1. Hur fungerar ***stacken*** och ***heapen*** ? Förklara gärna med exempel eller skiss på dess grundläggande funktion

Stacken är ett minnesarray... Det är en LIFO (Last In First Out) datastruktur. Man kan addera eller radera från den enbart från högsta adel av minnes strukturen.

2. Vad är ***Value Types*** respektive ***Reference Types*** och vad skiljer dem åt?

Value Types har sin data på dennes tilldelad minnesplats
Reference Types har en pekare som hänvisar till ett annat minnes del.

3. Följande metoder ( ***se bild nedan*** ) genererar olika svar. Den första returnerar 3, den andra returnerar 4, varför?

```
public int ReturnValue()
{
    int x = new int();
    x = 3;
    int y = new int();
    y = x;
    y = 4;
    return x;
}

0 references
public int ReturnValue2()
{
    MyInt x = new MyInt();
    x.MyValue = 3;
    MyInt y = new MyInt();
    y = x;
    y.MyValue = 4;
    return x.MyValue;
}
```

I första metoden ReturnValue() är x deklarerat och  tilldelad värdet 3  (x=3).
I andra metodenReturnValue2() sker en deklaration och en tilldelning i raden y.MyValue = 4; Detta ger y värdet 4 och därmed blir x lika med 4.

### *Övning 1: ExamineList()*

En lista är en ***abstrakt datastruktur*** som kan implementeras på flera olika sätt. Vid det här laget bör ni vara bekanta med ***listklassen*** . Till skillnad från ***arrayer*** har ***listor*** inte en förbestämd storlek utan storleken ökar i takt med att antalet element i listan ökar. ***Listklassen*** har dock en underliggande ***array*** som ni nu ska undersöka. För att se storleken på listans underliggande array används ***Capacitymetoden*** i ***Listklassen*** .

1. Skriv klart implementationen av ***ExamineList-metoden*** så att undersökningen blir genomförbar.
2. När ökar listans kapacitet? (Alltså den underliggande arrayens storlek)
3. Med hur mycket ökar kapaciteten?
4. Varför ökar inte listans kapacitet i samma takt som element läggs till?
5. Minskar kapaciteten när element tas bort ur listan?
6. När är det då fördelaktigt att använda en egendefinierad ***array*** istället för en lista?

Svar:

1)

```
class list
{       //skapa list
        List list = new List()
        Count(elemen)       // räknar elementen i list
        Capacity[]    // Bestämmer hur många element underliggande array kan ta

        ExamineList()
                {
                   Count elements in list
                   Skriva ut Capacity of list
                }
}
```

2) När Capacity arrayen ökar
3) Kapaciteten ökar efter användarens beslut. Alltså att det är Användaren som bestämmer storleken på underliggande arrayen
4) listan är i princip gränslös (upp to hela mängden minne tillgänglig) medan mängden element kan ändras (ökas eller sänkas)
5) Nej. Kapaciteten kan vara gränslös som till example:
   1. en array av fyra element utan bestämd storlek.
      String[] cars = new string[] {”Volvo” , ”BMW”, ”Ford”, ”Mazda”}

6) När man vet antalet element. Men när antalet kan ändras då är det bättre med en lista som kan ökas efter behöv.

*Övning 2: ExamineQueue()*

Datastrukturen ***kö*** (implementerad i ***Queue-klassen*** ) fungerar enligt Först In Först Ut ( ***FIFO*** ) principen. Alltså att det element som läggs till först kommer vara det som tas bort först.

1. Simulera följande kö på papper:
a. ICA öppnar och kön till kassan är tom
b. Kalle ställer sig i kön
c. Greta ställer sig i kön
d. Kalle blir expedierad och lämnar kön
e. Stina ställer sig i kön
f. Greta blir expedierad och lämnar kön
g. Olle ställer sig i kön
h. …

2. Implementera metoden ***TestQueue*** . Metoden ska simulera hur en ***kö*** fungerar genom att tillåta användaren att ställa element i kön ( ***enqueue*** ) och ta bort element ur kön ( ***dequeue*** ). Använd ***Queue-klassen*** till hjälp för att implementera metoden. Simulera sedan ICA-kön med hjälp av ditt program.

1)
- a) queue = 0;
- b) Kalle.enqueue;
  AntaletElementIKö =+ AntaletElementIKö;
- c) Greta.enqueue;
  AntaletElementIKö =+AntaletElementIKö;
- d) Kalle.expedieras;
  AntaletElementIKö =- AntaletElementIKö;
- e) Stina.Enque;
  AntaletElementIKö =+ AntaletElementIKö;
- f) Greta expedias;
  Greta.dequeue
- g) Olle.enqueue;
  AntaletElementIKö=+ AntaletElementIKö;
- g …

2)

```
Queue queue = new Queue;
String[] element = new String[] {”Kalle”,”Greta”, ”Stina”, ”Olle”, …, …MAX}
count(){ int AntaletElementIKö;AntaletElementIKö ++}
for (AntaletElementIKö=0, AntaletElementIKö< MAX, AntaletElementIKö++)
{ element =+ element;}
while (queue != MAX)
        {
                enqueue.element;
                expediera.element;
                dequeue.element;
        }
TestQueue()
        {
                queue.length= MAX;
                if (AntaletElementIKö = MAX)
                {
                        exit();
                }
        }
```

### *Övning 3: ExamineStack()*

Stackar påminner om köer, men en stor skillnad är att stackar använder sig av Först In Sist Ut ( ***FILO*** ) principen. Alltså gäller att det element som stoppas in först ( ***push*** ) är det som kommer tas bort sist ( ***pop*** ).

1. Simulera ännu en gång ICA-kön på papper. Denna gång med en ***stack*** . Varför är det inte så smart att använda en ***stack*** i det här fallet?
2. Implementera en ReverseText-metod som läser in en sträng från användaren och med hjälp av en stack vänder ordning på teckenföljden för att sedan skriva ut den omvända strängen till användaren.

Svar:

1)
1. Simulera följande kö på papper:
a. ICA öppnar och kön till kassan är tom
b. Kalle ställer sig i kön
c. Greta ställer sig i kön
d. Kalle blir expedierad och lämnar kön
e. Stina ställer sig i kön
f. Greta blir expedierad och lämnar kön
g. Olle ställer sig i kön
h. ...

1) Följande simulera ICA-kön när man använder sig av en stack.
   Stacken använder sig av en FIFO (First In First Out) scheme.
   - Kalle first
   - Greta second
   - Kalle måste låta Greta expedieras och dequeueas first innan han kan dequeueras. Det gör att fast Kalle är först i queue han dequeras efter den som kommer efter honom. Orättvist, den som kommer först ska dequeras först.
2) Följande är en implementation som tar en sträng och vänder ordning på teckenföljden och skriver ut den omvända strängen till användaren

using System;

```
namespace StringManipulation
{
    class Program
    {
        static void Main(string[] args)
        {
            bool showMenu = true;
            while (showMenu)
            {
                showMenu = MainMenu();
            }
        }
        private static bool MainMenu()
        {
            Console.Clear();
            Console.WriteLine("Choose an option:");
            Console.WriteLine("1) Reverse String");
            Console.WriteLine("2) Exit");
            Console.Write("\r\nSelect an option: ");
```

```
            switch (Console.ReadLine())
            {
                case "1":
                    ReverseString();
                    return true;

                case "2":
                    return false;
                default:
                    return true;
            }
        }

        private static string CaptureInput()
        {
            Console.Write("Enter the string you want to modify: ");
            return Console.ReadLine();
        }

        private static void ReverseString()
        {
            Console.Clear();
            Console.WriteLine("Reverse String");

            char[] charArray = CaptureInput().ToCharArray();
            Array.Reverse(charArray);
            DisplayResult(String.Concat(charArray));
        }


        private static void DisplayResult(string message)
        {
            Console.WriteLine($"\r\nYour modified string is: {message}");
            Console.Write("\r\nPress Enter to return to Main Menu");
            Console.ReadLine();
        }
    }
}
```

### *Övning 4: CheckParanthesis()*

Ni bör nu ha tillräckliga kunskaper om ovan nämnda datastrukturer för att lösa följande problem.

Vi säger att en sträng är ***välformad*** om alla parenteser som öppnas även stängs korrekt.

Att en parentes öppnas och stängs korrekt dikteras av följande regler:

• ), }, ] får enbart förekomma efter respektive (, {, [

• Varje parantes som öppnas måste stängas dvs ”(” följs av ”)”

Exempelvis är ([{}]({})) välformad men inte ({)}. Vidare kan en sträng innehålla andra tecken,

t ex är ”List<int> lista = new List<int>(){2, 3, 4};” välformad. Vi bryr oss alltså enkom om paranteser!

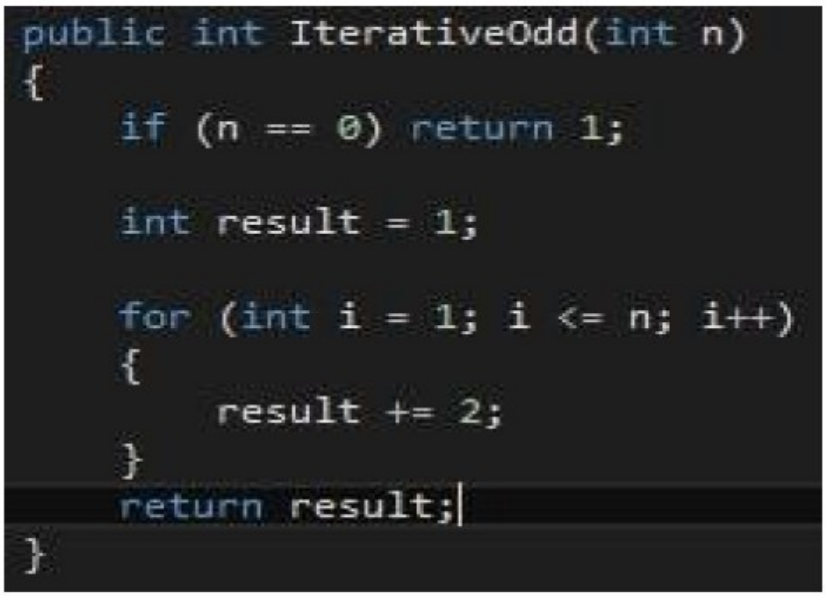
1. Skapa med hjälp av er nya kunskap funktionalitet för att kontrollera en välformad sträng på papper. Du ska använda dig av någon eller några av de datastrukturer vi precis gått igenom. Vilken datastruktur använder du?
2. Implementera funktionaliteten i metoden ***CheckParantheses*** . Låt programmet läsa in en ***sträng*** från användaren och returnera ett svar som reflekterar huruvida strängen är välformad eller ej.

### *Övning 5: Rekursion*

1. Illustrera förloppen för RecursiveOdd(1), RecursiveOdd(3) och RecursiveOdd(5) på papper för att förstå den rekursiva loopen.
2. Skriv en ***RecursiveEven(int n)*** metod som rekursivt beräknar det ***n*** :te jämna talet.
3. Implementera en rekursiv funktion för att beräkna tal i ***fibonaccisekvensen: (f(n) = f(n-1) + f(n-2))***

***Övning 6: Iteration***

Nu när ni är bekanta med rekursion är det dags att kolla på iteration. Iteration är en funktion som upprepar samma sak till dess att målet är uppnått. Så en iterativ funktion för att göra föregående beräkning om det n:te udda talet skulle se ut:



```
public int IterativeOdd(int n)
{
    if (n == 0) return 1;

    int result = 1;

    for (int i = 1; i <= n; i++)
    {
        result += 2;
    }
    return result;
}
```

Denna metod ser om ***n*** är ***noll*** , i så fall returnerar den det första udda talet ***1*** . Annars tar den ***1 + 2*** till dess att resultat blir det ***n*** :te udda talet.

1. Illustrera på papper förloppen för ***IterativeOdd(1)*** , ***IterativeOdd(3)*** och ***IterativeOdd(5)*** för att förstå iterationen.
2. Skapa en ***IterativeEven(int n)*** funktion för att iterativt beräkna det ***n*** :te jämna talet.
3. Implementera en iterativ version av ***fibonacciberäknaren*** .

Fråga :

Utgå ifrån era nyvunna kunskaper om iteration, rekursion och minneshantering. Vilken av ovanstående funktioner är mest minnesvänlig och varför?