Datastructures and Algorithms GD

Invertorio

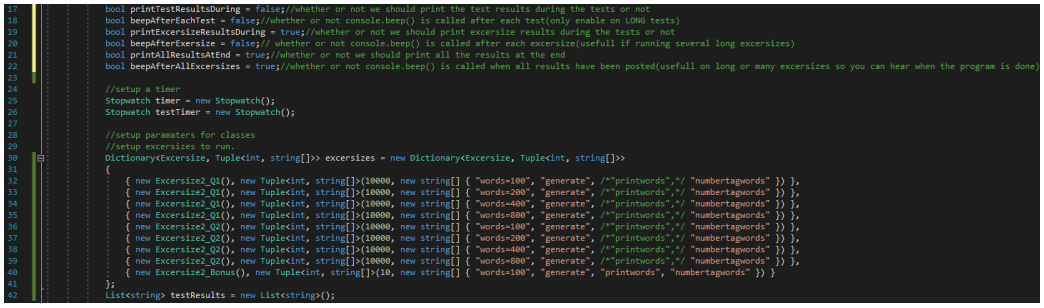
HVA

7 oktober 2017
Sjors Gielen (500765899)

Datastructures and Algorithms GD

Invertorio

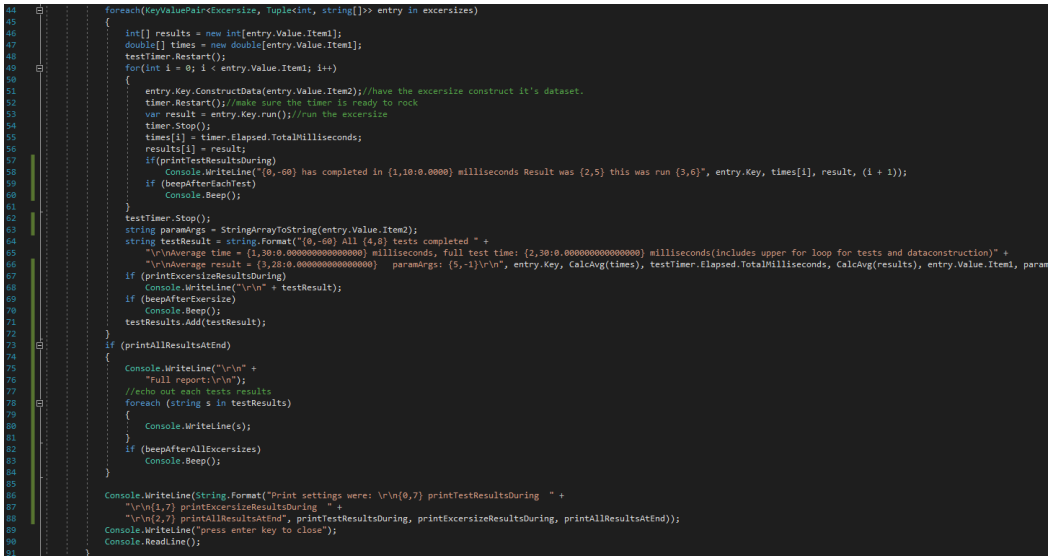
The test suite from last exercise was expanded in this exercise to support param args. Any param args with an equals in it is used to set a property on the exercise class underneath. Other param args are to be handled within the construct data method. This was mostly done to ensure that it's easy to read in the console what the current test running is. For complete context this is the test suite's current code:



```
bool printTestResultsDuring = false;//whether or not we should print the test results during the tests or not
bool beepAfterEachTest = false;//whether or not console.beep() is called after each test(only enable on LONG tests)
bool printExcersizeResultsDuring = true;//whether or not we should print excersize results during the tests or not
bool beepAfterExersize = false;// whether or not console.beep() is called after each excersize(usefull if running several long excersizes)
bool printAllResultsAtEnd = true;//whether or not we should print all the results at the end
bool beepAfterAllExcersizes = true;//whether or not console.beep() is called when all results have been posted(usefull on long or many excersizes so you can hear when the program is done)

//setup a timer
Stopwatch timer = new Stopwatch();
Stopwatch testTimer = new Stopwatch();

//setup paramaters for classes
//setup excersizes to run.
Dictionary<Excersize, Tuple<int, string[]>> excersizes = new Dictionary<Excersize, Tuple<int, string[]>>
{
    { new Excersize2_Q1(), new Tuple<int, string[]>(10000, new string[] { "words=100", "generate", /*"printwords",*/ "numbertagwords" }) },
    { new Excersize2_Q1(), new Tuple<int, string[]>(10000, new string[] { "words=200", "generate", /*"printwords",*/ "numbertagwords" }) },
    { new Excersize2_Q1(), new Tuple<int, string[]>(10000, new string[] { "words=400", "generate", /*"printwords",*/ "numbertagwords" }) },
    { new Excersize2_Q1(), new Tuple<int, string[]>(10000, new string[] { "words=800", "generate", /*"printwords",*/ "numbertagwords" }) },
    { new Excersize2_Q2(), new Tuple<int, string[]>(10000, new string[] { "words=100", "generate", /*"printwords",*/ "numbertagwords" }) },
    { new Excersize2_Q2(), new Tuple<int, string[]>(10000, new string[] { "words=200", "generate", /*"printwords",*/ "numbertagwords" }) },
    { new Excersize2_Q2(), new Tuple<int, string[]>(10000, new string[] { "words=400", "generate", /*"printwords",*/ "numbertagwords" }) },
    { new Excersize2_Q2(), new Tuple<int, string[]>(10000, new string[] { "words=800", "generate", /*"printwords",*/ "numbertagwords" }) },
    { new Excersize2_Bonus(), new Tuple<int, string[]>(10, new string[] { "words=100", "generate", "printwords", "numbertagwords" }) }
};
List<string> testResults = new List<string>();
```



```
foreach(KeyValuePair<Excersize, Tuple<int, string[]>> entry in excersizes)
{
    int[] results = new int[entry.Value.Item1];
    double[] times = new double[entry.Value.Item1];
    testTimer.Restart();
    for(int i = 0; i < entry.Value.Item1; i++)
    {
        entry.Key.ConstructData(entry.Value.Item2);//have the excersize construct it's dataset.
        timer.Restart();//make sure the timer is ready to rock
        var result = entry.Key.run();//run the excersize
        timer.Stop();
        times[i] = timer.Elapsed.TotalMilliseconds;
        results[i] = result;
        if(printTestResultsDuring)
            Console.WriteLine("{0,-60} has completed in {1,10:0.0000} milliseconds Result was {2,5} this was run {3,6}", entry.Key, times[i], result, (i + 1));
        if (beepAfterEachTest)
            Console.Beep();
    }
    testTimer.Stop();
    string paramArgs = StringArrayToString(entry.Value.Item2);
    string testResult = string.Format("{0,-60} All {4,8} tests completed " +
        "\r\nAverage time = {1,30:0.000000000000000} milliseconds, full test time: {2,30:0.000000000000000} milliseconds(includes upper for loop for tests and dataconstruction)" +
        "\r\nAverage result = {3,28:0.000000000000000}   paramArgs: {5,-1}\r\n", entry.Key, CalcAvg(times), testTimer.Elapsed.TotalMilliseconds, CalcAvg(results), entry.Value.Item1, param
    if (printExcersizeResultsDuring)
        Console.WriteLine("\r\n" + testResult);
    if (beepAfterExersize)
        Console.Beep();
    testResults.Add(testResult);
}
if (printAllResultsAtEnd)
{
    Console.WriteLine("\r\n" +
        "Full report:\r\n");
    //echo out each tests results
    foreach (string s in testResults)
    {
        Console.WriteLine(s);
    }
    if (beepAfterAllExcersizes)
        Console.Beep();
}

Console.WriteLine(String.Format("Print settings were: \r\n{0,7} printTestResultsDuring  " +
    "\r\n{1,7} printExcersizeResultsDuring  " +
    "\r\n{2,7} printAllResultsAtEnd", printTestResultsDuring, printExcersizeResultsDuring, printAllResultsAtEnd));
Console.WriteLine("press enter key to close");
Console.ReadLine();
```

And the exercise base-class now looks like this:

```
class Excersize
{
    private List<string> missingPorperties = new List<string>();
    private int n;
    private int words;
    public Excersize()
    {

    }

    public int _n { get => n; set => n = value; }
    public int _words { get => words; set => words = value; }

    //override this method and construct the class's data set within it.
    virtual public void ConstructData(string[] paramArgs = null)
    {
        foreach (string param in paramArgs)
        {
            if (param != String.Empty)//filter out acidental empty strings
            {
                int indexOfEquaals = param.IndexOf('=');
                if (indexOfEquaals != -1)//if it does not have an = it is meant for the other method to be dealth with, so we can ignore it
                {
                    string variableName = param.Substring(0, indexOfEquaals);
                    if (!missingPorperties.Contains(variableName))//if we already know we don't have this property no need to try it again.
                    {
                        try
                        {
                            //use reflection to set var
                            this.GetType().GetProperty("_" + variableName).SetValue(this, Convert.ToInt32(param.Substring(indexOfEquaals + 1, param.Length - (indexOfEquaals + 1))));
                        }
                        catch (Exception e)
                        {
                            Console.WriteLine("Missing property: _" + variableName + "\nOccured in " + this + " please check the param args and remove or add the property too the base-class if in
                            Console.WriteLine(e.StackTrace);
                            missingPorperties.Add(variableName);
                            Console.Beep();
                        }
                    }
                }
            }
        }
    }
```

By using some simple reflection I find the property if it exists and assign it the value. Currently it only supports int32's but can be built out to include more types

Question 1:

Write a program that reads Strings as user input from Console and stores them in a Stack. When the user enters an empty line and presses enter, your program prints the inventory items in reverse order:

If the param generated is not being used the construct data method will roll into the while input loop. Otherwise it will use the generateRadonmString method to make words amount of words.

```
public override void ConstructData(string[] paramArgs = null)
{
    base.ConstructData(paramArgs);

    dataBuffer = new List<string>();
    bool input = true;
    foreach (string param in paramArgs)
    {
        if(param == "generate")
        {
            input = false;
            Random random = new Random(Guid.NewGuid().GetHashCode());
            for(int i = 0; i < this._words; i++)//build words equaal too the param provided
            {
                var length = random.Next(3, 10);
                var word = generateRandomString(length, random);
                if (paramArgs.Contains("numbertagwords"))
                {
                    word = (i + 1) + " " + word;
                }
                if (paramArgs.Contains("printwords"))
                {
                    Console.WriteLine(i + " " + word);
                }
                dataBuffer.Add(word);
            }
        }
    }
    while (input)
    {
        var line = Console.ReadLine();
        if(line == null || line == "")
        {
            input = false;
            break;
        }

        if (paramArgs.Contains("numbertagwords"))
            line = (dataBuffer.Count() + 1) + " " + line;
        dataBuffer.Add(line);

    }
    data = new Stack<string>(dataBuffer);//commit the buffer into the stack
}
```

After the data is generated we run a simple recursive method to build the string. In the snippet I have commented out the WriteLine to prevent the output buffer from intervening with the time-tracking.

```
public override int run()
{
    var s = GenerateContentsString(data);
     //Console.WriteLine("contents:\n" + s);

    return base.run();
}

public string GenerateContentsString(Stack<string> data)
{
    if (data.Count() == 0)
        return "";
    if(data.Count() == 1)
    {
        return data.Pop();
    }
    else
    {
        return data.Pop() + "\n" + GenerateContentsString(data);
    }
}
```

Question 2:

Change your program such that it uses a Queue to store the inventory items.

The assignemtn remains mostly the same. Just need to keep track of the stackframe's data as it isn't held neatly by the order of how the line must be written.

```
public override int run()
{
    var s = GenerateContentsString(data);
    //Console.WriteLine("contents:\n" + s);

    return base.run();
}

public string GenerateContentsString(Queue<string> data)
{
    if (data.Count() == 0)
        return "";
    if (data.Count() == 1)
    {
        return data.Dequeue();
    }
    else
    {
        var stackBuffer = data.Dequeue();
        return GenerateContentsString(data) + "\n" + stackBuffer;
    }
}
```

Question 3

Which of these two datastructures leads to the most time efficient code? Base your argument on how stacks and queues work. Can we predict the Big-O for each method?

They are actually the same. The built in stack and queue are both written to do their pop and dequeue methods both operate under O(1). This means that both GenerateContentsString methods operate on O(n). However string concatenation takes O(n). Seeing as the amount of words constantly increases by a the words already in the buffer + the new word. This results in the methods going to O(!). This could have been improved allot by having the content be throw out into a List<T> first and returning the inverted structure. Then foreaching over that to print all the data would ultimately achieve the same effect.

Question 4

Run an experiment where you estimate the Big-O of each method (Q1 and Q2)

```
Datastruct_and_algo_excersizes.Excersize2_Q1               All    10000 tests completed
Average time =          0,035080620000000 milliseconds, full test time:          1254,464600000000000 milliseconds(includes upper for loop for tests and dataconstruction)
Average result =        0,000000000000000   paramArgs: words=100 | generate | numbertagwords

Datastruct_and_algo_excersizes.Excersize2_Q1               All    10000 tests completed
Average time =          0,057574170000000 milliseconds, full test time:          1464,650500000000000 milliseconds(includes upper for loop for tests and dataconstruction)
Average result =        0,000000000000000   paramArgs: words=200 | generate | numbertagwords

Datastruct_and_algo_excersizes.Excersize2_Q1               All    10000 tests completed
Average time =          0,218469520000001 milliseconds, full test time:          3975,102500000000000 milliseconds(includes upper for loop for tests and dataconstruction)
Average result =        0,000000000000000   paramArgs: words=400 | generate | numbertagwords

Datastruct_and_algo_excersizes.Excersize2_Q1               All    10000 tests completed
Average time =          0,949915639999998 milliseconds, full test time:         13093,691700000000000 milliseconds(includes upper for loop for tests and dataconstruction)
Average result =        0,000000000000000   paramArgs: words=800 | generate | numbertagwords

Datastruct_and_algo_excersizes.Excersize2_Q2               All    10000 tests completed
Average time =          0,022511050000000 milliseconds, full test time:           714,527100000000000 milliseconds(includes upper for loop for tests and dataconstruction)
Average result =        0,000000000000000   paramArgs: words=100 | generate | numbertagwords

Datastruct_and_algo_excersizes.Excersize2_Q2               All    10000 tests completed
Average time =          0,068277789999999 milliseconds, full test time:          1763,603500000000000 milliseconds(includes upper for loop for tests and dataconstruction)
Average result =        0,000000000000000   paramArgs: words=200 | generate | numbertagwords

Datastruct_and_algo_excersizes.Excersize2_Q2               All    10000 tests completed
Average time =          0,235137320000002 milliseconds, full test time:          4292,986800000000000 milliseconds(includes upper for loop for tests and dataconstruction)
Average result =        0,000000000000000   paramArgs: words=400 | generate | numbertagwords

Datastruct_and_algo_excersizes.Excersize2_Q2               All    10000 tests completed
Average time =          1,004149600000000 milliseconds, full test time:         13876,203400000000000 milliseconds(includes upper for loop for tests and dataconstruction)
Average result =        0,000000000000000   paramArgs: words=800 | generate | numbertagwords
```

A quick run of the test suite shows the O(!) behaviour perfectly. The words constantly double but the time rises much faster than $O(n^2)$ would. Not included in the screenshot but a version with 1600 words would jump up to 5 milliseconds per test.

Question 5

Does your program use dynamic programming (recursion)? If yes, show the relevant code snippets. If no, change part of your program so that works recursively.

Yes. Screenshots were already provided earlier in this report.

Question 6

What is the stopping criterion of your recursive function(s)?

For both methods the stopping criterion is countable<T>.count() == 0 or countable<T>.count() == 1. In the case an empty stack or queue is supplied we can return an empty string. Otherwise we keep popping/dequeuing until there is only one entity left, then we simply pop/dequeue that last entity and we can stop.

Bonus

Show how to implement a Queue using two Stacks.

```
class StackQueue<T>
{
    private Stack<T> outputStack;
    private Stack<T> inputStack;
    private int size;

    public StackQueue()
    {
        inputStack = new Stack<T>();
        outputStack = new Stack<T>();
        this.size = 0;
    }

    public StackQueue(IEnumerable<T> data)
    {
        inputStack = new Stack<T>();
        outputStack = new Stack<T>();
        this.size = 0;
        foreach(T element in data)
        {
            this.Enqueue(element);
        }
    }

    public void Enqueue(T e)
    {
        inputStack.Push(e);
        size++;
    }

    public T Dequeue()
    {
        // fill out all the Input if output stack is empty
        if (outputStack.Count == 0)
            while (inputStack.Count != 0)
                outputStack.Push(inputStack.Pop());

        T temp = default(T);
        if (outputStack.Count != 0)
        {
            temp = outputStack.Pop();
            size--;
        }

        return temp;
    }
```

Like so.