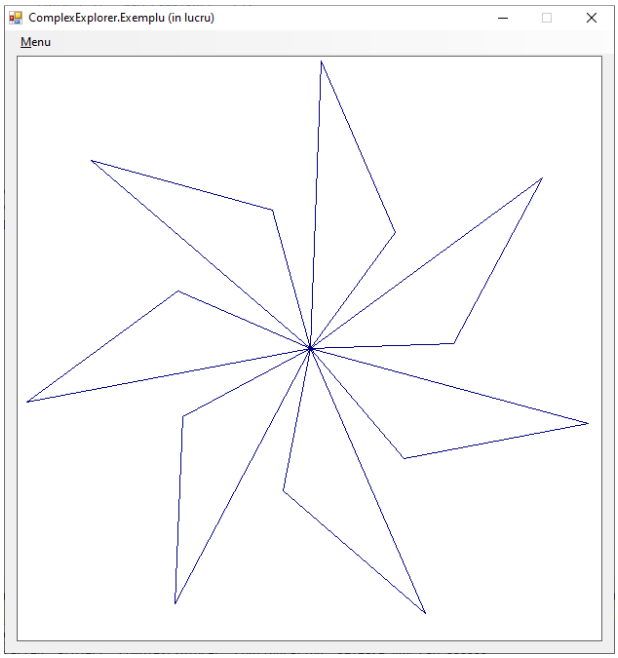
Liste în C#

După cum am văzut, în C#, când avem de memorat o mulţime de obiecte de acelaşi tip putem folosi cu succes tablourile, chiar dacă numărul de obiecte este cunoscut doar la rulare, tablourile din C# fiind asemănătoare tablourilor alocate dinamic în C++, dar mult mai uşor de folosit. Totuşi, atunci când mulţimea obiectelor memorate îsi modifică foarte des volumul în timpul rulării este mult mai convenabil să utilizăm liste în loc de tablouri, de exemplu la realizarea desenelor recursive, atunci când numărul punctelor memorate creşte în progresie geometrică de la o etapă la alta.

Aici vom prezenta utilizarea listelor puse la dispoziţie de C#, mai precis vom vorbi, mărginindu-ne doar la strictul necesar, despre utilizarea listelor generice din clasa `List<T>` vezi: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=netframework-4.8

Exemplul 1. *Următorul program roteşte înainte şi înapoi o morişcă. Rulaţi programul, studiaţi codul şi apoi încercaţi să daţi o implementare fără liste, cu tablouri.*




```
using System;
using System.Drawing;
using System.Collections.Generic; //pentru List<T>
```

```
namespace ComplexExplorer
{
    public class Exemplu : ComplexForm
    {
        Complex i = new Complex(0, 1);
        void traseaza(List<Complex> li)
        {
            for (int k = 1; k < li.Count; k++)
            {
                setLine(li[k - 1], li[k], PenColor);
            }
        }
        void roteste(ref List<Complex> li)
        {
            Complex rotor = Complex.setRoTheta(0.999, -0.005);
            List<Complex> rez = new List<Complex>();
            foreach (Complex z in li)
            {
                rez.Add(rotor * z);
            }
            li = rez;
        }
        void roteste(List<Complex> li)
        {
            Complex rotor = Complex.setRoTheta(1.001, 0.005);

            for (int k = 0; k < li.Count; k++)
            {
                li[k] *= rotor;
            }
        }
        public override void makeImage()
        {
            setXminXmaxYminYmax(-5, 5, -5, 5);
            ScreenColor = Color.White;
            PenColor = Color.Navy;
            int N = 7;
            List<Complex> li = new List<Complex>();
            double theta = 2 * Math.PI / N;
            Complex omega = Complex.setRoTheta(2, 2 * theta / 3);
            for (int k = 0; k < N; k++)
            {
                Complex z = Complex.setRoTheta(1, k * theta);
                li.Add(z);
                li.Add(omega * z);
                //li.Add(i * z);
                li.Add(0);
            }
            li.Add(1);
            for (int k = 0; k < 10000; k++)
            {
                initScreen();
                if ((k / 1000) % 2 == 0) roteste(li);
                else roteste(ref li);
                traseaza(li);
                if (!resetScreen()) return;
            }
        }
    }
}
```

Rezolvare. Studiem codul. Observăm că morişca din imagine este trasată ca o linie poligonală cu vârfurile stocate în lista `li`, declarată şi instanţiată cu instrucţiunea

```
List<Complex> li = new List<Complex>();
```

Clasa `List<Complex>` este o specializare a clasei generice[1] `List<T>` prin urmare `List<Complex>()` nu poate fi decât constructorul fără parametri al clasei. Limbajul C# nu dispune de operatorul `typedef` din C++, totuşi, pentru a introduce o denumire mai scurtă se poate utiliza directiva `using` cu egalitate amplasată imediat după namespace:

```
namespace ComplexExplorer
{
    using MyList = List<Complex>;

    public class Exemplu : ComplexForm
    {
...
```

În acest caz declaraţia listei `li` poate fi scrisă astfel

```
MyList li = new MyList();
```

dar acum vom lăsa în continuare codul sursă aşa cum a fost scris.

Să obsevăm că în preambulul fişierului sursă apare directiva

```
using System.Collections.Generic;
```

necesară pentru accesul la `List<T>`.

Revenim la lista `li`, ea este încărcată în ciclul

```
for (int k = 0; k < N; k++)
    {
        Complex z = Complex.setRoTheta(1, k * theta);
        li.Add(z);
        ...
    }
```

folosind metoda `Add()`, apoi este trimisă în mod repetat la prelucrare şi trasare. Observăm că în metoda

```
void traseaza(List<Complex> li)
{
    for (int k = 1; k < li.Count; k++)
    {
        setLine(li[k - 1], li[k], PenColor);
    }
}
```

elementele listei sunt *citite* ca şi cum ar fi elementele unui tablou, iar în

```
void roteste(List<Complex> li)
{
    Complex rotor = Complex.setRoTheta(1.001, 0.005);

    for (int k = 0; k < li.Count; k++)
    {
        li[k] *= rotor;
    }
}
```

ele sunt *scrise*, modificate, tot ca şi cum ar fi elementele unui tablou.

[1] clasele generice sunt varianta C# , mai slabă, a şabloanelor de clase din C++, se comportă la fel dar între ele sunt diferenţe esenţiale, nu numai de denumire.

Atragem atenţia că aceasta este o facilitate pusă la dispoziţia utilizatorilor clasei `List<T>` prin intermediul metodelor de tip proprietate. Structura internă a unui un obiect din această clasă se bazează pe un tablou dinamic, realocat ori de câte ori este depăşită o anumită capacitate, dar utilizatorul nu poate accesa direct elementele tabloului[2]. Vom reveni.

Să observăm şi utilizarea instrucţiunuii `foreach` în metoda care primeşte lista prin referinţă

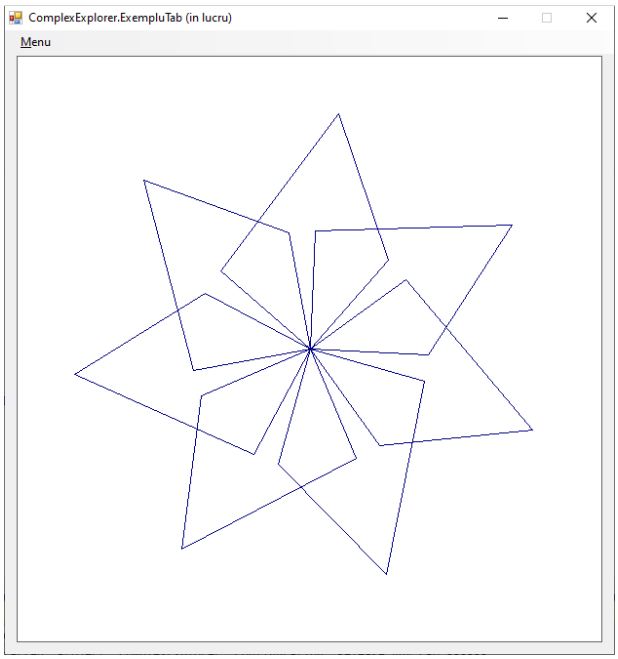
```
void roteste(ref List<Complex> li)
 {
     Complex rotor = Complex.setRoTheta(0.999, -0.005);
     List<Complex> rez = new List<Complex>();
     foreach (Complex z in li)
     {
         rez.Add(rotor * z);
     }
     li = rez;
 }
```

În cealaltă metodă `roteste`, cea în care lista ajunge prin valoare, ciclarea a fost implementată cu un `for` obişnuit deoarece variabila de iterare dintr-un `foreach` este de tip `readonly` şi nu poate fi modificată.

Forma elicei poate fi schimbată foarte uşor, de exemplu este suficient să scoatem din comentariu linia

```
//li.Add(i * z);
```

ca să obţinem la rulare figura următoare:



Această modificare este mai greu de realizat în cazul utilizării tablourilor, unde trebuie să precizăm din start dimensiunea tabloului alocat:

[2] clasa `List<T>` este varianta generică a clasei `ArrayList`, obiectele lor fiind nişte tablouri deghizate în liste, dar există şi clase de liste propriu-zise, cum ar fi `LinkedList<T>`, recomantate în cazul volumelor mari de date (la alocarea dinamică tablourile cer blocuri mari de memorie formate din adrese consecutive, blocuri mai greu de găsit decât cele necesare nodurilor unei liste).

```
int N = 7;
Complex[] tab = new Complex[3 * N + 1];
```

pentru morişca iniţială, şi

```
Complex[] tab = new Complex[4 * N + 1];
```

pentru cea modificată. În plus, ciclul `for` din `makeImage()` pentru formarea elicei, care ar trebui să arate cam aşa:

```
for (int k = 0; k < N; k++)
{
    Complex z = Complex.setRoTheta(1, k * theta);
    tab[3 * k] = z;
    tab[3 * k + 1] = omega * z;
    tab[3 * k + 2] = 0;
}
tab[3 * N] = 1;
```

trebuie schimbat în

```
for (int k = 0; k < N; k++)
{
    Complex z = Complex.setRoTheta(1, k * theta);
    tab[4 * k] = z;
    tab[4 * k + 1] = omega * z;
    tab[4 * k + 2] = i * z;
    tab[4 * k + 3] = 0;
}
tab[4 * N] = 1;
```

cu modificări consistente. Este vizibil acum avantajul utilizării listelor faţă de tablouri atunci când sunt implementate operaţii de inserare.

Reţinem: listele în C# sunt foarte uşor de utilizat, pe lângă metodele specifice listelor ele sunt dotate şi cu accesul prin indexare, dar ele nu sunt tablouri:

Exemplul 2. *Următoarea aplicaţie în mod consolă primeşte la compilare mesajul pus în comentariu. Explicaţi eroarea.*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    public struct Punct
    {
        public double x, y;
        public Punct(double xx, double yy)
        {
            x = xx; y = yy;
        }
    }

    class Program
    {
        static void Arata(string mesaj, Punct p)
        {
            Console.WriteLine(mesaj + ": x={0} y={1}", p.x, p.y);
        }
```

```
        static void Main(string[] args)
        {
            Punct p = new Punct(0, 0), q = new Punct(2, 2);

            Punct[] tab = new Punct[2] { p, q };
            Arata("tab[0]", tab[0]);    //x=0 y=0
            tab[0].x = 17;
            Arata("tab[0]", tab[0]);    //x=17 y=0
            List<Punct> li = new List<Punct>() { p, q };
            Arata("li[0]", li[0]);      //x=0 y=0
            li[0].x = 17;
            //Error:Cannot modify the return value of
            //'System.Collections.Generic.List<ConsoleApplication1.Punct>.this[int]'
            //because it is not a variable
            li[0] = q;
            Arata("li[0]", li[0]);      //x=2 y=2
        }
    }
}
```

Rezolvare. În cazul unui tablou, expresia `tab[0]` este o referinţă (exact ca în C++, ţinta unui pointer) către primul element al tabloului, se comportă ca o variabilă de tip `Punct` şi poate fi modificată câmp cu câmp prin operatorul de selecţie membru.

În cazul unei liste, expresia `li[0]` desemnează o proprietate publică de acces la primul nod al listei, proprietate care, conform documentaţiei[3], are forma

```
public T this[int index] { get; set; }
```

Prin urmare, în cazul nostru, prin `li[0]` se înţelege proprietatea

```
public Punct li[0] { get; set; }
```

şi astfel, când întâlneşte expresia `li[0].x`, compilatorul încearcă să aplice operatorul . de selecţie membru operandului format din rezultatul returnat de `get`-ul proprietăţii `li[0]`, care returnează o copie a obiectului stocat în primul nod al listei. Metoda de acces `set` nu poate fi folosită: ea nu returnează nimic.

Metoda `get` returnează prin valoare un obiect cu o existenţă temporară, obiect care poate fi doar copiat de către funcţia apelantă, nu şi modificat. Ca dovadă, dacă adăugăm în clasa `Program` următoarea funcţie `test()`, care returnează prin valoare un `Punct`, reuşim să obţinem la compilare exact acelaşi mesaj de eroare:

```
    class Program
    {
        static void Arata(string mesaj, Punct p)
        {
            Console.WriteLine(mesaj + ": x={0} y={1}", p.x, p.y);
        }
        static Punct test(Punct p)
        {
            return p;
        }
```

[3] https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.item?view=netframework-4.8

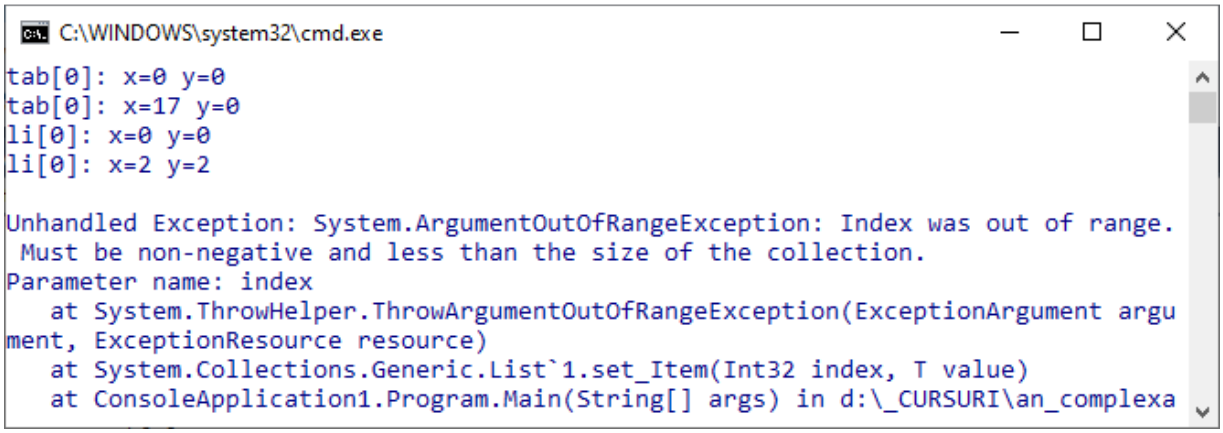
```
        static void Main(string[] args)
        {
            Punct p = new Punct(0, 0), q = new Punct(2, 2);

            Punct[] tab = new Punct[2] { p, q };
            Arata("tab[0]", tab[0]);    //x=0 y=0
            tab[0].x = 17;
            Arata("tab[0]", tab[0]);    //x=17 y=0
            List<Punct> li = new List<Punct>() { p, q };
            Arata("li[0]", li[0]);      //x=0 y=0
            //li[0].x = 17;
            //Error:Cannot modify the return value of
            //'System.Collections.Generic.List<ConsoleApplication1.Punct>.this[int]'
            //because it is not a variable
            li[0] = q;
            Arata("li[0]", li[0]);      //x=2 y=2
            test(p).x = 17;
            //Error: Cannot modify the return value of
            //'ConsoleApplication1.Program.test(ConsoleApplication1.Punct)'
            //because it is not a variable
        }
}
```

Menţionăm că atribuirea `li[0] = q` funcţionează fără probleme deoarece compilatorul o traduce în apelul `li[0].set(q)` şi astfel primul element din listă este modificat corespunzător.

Reţinem: în cazul listelor de obiecte de tip structură, un element accesat cu proprietatea de indexare poate fi citit pe bucăţi, câmp cu câmp, dacă vrem, dar nu poate fi modificat decât global, tot elementul odată.

Atenţie: prin accesarea cu indici se pot modifica numai elementele deja prezente în listă, nu se pot face noi alocări. Accesarea unui element nealocat trece de compilare dar la rulare provocă întreruprea execuţiei. Schimbaţi mai sus `li[0] = q;` în `li[3] = q;` şi veţi vedea:



În final, o întrebare: de ce nu mai apare nici o eroare la compilare dacă în programul iniţial scriem `public class Punct` în loc de `public struct Punct`? Mai mult, trece de compilare, dar rezultatul este cel aşteptat?

Indicaţie. Răspunsul la aceste întrebări îl veţi afla când veţi înţelege de ce nu apare nicăieri numărul `13` în rezultatul

```
/*REZULTAT:
p1 : x=1 y=0
simetric: x=0 y=1
p2 : x=7 y=5
a=5
p3 : x=7 y=5
p3 simetric: x=5 y=7
Press any key to continue . . .*/
```

al programului următor, în care am înlocuit proprietăţile indexate cu o proprietate obişnuită, mai precis cu proprietatea `public Punct Sim` care, la citire, returnează simetricul punctului curent faţă de prima bisectoare iar la scriere setează punctul curent astfel încât simetricul său sa fie cel precizat la apel.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication0
{
    public class Punct
     //public struct Punct
    {
        public double x, y;
        public Punct(double xx, double yy)
        {
            x = xx;
            y = yy;
        }

        public Punct Sim
        {
            get
            {
                return new Punct(y, x);
            }
            set
            {
                x = value.y;
                y = value.x;
            }
        }

    }


    class Program
    {
        static void Arata(string mesaj, Punct p)
        {
            Console.WriteLine(mesaj + ": x={0} y={1}", p.x, p.y);
        }
```

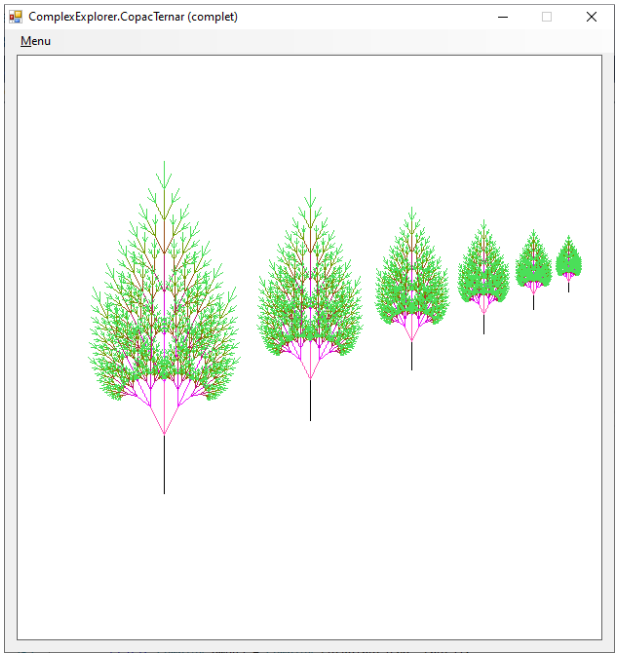
```
        static void Main(string[] args)
        {
            Punct p = new Punct(0, 0);
            p.x = 1;
            Arata("p1 ", p);//x=1 y=0
            Arata("simetric", p.Sim);//x=0 y=1

            p.Sim = new Punct(5, 7);
            Arata("p2 ", p);//x=7 y=5

            double a = p.Sim.x;
            Console.WriteLine("a={0}", a);//a=5
            p.Sim.x = 13;
            // daca Punct este o structura apare mesajul urmator:
            // Error:Cannot modify the return value of 'ConsoleApplication1.Punct.Sim'
            // because it is not a variable
            Arata("p3 ", p);//x=7 y=5
            Arata("p3 simetric", p.Sim);//x=5 y=7
        }
    }
}
```

Revenim la utilizarea listelor, exersând pe următorul exemplu:

Exemplul 3. *Presupunem că un copac creşte după următoarea schemă simplificată: mugurii de creştere se află numai în vârfurile rămurelelor de un an, câte trei în fiecare vârf, şi din fiecare mugur creşte în fiecare an câte o nouă rămurică. Să se deseneze un astfel de copac colorând diferit fiecare generaţie de ramuri şi apoi imaginea obţinută să fie mutată şi micşorată de mai multe ori, ca în figura următoare*:



Rezolvare. Simplificăm şi mai mult modelul: toate creşterile sunt în linie dreaptă, mugurul din mijloc este în prelungirea ramurii iar ceilalţi doi fac acelaşi unghi θ cu acesta. Un copac va fi, prin urmare, o mulţime de segmente colorate, pe care le vom memora într-o listă. Un astfel de segment îl vom numi *ram*, indiferent dacă este un segment internoduri sau o rămurică terminală, şi îl vom implementa în următoarea structură

```
public struct Ram
{
    public Complex q, delta;
    public Color col;
    public Ram(Complex q, Complex delta, Color col)
    {
        this.q = q; this.delta = delta; this.col = col;
    }
}
```

Câmpurile au următoarea semnificaţie: `Complex q` este afixul punctului de plecare al ramului, `Complex delta` este vectorul care dă direcţia şi lungimea ramului, astfel că vârful ramului are afixul `q+delta`, iar `Color col` este culoarea primită de ram la formare şi pe care şi-o păstrează toată viaţa.

Vom memora întregul copac în lista

```
List<Ram> copac;
```

şi vom păstra separat lista ramurilor de un an, în

```
List<Ram> ultimaGeneratie;
```

deoarece numai acestea au muguri de creştere. Presupunând că am implementat deja metoda

```
void genereaza(ref List<Ram> li, Color col){}
```

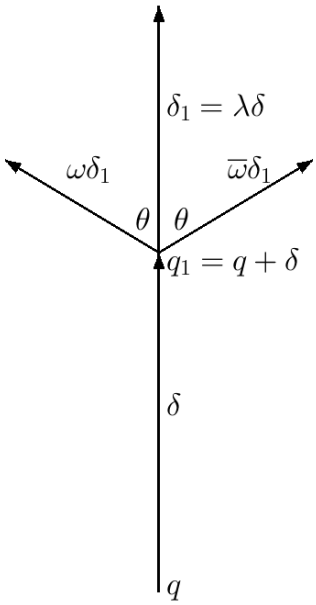
care primeşte lista ramurilor de un an şi returnează lista celor nou crescute, colorate toate în culoarea `col` , atunci copacul care creşte din ramul iniţial `Ram r` este format de metoda următoare

```
void formeazaCopac(Ram r, out List<Ram> copac, int etate = 7)
{
    List<Ram> ultimaGeneratie = new List<Ram>() { r };
    copac = new List<Ram>(ultimaGeneratie);
    copac.ForEach(printeaza);
    for (int k = 0; k < etate; k++)
    {
        genereaza(ref ultimaGeneratie, getColor(350 + 50 * k));
        copac.AddRange(ultimaGeneratie);
        ultimaGeneratie.ForEach(printeaza);
        delaySec(0.1);
        if (!resetScreen()) return;
    }
}
```

Aici acţiunea[4] `printeaza` livrată metodei `copac.ForEach()` este

```
void printeaza(Ram r)
{
    setLine(r.q, r.q + r.delta, r.col);
}
```

În ciclul `for (int k = 0; k < etate; k++)` din `formeazaCopac` generăm la fiecare etapă generaţia anuală de ramuri pe care apoi o adăugăm, cu metoda `AddRange()` la lista întregului copac.



Să prezentăm modelul matematic necesar metodei de generare a creşterilor anuale. Fie q şi $q1 = q + \delta$ baza, respectiv vârful unui ram din ultima generaţie, din care vor creşte trei segmente noi. Cel din mijloc va fi coliniar cu qq_1, prin urmare va avea vectorul director de forma $\delta_1 = \lambda\delta$, cu scalar subunitar. Condiţia $\lambda \in (0,1)$ asigură mărginirea copacului (de ce?), şi îl vom fixa la valoarea

```
static double lambda = 0.9;
```

aceeaşi pentru toţi mugurii centrali.

Fie δ_2 vectorul director al ramului nou din din stânga. Definim $\omega = \frac{\delta_2}{\delta_1}$ şi avem

$$\omega = \rho(\cos\theta + i\sin\theta)$$

unde $\theta \in (0, \frac{\pi}{2})$ este unghiul dintre un ram lateral şi cel din mijloc iar ρ este raportul lungimilor celor două ramuri, raport pe care îl vom fixa subunitar:

```
static double ro = 0.6;
static double theta = Math.PI / 7;
static Complex omega = Complex.setRoTheta(ro, theta);
```

Cu aceste precizări, formarea noii generaţii este dată de metoda

[4] vezi https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.foreach?view=netframework-4.8

```
void genereaza(ref List<Ram> li, Color col)
{
    List<Ram> rez = new List<Ram>();
    foreach (Ram r in li)
    {   //r este ramura veche, formam 3 ramuri noi
        Complex q1 = r.q + r.delta;
        Complex delta1 = lambda * r.delta;

        rez.Add(new Ram(q1, omega * delta1, col));
        rez.Add(new Ram(q1, delta1, col));
        rez.Add(new Ram(q1, omega.conj * delta1, col));
    }
    li = rez;
}
```

Am reuşit să creştem un copac, l-am şi afişat, acum trebuie să-l mutăm de mai multe ori în desenul nostru printr-o translaţie un vector u şi o scalare cu un factor σ.

```
void mutaImaginea(List<Ram> li, ref Complex u, double sigma)
{
    Complex q0 = li[0].q;
    for (int k = 0; k < li.Count; k++)
    {
        Ram ram = li[k];
        ram.q = q0 + sigma * (ram.q - q0) + u;
        ram.delta *= sigma;
        li[k] = ram;
    }
    u *= sigma;
}
```

Aici fiecărui ram din listă i se aplică o omotetie cu centru în q_0 şi raport σ, unde q_0 este punctul de bază al ramului iniţial, urmată de o translaţie de vector u. În final vectorul u este scalat şi el, ca să obţinem efectul de perspectivă. Iată fişierul sursă:

```
using System;
using System.Drawing;
using System.Collections.Generic; //pentru List<T>

namespace ComplexExplorer
{
    public class CopacTernar : ComplexForm
    {
        public struct Ram
        {
            public Complex q, delta;
            public Color col;
            public Ram(Complex q, Complex delta, Color col)
            {
                this.q = q; this.delta = delta; this.col = col;
            }
        }
        static Complex i = new Complex(0, 1);
        static double lambda = 0.9;
        static double ro = 0.6;
        static double theta = Math.PI / 7;
        static Complex omega = Complex.setRoTheta(ro, theta);
```

```
void genereaza(ref List<Ram> li, Color col)
{
    List<Ram> rez = new List<Ram>();
    foreach (Ram r in li)
    {   //r este ramura veche, formam 3 ramuri noi
        Complex q1 = r.q + r.delta;
        Complex delta1 = lambda * r.delta;
        rez.Add(new Ram(q1, omega * delta1, col));
        rez.Add(new Ram(q1, delta1, col));
        rez.Add(new Ram(q1, omega.conj * delta1, col));
    }
    li = rez;
}


void printeaza(Ram r)
{
    setLine(r.q, r.q + r.delta, r.col);
}


void formeazaCopac(Ram r, out List<Ram> copac, int etate = 7)
{
    List<Ram> ultimaGeneratie = new List<Ram>() { r };
    copac = new List<Ram>(ultimaGeneratie);
    copac.ForEach(printeaza);
    for (int k = 0; k < etate; k++)
    {
        genereaza(ref ultimaGeneratie, getColor(350 + 50 * k));
        copac.AddRange(ultimaGeneratie);
        ultimaGeneratie.ForEach(printeaza);
        delaySec(0.1);
        if (!resetScreen()) return;
    }
}

void mutaImaginea(List<Ram> li, ref Complex u, double sigma)
{
    Complex q0 = li[0].q;
    for (int k = 0; k < li.Count; k++)
    {
        Ram ram = li[k];
        ram.q = u + q0 + sigma * (ram.q - q0);
        ram.delta *= sigma;
        li[k] = ram;
    }
    u *= sigma;
}

public override void makeImage()
{
    setXminXmaxYminYmax(-1, 3, -1, 3);
    ScreenColor = Color.White;

    List<Ram> copac;
    Ram trunchi = new Ram(0, 0.4 * i, Color.Black);
    formeazaCopac(trunchi, out copac);
    Complex u = 1 + i / 2;
```

```
            for (int k = 0; k < 5; k++)
            {
                mutaImaginea(copac, ref u, 0.7);
                copac.ForEach(printeaza);
                resetScreen();
                delaySec(0.1);
            }
        }
    }
}
```

Copacii desenaţi până acum au un aspect supărător de regulat. Pentru a obţine un aspect cât mai natural, trebuie să introducem un anumit grad de dezordine în regulile de generare, de exemplu: unghiurile dintre ramurile noi să varieze aleator în jurul valorii θ, ramul nou central să fie şi el înclinat aleator faţa de cel vechi, mugurii laterali să germineze doar cu o anumită probabilitate. In acest scop se poate folosi metoda

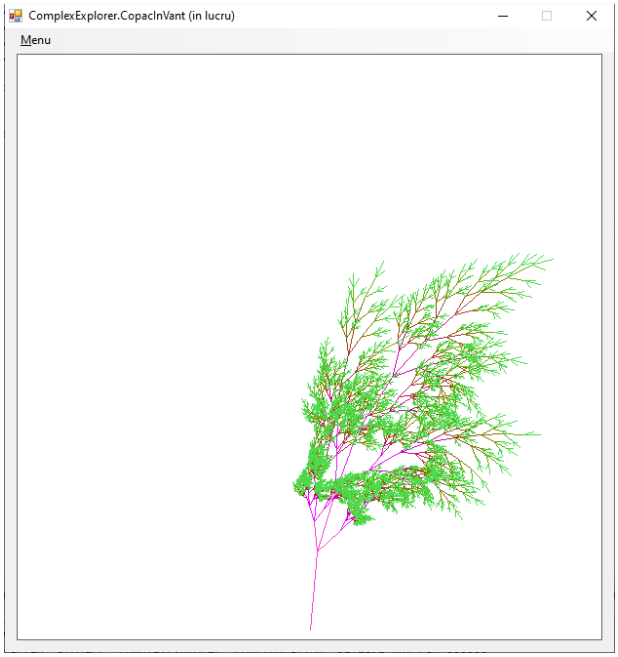
```
static Random rand = new Random();

double zar()
{
    return (rand.NextDouble() - 0.5) / 5;
}
```

care returnează un număr aleator, uniform distribuit, în intervalul $[-0.1, +0.1]$. Desenul următor a fost obţinut mutând numai punctul iniţial şi generând la fiecare mutare un copac aleator cu trunchiul iniţial micşorat corespunzător. Încercaţi şi voi.

Acum, după ce am reuşit să plantăm copaci într-o livadă, vrem să-i facem să se legene în vânt.

Exemplul 4. *Să se realizeze o animaţie care să simuleze mişcarea oscilatorie a unui arbore sub acţiunea vântului.*



Rezolvare. Dacă rotim înainte şi înapoi tot copacul în jurul punctului de bază q_0 obţinem o mişcare rigidă, nefirească. Pentru a simula unduirile ramurilor unui copac trebuie să rotim câte puţin, înainte şi înapoi, fiecare ram în jurul propriului punct de bază. Rotirea rămurelelor terminale nu ridică probleme, dar rotirea unui segment internoduri desprinde vârful său de baza celor trei ramuri crescute din el, şi astfel distrugem copacul. Suntem nevoiţi ca pentru fiecare cadru al animaţiei să generăm în memorie din nou întregul copac cu ramurile rotite puţin faţă de propria lor bază şi apoi să-l afişăm.

În cazul unui copac artificial dat de clasa `CopacTernar` modificările sunt puţine şi uşor de implementat, prin urmare sunt lăsate în seama cititorului, noi ne propunem să mişcăm un copac cu aspect natural, unul cu ramurile dispuse aleator. Dificultatea suplimentară este următoarea: ca să reuşim să generăm din nou acelaşi copac dar cu ramurile rotite puţin, va trebui să memorăm cumva înclinaţiile ω alese în mod aleator la prima generare a copacului, şi apoi să le folosim, modificate corespunzător.

Pentru a păstra aceste informaţii folosim structura

```
public struct Mugur
 {
     public int nivel;
     public Complex omStg,omega,omDrp;   //unghiurile noilor ramuri
     public bool areStg, areDrp;   //decid daca vor creste ram stg si ram drp
     public Mugur(int nivel, Complex omStg, Complex omega, Complex omDrp,
                             bool areStg = true, bool areDrp = true)
     {
         this.nivel = nivel;
         this.omStg = omStg; this.omega = omega; this.omDrp = omDrp;
         this.areDrp = areDrp; this.areStg = areStg;
     }
```

```
}
```

în care memorăm `nivel` -ul mugurului curent (etapa de generare, nivelul 0, corespunde trunchiului iniţial) şi cele trei numere complexe ω corespunzătoare celor trei ramuri noi (atenţie: în clasa `CopacTernar` s-au folosit numai două, `omega` pentru ramul stâng şi `omega.conj` pentru ramul drept, aici `omega` caracterizează ramul nou central, care nu mai este coliniar cu ramul vechi) . In plus apar şi două câmpuri booleene, `bool areStg, areDrp`, care decid dacă din vârful curent vor creşte sau nu ramurile din stânga şi, respectiv, din dreapta.

Un astfel de `Mugur`, format de fapt din trei muguraşi de creştere la un loc, va fi iniţializat aleator cu următoarea metodă:

```
static Random rand = new Random();

double zar()
{
    return (rand.NextDouble() - 0.5) / 4;
}

Mugur rndMugur(int niv)    //returneaza un mugur aleator
{
    Mugur mg = new Mugur();
    mg.nivel = niv;
    mg.omStg = Complex.setRoTheta(ro + zar(), theta + zar());
    mg.omega = Complex.setRoTheta(lambda + zar(), zar());
    mg.omDrp = Complex.setRoTheta(ro + zar(), -theta + zar());
    mg.areStg = rand.Next(10) < 9;
    mg.areDrp = rand.Next(10) < 9;
    return mg;
}
```

iar trecerea de la o generaţie de muguri la alta va fi de forma

```
void genereaza(ref List<Mugur> li, int niv)
{
    List<Mugur> rez = new List<Mugur>();
    foreach (Mugur mug in li)
    {   //mug este pe ramura veche, formam 3 muguri noi
        if (mug.areStg) rez.Add(rndMugur(niv));
        rez.Add(rndMugur(niv));
        if (mug.areDrp) rez.Add(rndMugur(niv));
    }
    li = rez;
}
```

Aici se observă foarte limpede rolul variabilelor `areStg` şi `areDrp` în formarea copacului.

Pentru a obţine copacul, vom proceda astfel: mai întâi vom genera, nivel cu nivel, toţi mugurii care descriu direcţiile de creştere şi mărimile noilor ramuri, şi îi vom păstra într-o listă numită

```
List<Mugur> planDirector,
```

apoi vom folosi această listă în generarea dintr-o singură parcurgere a întregului copac.

Prima metodă este

```
List<Mugur> formeazaPlanDirector( int nivMax = 8 )
{
    Mugur m0 = rndMugur(0);
    m0.areDrp = m0.areStg = true;
    List<Mugur> ultimaGeneratie = new List<Mugur>() { m0 };
    List<Mugur> planDirector = new List<Mugur>(ultimaGeneratie);
    for (int k = 1; k <= nivMax; k++)
    {
        genereaza(ref ultimaGeneratie, k);
        planDirector.AddRange(ultimaGeneratie);
    }
    return planDirector;
}
```

iar cea care foloseşte acest plan pentru a forma copacul cu ramurile rotite de un `Complex rotor` şi care apoi îl trasează este următoarea:

```
void traseazaCopac(Ram tr, List<Mugur> planDirector, Complex rotor)
{
    List<Ram> copac = new List<Ram> { tr };

    //construim copacul cu mugurii rotiti
    for (int k = 0; k < planDirector.Count; k++)
    {
        Mugur mug = planDirector[k];
        Ram ram = copac[k];
        Color col = intToColor(mug.nivel);
        Complex q1 = ram.q + ram.delta;
        Complex delta1 = rotor * mug.omega * ram.delta;
        if (mug.areStg) copac.Add(new Ram(q1, rotor * mug.omStg * delta1, col));
        copac.Add(new Ram(q1, delta1, col));
        if (mug.areDrp) copac.Add(new Ram(q1, rotor * mug.omDrp * delta1, col));
    }

      //trasam copacul
   copac.ForEach(printeaza);
}
```

În sfârşit, ciclul `for (double t = 0; t < 10000; t += 0.1)` în care sunt formate şi afişate cadrele animaţiei apare în `public override void makeImage()`, conform următorului text sursă:

```
using System;
using System.Drawing;
using System.Collections.Generic; //pentru List<T>

namespace ComplexExplorer
{
    public class CopacInVant : ComplexForm
    {

        public struct Mugur
        {
            public int nivel;
            public Complex omStg, omega, omDrp;   //unghiurile noilor ramuri
            public bool areStg, areDrp;   //decide daca vor creste ram stg si ram drp
```

```
    public Mugur(int nivel, Complex omStg, Complex omega, Complex omDrp,
                     bool areStg = true, bool areDrp = true)
    {
        this.nivel = nivel;
        this.omStg = omStg; this.omega = omega; this.omDrp = omDrp;
        this.areDrp = areDrp; this.areStg = areStg;
    }
}


public struct Ram
{
    public Complex q, delta; // baza si deplasarea
    public Color col;

    public Ram(Complex q, Complex delta, Color col)
    {
        this.q = q; this.delta = delta; this.col = col;
    }
}


static Complex i = new Complex(0, 1);
static double lambda = 0.9;
static double ro = 0.6;
static double theta = Math.PI / 7;

static Random rand = new Random();
double zar()
{
    return (rand.NextDouble() - 0.5) / 4;
}


Mugur rndMugur(int niv)    //returneaza un mugur aleator
{
    Mugur mg = new Mugur();
    mg.nivel = niv;
    mg.omStg = Complex.setRoTheta(ro + zar(), theta + zar());
    mg.omega = Complex.setRoTheta(lambda + zar(), zar());
    mg.omDrp = Complex.setRoTheta(ro + zar(), -theta + zar());
    mg.areStg = rand.Next(10) < 9;
    mg.areDrp = rand.Next(10) < 9;
    return mg;
}


void genereaza(ref List<Mugur> li, int niv)
{
    List<Mugur> rez = new List<Mugur>();
    foreach (Mugur mug in li)
    {   //mug este pe ramura veche, formam 3 muguri noi
        if (mug.areStg) rez.Add(rndMugur(niv));
        rez.Add(rndMugur(niv));
        if (mug.areDrp) rez.Add(rndMugur(niv));
    }
    li = rez;
}
```

```
        List<Mugur> formeazaPlanDirector(int nivMax = 8)
        {
            Mugur m0 = rndMugur(0);
            m0.areDrp = m0.areStg = true;
            List<Mugur> ultimaGeneratie = new List<Mugur>() { m0 };
            List<Mugur> planDirector = new List<Mugur>(ultimaGeneratie);
            for (int k = 1; k <= nivMax; k++)
            {
                genereaza(ref ultimaGeneratie, k);
                planDirector.AddRange(ultimaGeneratie);
            }
            return planDirector;
        }

        void printeaza(Ram r)
        {
            setLine(r.q, r.q + r.delta, r.col);
        }
        Color intToColor(int k)
        {
            return getColor(365 + 35 * k);
        }
        void traseazaCopac(Ram tr, List<Mugur> planDirector, Complex rotor)
        {
            List<Ram> copac = new List<Ram> { tr };
            //construim copacul cu mugurii rotiti
            for (int k = 0; k < planDirector.Count; k++)
            {
                Mugur mug = planDirector[k];
                Ram ram = copac[k];
                Color col = intToColor(mug.nivel);
                Complex q1 = ram.q + ram.delta;
                Complex delta1 = rotor * mug.omega * ram.delta;
                if (mug.areStg) copac.Add(new Ram(q1, rotor * mug.omStg * delta1, col));
                copac.Add(new Ram(q1, delta1, col));
                if (mug.areDrp) copac.Add(new Ram(q1, rotor * mug.omDrp * delta1, col));
            }
            //trasam copacul
            copac.ForEach(printeaza);
        }
        public override void makeImage()
        {
            setXminXmaxYminYmax(-3, 3, 0, 6);
            ScreenColor = Color.White;
            List<Mugur> planDir = formeazaPlanDirector();
            Complex q0 = 0.1 * i;
            Complex delta0 = 0.8 * i;
            Ram trunchi = new Ram(q0, delta0, intToColor(0));
            for (double t = 0; t < 10000; t += 0.1)
            {
                initScreen();
                Complex rot = Complex.setRoTheta(1, Math.Sin(t) / 10);
                trunchi.delta = rot * delta0;
                traseazaCopac(trunchi, planDir, rot);
                if (!resetScreen()) return;
                delaySec(0.05);
            }
        }
    }
}
```