Joanna Klukowska
joannakl@cs.nyu.edu

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

Joanna Klukowska
joannakl@cs.nyu.edu

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

Joanna Klukowska
joannakl@cs.nyu.edu

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.peek();
stack.pop();
stack.push('%');
stack.peek();
stack.push('A');
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');

`[ c, , , ]` return value: none

stack.push(new Character('s'));

`[ c, s, , ]` return value: none

stack pop();

`[ c, , , ]` return value: `'s'`

char p = 's'; stack.push(p);

`[ c, s, , ]` return value: none

stack.push(p);

`[ c, s, s, ]` return value: none

stack.push(new Character('1'));

`[ c, s, s, 1]` return value: none

stack.peek();

`[ c, s, s, 1]` return value: `'1'`

stack.pop();

`[ c, s, s, ]` return value: `'1'`

stack.push('%');

`[ c, s, s, %]` return value: none

stack.peek();

`[ c, s, s, %]` return value: `'%'`

stack.push('A');

`[ c, s, s, %, A, , , ]` return value: none

stack.push('X');

Joanna Klukowska
joannakl@cs.nyu.edu

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

```
void orderedInsert( T item )
```

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

```
/**
*
*Inserts a T object keeping the order of the list
*
*@param item the item to insert
*/
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 )
  if (item == null )
    return;

  //create new node and set its next reference to null
  GenericNode<T> newNode = new GenericNode <T> ( item, null );

  //special case for an empty list
  if (head == null )
    head = newNode;
  //special case when new node becomes first node
  else if(newNode.compareTo(head)<0){
    newNode.setNext(head);
    head = newNode;
  }else{
    //create the current reference and advance it to the last node
    GenericNode<T> current = head;
    while (current.getNext() != null && newNode.compareTo(current.getNext())>0)
        current = current.getNext();

    if(current.getNext()==null)
      //make the last node point to the new last node
      current.setNext(newNode);
    else{
      //insert the new node after the current
      newNode.setNext(current.getNext());
      current.setNext(newNode);
    }
  }
  numOfElements++;
}
```

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.

Joanna Klukowska
joannakl@cs.nyu.edu

- 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.

```
int size ()
  if (head == null ) return 0;
  return 1 + size ( head.next )


int size ( Node n )
  //this should not happen in a circular list
  if (n == null ) throw new IllegalStateException () ;
  if (n == head ) return 0;
  return 1 + size (n.next )

```

Note: you could also use an iterative solution.

Problem 5a

Write a method that detects if a given linked list is a circular list or if it ends with the traditional **null** reference.

```
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 )
  if (data == null ) throw new IllegalArgumentException("null parameter detected");
  Node n = new Node ( data )
  if ( head == null )
    head = n
    tail = n
  else
    n.next = head
    head.previous = n
    head = n
```

Joanna Klukowska
joannakl@cs.nyu.edu

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

```
void addBack ( data )
  if (data == null ) throw new IllegalArgumentException("null parameter detected");
  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
    tail = null
    return tmp
  else
    tmp = head.data
    head = head.next
    head.previous = null
    return tmp
```

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
    tail = null
    return tmp
  else
    tmp = tail.data
    tail = tail.previous
    tail.next = null
    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 )
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
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.

- $a - b + c$
  $a\,b\,-\,c+$ $\quad + - a\,b\,c$
- $a - (b/c \times d)$
  $a\,b\,c\,/\,d\,*\,-$ $\quad - a * / b\,c\,d$
- $(a - b/c) \times d$
  $a\,b\,c\,/\,-\,d\,*$ $\quad * - a\,/\,b\,c\,d$
- $a - b/(c \times d)$
  $a\,b\,c\,d\,*\,/\,-$ $\quad - a\,/\,b * c\,d$
- $a - (b + c \times d)/e$
  $a\,b\,c\,d\,*\,+\,e\,/\,-$ $\quad - a\,/\,+\,b * c\,d\,e$
- $a - (b + c) \times d/e$
  $a\,b\,c\,+\,d\,*\,e\,/\,-$ $\quad - a\,/\,*\,+\,b\,c\,d\,e$
- $a - (b + c) \times (d/e)$
  $a\,b\,c\,+\,d\,e\,/\,*\,-$ $\quad - a * + b\,c\,/\,d\,e$
- $(a - b) + c \times d/e$
  $a\,b\,-\,c\,d\,*\,e\,/\,+$ $\quad + - a\,b\,/\,*\,c\,d\,e$

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

Joanna Klukowska
joannakl@cs.nyu.edu

Extra Challenge 1[1]

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

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.

[1] This 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.