Examine List

Count is the number of elements that are currently in the list.
Capacity is the number of elements that can be stored in the list before resizing is required.
2) When count exceeds the current capacity then .NET framework resizes the list.
3) The increase in the lists capacity is double the previous capacity.
4) Resizing a list has trade offs. Memory is a resource and needs managing. Resizing the list after an element has been added to the list could increase the overhead of the application. The overhead could be increased hardware resources and or more code to write and maintain. (This question is a little vague. Are we meant to answer why the underlying .NET framework doesn't increase the capacity or why we should adjust the capacity of the list!)
5) No.
6) Arrays are faster when accessing elements of the array because of the predicted size of the array at compile time. Direct access to a specific element in the array using its index. Thus, if you know ahead of time what you want to store in memory then an array has a performance advantage over a list.

Examine Queue			
		FiFo queue	`FIFOQueue.Count`
ICA öppnar och kön till kassan är tom	`Queue<string> FIFOQueue = new();`		0
Kalle ställer sig i kön	`FIFOQueue.Enqueue(”Kalle”);`	Kalle	1
Greta ställer sig i kön	`FIFOQueue.Enqueue(”Greta”);`	Greta, Kalle	2
Kalle blir expedierad och lämnar kön	`FIFOQueue.Dequeue();`	Greta	1
Stina ställer sig i kön	`FIFOQueue.Enqueue(”Stina”);`	Stina, Greta	2
Greta blir expedierad och lämnar kön	`FIFOQueue.Dequeue();`	Stina	1
Olle ställer sig i kö	`FIFOQueue.Enqueue(”Olle”);`	Olle, Stina	2

<table>
<tr><td colspan="4">Examine Stack</td></tr>
<tr><td></td><td></td><td>FiLo queue</td><td>stack.Count</td></tr>
<tr><td>ICA öppnar och kön till kassan är tom</td><td>Stack&lt;string> stack = new();</td><td></td><td>0</td></tr>
<tr><td>Kalle ställer sig i kön</td><td>stack.Push(”Kalle”);</td><td>Kalle</td><td>1</td></tr>
<tr><td>Greta ställer sig i kön</td><td>stack.push(”Greta”);</td><td>Greta, Kalle</td><td>2</td></tr>
<tr><td>Kalle blir expedierad och lämnar kön</td><td>stack.pop();</td><td>Kalle</td><td>1</td></tr>
<tr><td>Stina ställer sig i kö n</td><td>stack.push(”Stina”);</td><td>Stina, Kalle</td><td>2</td></tr>
<tr><td>Greta blir expedierad och lämnar kön</td><td>stack.pop();</td><td>Kalle</td><td>1</td></tr>
<tr><td>Olle ställer sig i kö</td><td>stack.push(”Olle”);</td><td>Olle, Kalle</td><td>2</td></tr>
<tr><td colspan="4">A queue based on a stack means that the first person in the queue, ”Kalle”, will remain in the queue for as long the queue continues to grow. Thus if someone arrives at the last minute they will be prioritised and served first. While, Kalle, who was early, will be penalised and as a consequence will receive poor service from ICA.</td></tr>
</table>

Check Parenthesis				
Input	Examine		FILO Queue	Stack.count
([{}]({})		`Stack<char> stack = new();`		0
[{}]({})	’(’	stack.push(’)’);	)	1
{}]({})	’[’	stack.push(’]’);	],)	2
}]({})	’{’	stack.push(’}’);	},],)	3

]({})	’}’	`if(stack.TryPeek(out char testCharacter))`
		`    if(testCharacter == ’}’)`
		`        stack.Pop();`
			],)	2

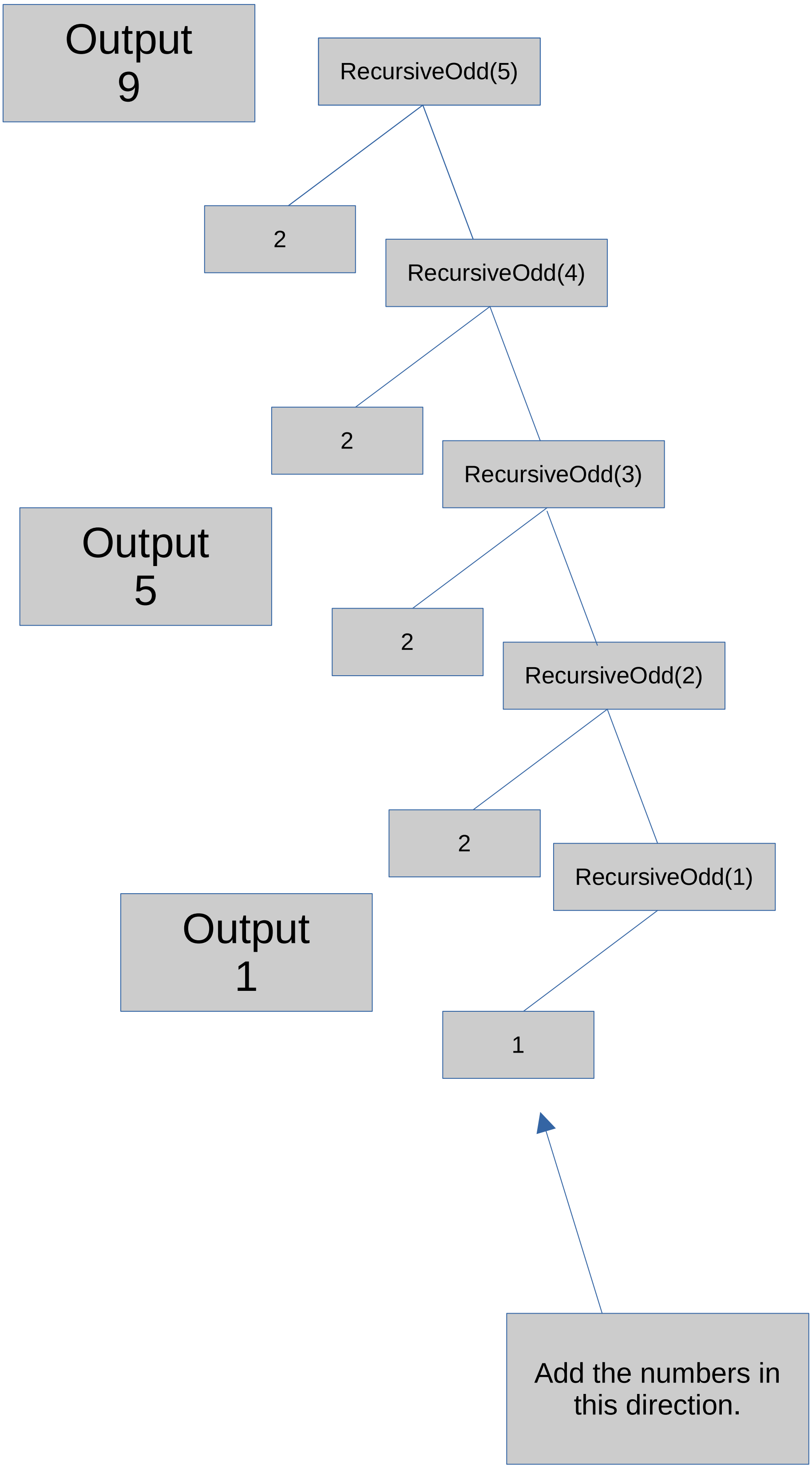
({})	’]’	`if(stack.TryPeek(out char testCharacter))`
		`    if(testCharacter == ’]’)`
		`        stack.Pop();`
			)	1

{})	’(’	stack.push(’)’);	),)	2
})	’{’	stack.push(’}’);	},),)	3

)	’}’	`if(stack.TryPeek(out char testCharacter))`
		`    if(testCharacter == ’}’)`
		`        stack.Pop();`
			),)	2

	’)’	`if(stack.TryPeek(out char testCharacter))`
		`    if(testCharacter == ’)’)`
		`        stack.Pop();`
			)	1

Note that the FILO queue is not empty after reading the last parenthesis in the input. Thus ([{}]({}) fails the paranthesis test.				
Note: In my implementation when a left handed parenthesis is detected a right handed parenthesis is pushed onto the stack. This makes easier to compare a right handed parenthesis to what is at the head of the stack.				



Output
9

Output
5

Output
1

Add the numbers in this direction.