Wydział Elektryczny
POLITECHNIKA WARSZAWSKA

Krzysztof DĄBROWSKI 293101 19 października 2018 r.

SPRAWOZDANIE
Lista

Cel zadania
Implementacja listy jednokierunkowej oraz napisanie testów czasu wyszukania liniowego i rekurencyjnego.

Repozytorium z kodem
https://github.com/SiwyKrzysiek/CS-List

Zaimplementowana funkcjonalność
- Genetyczność listy
- Przechodzenie i reprezentacja w formie tekstowej
- Dodawanie elementów na początek, koniec i w wybranym miejscu
- Usuwanie elementów na początku, końcu i w wybranym miejscu
- Znajdowanie elementu o podanym indeksie (rekurencyjnie i iteracyjnie)
- Znajdowanie pierwszego elementu spełniającego warunek (rekurencyjnie i iteracyjnie)
- Seria testów jednostkowych poprawności operacji na liście
- Testy badające czas wyszukiwani iteracyjnych i rekurencyjnych oraz analiza statystyczna otrzymanych wyników

Przykład wyników programu
```
Testy jednostkowe poprawne: True

Seraia pomiarów wyszukiwania elementu o losowo wybranym indeskie w liście
Testowane jest wyszukiwanie iteracyjne i rekurencyjne
--------------------------------------------------------------------------------

Test dla listy dlugosci 500

Przykladowe wyniki:
<czas iteracyjnego> <czas rekurencyjnego>

3023 3941
22 134
25 176
11 25
31 191
6 13
20 51
33 159
36 158
30 86

Srednie wartosci z 5000 próbek:
24,8878 65,8348
--------------------------------

Test dla listy dlugosci 2000
```

Przykladowe wyniki:
<czas iteracyjnego> <czas rekurencyjnego>

93 1292
32 94
96 517
101 461
78 244
57 155
113 577
20 55
18 53
59 179

Srednie wartosci z 5000 próbek:
74,8082 224,2216

Test dla listy dlugosci 6000

Przykladowe wyniki:
<czas iteracyjnego> <czas rekurencyjnego>

293 3988
115 355
318 1763
355 1966
365 2254
945 648
231 815
206 853
350 1265
195 638

Srednie wartosci z 5000 próbek:
208,7712 697,8996

Pliki źródłowe

Program.cs

```
using System;

namespace Lista
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Testy jednostkowe poprawne: {0}\n",
Tests.RunAllTests());

            Tests.FullSpeedTest();
             }
    }
}
```

Node.cs

```
using System;

namespace Lista
{
```

```
    public class Node<T>
    {
        public T Value { get; set; }
        public Node<T> Next { get; set; }

        public Node(T value)
        {
            Value = value;
        }
    }
}
```

List.cs

```
using System;
using System.Text;

namespace Lista
{
    /// <summary>
    /// Klasa listy jednokierunkowej.
    /// Zawiera podstawowe operacje na listach
    /// </summary>
    public class List<T>
    {
        private Node<T> head;
        private Node<T> tale;

        /// <summary>
        /// Ilosc elementow listy
        /// </summary>
        public int Lenght { get; private set; }

        public List()
        {
            //Pola sa automatycznie inicjalizowane poprawnymi wartosciami
        }

        /// <summary>
        /// Dodaje element na koniec listy
        /// </summary>
        public void PushBack(T element)
        {
            Lenght++;

            Node<T> newNode = new Node<T>(element);
            if (head == null) //Lista pusta
            {
                head = newNode;
                tale = newNode;
            }
            else
            {
                tale.Next = newNode;
                tale = newNode;
            }
        }

        /// <summary>
        /// Dodaje element na poczatek listy
        /// </summary>
        public void PushFront(T element)
        {
```

```
            Lenght++;

            Node<T> newNode = new Node<T>(element);
            if (head == null) //Lista pusta
            {
                head = newNode;
                tale = newNode;
            }
            else
            {
                newNode.Next = head;
                head = newNode;
            }
        }

        /// <summary>
        /// Usuwa z listy pierwszy element
        /// </summary>
        public void PopFront()
        {
            head = head.Next;
            Lenght--;
        }

        /// <summary>
        /// Usuwa z listy ostatni element
        /// </summary>
        public void PopBack()
        {
            if (Lenght <= 0)
                throw new InvalidOperationException("Can't pop from empty list");
            if (Lenght == 1)
            {
                head = null;
                tale = null;
                Lenght--;

                return;
            }

            Node<T> previous=head, i = head;

            while(i.Next != null)
            {
                previous = i;
                i = i.Next;
            }

            previous.Next = null;
            tale = previous;
            Lenght--;
        }

        /// <summary>
        /// Usuwa z lisyt element o podanym indeksie
        /// </summary>
        public void RemoveAtIndex(int index)
        {
                    if (index >= Lenght)
                                throw new IndexOutOfRangeException("Element with
given index does not exist");
            if (index == Lenght-1)
```

```
            {
                PopBack();
                return;
            }
            if (index == 0)
            {
                PopFront();
                return;
            }

            int currentIndex = 0;
            Node<T> previous = head;
                    for (Node<T> i = head; i != null; i = i.Next)
                    {
                if (currentIndex == index)
                {
                    previous.Next = i.Next;
                    Lenght--;
                    return;
                }

                previous = i;
                currentIndex++;
                    }
        }

        /// <summary>
        /// Wstawia element do listy po elemencie o danym indeksie
        /// </summary>
        public void InsertAfterIndex(int index, T value)
        {
                    if (index >= Lenght)
                            throw new IndexOutOfRangeException("Element with given
index does not exist");

            Node<T> temp = head;
            for (int i = 0; i < index; i++)
            {
                temp = temp.Next;
            }

            Node<T> newNode = new Node<T>(value);
            newNode.Next = temp.Next;
            temp.Next = newNode;

            Lenght++;
        }

        /// <summary>
        /// Zwraca element o podanym indeksie
        /// </summary>
        public T FindAtIndex(int index)
        {
            if (index >= Lenght)
                            throw new IndexOutOfRangeException("Element with given
index does not exist");

                    int currentIndex = 0;
            for (Node<T> i = head; i != null; i = i.Next)
            {
                if (index == currentIndex++)
                    return i.Value;
```

```
            }

            throw new IndexOutOfRangeException("Element with given index does not
exist");
        }

        /// <summary>
        /// Zwraca element o podanym indeksie.
        /// Dziala rekurencyjnie
        /// </summary>
        public T RecurentFindAtIndex(int index)
        {
            if (index >= Lenght)
                throw new IndexOutOfRangeException("Element with given index does not
exist");

            return RecurentHidenFindAtIndex(index, head);
        }

        private T RecurentHidenFindAtIndex(int index, Node<T> myHead)
        {
            if (index > 0)
                return RecurentHidenFindAtIndex(index - 1, myHead.Next);

            return myHead.Value;
        }

        /// <summary>
        /// Zwraca wartosc ktora wraz z element spelni funkcje whatToFind.
        /// Dziala rekurencyjnie
        /// </summary>
        /// <returns>Pierwszy element spelniajacy funkcje</returns>
        public T RecurentFindElement(Func<T, bool> whatToFind)
        {
            return RecurentHiddenFindElement(whatToFind, head);
        }

        private T RecurentHiddenFindElement(Func<T, bool> whatToFind, Node<T> myHead)
        {
            if (whatToFind(myHead.Value))
                return myHead.Value;

            if (myHead.Next != null)
                return RecurentHiddenFindElement(whatToFind, myHead.Next);

            throw new Exception("No element found");
        }

        /// <summary>
        /// Zwraca wartosc ktora wraz z element spelni funkcje whatToFind
        /// </summary
        /// <returns>Pierwszy element spelniajacy funkcje</returns>
        public T FindElement(Func<T, bool> whatToFind)
        {
                    for (Node<T> i = head; i != null; i = i.Next)
                    {
                if (whatToFind(i.Value))
                    return i.Value;
                    }

            throw new Exception("No element found");
        }
```

```
        /// <summary>
        /// Zwraca postac tekstowa listy
        /// </summary>
        public override string ToString()
        {
            StringBuilder stringBuilder = new StringBuilder();

            for (Node<T> i = head; i != null; i = i.Next)
            {
                stringBuilder.Append($"{i.Value.ToString()}->");
            }

            stringBuilder.Remove(stringBuilder.Length-2, 2);

            return stringBuilder.ToString();
        }
    }
}
```

Tests.cs

```
using System;
using System.Diagnostics;

namespace Lista
{
    public class Tests
    {
        private static readonly Random random = new Random();

        /// <summary>
        /// Wykonanie testow jednostkowych
        /// </summary>
        /// <returns><c>true</c> if all tests were passed correctly <c>false</c>
otherwise.</returns>
        public static bool RunAllTests()
        {
            return ToStringTest() && AddRemoveTest() && AddRemoveTest(50) &&
PushBackTest()
                && PushBackTest() && InsertAfterIndexTest();
        }

        /// <summary>
        /// Sprawdza przechodzenie listy i wypisanie jej elementow
        /// </summary>
        /// <returns><c>true</c> if test was passed correctly<c>false</c>
otherwise.</returns>
        public static bool ToStringTest()
        {
                    List<int> list = new List<int>();

                    list.PushBack(3);
                    list.PushBack(24);
                    list.PushBack(-4);

            return list.ToString() == "3->24->-4";
             }

             /// <summary>
             /// Test wstawiania elementu do listy
             /// </summary>
```

```
        /// <returns><c>true</c> if test was passed correctly<c>false</c>
otherwise.</returns>
        public static bool InsertAfterIndexTest()
    {
            List<int> list = new List<int>();

            list.PushBack(3);
            list.PushBack(24);
            list.PushBack(17);

        list.InsertAfterIndex(1, 8);

        return list.ToString() == "3->24->8->17";
    }

        /// <summary>
        /// Test wstawiania elementu na koniec listy
        /// </summary>
        /// <returns><c>true</c> if test was passed correctly<c>false</c>
otherwise.</returns>
        public static bool PushBackTest()
    {
        int howManyElements = random.Next(300);
        List<int> list = new List<int>();

        for (int i = 0; i < howManyElements; i++)
        {
            list.PushBack(random.Next(99));
        }

        return list.Lenght == howManyElements;
    }

        /// <summary>
        /// Test usuwania elementu z listy
        /// </summary>
        /// <returns><c>true</c> if test was passed correctly<c>false</c>
otherwise.</returns>
        public static bool RemoveAtIndexTest()
    {
        List<int> list = new List<int>();

            list.PushBack(4);
            list.PushBack(8);
            list.PushBack(2);
            list.PushBack(17);
            list.PushBack(11);
            list.PushBack(9);

        list.RemoveAtIndex(2);

        return (list.ToString() == "4->8->17->11->9" && list.Lenght == 5);
    }

        /// <summary>
        /// Test zapelnienia i oproznienia listy
        /// </summary>
        /// <returns><c>true</c> if test was passed correctly<c>false</c>
otherwise.</returns>
        public static bool AddRemoveTest(int howManyElementsToAdd = 1)
    {
        List<int> list = new List<int>();
```

```
            for (int i = 0; i < howManyElementsToAdd; i++)
            {
                list.PushBack(random.Next(100));
            }

                    for (int i = 0; i < howManyElementsToAdd; i++)
                    {
                           list.PopBack();
                    }

            return list.Lenght == 0;
        }

        /// <summary>
        /// Sprawdza ktore przeszukiwanie listy jest szybsze
        /// </summary>
        public static void FullSpeedTest()
        {
            const int howManyTestsPerSires = 5000;
            int[] listLengthsToTest = {500, 2000, 6000};

            //SingleSpeedTest(6000, howManyTestsPerSires);

            string message = "Seraia pomiarów wyszukiwania elementu o losowo wybranym
indeskie w liście\n" +
                             "Testowane jest wyszukiwanie iteracyjne i rekurencyjne\n"
+
                             "----------------------------------------------------------
------------------\n";
            Console.WriteLine(message);

            foreach (int listLength in listLengthsToTest)
            {
                SingleSpeedTest(listLength, howManyTestsPerSires);
            }
        }

        /// <summary>
        /// Wykonuje serie pomiarow rekurencyjnego i iteracyjnego wyszukania elementu.
        /// Wyświetla kilka przykladowych czasow i srednia wszystkich wynikow
        /// </summary>
        /// <param name="listSize">Dlugosc testowanej listy</param>
        /// <param name="testsAmount">Ilosc testow do przeprowadzenia</param>
        private static void SingleSpeedTest(int listSize, int testsAmount)
        {
            Tuple<long, long>[] results = new Tuple<long, long>[testsAmount];

            for (int i = 0; i < testsAmount; i++)
            {
                results[i] = MeasureSearchTime(listSize);
            }
            Tuple<double, double> means = CalculateMeanOfResults(results);

            int resultsToDisplay = Math.Min(10, testsAmount); //Wyswietlam co najwyzej
10 wynikow

            string beginningMessage = $"Test dla listy dlugosci {listSize}\n\n" +
                                      "Przykladowe wyniki:\n" +
                                      "<czas iteracyjnego> <czas rekurencyjnego>\n";
            Console.WriteLine(beginningMessage);
```

```
            for (int i = 0; i < resultsToDisplay; i++)
            {
                Console.WriteLine($"{results[i].Item1} {results[i].Item2}");
            }

            Console.WriteLine($"\nSrednie wartosci z {testsAmount}
próbek:\n{means.Item1} {means.Item2}\n" +
                              "------------------------------\n");
        }

        /// <summary>
        /// Liczy srednia z wynikow pomiarow
        /// </summary>
        /// <returns>Krotka (srednia iteracji, srednia rekurencji)</returns>
        public static Tuple<double, double> CalculateMeanOfResults(Tuple<long, long>[]
results)
        {
            long sumOfIterationTimes = 0;
            long sumOfRecursiveTimes = 0;

            foreach (Tuple<long, long> result in results)
            {
                sumOfIterationTimes += result.Item1;
                sumOfRecursiveTimes += result.Item2;
            }

            double iterationsMean = (double) sumOfIterationTimes / results.Length;
            double recursivesMean = (double) sumOfRecursiveTimes / results.Length;

            return Tuple.Create(iterationsMean, recursivesMean);
        }

        /// <summary>
        /// Mierzy czas znalezienie losowego elementu w liscie o podanej dlugosci.
        /// Testowane jest przeszukanie iteracyjne i rekurencyjne
        /// </summary>
        /// <returns>
        /// Krotka (czas szukania iteracyjnego, czas szukania rekurencyjnego)
        /// Zwracane wartosci sa w cyklach zegara
        /// </returns>
        private static Tuple<long, long> MeasureSearchTime(int listSize)
        {
            List<int> list = GenerateRandomList(listSize);
            int indexToFind = random.Next(listSize);

            long timeOfInternationalSearch;
            long timeOfRecursiveSearch;

            Stopwatch stopwatch = Stopwatch.StartNew();
            list.FindAtIndex(indexToFind);
            stopwatch.Stop();
            timeOfInternationalSearch = stopwatch.ElapsedTicks;

            stopwatch = Stopwatch.StartNew();
            list.RecurentFindAtIndex(indexToFind);
            stopwatch.Stop();
            timeOfRecursiveSearch = stopwatch.ElapsedTicks;

            return Tuple.Create(timeOfInternationalSearch, timeOfRecursiveSearch);
        }

        /// <summary>
```

```
        /// Tworzy liste losowych liczb calokowitych
        /// </summary>
        /// <param name="length">Dlugosc generowanej listy</param>
        public static List<int> GenerateRandomList(int length)
        {
            List<int> list = new List<int>();

            for (int i = 0; i < length; i++)
            {
                list.PushBack(random.Next(1000000));
            }

            return list;
        }
    }
}
```