
 }  
  
 private boolean hasNext() {  
 return next != null;  
 }  
  
 private boolean hasPrevious() {  
 return previous != null;  
 }  
  
 private Integer next() {  
 return next;  
 }  
  
 private Integer previous() {  
 return previous;  
 }  
  
 public int index() {  
 return index;  
 }  
  
 private int value() {  
 return value;  
 }  
  
 private void setNext(Integer o) {  
 next = o;  
 }  
  
 private void setPrevious(Integer o) {  
 previous = o;  
 }  
  
 private void setIndex(int i) {  
 index = i;  
 }  
  
 private void setValue(int value) {  
 this.value = value;  
 }  
  
 public String toString() {  
 return value + "";  
 }  
 }  
}