California State University, Sacramento
College of Engineering and Computer Science

Computer Science 130: Data Structures and Algorithm Analysis

Assignment #1 – Proper Stack and Queue

Revision 2

Overview

For this assignment, you are going to create a well-written, efficient, and versatile Linked-List class. You will then use this Linked-List to create an equally efficient Stack and Queue.

Part 1: Linked List Class

The first class you need to write an efficient Linked-List class. You will use this class to construct your Stack and Queue (and use it throughout this semester). So, do a good job on it.

Your class **must** be O(1) in all cases. So, you **must** maintain a reference (link) to the tail.

For the class, you will need an internal node class. Use the following definition. Notice that it will be defined using "object" – this will work for all data types.

```
class Node
    public Object Value
    public Node Next
end class
```

The following is the required interface for class. This is not a deque, but it has many of the same features. It is a singly-linked list.

public class LinkedList		
string	**About()**	Returns text about you – the author of this class.
void	**AddHead(Object item)**	Adds an object to the head of the list. This must be O(1).
void	**AddTail(Object item)**	Adds an object to the tail of the list. This must be O(1).
object	**RemoveHead()**	Removed an object from the head of the list. This must be O(1).
object	**PeekHead()**	Returns the value from the head of the list. It returns the value, not the node itself. This must be O(1).

Part 2: The Stack and Queue

Once you have your Linked-List class working, it is time to create a stack and queue class. Again, you **must** use your class from Part 1. If you use another data structure (such as an array or one of the language libraries built-in classes), you will receive a zero.

The Stack Class

The following is the interface for the Stack Class. This will be fairly easy to write – since it wraps around your Linked-List class. Don't inherit. The objects from the Linked-List will be cast to doubles.

`public class Stack`		
`string`	**`About()`**	Returns text about you – the author of this class.
`void`	**`Push(double item)`**	Pushes a double (number) onto the stack.
`double`	**`Pop()`**	Pops (removes) an item from the top of the stack.
`double`	**`Peek()`**	Returns the value on the top of the stack. Do not return the node itself.
`boolean`	**`IsEmpty()`**	Returns true of the stack is empty.

The Queue Class

The following is the interface for the Queue Class. This will be fairly easy to write – since it wraps around your Linked-List class. Don't inherit. The objects from the Linked-List will be cast to strings.

`public class Queue`		
`string`	**`About()`**	Returns text about you – the author of this class.
`void`	**`Enqueue(string item)`**	Enqueues a string onto the queue.
`string`	**`Dequeue()`**	Dequeues (removes) a string from the front of the queue.
`string`	**`Peek()`**	Returns the value on the front of the queue. Do not return the node itself.
`boolean`	**`IsEmpty()`**	Returns true of the stack is empty.

Part 3: The Pool

Only start this section once you have thoroughly tested your Linked-List, Stack, and Queue.

Now that you have a well-written and efficient Linked List, you can make it "heap friendly". You might want to make a backup of your Linked-List class, since it is going to be modified. You are going to add a "pool" to your class. You are welcome to implement this any way you like: an array or a linked list. I do recommend using a linked list, the logic is already there with the Node class.

Your Pool should contain at least 10 nodes.

Using a Linked-List (recommended)

In your Linked-List class, you maintain a reference (link) to both the head and tail. You probably named them as such! Well, you will create another one called "pool". It will function as the head of the pool of nodes – and we can link them together like before.

The logic is as follows:

- You will need to maintain an integer that tracks the *size* of the pool.
- When a value is added to the list, check the pool. If there is a node, remove it. You will use this node. If no node exists ("pool" is null), then you create one using *new*.
- When a value is removed, the node (that contained it) will either be put back into the pool or discarded. This is the point of the *size* variable. If the maximum hasn't reached, store it back on the head of the pool. If the maximum was reached, let the node get cleaned up by the Garbage Collector.
- **Note:** Whenever you store a node back into the Pool, set the nodes Value field to null. Otherwise, you can create loitering.

Using an Array

The logic for an array is pretty much the same as above. You will simply maintain a partially filled array.

Testing is Tricky

So, how do you know the pool is working? Well, for testing, it might be a good idea to add some "strategic" print statements. You can, for example, print a message to the screen whenever a new node is created or a node allowed to be discarded.

Requirements

You must write your linked-list, stack, and queue yourself.

Do not use any built-in library classes for stacks, arraylists, templates, etc…
If you use any of these, you will receive a zero. No exceptions. No resubmissions.

- This **must** be completely all your code. If you share your solution with another student or re-use code from another class, you will receive a zero.
- You **must** write your own Linked-List, Stack and Queue classes. They must all be **O(1)** for enqueue/dequeue and push/pop operations. This means your Linked-List class must be O(1).
- Your Stack must use your Linked-List class.
- Your Queue must use your Linked-List class.
- You may use any programming language you are comfortable with. I strongly recommend not using C (C++, Java, C#, Visual Basic are all good choices).
- Create some excellent testing for your class.

Due Date

Due **September 26, 2021** by 11:59 pm.

Given you already have developed excellent programming skills in CSc 20, this shouldn't be a difficult assignment. **Do not send it to canvas.**

E-Mail the following to dcook@csus.edu:

- The source code.
- The main program that runs the tests.
- Output generated by your tests

⚠ **The e-mail server will delete all attachments that have file extensions it deems dangerous. This includes .py, .exe, and many more.**

So, please send a ZIP File containing all your files.

Some Helpful Pseudocode

LinkedList Pseudocode

```
class Linked List
    private Node head
    private Node tail
    private Node pool   ... Don't worry until Part 3

    ... The methods go here, of course.

end class
```

AddHead Pseudocode

```
procedure AddHead(object value)
    add = new Node(value)    ... Create a new node. This will be modified in Part 3
                             ... to use the pool.

    if the list is empty     ... is head null?
        head = add           ... Link both head and tail to the new node
        tail = add
    else
        add.next = head      ... Link the new node to the head
        head = add           ... The head is now the new node
    end if

end procedure
```

AddTail Pseudocode

```
procedure AddTail(object value)
    add = new Node(value)  ... Create a new node. This will be modified in Part 3
                           ... to use the pool.

    if list is empty       ... is head/tail null?
        head = add
        tail = add
    else
        tail.next = add     ... Link the old tail to the new node
        tail = add          ... The new node is the new tail
    end if

end procedure
```

RemoveHead Pseudocode

```
function object RemoveHead()
    old = head              ... Save a reference to the old head.
    result = old.value      ... Save the value to return later

    head = head.next        ... The "second" item is the new head.

    if head is null         ... Just removed the last node
        tail = null         ... Remove the tail reference
    end if

    ... In part 3, you will add the old head to the pool (if it fits)
    ... Remember to set the value and next fields to null.

    return result
end procedure
```

Part 3 Helpful Pseudocode

addPool Pseudocode

```
procedure addPool(Node add)

    if pool count is not at the maximum value
        add.next = pool    ... Link the kept node to the head of the pool chain
        pool = add         ... The head is now the new node
        increment pool count
    end if

end procedure
```

removepool Pseudocode

```
function object removePool()

    if pool is null         ... The pool is empty
        result = new Node()
    else
        result = pool       ... Remove a node
        pool = pool.next    ... The "second" item is the new pool head
        decrement pool count
    end if

    return result
end procedure
```