

DNHI Homework 3 Solutions List, Stacs and Queues

Problem 1

Given the IntegerQueue ADT below state the return value and show the content of the, initially empty, queue of Integer objects after each of the following operations. If any of the operations below is invalid or would cause program to crash, state it and explain what is wrong with it (then proceed with the next step ignoring the invalid line). Assume that queue is a reference of type IntegerQueue.

Informal Specification The IntegerQueue contains a (possibly empty) collection of objects of type Integer. The queue supports the following operations:

insert/enqueue This operation adds a Integer object, given as a parameter, to the end of the queue of integers.

remove/dequeue This operation removes and returns a Integer object from the front of the queue of integers. If the queue is empty, null should be returned.

toString This operation produces a String that contains all Integer objects stored in the queue from font to back separated by a single comma and a single space. If the queue is empty, an empty string should be returned.

```
queue.insert( 15 );
queue.insert( 3 );
queue.insert( -15 );
queue.insert( 35 );
queue.remove();
queue.remove();
queue.remove();
queue.insert( 13 );
queue.insert( 13 );
queue.remove();
queue.remove();
queue.remove();
queue.remove();
queue.insert( 3 );
```

A Assume that the implementation is array based and follows the efficiency ideas that we discussed in class. Assume that the initial capacity of the array to store the queue is equal to 4 and that its size is doubled whenever the array runs out of room.

```
queue.insert(15);
          [ 15,
                                       return value: none
queue.insert(3);
                                       return value: none
         [ 15,
queue.insert( -15);
                                       return value: none
         [ 15,
                   3. -15.
queue.insert(35);
         [ 15,
                                       return value: none
                   3, -15, 351
queue.remove();
                   3, -15, 351
                                       return value: 15
         [
queue.remove();
                                       return value: 3
         [
                     ,-15, 351
queue.remove();
                                       return value: -15
                             351
         ſ
queue.insert(13);
                             35]
                                       return value: none
         [ 13,
queue.remove();
```

```
[ 13, , , ] return value: 35

queue.remove();
        [ , , , ] return value: 13

queue.remove();
        [ , , , ] return value: null (or an exception thrown)

queue.insert( 3 );
        [ , 3, , ] return value: none

or
        [ 3, , , ] return value: none
```

B Assume that the implementation is reference based and the head reference points of the front of the queue.

```
queue.insert(15);
            head -> [15]-> null
                                                  return value: none
queue.insert(3);
            head \rightarrow [15]\rightarrow [3]\rightarrow null
                                                            return value: none
queue.insert( -15);
            head \rightarrow [15]\rightarrow [3]\rightarrow [-15]\rightarrow null
                                                                       return value: none
queue.insert(35);
            head \rightarrow [15] \rightarrow [3] \rightarrow [-15] \rightarrow [35] \rightarrow null
                                                                                  return value: none
queue.remove();
            head \rightarrow [3]\rightarrow [-15]\rightarrow [35]\rightarrow null
                                                                       return value: 15
queue.remove();
            head \rightarrow [-15]-> [35]-> null
                                                              return value: 3
queue.remove();
            head -> [35]-> null
                                                 return value: -15
queue.insert(13);
            head \rightarrow [35]\rightarrow [13]\rightarrow null
                                                            return value: none
queue.remove();
                                                 return value: 35
            head -> [13]-> null
queue.remove();
            head -> null
                                      return value: 13
queue.remove();
            head -> null
                                      return value: null (or an exception thrown)
queue.insert(3);
            head \rightarrow [3] \rightarrow null
                                                 return value: none
```

C Discuss efficiency/performance of each the insert and remove operations for each implementation (for the reference based implementation, consider the difference in performance if you have an additional reference to the end of the queue).

Array based implementation:

assumption: we know the indexes of the first and last elements in the queue insertion is O(1) as long as there is room in the array, when array needs to be resized, it is O(N) removal is always O(1)

Reference based implementation:

assumption: we only have a reference to the head and it points to the first element in the queue insertion is O(N) since we need to traverse the queue to get to the end remove is O(1) since we have a reference to the first element

Reference based implementation:

assumption: we have references to the front and back of the queue, front points to the first element and back points to the last element insertion is O(1) since have a reference to the last element remove is O(1) since we have a reference to the first element



Problem 2

Given the CharStack ADT that we discussed in class show the content of the, initially empty, stack of Character objects after each of the following operations. If any of the operations below is invalid or would cause program to crash, state it and explain what is wrong with it. Assume that stack is a reference of type CharStack.

Informal Specification The CharStack contains a (possibly empty) collection of objects of type Character. The stack supports the following operations:

insert/push This operation adds a Character object, given as a parameter, to the top of the stack of characters.

remove/pop This operation removes and returns a Character object from the top of the stack of characters.

peek This operation returns a Character object from the top of the stack of characters.

toString This operation produces a meaningful String representation of the stack.

```
stack.push('c');
stack.push( new Character('s') );
stack pop();
char p = 's';
stack.push( p );
stack.push( p );
stack.push( new Character('1') );
stack.push();
stack.peek();
stack.pop();
stack.push('%');
stack.push('%');
stack.push('A');
stack.push('X');
stack.push('X');
stack.pop();
stack.pop();
```

A Assume that the implementation is array based and follows the efficiency ideas that we discussed in class. Assume that the initial capacity of the array to store the stack is equal to 4 and that its size is doubled whenever the array runs out of room.

```
stack.push('c');
                                      return value: none
          [ c,
stack.push( new Character('s') );
                                     return value: none
          [ c, s,
stack pop();
                                      return value: 's'
          [ c,
char p = 's'; stack.push( p );
          [ c, s,
                                     return value: none
stack.push( p );
                           -1
                                   return value: none
          [ c, s, s,
stack.push( new Character('1') );
          [c, s, s, 1]
                                 return value: none
stack.peek();
                                  return value: '1'
            c, s, s, 1]
stack.pop();
                                   return value: '1'
          [ c, s, s,
stack.push('%');
          [ c, s, s, %]
                                  return value: none
stack.peek();
                                  return value: '%'
                s, s, %]
stack.push('A');
                s, s, %, A,
                                                       return value: none
          [ c,
stack.push('X');
```



```
[ c, s, s, %, A, X, , ] return value: none stack.pop();
        [ c, s, s, %, A, , , ] return value: 'X' stack.pop();
        [ c, s, s, %, , , , ] return value: 'A'
```

B Assume that the implementation is reference based. Where should the head reference point to (bottom of the stack or top of the stack).

The head reference needs to point to the top of the stack in order to provide efficient implementation.

```
stack.push('c');
              head \rightarrow [c] \rightarrow null
                                                      return value: none
stack.push( new Character('s') );
              head \rightarrow [s]\rightarrow [c]\rightarrow null
                                                                return value: none
stack pop();
              head \rightarrow [c] \rightarrow null
                                                     return value: 's'
char p = 's'; stack.push( p );
              head \rightarrow [s] \rightarrow [c] \rightarrow null
                                                                return value: none
stack.push( p );
                                                                           return value: none
              head \rightarrow [s]\rightarrow [c]\rightarrow null
stack.push( new Character('1') );
              head \rightarrow [1] \rightarrow [s] \rightarrow [c] \rightarrow null
                                                                                      return value: none
stack.peek();
              head \rightarrow [1] \rightarrow [s] \rightarrow [c] \rightarrow null
                                                                                      return value: '1'
stack.pop();
            head \rightarrow [s]\rightarrow [c]\rightarrow null
                                                                       return value: '1'
stack.push('%');
            head \rightarrow [%]\rightarrow [s]\rightarrow [c]\rightarrow null
                                                                                  return value: none
stack.peek();
            head \rightarrow [%]\rightarrow [s]\rightarrow [c]\rightarrow null
                                                                                  return value: '%'
stack.push('A');
            head \rightarrow [A]\rightarrow [%]\rightarrow [s]\rightarrow [c]\rightarrow null
                                                                                            return value: none
stack.push('X');
            head \rightarrow [X]\rightarrow [A]\rightarrow [%]\rightarrow [s]\rightarrow [s]\rightarrow null
                                                                                                       return value: none
stack.pop();
            head \rightarrow [A]\rightarrow [%]\rightarrow [s]\rightarrow [c]\rightarrow null
                                                                                            return value: 'X'
stack.pop();
            head \rightarrow [%]\rightarrow [s]\rightarrow [c]\rightarrow null
                                                                                  return value: 'A'
```

C Discuss efficiency/performance of each the insert/push and remove/pop operations for each implementation.

Array based implementation:

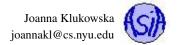
assumption: we know the indexes of the top element in the stack, bottom of the stack is anchored to the zeroth index insertion is O(1) as long as there is room in the array, when array needs to be resized, it is O(N) removal is always O(1)

Reference based implementation:

assumption: we have a reference to the head and it points to the top element in the stack insertion is O(1) since we have a reference to the top element remove is O(1) since we have a reference to the top element

Problem 3

The GenericList interface and its GenericLinkedList implementation used in class (see the source code for lecture 5) provide an insertion method that always adds a new node to the back of the list. Assuming that the data item that is stored in the GenericNode object implements the Comparable interface (i.e. has standard definitions of comp



void orderedInsert(T item)

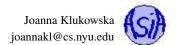
that adds a new node to the list in a sorted order (from smallest to largest).

```
2 *
 3 * Inserts a T object keeping the order of the list
4 4
5 * @param item the item to insert
 7 public void insert(T item) {
    //add node only if item is not null
    //(we do not want to have nodes storing null reference as the data value )
10
    if (item == null )
11
      return;
12
13
    //create new node and set its next reference to null
14
    GenericNode<T> newNode = new GenericNode <T> ( item, null );
15
16
    //special case for an empty list
    if (head == null )
17
      head = newNode:
18
    //special case when new node becomes first node
19
    else if (newNode.compareTo(head) < 0) {</pre>
20
21
      newNode.setNext(head);
22
      head = newNode;
    }else{
23
24
      //create the current reference and advance it to the last node
      GenericNode<T> current = head;
25
      while (current.getNext() != null && newNode.compareTo(current.getNext())>0)
26
          current = current.getNext();
2.7
28
29
      if(current.getNext() ==null)
        //make the last node point to the new last node
30
31
        current.setNext(newNode);
32
      else{
        //insert the new node after the current
33
        newNode.setNext(current.getNext());
34
        current.setNext(newNode);
35
36
37
    numOfElements++;
38
39 }
```

Problem 4

Answer the following true/false questions. Explain your answers.

- F A stack is a first in, first out structure.
- F A queue is a last in, first out structure.
- F A Java interface must have at least one defined constructor.
- T A class that implements an interface has to implement ALL methods listed in the interface.
- T In a non-empty stack, the item that has been in the stack the longest is at the bottom of the stack.
- F A recursive solution to a problem is always better than an iterative solution.
- T A general case of a valid recursive algorithm must eventually reduce to a base case.
- F Recursive methods should be used whenever execution speed is critical.
- T If you enqueue 5 elements into an empty queue, and then perform the dequeue operation 5 times, the queue will be empty again.
- F If you push 5 elements into an empty stack, and then perform the peek operation 5 times, the stack will be empty again.
- T In a singly linked list based implementation of a queue, at least one of the operations that add or remove an item has to traverse all the elements in the queue.



• F In a singly linked list based implementation of a stack, , at least one of the operations that add or remove an item has to traverse all the elements in the stack.

Problem 5

In a circular singly linked list, the next reference of the last node points back to the first node (instead of null). Write a method that counts the number of nodes in a circular singly lined list. Assume that there is a head reference that points to the first node in the list.

```
boolean isCircular ()
if (head == null ) return false
return isCircular ( head.next )

boolean isCircular ( Node n )
if (n == null ) return false
if (n == head ) return true
return isCircular (n.next )
```

Note: you could also use an iterative solution.

Problem 6

A doubly linked list mantains a reference to the head and to the tail of the list (first and last nodes, respectively). Each node has a next reference and a previous reference that point to the nodes after it and before it. Write the following methods that operate on a doubly linked list. Make sure that the list is still valid after the operation is performed. Make sure that your operations can handle special cases of an empty list and one element list.

A Write a method that adds a node at the beginning of the doubly linked list

```
void addFront ( data )
Node n = new Node ( data )

if ( head == null )
head = n
tail = n

else

n.next = head
head.previous = n
head = n
```

B Write a method that adds a node at the end of the doubly linked list

```
void addBack ( data )

Node n = new Node ( data )

if ( head == null )

head = n

tail = n

else

n.previous = tail

tail.next = n

tail = n
```

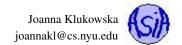
C Write a method that removes a node at the beginning of the doubly linked list

```
typeOfData removeFront ( )
if ( head == null ) //zero nodes

return null

if ( head == tail ) //only one node

tmp = head.data
```



```
head = null
6
7
        tail = null
8
        return tmp
9
      else
10
        tmp = head.data
        head = head.next
11
12
        head.previous = null
        return tmp
13
```

D Write a method that removes a node at the end of the doubly linked list

```
typeOfData removeFront ( )
      if ( head == null ) //zero nodes
        return null
      if ( head == tail ) //only one node
        tmp = head.data
        head = null
6
        tail = null
8
        return tmp
9
      else
        tmp = tail.data
10
        tail = tail.previous
11
12
        tail.next = null
13
        return tmp
```

Problem 7

Write a method of a LinedList class that computes and returns the number of nodes stored in the linked list. Assume that there is a data field called head that contains the reference to the first node. Provide both iterative and recursive implementations.

```
int size ()
Node current = head
int counter = 0
while current != null
counter ++
return counter

int size ()
return size ( head )

int size ( Node n )
if (n == null ) return 0
else return 1 + size (n.next )
```

Problem 8

Write a method of a LinkedList class that computes and returns the sum of the values stored in the nodes. Assume that the node definition is as follows:

```
class Node {
  int data;
  Node next;
}

int sum ()
  Node current = head
```



```
int sum = 0
while current != null
sum += current.data
return sum
```

Problem 9

Convert the following infix expressions to postfix and to prefix.

Problem 10

Evaluate the following postfix expressions for the following values of the variables: a = 7, b = 3, c = 12, d = -5 and e = 1.

- a b c + -8• a b c - d * + 52• a b + c - d e * + -7

Extra Challenge 1¹

Write a method of a linked list that performs a "partition" of the values stored in the list around a value *. All nodes less than * should come before all nodes greater than or equal to *.

Extra Challenge 2

Implement a method to check if a linked list is a palindrome (in the broad sense: first element is the same as last, etc). You should pass through the list only once. You may use other data structures. You may assume that the length of the list is known.

¹This is a challenging problem that maybe useful for a job interview. It is not the type of problem that you may see on the exam.