SVEUČILIŠTE U SPLITU

FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

ALGORITMI I STRUKTURE PODATAKA (220)

Izvještaj laboratorijskih vježbi

Tomislav Babac

Split, veljača 2024.

Vježba 1. Metode i kriteriji

1. Zadatak:

U prvom zadatku trebamo napraviti projekt tako da koristimo C# jezik te Console app(.NET Framework). Cilj prvog zadatka je implementirati metodu za zamjenu vrijednost dvaju cijelih brojeva (odnosno SWAP metoda). Također upotrebu implementirane swap funkcije ćemo primjeniti na nekom nizu brojeva.

Programski kod:

```
namespace Parameters
{
    class Program
    {
        public static void Swap(ref int a, ref int b)
        {
            int temp = a;
            a = b;
            b = temp;
        }

        static void Main(string[] args)
        {
            int a = 1;
            int b = 2;

            Console.WriteLine("a= "+a);
            Console.WriteLine("b= "+b);

            Swap(ref a, ref b);

            Console.WriteLine("a= " + a);
            Console.WriteLine("b= " + b);

            int[] numbers = { 2, 3, 4, 1, 8, 6, 5, 7 };

            foreach(int num in numbers)
            {
                Console.WriteLine(num);
            }

            Swap(ref numbers[2], ref numbers[3]);

            foreach (int num in numbers)
            {
                Console.WriteLine(num);
            }
        }
    }
}
```

ISPIS:

Komentar:

Swap medota realizirana je na način da prima dva argumenta nazvani *a* i *b* putem reference. Kada metoda prima argumente putem reference, to znači da prima njihove adrese te sve promjene koje se događaju unutar te metode sa tako prenesenim argumentima će se preslikati i u memoriji. U ovom slučaju, metoda *Swap* će zamijeniti vrijednosti prosljeđenim argumentima (vrijednost prosljeđena putem argumenta a će biti zapisana na memorijsku lokaciju argumenta *b* i obrnuto. Da nema ključne riječi *ref* ispred argumenata funkcije, varijable *a* i *b* koje su sada lokalne varijable funkcije i nemaju nikakve veze sa onima unutar *Main* funkcije bi zamijenile vrijednosti i po završetku funkcije bi se izbrisale što naravno ne bi promijenilo vrijednosti istoimenih varijabli koje su proslijeđene kao argument toj funkciji.

Unutar *Main* programa provjerena je funkcionalnost implementirane metode na primjeru zamjene dvije varijable i dva elementa niza.

2. Zadatak:

Cilj ovog projekta bio je implementirati C# konzolnu aplikaciju koja demonstrira upotrebu sučelja IComparable za usporedbu i sortiranje objekata klase Student prema ocjenama. Projekt se sastoji od dvije klase, Student i Bubble, te koristi sortiranje mjehurićem za sortiranje niza objekata Student prema njihovim ocjenama.

Programski kod (Class Student):

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

namespace Comparables
{
    class Student:IComparable
    {
        private string name;
        private double grade;

        public Student(string name,double grade)
        {
            this.name = name;
            this.grade = grade;
        }

        public override string ToString()
        {
            return name + ": " + grade;
        }

        public int CompareTo(object obj)
        {
            Student other = obj as Student;

            if (grade > other.grade)
                return -1;
            else if (grade < other.grade)
                return 1;
            return 0;
```

```
            }
        }
    }
```

Komentar:

Novokreirana klasa Student sadrži privatne članove za ime(**string** tipa) i ocjenu(**double** tipa) studenta te istoimena klasa će nasljeđivat **IComparable** interface. Definiramo konstruktor koji će postavljati privatne varijable na određenu vrijednost. Implementirali smo public metodu **CompareTo** koji prima objekt i vraća vrijednost **-1** u slučaju da je vrijednost **grade** veća od **other.grade,** a u suprotnom vraća **1.** U else funkciji imamo ukoliko su vrijednosti jednake vraća **0**.

```
public int CompareTo(object obj)
        {
            Student other = obj as Student;

            if (grade > other.grade)
                return -1;
            else if (grade < other.grade)
                return 1;
            return 0;
```

Programski kod (Class Bubble):

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

namespace Comparables
{
    class Bubble
    {
        public static void Sort(IComparable[] array)
        {
            for (int i = 0; i < array.Length - 1; i++)
                for (int j = i + 1; j < array.Length; j++)
                    if (array[j].CompareTo(array[i]) > 0)
                    {
                        IComparable temp = array[j];
                        array[j] = array[i];
                        array[i] = temp;
                    }
        }

}
}
```

Komentar:

Klasa Bubble sadrži statičku metodu Sort koja koristi algoritam „Bubble Sort“ koju koristimo za sortiranje zadanog niza objekata uz nasljeđivanje IComparable sučelja. Ovaj algoritam je realiziran pomoću dvije for petlje koje funkcioniraju tako da u prvoj petlji se uzme prvi član niza (startna vrijednost iteratora **i**=0) te u drugoj petlji pomoću **j** iteratora prođe sve članove niza i usporedi ih sa prvim članom. Ukoliko je član array[j] manji od array[i], oni se zamjenjuju tako da se dobiva niz sortiran od manjeg prema većem. Nakon što je petlja prošla sve članove inkrementira se **i** iterator i ponovno se uspoređuju svi članovi sa **i++**.

Programski kod (Main):

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

namespace Comparables
{
    class Program
    {
        static void Main(string[] args)
        {
            Student[] students = {new Student ("Ivo", 4.1),new Student ("Ana",
4.9),new Student ("Iva", 4.3),new Student ("Bob", 4.5),new Student ("Joe", 4.7)};


            foreach(Student s in students)
            {
                Console.WriteLine(s);
            }
            Bubble.Sort(students);

            Console.WriteLine("---------------");

            foreach (Student s in students)
            {
                Console.WriteLine(s);
            }

            Console.ReadKey();
        }
    }
}
```

Komentar:

Unutar Main funkcije deklariramo niz studenata. Obavimo sortiranje niza (po ocjeni budući da je tako implementirana CompareTo metoda). Na kraju, ispišemo sortirani niz studenata.

ISPIS:

3. Zadatak:

U ovom zadatku cilj nam je sortirati zadani niz studenata prema zadano **kriteriju**(„Criterion“). Kriterij nam je sortiranje ili po ocjeni ili po nizu. Implementiramo sustav klasa koristeći **Icomparer** interface i njegovu metodu **CompareTo** za usporedbu dva objekta, te enumeraciju.

Programski kod (Class Student):

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

namespace Comparers
{
    class Student
    {
        public string name;
        public double grade;

        public Student(string name, double grade)
        {
            this.name = name;
            this.grade = grade;
        }

        public override string ToString()
        {
            return name + ": " + grade;
        }
    }
}
```

Komentar:
Klasa Student je implementirana kao i u prošlom zadatku, ali ne koristi nikakav interface. U posljednjem redu smo napravili *override* metodu **ToString** za ispis. Klasa Student sadrži dvije varijable **name** i **grade** te pomoću konstruktora postavljamo željenu vrijednost.

Programski kod (Class StudentComparer):

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

namespace Comparers
{
    enum StudentComparerType { Name, Grade }
    class StudentComparer : IComparer
    {
        private StudentComparerType criterion;
```

```
        public StudentComparer(StudentComparerType criterion)
        {
            this.criterion = criterion;
        }

        public int Compare(object o1, object o2)
        {

            Student s1 = o1 as Student;
            Student s2 = o2 as Student;
            switch (criterion)
            {
                case StudentComparerType.Name:
                    return s1.name.CompareTo(s2.name);
                case StudentComparerType.Grade:
                    return s2.grade.CompareTo(s1.grade);
                default:
                    throw new Exception("Kriterij ne postoji");
            }
        }
    }
}
```

Komentar:
Klasa StudentComparer koristi **IComparer** interface radi usporedbe studenata. **Compare** metoda je implementirana na način da ukoliko je kao kriterij poslana var. **Name** onda uspoređuje dva studenta po imenu i vraća iste vrijednosti kao **CompareTo**. Ukoliko je kriterij **Grade** onda uspoređuje studente po ocjenama.

Criterion metoda može biti **Name** ili **Grade** koje su pohranjenje u enumeraciji **StudentComparerType.**

```
        enum StudentComparerType { Name, Grade }
```

Programski kod (Class Bubble):

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

namespace Comparers
{
    class Bubble
    {
        public static void Sort(object[] array, IComparer comparer)
        {
            for (int i = 0; i < array.Length - 1; i++)
                for (int j = i + 1; j < array.Length; j++)
                    if (comparer.Compare(array[j], array[i]) < 0)
                    {
                        object temp = array[j];
                        array[j] = array[i];
                        array[i] = temp;
                    }
        }
    }
}
```

Komentar:
Klasa Bubble je implementirana kao i u prethodnom zadatku uz malu promjenu, a to je da umjesto **CompareTo** metode (*IComparable* interface) koristi **Compare** metoda (*IComparer* interface) koja kao argument prima dva objekta koja uspoređuje.

Programski kod (Main):

```
using System;

namespace Comparers
{
    class Program
    {
        static void Main(string[] args)
        {
           Student[] students = {
               new Student ("Ivo", 4.1),
               new Student ("Ana", 4.9),
               new Student ("Iva", 4.3),
               new Student ("Bob", 4.5),
               new Student ("Joe", 4.7)
 };
            StudentComparer comparer = new
StudentComparer(StudentComparerType.Name);
            Bubble.Sort(students, comparer);

            foreach(Student s in students)
                Console.WriteLine(s);

            Console.WriteLine("-----------");

            StudentComparer comparer2 = new
StudentComparer(StudentComparerType.Grade);
            Bubble.Sort(students, comparer2);

            foreach (Student s in students)
                Console.WriteLine(s);

        }
    }
}
```

Komentar:
Inicijaliziramo niz Student kojemu pridjelimo vrijednosti kao što su ime i pripadajuća ocjena (Odnosno sortiran po našem *Kriteriju,* po **imenu** i po **ocjeni**). Prvo ćemo ispisati originalni niz te onda stvoriti StudentComparer objekt za usporedbu po **imenu**, sortiranje i prikaz rezultata. Radimo istu stvar ali samo je sad *kriterijusporedba* po **ocjeni**.

4. Zadatak:

Cilj ovog zadatka je pozivati funkcije putem delegata. Delegati u programskom jeziku C# su referentni tipovi koji se koriste za učahurivanje metoda. U delegata se može učahuriti bilo koja odgovarajuća metoda (s određenim potpisom i povratnim tipom) – slično kao pointeri na funkcije u C++-u. Delegat se stvara pomoću ključne riječi delegate iza koje se navodi povratni tip i potpis metoda koje mu se mogu delegirati.

Programski kod (Main):

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

namespace Delegates
{
    class Program
    {
        delegate void Invoker(int i);

        static void Method(Invoker invoker, int value)
        {
            invoker(value);
        }
        static void Display(int i)
        {
            Console.WriteLine("Displaying " + i);
        }
        static void Print(int i)
        {
            Console.WriteLine("Printing " + i);
        }

        static void Main(string[] args)
        {

            //Invoker invoker=new Invoker(Display);
            Invoker invoker = Display;

            // invoker.Invoke(1);
            invoker(1);

            invoker = Print;
            invoker(2);

            invoker += Display;
            invoker(3);

            // invoker-=Display; invoker(35);

            Method(Display, 4);

            Method(Print, 5);

            invoker = delegate (int i)
            {
                Console.WriteLine("Annonimously: " + i);
            };
```

```
            invoker(6);

            invoker = i => Console.WriteLine("Lambdaing " + i);
            invoker(7);

        }
    }
}
```

Komentar:
U ovom projektu prvo smo definirali delegat imena **Invoker** koji prima varijablu **int i**, te nema povratnu vrijednosti, odnosno *void*

```
delegate void Invoker(int i);
```

Delegatu imena **invoker** ćemo pridružiti funkciju **Display** koju smo već prije implementirali, a one su jednostavne metode koje ispisuju poruke na konzolu. **Method** poziva delegat koji mu je proslijeđen kao argument poruke na konzolu. Delegati nisu striktno vezani za jednu funkciju, kao što vidimo na primjeru ispod da smo pridjelili **invoker** delegatu *Display* i *Print* funkciju

```
Invoker invoker = Display;
invoker = Print;
```

Zgodno je koristiti M**ulticast Delegate,** odnosno višestruke delegate koji nam omogućavaju da za jedan delegat možemo pridjeliti više funkcija, odnosno pozivom jednog delegata možemo izvršiti dvije ili više funkcija. Pridjeljivanje ili oduzimanje funkcija delegatu se odrađuje „**+**“ ili „**-**„ matematičkom operacijom.

```
invoker += Display;
```

Moguće je delegatu pridružiti metodu bez striktnog navođenja imena metode:

```
invoker = delegate (int i)
            {
                Console.WriteLine("Annonimously: " + i);
        };
```

ili putem lambda expressiona:

```
invoker = i => Console.WriteLine("Lambdaing " + i);
```

5. Zadatak:

Cilj ovog zadatka je korištenje *delegata* kako bi implementirali sortiranje niza uz zadani kriterij

Klasa Student je identična kao i u prethodnim zadacima, uz dodatak:

```
public static bool CompareName(object a, object b)
        {
            Student st1 = a as Student;
            Student st2 = b as Student;

            if (st1.name.CompareTo(st2.name) < 0) return true;
            else return false;

        }
        public static bool CompareGrade(object a, object b)
        {
            Student st1 = a as Student;
            Student st2 = b as Student;

            return st1.grade > st2.grade;

        }
```

Komentar:

U 5. zadatku u klasi Student ne koristimo više interface već implementiramo dvije nove članske metode **CompareGrade** i **CompareName** koja uspoređuje dva studenta, odnosno njegove ili ocjene ili ime. Obje funkcije su bool tipa, što znači da vraćaju *jedan* ukoliko je uvjet zadovoljen ili *nula* ukoliko ne zadovoljava uvjet.

Programski kod (class Bubble):

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

namespace Comparisons
{
    delegate bool Comparison(object a, object b);
    class Bubble
    {
        public static void Sort(object[] array, Comparison comparison)
        {
            for (int i = 0; i < array.Length - 1; i++)
            {
                for (int j = i + 1; j < array.Length; j++)
                {
                    if (comparison(array[j], array[i]))
                    {
                        object temp = array[i];
                        array[i] = array[j];
                        array[j] = temp;
                    }
                }
            }
        }
    }
```

```
}
```

Komentar:
Van klase definiran je delegat čiji je povratni tip *bool* (vraća 0 ili 1) te prima dva objekta (*a* i *b*):

```
delegate bool Comparison(object a, object b);
```

Klasa Bubble sadrži delegat **Comparison** i statičku metodu **Sort** koja koristi *Bubble algoritam* za sortiranje niza objekata. Također kod sortiranja metoda prima i kriterij po kojem će vršiti sljedeće sortiranje, te putem tog delegata (Comparison) imat ćemo dvije metode **CompareName** ili **CompareGrade** u ovisnosti o *kriteriju*.

Programski kod (Main):

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

namespace Comparisons
{
    class Program
    {
        static void Main(string[] args)
        {
            Student[] students = {
 new Student ("Ivo", 4.1),
 new Student ("Ana", 4.9),
 new Student ("Iva", 4.3),
 new Student ("Bob", 4.5),
 new Student ("Joe", 4.7)
 };

            foreach (Student st in students)
            {
                Console.WriteLine(st);
            }

            Bubble.Sort(students, Student.CompareGrade);
            Console.WriteLine("after sort by grade");
            foreach (Student st in students)
                Console.WriteLine(st);

            Bubble.Sort(students, Student.CompareName);
            Console.WriteLine("after sort by name");
            foreach (Student st in students)
                Console.WriteLine(st);

        }
    }
}
```

Komentar:
U main funkciji kao i u prethodnim zadacima inicijaliziramo niz Student sa njihovim pripadnim imenima i ocjena kao što nam i kriterij zahtjeva. Metodom **Sort** u klasi **Bubble** sortiramo cijeli niz na temelju *Bubble Sort algoritma*. Dva puta pozivamo **Bubble.Sort**, prvi put sa kriterijom

ocjene (Student.CompareGrade) putem delegata, a drugi put sa kriterijom *imena* (Student.CompareName)putem delegata, te nakon usporedbe ispisujemo na konzolu.

Vježba 2. Pretraživanje

1. Zadatak:

Cilj ovog zadatka je bila implementacija i poziv **rekurzivne metode** te vrijeme i efikasnost njenog izvođenja u odnosnu na **iterativne metode**. Pomoću rekurzije napisat ćemo kod za računanje faktorijel nekog broja i ispis Fibonaccijevog niza na određenoj poziciji.

Programski kod:

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

namespace Recursion
{
    class Program
    {

        static int Factorial(int n)
        {
            if (n <= 1) return 1;
            return n * Factorial(n - 1);
        }

        static int Fibonacci(int n)
        {
            if (n <= 2)
                return 1;
            else
                return Fibonacci(n - 1) + Fibonacci(n - 2);
        }

static int FibonacciIter(int n)
        {
            int[] fibos = new int[n + 1];
            fibos[0] = 0;

            if (n > 0)
            {
                fibos[1] = 1;
                for (int i = 2; i <= n; i++)
                    fibos[i] = fibos[i - 1] + fibos[i - 2];
            }
            return fibos[n];
        }

        static void Main(string[] args)
        {
            Stopwatch sw = new Stopwatch();
            sw.Start();
            int result = Factorial(6);
            sw.Stop();
```

```
            Console.WriteLine("Result of factorial: " + result + " exec. time:
" + sw.Elapsed);

            sw.Start();
            result = Fibonacci(6);
            sw.Stop();
            Console.WriteLine("Result of fibonacci: " + result + " exec. time:
" + sw.Elapsed);


            sw.Start();
            result = FibonacciIter(6);
            sw.Stop();
            Console.WriteLine("Result of fibonacci iterative: " + result + "
exec. time: " + sw.Elapsed);

        }
    }
}
```

Komentar:

U zadatku imamo implementaciju rekurzivne funkcije kojom smo realizirali računanje faktorijela i Fibonaccijevog niza. **Rekurzivna funkcija** ima *osnovni slučaj* koji rješava i vraća traženu vrijednost i *rekurzivni slučaj* kada funkcija poziva samu sebe i izvršava osnovni slučaj i distribuciju zadataka. Metoda **Factoriel** računa faktorijel primljenog **n** broja. Ukoliko je broj 0 ili 1 faktorijel iznosi 1. Za sve ostale brojeve se matematički računa kao npr 3! = 2!*3 (2!=1*2). Zbog toga rekurzivno definiramo funkc. Factoriel da sama sebe poziva za neki broj n:

```
return n * Factorial(n - 1);
```

Metoda **Fibonacci** računa vrijednost elementa Fibonaccijevog niza na određenoj poziciji. Sljedeći broj u Fibonaccijevom nizu se dobije kao zbroj prethodna dva (Fibonaccijev niz: 1 1 2 3 5 8,..) Ukoliko se traži vrijednost Fibonaccijevog niza na poziciji 1 ili 2 funkcija vraća 1. Vrijednost elementa dobijemo zbrojem prethodna 2 broja u nizu te se ta metoda poziva rekurzivno:

```
return Fibonacci(n - 1) + Fibonacci(n - 2);
```

Metoda **Fibonaccilter** je iterativna implementacija kod koje vidimo da više nemamo samostalno pozivanje funkcije, već postoji iteracija kroz koju funkcija prolazi dok ne zadovolji uvjet koji smo postavili. Kada funkcija zadovolji uvjet, ona se prekida i ispisuje traženu vrijednost.

```
for (int i = 2; i <= n; i++){
       fibos[i] = fibos[i - 1] + fibos[i - 2];
            }
```

Uglavnom rekurzivne funkcije je lakše implementirati od iterativnih, ali su dosta *sporije* kod funkcija sa velikim brojem elemenata koje se računaju. Dok je kod iterativnih funkcija obrnuto.

2. Zadatak:

Cilj ovog zadatka je implementirati algoritme za pretraživanje: Sequential Search (iterativno) i Binary Search (iterativno i rekurzivno).

Programski kod:

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

namespace Searching
{
    class Program
    {

        static public int SequentialSearch(int[] array, int num)
        {
            for (int i = 0; i < array.Length; i++)
            {
                if (array[i] == num)
                {
                    return i;
                }
            }
            return -1;
        }

        static public int BinarySearch(int[] array, int num)
        {
            int low = 0;
            int high = array.Length - 1;

            while (low <= high)
            {

                int mid = (low + high) / 2;

                if (num == array[mid])
                {
                    return mid++;
                }
                else if (num < array[mid])
                {
                    high = mid - 1;
                }
                else low = mid + 1;


            }
            return -1;
        }

        public static int BinarySearchRec(int[] inputArray, int key, int
istart, int iend)
        {
            istart -= 1;
            iend -= 1;
```

```
            while (istart <= iend)
            {
                int mid = (istart + iend) / 2;
                if (key == inputArray[mid])
                {
                    return mid;
                }
                else if (key < inputArray[mid])
                {
                    iend = mid - 1;
                    BinarySearchRec(inputArray, key, istart, iend);
                }
                else
                {
                    istart = mid + 1;
                    BinarySearchRec(inputArray, key, istart, iend);
                }
            }
            return -1;
        }

        static void Main(string[] args)
        {
            int[] array = new int[7] { 5, 6, 9, 12, 45, 8, 7 };
            Array.Sort(array);
            Stopwatch sw = new Stopwatch();

            foreach (int i in array)
            {
                Console.WriteLine(i + ", ");
            }
            sw.Start();
            int index = SequentialSearch(array, 45);
            sw.Stop();
            Console.WriteLine("Index: " + index + " exec time: " + sw.Elapsed);

            sw.Start();
            index = BinarySearch(array, 45);
            sw.Stop();
            Console.WriteLine("Index: " + index + " exec time: " + sw.Elapsed);

            sw.Start();
            index = BinarySearchRec(array, 45, 0, 7);
            sw.Stop();
            Console.WriteLine("Index: " + index + " exec time: " + sw.Elapsed);
        }
    }
}
```

Komentar:

Sekvencijalno pretraživanje se koristi za traženje elemenata u *nesortiranim nizovima*. Mora se provjeriti svaki element. Algoritamska kompleksnost O(n).
Binarno pretraživanje se koristi za traženje elemenata u *sortiranim nizovima* (i samo se na njima može primjeniti.Ne mora se provjeriti svaki element. Algoritamska kompleksnost O(log n).

Metoda **SequentialSearch** je implementirana tako da kao ulazne argumente prima cijeli niz int[] **array** i traženu vrijednost **num**. *For* petljom prolazimo kroz cijeli niz

provjeravajuci stalno je li je trenutna vrijednost niza jednaka vrijednost **num** koju tražimo. Ukoliko je pronađemo, vraćamo poziciju tog elementa u nizu. U suprotnome, vraćamo vrijednost -1 koja simbolizira da se cijela petlja izvrtila, odnosno prošli smo kroz sve elemente i nismo našli traženu vrijednost.

Metoda **BinarySearch** se može implementirati samo na već sortiranim nizovima. Prvo ćemo *originalni niz* podijeliti na 2 podniza i pamtimo početni i krajnji ideks, te indeks srednjeg el. tog niza. Definiramo *mid* (srednju) vrijednost mid= (low+high)/2. *Low* je najmanja vrijednost, a *High* je najveća vrijednost niza. Ukoliko je vrijednost **num** koju tražimo u nizu manja od *mid* tada uzimamo lijevu stranu niza, odnosno „prvi dio“ niza(1. podniz) te s njim dalje radimo i nastavljamo tražiti, eliminirajući drugi podniz. Isto vrijedi ukoliko je **num** veći ili jednak *mid* uzimamo „drugi dio“ niza(2. podniz). Ako nismo pronašli traženu vrijednost, vraćamo -1.

Vidimo da je **BinarySearch** efikasnija metoda jer u startu eliminiramo dio niza i nastavljamo raditi samo s polovicom niza, što nam može puno uštedjeti na vremenu za razliku od **SequencialSearch** gdje moramo proći kroz svaki element niza. Velika mana kod binarnog pretraživanja je šta striktno mora biti sortirani niz, a kod sekvencijalnog ne mora što nam govori da ovisno što nam treba uzimamo i implementiramo u svoj kod.

3. Zadatak:

Cilj ovog zadatka je implementirati klasu Smart Array („pametni niz“) koji ima mogućnost promjene veličine niza prilikom uklanjanja ili dodavanja elementa u niz.

Programski kod (class Smart Arrays):

```
using System;

using System.Collections;

namespace Smart_Arrays
{
    class Smart_Arrays
    {
        public int size;
        public int last = -1;
        int[] array;

        public Smart_Arrays(int size)
        {
            this.array = new int[size];
            this.size = array.Length;
        }
        public int this[int index]
        {
            get { return array[index]; }
            set { array[index] = value; }
        }
        public int Length
        {
            get { return last + 1; }
```

```
        }
        public void Add(int item)
        {
            if (last == (size - 1))
            {
                Array resized =
               Array.CreateInstance(typeof(int), size * 2);
                Array.Copy(array, resized, size);

                array = (int[])resized;
                size = array.Length;
            }
            array[++last] = item;
        }
        public void Remove(int item) {
        for(int i=0;i<this.Length; i++)
            {
                if (array[i] == item)
                {
                    Array.Copy(array, i + 1, array, i, last - i);
                    last--;
                    break;
                }
            }
        }
        public IEnumerator GetEnumerator()
        {
            return new SmartEnumerator(this);
        }
        private class SmartEnumerator : IEnumerator, IDisposable
        {
            int index = -1;
            Smart_Arrays smarty;

            public SmartEnumerator(Smart_Arrays smarty)
            {
                this.smarty = smarty;
            }
            public bool MoveNext()
            {
                index++;
                return index < smarty.Length;
            }
            public void Reset()
            {
                throw new NotSupportedException();
            }
            public void Dispose() { }
            public object Current
            {
                get { return smarty.array[index]; }
            }


        }


    }
}
```

Programski kod (Main):

```
using System;
```

```
using System.Collections;

namespace Smart_Arrays
{
    class Program
    {
        static void Main(string[] args)
        {
            Smart_Arrays smarty = new Smart_Arrays(4);
            for(int i = 0; i < 8; i++)
            {
                smarty.Add(i);
            }
            for(int i=0;i<smarty.Length;i++)
                Console.Write(smarty[i]+" ");

            Console.WriteLine();

            IEnumerator enumerator = smarty.GetEnumerator();
            while (enumerator.MoveNext())
            {
                int i = (int)enumerator.Current;
                Console.Write(i+" ");
            }
            Console.WriteLine();

            for(int i = 0; i < 8; i++)
            {
                Console.WriteLine("Removing "+i);
                smarty.Remove(i);

                foreach(var s in smarty)
                {
                    Console.Write(s + " ");
                }
                Console.WriteLine();
            }
        }
    }
}
```

Komentar:

Unutar klase imamo nekoliko definiranih varijabli: *Size* – veličina niza, *last* – indeks posljednjeg elementa u nizu, *array*- definirani niz s svim vrijednostima koji također sadržava **Smart Array,** *Length* – duljina niza (u našem slučaju duljina niza je posljednji niz uvećan za jedan)**.** Funkcija **this** ima svoj *get* i *set* pomoću kojeg postavljamo i dohvaćamo vrijednosti vrijednosti niza na određenom indeksu.

Metode **Add** i **Remove** nam omogućavaju da kao što i ime kaže dodavanje elementa u niz ili uklanjanje elementa iz niza za određeni indeks. Ukoliko je posljednji element na indeksu koji odgovara veličini niza umanjen za jedan (size - 1), potrebno je promijeniti veličinu niza (resize) budući da nemamo mjesta za umetanje novog elementa u niz. Kreiramo niz koji je duplo veći od prethodnoga kako bi mogli ubaciti novi element. Zatim je potrebno taj novi niz upisati u našu varijablu za pohranu niza (array). Također, potrebno je promijeniti veličinu niza tj. postaviti novi *size*. Nakon što povećamo niz (ili ukoliko niz prethodno već ima dovoljno praznih mjesta za pohranu novog elementa) potrebno je ubaciti element na idući prazan indeks (indeks ++last)

SmartEnumerator interface uključujemo u ovu klasu i koristimo **IEnumerato**r kao iterator za „šetanje“ kroz zadani niz. **IDisposable** nam služi za oslobađanje memorije koje više ne koristimo ili nam ne treba.
Metoda **MoveNext** nam omogućava pomicanje po nizu pomoću iteratora uvećavanjem za *jedan* i vraća nam **1** (true) ukoliko je indeks elementa manji od duljine samog niza što nas osigurava da ne bismo premašili veličinu niza.
Metoda **Current** vraća vrijednost elementa u nizu na indeksu na koji trenutno pokazuje iterator.

Unutar Main programa kreiran je objekt smarty unutar kojega dodajemo 8 elemenata. Nakon toga kreiramo iterator enumerator tome objektu kako bi pomoću njega šetali po nizu. Ukoliko nam MoveNext metoda enumeratora vraća true (tj. da nismo premašili veličinu niza prilikom šetnje) ispisujemo elemente pomoću prethodno implementirane metode Current. Nakon toga je isprobana funkcionalnost implementirane metode Remove za uklanjanje elemenata u nizu.!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Vježba 3. Sortiranje

1. Zadatak:

U ovom zadatku ćemo implementirati rekurzivni **SelectionSearch** algoritam za sortiranje niza cijelih brojeva.

Programski kod:

```
using System;

namespace Selection
{
    class Program
    {
        static void Sort(int[] arr, int first)
        {
            int n = arr.Length;

            if (first== n - 1)
                return;
            int min =first;
            for (int i = min + 1; i < arr.Length; i++)
            {
                if (arr[i] < arr[min])
                {
                    // swap
                    int temp = arr[min];
                    arr[min] = arr[i];
                    arr[i] = temp;
                }
            }
            // recursive call
            Sort(arr,first + 1);
        }


        static void Main(string[] args)
        {

            int[] array = { 9, 8, 4, 11, 5, 7, 3, 1 };

            foreach (int i in array)
            {
                Console.WriteLine(i);

            }

            Sort(array, 0);

            Console.WriteLine("---------");


            foreach (int i in array)
            {
                Console.WriteLine(i);

            }

        }
    }
}
```

Komentar:

Cilj nam je implementirati rekurzivni algoritam za sortiranje.Metoda **Sort** prima 2 argumenta, a to su zadani niz koji treba sortirati **arr** i **first** indeks niza od kojeg je potrebno sortirati zadani niz.

Prvo kontroliramo je li je indeks **first** veći od duljine zadanog niza **arr.Length.** Ukoliko ne premašuje duljina niza nastavljamo dalje. Varijabla **min** predstavlja indeks minimalnog elementa u nizu.

Sada implementiramo for petlju u kojoj prolazimo kroz svaki element i provjeravamo je li postoji manji element od elementa zapisanog u **min** varijabli. Ukoliko ne postoji niti jedan el. koji je manji od **min** petlja završava i trenutna varijabla se postavlja kao najmanja i prva u nizu. Ukoliko postoji broj koji je manji el. od **min,** zamjenjujemo mjesta trenutnog elementa i varijable **min**.

Funkcija se rekurzivno poziva tako da se inkrementira argument *(first+1)* kako bi provjerili sljedeći broj u nizu i sortirali ga u odnosu je li veći ili manji od već sortiranog prijašnjeg broja u nizu.

2. Zadatak:

U ovom zadatku ćemo implementirati rekurzivni **SelectionSearch** algoritam za sortiranje niza cijelih brojeva.

Programski kod (class Student):

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

namespace Quick_Sort
{
    class Student
    {
        string name;
        double grade;

        public Student(string name, double grade)
        {
            this.name = name;
            this.grade = grade;
        }
        public override string ToString()
        {
            return "Student: " + name + " ,grade: " + grade;
        }
        public static bool CompareName(object a, object b)
        {
            Student s1 = (Student)a;
            Student s2 = (Student)b;
            if (s1.name.CompareTo(s2.name) < 0) return true;
            else return false;
```

```
        }
        public static bool CompareGrade(object a,object b)
        {
            Student s1 = (Student)a;
            Student s2 = (Student)b;
            if (s1.grade.CompareTo(s2.grade) <= 0) return true;
            else return false;
        }

    }
}
```

Komentar:
Kao što smo imali i u prethodnim zadacima u klasi Student imamo 2 varijable **name** i **grade** kojima vrijednost pridjelimo pomoću konstruktora. Metode **CompareName** i **CompareGrade** koje koriste CompareTo metodu pomoću kojih uspoređujemo *studente* po imenu i ocjenama.Ukoliko je ime/ocjena prvog *studenta* (objekt a) veća od drugog *studenta* (objekt b) vraća FALSE, u suprotnom TRUE.

Programski kod (class Quick):

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

namespace Quick_Sort
{
    delegate bool Comparison(object a, object b);

    class Quick
    {
        public static int Partition(object[] array, int left, int right,
Comparison cmp)
        {
            object pivot = array[left];
            int last = left;
            for (int i = last + 1; i <= right; i++)
                if (cmp(array[i], pivot))
                    Swap(array, ++last, i);
            Swap(array, left, last);
            return last;
        }
        static void Swap(object[] array, int first, int second)
        {
            object temp = array[second];
            array[second] = array[first];
            array[first] = temp;
        }
        public static void Sort(object[] array,int left,int right,Comparison
cmp)
        {
            if (left >= right) return;
            int last = Partition(array, left, right, cmp);
            Sort(array, left, last - 1, cmp);
            Sort(array, last + 1, right, cmp);
        }
    }
}
```

Komentar:

Delegat **Comparison** ("pokazivač" na funkciju koja vraća vrijednost *true* ili *false* i prima dva objekta kao argument je deklariran izvan klase. Preko njega ćemo koristiti metode **Sort** i **Partition** za slanje kriterija prema kojima sortiramo (bilo po imenu ili ocjenama učenika putem metoda *CompareName* ili *CompareGrade*) .
Funkcija **Partition** ima element **pivot** pomoću kojeg niz dijelimo na dva podniza. Dijelimo na elemente koji su manji od **pivota** (*s lijeve strane pivota*) i elementi veći od **pivota** (*s desne strane pivota*). **Partition** vraća kao konačnu vrijednost indeks *pivota.*
Uspoređujemo **pivot** sa sljedećim elementima niza. Budući da uspoređujemo objekte tipa *student*, potrebno je proći kriterij po kojem sortiramo kroz argument **cmp** (*Comparison delegate*). Ako je rezultat usporedbe točan (odnosno ime/razred prvog učenika manji je od imena/razreda drugog učenika), vršimo razmjenu metodom **Swap**.
Funkcija **Swap** mijenja elemente niza na određenim indeksima niza (navedenim s *first* i *second* argumentom).
Algoritam za *brzo sortiranje* koristi *Partition* rekurzivno, tj. poziva *Partition* za desni i lijevi podniz oko *pivota*. **Pivot** se odabire preko posljednje varijable pozivom **Partition-a**. Nadalje, **Sort** se poziva rekurzivno za desnu (od elementa nakon pivota: *last+1* do kraja niza) i lijevu (od početka niza do elementa prije stožera: *last-1*) podniz *pivot* posljednji.

Programski kod (Main):

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

namespace Quick_Sort
{
    class Program
    {
        static void Main(string[] args)
        {
            Student[] students = {
             new Student ("Ivo", 4.1),
             new Student ("Ana", 4.9),
             new Student ("Iva", 4.3),
             new Student ("Bob", 4.5),
             new Student ("Joe", 4.7),
             new Student ("Tom", 4.4),
             new Student ("Iko", 4.6)
             };

            for (int i = 0; i < students.Length; i++)
            {
                Console.WriteLine(students[i]);
            }

            Quick.Partition(students, 0, students.Length - 1,
Student.CompareName);
            Console.WriteLine("------------");

            for (int i = 0; i < students.Length; i++)
            {
                Console.WriteLine(students[i]);
            }

            Quick.Sort(students, 0, students.Length - 1, Student.CompareName);
            Console.WriteLine("------------");
```

```
                for (int i = 0; i < students.Length; i++)
                {
                    Console.WriteLine(students[i]);
                }

                Quick.Sort(students, 0, students.Length - 1, Student.CompareGrade);
                Console.WriteLine("------------");

                for (int i = 0; i < students.Length; i++)
                {
                    Console.WriteLine(students[i]);
                }

            }
        }
    }
```

Komentar:

Unutar **Main** programa isprobana je funkcionalnost implementiranih metoda **Quick Sort** (*algoritam za brzo sortiranje*). Kao argumenti funkcije šalju se metode *CompareName* ili *CompareGrade*, te **Sort** metoda ih prima preko *delegata*.

Vježba 5. Stack i Queue

1. Zadatak

U ovoj vježbi implementirat ćemo **stog** delegiranjem poziva klasi **List**. **Stack** (stog) ćemo implementirati pomoću jednostruke vezane liste iz prethodne vježbe. **Stack** je organiziran **FILO** (*First-In-Last-Out*) metodom.

Programski kod (class Stack):

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

namespace Stack
{
    class Stack
    {

        private List list;
        public bool IsEmpty()
        {
            return list.IsEmpty();
        }
        public void Display()
        {
            list.Display();
        }
        public Stack()
        {
            list = new List();
        }
        public void Push(object element)
        {
            list.InsertFront(element);
        }
        public object Pop()
        {
            if (IsEmpty())
                throw new Exception("Prazan stack");
            return list.RemoveFront();
        }

    }
}
```

Komentar:

Kako bi koristili prethodno implementiranu jednostruku vezanu listu i njene metode, moramo ju dodati kao referenca na ovaj projekt (`using Single_List`).

Kako bi ostvarili **FILO** strukturu dovoljne su nam dvije metode, **Push** (*dodavanje elementa*) i **Pop** (*uklanjanje elementa*). **Push** metoda je bazirana na *InsertFront* kod vezanih lista, znači da kod ove metode dodajemo element na vrh *Stack-a*. **Pop** metoda je bazirana na *RemoveFront* kod vezanih lista, znači da kod ove metode uklanjamo element sa vrha *Stack-a.* Osim složenosti metoda koje možemo koristiti, moramo voditi računa i o tome je li je **Stack** s kojim radimo *prazan.*

Programski kod (Main):

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

namespace Stack
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack stack = new Stack();
            stack.Push("Ivo");
            stack.Push("Ana");
            stack.Push("Bob");
            stack.Push("Iva");
            stack.Push("Kim");
            stack.Display();
            try
            {
                for (int i = 0; i < 6; i++)
                {
                    stack.Pop();
                    stack.Display();
                }
            }
            catch (Exception x)
            {
                Console.WriteLine(x.Message);
            }
            stack.Display();
        }

    }
}
```

Komentar:

Unutar **Main** programa isprobana je funkcionalnost implementiranog **Stacka**.

2. Zadatak

U ovom zadatku ćemo implementirati **Queue** (*red*) korištenjem prethodne implementacije jednostruke vezane liste i to tako da poštujemo **FIFO** (*First-In-First-Out*) princip.

Programski kod (class Queue):

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

namespace Queue
{
    class Queue
    {
        private List list;
        public bool IsEmpty() { return list.IsEmpty(); }
        public void Display() { list.Display(); }

        public Queue()
        {
            list = new List();
        }
        public void Enqueue(object element)
        {
            list.InsertEnd(element);
        }
        public void Dequeue()
        {

            if (IsEmpty())
                throw new Exception("Queue is empty!");
            else
                list.RemoveFront();
        }

    }
}
```

Komentar:

Ponovno koristimo jednostruko vezane liste pa moramo dodati referencu na ovaj projekt (`using Single_List`). Moramo implementirati dvije funkcije **Enqueue** (dodavanje elementa reda) i **Dequeue** (uklanjanje elementa reda). Moramo voditi računa o složenosti algoritama pa zato koristimo iz prethodne vježbe *InsertEnd* i *RemoveFront*.

Kod metode **Enqueue** ćemo iskoristiti već implementiranu i spomenutu *InsertEnd* metodu kako bi dodali željeni element u **Queuea** (*na kraj liste*). Kod metode **Dequeue** ćemo iskoristiti već implementiranu i spomenutu *RemoveFront* metodu kako bi uklonili željeni element iz **Queuea** (*s vrha liste*). Bitno da pripazimo da **Queue** nije prazan kada koristimo **Dequeue.**

Programski kod (Main):

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

```
namespace Queue
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue queue = new Queue();
            queue.Enqueue("Ivo");
            queue.Enqueue("Ana");
            queue.Enqueue("Bob");
            queue.Enqueue("Iva");
            queue.Enqueue("Leo");
            queue.Display();
            try
            {
                for (int i = 0; i < 6; i++)
                {
                    queue.Dequeue();
                    queue.Display();
                }
            }
            catch (Exception x)
            {
                Console.WriteLine(x.Message);
            }
            queue.Display();
        }
    }

}
```

Komentar:

Unutar **Main** programa isprobana je funkcionalnost implementiranog **Queue**.

Vježba 7. Hash Tablice

Programski kod (class Node):

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

namespace Hash_Table
{
    class Node
    {
        public string Name { get; set; }
        public int Value { get; set; }
        public Node Next { get; set; }
        public Node(string name, int value, Node next)
        {
            Name = name;
            Value = value;
            Next = next;
        }
    }
}
```

Programski kod (class HashTable):

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

namespace Hash_Table
{
    class HashTable
    {
        Node[] buckets;
        int length;
        public HashTable(int length)
        {
            this.length = length;
            buckets = new Node[length];
        }
        public void Display()
        {
            for (int bucket = 0; bucket < buckets.Length; bucket++)
            {
                Node current = buckets[bucket];
                Console.Write(bucket + ": ");
                while (current != null)
                {
                    Console.Write("[" + current.Name + "," + current.Value + "]
");
                    current = current.Next;
                }
                Console.WriteLine();
            }
        }
        private int Hash(string str)
        {
```

```
            int total = 0;
            char[] c;
            c = str.ToCharArray();
            for (int k = 0; k <= c.GetUpperBound(0); k++)
                total += (int)c[k];

            return total % this.length;
        }

        public void Insert(string name, int value)
        {
            int bucket_index = Hash(name);
            buckets[bucket_index] = new Node(name, value,
buckets[bucket_index]);
        }
public int Search(string name)
        {
            Node current = buckets[Hash(name)];
            while (current != null)
            {
                if (current.Name.Equals(name))
                {
                    return current.Value;
                }
                current = current.Next;
            }
            throw new Exception(name + " Not Found.");
        }
        public void Delete(string name)
        {
            int bucket = Hash(name);
            Node current = buckets[bucket];
            Node previous = null;
            while (current != null)
            {
                if (current.Name.Equals(name))
                {
                    if (previous == null)
                        buckets[bucket] = current.Next;
                    else
                        previous.Next = current.Next;
                    return;
                }
                previous = current;
                current = current.Next;
            }
            throw new Exception(name + " Not found.");
        }


    }

}
```

Programski kod (Main):

```
using System;

namespace Hash_Table
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            HashTable table = new HashTable(9);
            table.Insert("Ivo", 5);
            table.Insert("Ana", 1);
            table.Insert("Joe", 7);
            table.Insert("Bob", 4);
            table.Insert("Kim", 3);
            table.Insert("Ena", 8);
            table.Insert("Iva", 2);
            table.Insert("Tea", 9);
            table.Insert("Lea", 6);
            table.Display();
            string name = "Bob";
            try
            {
                int value = table.Search(name);
                Console.WriteLine(name + ", " + value);
            }
            catch (Exception x)
            {
                Console.WriteLine(x.Message);
            }
            try
            {
                table.Delete(name);
                table.Delete("Joe");
            }
            catch (Exception x)
            {
                Console.WriteLine(x.Message);
            }
            table.Display();
        }
    }
}
```

Komentar:

3. Zadatak

Prvo smo definirali klasu Node koja nam služi za definiranje čvorova koje ćemo koristiti kod Hash tablice. U klasi **HashTable** smo implementirali svoju hash tablicu koja će primat ključ (*key*) i vrijednost (*value*). **Hash tablica** (eng. *hash table*) ili **hash** mapa (eng. *Hash map*) je podatkovna struktura koja rabi ***hash*** funkciju za učinkovito preslikavanje određenih ključeva (na primjer imena ljudi) u njima pridružene vrijednosti (na primjer telefonske brojeve). **Hash** funkcija se koristi za transformiranje ključa u indeks (*hash*) to jest mjesto u nizu elemenata gde treba tražiti odgovarajuću vrijednost.

U klasi **Hash** imamo niz čvorova iz klase **Node** definirane imenom *buckets* unutar koje pohranjujemo ime i vrijednost.

Definirali smo dvije funkcije **Display** i **Hash**. **Display** f. nam služi kako bi ispisali zapisano ime i ocjenu studenta preko ključa i vrijednosti koje su im dodijeljene za upis u *Hash Tablicu* kako bi ih lako dohvatili. **Hash** funkcija je implementirana tako da generira *hash* vrijednost za primljeni string. Ukoliko je *hash tablica* prazna pridjeljuje najmanji iznos u tablici koji predstavlja prvu vrijednost u tablici te nakon nje generira sljedeću po redu generirajuci niz

kljuceva koji tvore redosljedno tablicu. Vrijednost računamo tako da svako slovo stringa pretvorimo u integer tj. zbrajamo ASCII vrijednosti svakog znaka od kojega je sačinjen string te vraća njen indeks kao ostatak cjelobrojnog dijeljenja kako ne bismo premašili veličinu niza unutar koji spremamo te podatke.

4. **Zadatak**

Implementiramo metodu **Insert** koja ubacuje element u Hash tablicu. Insert funkcija je implementirana tako da prima 2 argumenta, to je **name**(*ime* tipa string) i **value** (*vrijednost* tipa integer). Prvo inicijaliziramo varijablu **bucket_index** u kojoj ćemo pohraniti varijablu **name** preko hash metode *Hash(name).* Imamo varijablu buckets[bucket_index] te unutar nje postavljamo novi Node (čvor) sa svojim vrijednostima (name,value i pokazivač koji pokazuje na samog sebe tako da njegov prethodnik može imati pokazivač na trenutni čvor).

5. **Zadatak**

Implementiramo metodu **Search** koja traži odgovarajući element u nizu buckets te vraća njegovu vrijednost (*Value*). Kao argument funkcija prima ime elementa (*name*) kojeg tražimo. Taj čvor se pronalazi tako da se izvrši *hash* funkcija koja daje indeks na kojem se nalazi traženi podatak. Kako bi spremili „trenutnu“ vrijednost nekog čvora, inicijalizirat ćemo privremeni čvor **current** kojeg postavimo na element čiji se indeks dobije kao *hash* vrijednost ulaznog parametra **name**

Pomoćnim čvorom (*current*) se „šeta“ kroz cijeli niz te ukoliko naiđe na element koji odgovara traženome vrati njegovu vrijednost.

6. **Zadatak**

Implementiramo metodu **Delete** koja briše par **Name-Value** iz *hash* tablice. Funkcija kao parametar prima name, odnosno ime elementa kojeg želimo izbrisati iz hash tablice. Opet ćemo iskoristiti current element koji ćemo postaviti na mjesto gdje otprilike mislimo da će se nalaziti element kojeg želimo izbrisati. Prvo, prethodni pokazivač je postavljen da pokazuje na **null**. **Previous** služi da se zna koji čvor prethodi trenutnom, tako da se može lako zaobići.

Ako pronađemo element i ako je pokazivač na prethodni jednak **null**, prvi element je pronađen i početna *hash* tablica bit će sljedeći element. Prvi element je izbrisan.

Ako traženi element nije prvi u nizu, tada **Next** od prethodnog elementa pokazuje na mjesto gdje pokazuje **Next** od trenutnog i tako se zaobilazi. Ako se ne pronađe, prethodni pokazivač postaje stari trenutni, a trenutni se pomiče za jedno mjesto naprijed.

7. **Zadatak**

U **main** metodi stvara se instanca HashTable s određenom dužinom (*Length*). Koristimo sve funkcije koje smo definirali u prijašnjim zadacima, odnosno ispitujemo funkcionalnosti implementiranih metoda. Zatim se poziva metoda **Display** da bi se prikazala *hash* tablica.