Sveučilište u Splitu

Fakultet elektrotehnike, strojarstva i brodogradnje

Algoritmi i strukture podataka (220)

Izvještaj iz laboratorijskih vježbi

Boris Puljak (62-2020)

Split, prosinac 2023.

**Vježba 1: Methods and Criteria**

1. **Zadatak: Passing by Reference**

Programski kod:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

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, b = 2;

Console.WriteLine($"Prije zamjene: a = {a} b = {b}");

Swap(ref a, ref b);

Console.WriteLine($"Poslije zamjene: a = {a} b = {b}");

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

Console.WriteLine("\nNiz prije zamjene:");

foreach (int num in array)

{

Console.Write(num+"\n");

}

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

Console.WriteLine("\nNiz nakon zamjene:");

foreach (int num in array)

{

Console.Write(num+"\n");

}

}

}

}

Na početku samog zadatka je potrebno kreirati novi C# console projekt pod nazivom Parameters.

Nakon toga potrebno je implementirati javnu statičku metodu Swap koja preko reference prima dva argumenta a i b. Kada su argumenti primljeni putem reference, to znači da metoda Swap prima njihove adrese, a ne vrijednost argumenata. Tako da sve promjene nad argumentima a i b u metodi Swap će se odraziti na vrijednosti argumenata a i b zapisani u memoriji.

Metoda Swap će zamijeniti vrijednosti primljenih argumenata, tako što ćemo dodati novi argument temp koji će nam služiti kao privremeni spremnik za vrijednost argumenta a jer ćemo ga izjednačiti s argumentom a. Zatim ćemo vrijednost argumenta b upisati u argument a, a u argument b upisati vrijednost argumenta temp (u njemu je zapisana početna vrijednost argumenta a, a u argumentu a je trenutno početna vrijednost argumenta b).

Varijable a i b su lokalne varijable funkcije i nemaju nikakve veze sa onima unutar Main funkcije, osim što imaju isti naziv. Da nema ključne riječi ref ispred argumenata funkcije, varijable a i b bi zamijenile vrijednosti i po završetku funkcije bi se izbrisale (te dvije varijable u funkciji Swap), vrijednosti istoimenih varijabli a i b u mainu koje su proslijeđene kao argument Swap funkciji se ne bi promijenile i onda zadatak ne bi bio izvršen kako treba.

U main funkciji se deklariraju dvije varijable tipa integer, a i b, početnih vrijednosti 1 i 2. Potrebno je ispisati varijable u konzolnom prozoru odmah pri pokretanju programa. Ispis se vrši metodom Console.WriteLine (u kodu su vidljiva dva načina vršenja ispisa i oba su točna).Zatim se poziva funkcija Swap te joj se prosljeđuju varijable a i b. A zatim se ispisuju nove vrijednosti varijabli a i b.

Zatim deklariramo i inicijaliziramo niz Array. Ispisujemo sve vrijednosti niza funkcijom foreach. Pozivamo funkciju Swap kako bi zamijenili 3. i 4. element niza, te joj te iste elemente proslijedimo. Funkcija Swap zamijeni vrijednosti ta dva elementa. Zatim ponovno ispisujemo niz kojem su sada 3. i 4. element zamijenjenih vrijednosti.

Slika na kojoj se prikazuje tekst, snimka zaslona

Opis je automatski generiran

1. **Zadatak: Sorting via IComparable**

Programski kod (klasa Student):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Comparables

{

public 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;

}

}

}

Prvo kreiramo klasu student koja koristi interface IComparable. Ta klasa mora sadržavati javne varijable za pohranu imena i ocjene studenta. Definiran je konstruktor za postavljenje varijabli koje sadrži klasa Student, na način da primljene vrijednosti se upisuju na mjesto tih varijabli. Override metoda služi nam za ispis studenta i njegove ocjene. Interface IComparable definira metodu CompareTo koja prima objekt i vraća vrijednost -1 u slučaju da je vrijednost sadržana u this objektu manja od vrijednosti u drugom objektu. U slučaju da je obrnuto, metoda vraća vrijednost 1. U slučaju da su obe vrijednosti jednake, metoda vraća vrijednost 0. U ovom zadatku uspoređivani su objekti klase student po njihovim ocjenama. Metoda uspoređuje ocjenu studenta za koji je pozvana metoda i drugog studenta te prema gore navedenim kriterijima vrača određenu vrijednost.

Programski kod (klasa Bubble):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Comparables

{

public class Bubble

{

public static void Sort(IComparable[] array)

{

for (int i = 0; i < array.Length; 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;

}

}

}

}

}

}

U klasi Bubble implementirana je funkcija Sort koja nam služi za sortiranje, ona koristi Bubble Sort algoritam sortiranja koji sortira niz tako da uspoređuje svaki element sa svakim elementom. Funkcija prvo uzme jedan element niza (preko vanjske for petlje) i onda ga uspoređuje sa svim ostalim elementima (preko unutarnje for petlje). A za samu usporedbu dva elementa koristi se CompareTo metoda koju smo objasnili u klasi Student. Kada metoda pronađe element koji je veći od onog s kojim se uspoređuje, a koji se nalazi na nižoj poziciji u nizu, tada se zamijene vrijednosti tih elemenata. Nakon te operacije veći element bude na nižoj poziciji u nizu, pa tako dobijemo sortiran niz u kojem su poredani objekti klase student prema ocjeni od veće prema manjoj.

Programski kod (Main program):

using Parameters;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Comparables

{

class Program

{

static void Main()

{

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)

};

Console.WriteLine("Studenti prije sortiranja:");

DisplayStudents(students);

Bubble.Sort(students);

Console.WriteLine("\nStudenti nakon sortiranja:");

DisplayStudents(students);

}

static void DisplayStudents(Student[] students)

{

foreach (var student in students)

{

Console.WriteLine(student);

}

}

}

}

U main funkciji se deklarira niz objekata klase student. Prilikom stvaranja svakog novog objekta Student šalje se ime i ocjena tog studenta tako da objekt može biti u postupnosti definiran. U konzolni prozor ispišemo niz studenata, zatim sortiramo niz i ispišemo sortirani niz studenata. Dodali smo funkciju DisplayStudents koja prima niz objekata i ispisuje ih u konzolni prozor prolaskom kroz niz pomoću petlje foreach koja za svaki objekt student u nizu objekata ispiše objekt student, odnosno njegovo ime i ocjenu.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font, dizajn

Opis je automatski generiran

1. **Zadatak: Sorting via IComparer**

Programski kod (klasa Student):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Comparers

{

public 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}";

}

}

}

Klasa student sadrži dvije varijable, name i grade, koje se pomoću konstruktora postavljaju na primljene vrijednosti. Također kao u prethodnom zadatku override metoda nam služi za ispis studenta i njegove ocjene.

Programski kod (klasa StudentComparer):

using System;

using System.Collections.Generic;

using System.Text;

using System.Collections;

namespace Comparers

{

public class StudentComparer : IComparer

{

private StudentComparerType criterion;

public StudentComparer(StudentComparerType criterion)

{

this.criterion = criterion;

}

public int Compare(object x, object y)

{

Student studentX = (Student)x;

Student studentY = (Student)y;

switch (criterion)

{

case StudentComparerType.Name:

return studentX.name.CompareTo(studentY.name);

case StudentComparerType.Grade:

return studentY.grade.CompareTo(studentX.grade);

default:

throw new Exception("Kriterij ne postoji");

}

}

}

public enum StudentComparerType

{

Name, Grade

}

}

Klasa StudentComparer koristi interface IComparer i koristi njegovu metodu Compare. Kriterij po kojem vršimo usporedbu se pohranjuje u privatnu varijablu criterion koja se pomoću konstruktora postavlja na određenu vrijenost, a može biti Name ili Grade jer su te dvije opcije pohranjene u enumeraciji StudentComparerType. Compare metoda prima dva objekta koja uspoređuje, objekt x i objekt y, koja uspoređuje prema varijabli criterion, koristi CompareTo metodu i vraća vrijednost CompareTo. U slučaju da se varijabla criterion ne podudara s dva moguća kriterija za rješavanje zadatka javlja se greška da kriterija za usporedbu studenata ne postoji.

Programski kod (klasa Bubble):

using System;

using System.Collections.Generic;

using System.Text;

using System.Collections;

namespace Comparers

{

public class Bubble

{

public static void Sort(object[] array, IComparer comparer)

{

for (int i = 0; i < array.Length; 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;

}

}

}

}

}

}

Klasa Bubble je napravljena na isti princip kao i ona u prethodnom zadatku, samo što se sada koristi metoda Compare iz interfacea IComparer koja kao argument prima dva objekta koja uspoređuje (u prošlom zadatku koristio se interface IComparable i njegova metoda CompareTo). Implementacija metode Compare se nalazi u klasi StudentComparer.

Programski kod (Main program):

using Comparers;

using System;

namespace Comparers

{

class Program

{

static void Main()

{

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)

};

Console.WriteLine("Studenti:");

DisplayStudents(students);

StudentComparer comparerByName = new StudentComparer(StudentComparerType.Name);

Bubble.Sort(students, comparerByName);

Console.WriteLine("\nStudenti nakon sortiranja prema imenu:");

DisplayStudents(students);

StudentComparer comparerByGrade = new StudentComparer(StudentComparerType.Grade);

Bubble.Sort(students, comparerByGrade);

Console.WriteLine("\nStudenti nakon sortiranja prema ocjeni:");

DisplayStudents(students);

}

static void DisplayStudents(Student[] students)

{

foreach (var student in students)

{

Console.WriteLine(student);

}

}

}

}

U main funkciji se deklarira niz objekata klase student. Prilikom stvaranja svakog novog objekta Student šalje se ime i ocjena tog student. U konzolni prozor ispišemo niz studenata, zatim sortiramo niz prema imenima studenata i ispišemo sortirani niz studenata prema imenu, zatim sortiramo niz prema ocjenama studenata i ispišemo sortirani niz studenata prema ocjeni. Ispisivanje se vrši funkcijom DisplayStudents na isti način kao u prethodnom zadatku.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font

Opis je automatski generiran

**Vježba 2: Searching**

1. **Zadatak: Recursive Methods**

Programski kod:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Diagnostics;

namespace Recursions

{

class Program

{

static void Main()

{

int result = Factorial(6);

Console.WriteLine("6!="+result);

result = Fibonacci(6);

Console.WriteLine("Fibonacci of 6 (Recursive)="+result);

result = FibonacciIter(6);

Console.WriteLine("FibonacciIter of 6 (Iterative):"+result);

}

static int Factorial(int n)

{

if (n <= 1) return 1;

return n \* Factorial(n - 1);

}

static int Fibonacci(int n)

{

if (n <= 1) return n;

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

}

static int FibonacciIter(int n)

{

if (n <= 1) return n;

int a = 0, b = 1, result = 0;

for (int i = 2; i <= n; i++)

{

result = a + b;

a = b;

b = result;

}

return result;

}

}

}

Prvo je implementirana metoda Factorial koja računa faktorijel nekog broja primljenog preko argumenta n. U slučaju da je primljeni broj manji ili jednak 1, njegov faktorijel iznosi 1. Za sve ostale (veće od 1) faktorijel je umnožak svih brojeva do tog broja, odnosno faktorijel prethodnog broja pomnožen s tim brojem te baš zbog toga se u metodi Factorial rekurzivno poziva ta ista metoda Factorial za taj broj umanjen za 1. Tako ostvarujemo da se izračuna umnožak svih brojeva od 1 do tog u početku primljenog broja n.

Zatim je implementirana metoda Fibonacci za izračun vrijednosti određenog elementa Fibonaccijevog niza. To je niz u kojem se sljedeći broj dobije zbrajanjem prethodna dva broja (Fibonaccijev niz: 1 1 2 3 5 8 13 …). Vrijednost Fibonaccijevog niza za prvi i drugi element je 1, zato funkcija vraća 1. U svakom drugom slučaju kada se traži element na poziciji 3 ili višoj, tada se ta vrijednost dobije kao zbroj prethodna dva broja u nizu pozivajući tu istu metodu Fibonacci rekurzivno i tako sve dok ne dođemo do prva 2 elementa koji su nam poznati.

Metoda Fibonacciter je implementirana tako da računa vrijednost elementa u Fibonaccijevom nizu iterativnim prolaženjem kroz niz od prve do tražene vrijednosti. Opet vrijedi da je vrijednost niza za prvi i drugi element 1. Za računanje vrijednosti na svim sljedećim elementima, sve do onog kojeg tražimo, koristimo petlju koja računa vrijednosti elemenata niza.

Pozvana je svaka funkcija te zatim ispisano ono što ona vraća.

Slika na kojoj se prikazuje tekst, Font, snimka zaslona, broj

Opis je automatski generiran

1. **Zadatak: Sequential and Binary Search**

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 int SequentialSearch(int[] array, int value)

{

for (int i = 0; i < array.Length; i++)

{

if (array[i] == value)

return i;

}

return -1;

}

static int BinarySearch(int[] array, int value)

{

int low = 0;

int high = array.Length - 1;

while (low <= high)

{

int mid = (low + high) / 2;

if (array[mid] == value)

return mid;

else if (array[mid] < value)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

static int BinarySearchRecursive(int[] array, int value, int low, int high)

{

if (low <= high)

{

int mid = (low + high) / 2;

if (array[mid] == value)

return mid;

else if (array[mid] < value)

return BinarySearchRecursive(array, value, mid + 1, high);

else

return BinarySearchRecursive(array, value, low, mid - 1);

}

return -1;

}

static void Main()

{

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

int index = SequentialSearch(array, 6);

Console.WriteLine("Sequential Search: "+index);

index = BinarySearch(array, 6);

Console.WriteLine("Binary Search: "+index);

index = BinarySearchRecursive(array, 6, 0, 6);

Console.WriteLine("Binary Search (Recursive): "+index);

}

}

}

U ovom zadatku prvo smo implementirali algoritam za sekvencijalno pronalaženje određenog elementa u nekom nizu. Metoda zvana SequentialSearch prima niz iz kojeg traži element i vrijednost koja se traži. Pronalazak smo ostvarili na način da smo for petljom prošli kroz sve elemente niza te svaki od njih zasebno i po redu usporedili s vrijednošću koju tražimo. Kada smo pronašli element koji sadrži vrijednost koja se poklapa s onom koju tražimo tada vračamo indeks tog elementa, koji poslije u main dijelu programa ispisujemo u konzoli, naravno nakon što smo pozvali tu našu funkciju SequentialSearch i proslijedili joj niz te vrijednost koju tražimo u tom nizu. U slučaju da se vrijednost koja tražimo ne poklapa s ni jednom vrijednošću elemenata niza, tada vračamo vrijednost -1 (zbog toga što su indeksi elemenata niza od 0 pa do n-1 pa zato kada u konzoli vidimo da se ispisalo -1 znamo da te vrijednosti nema u nizu).

Iduća metoda koju smo implementirali je BinarySearch koja također prima niz i vrijednost koju tražimo u tom nizu. Da bi ovaj algoritam uopće mogli implementirati potrebno je prethodno sortirati niz jer jedino tada nam vrijednosti elemenata u nizu daju informaciju o pozicijama ostalih elemenata jer ćemo uspoređivat vrijednosti te prema toj usporedbi se odlučuje slijedeći korak. Mi smo odmah niz postavili kao sortiran s vrijednostima od 1 do 7 kao što je zadano u zadatku. Algoritam funkcionira tako da se niz podijeli na pola, odredi se sredina niza pomoću poznatih indexa početka i kraja niza te se traženi element uspoređuje s elementom koji ima indeks sredine niza. Na temelju te usporedbe odlučujemo o slijedećem koraku, u slučaju da vrijednost tog elementa odgovara traženoj vrijednosti vračamo indeks tog elementa, u slučaju da je srednji element manji ili veći od traženoga nastavljamo tražiti u lijevom ili desnom podnizu. Ako algoritam nastavlja gledati neki od podnizova bitno je voditi računa o početnom i krajnjem indexu tog podniza, odnosno tada više oni nisu jednaki ovisno o tome koji pozniz je izabran za daljnje promatranje, ako je izabran desni podniz onda početni indeks postaje jednak indeksu sredine prethodnog niza uvećanog za 1 dok krajnji indeks ostaje isti, a u slučaju da je izabran lijevi pozniz tada krajnji indeks postaje jednak srednjem indeksu umanjenom za 1 dok početni indeks ostaje isti. Taj podniz tretiramo identično kao i prvi niz, odnosno radimo istu proceduru. Tako radimo sve dok ne nađemo traženi element u nizu, a uvjet da se traženje izvršava je da je početni indeks manji od krajnjeg indexa niza to jest podniza koji promatramo.

Ovaj algoritam je također moguće implementirati rekurzivno. Sada metodu nazivamo BinarySearchRecursive te ona prima niz, vrijednost koju tražimo u tom nizu, početni indeks i krajnji indeks. Ona vrača indeks elementa ukoliko je tražena vrijednost jednaka elementu na srednjem indexu, ali u slučaju da se ne nalazi tu prelazimo na podniz tako što metoda poziva samu sebe rekurzivno. Ponovnim pozivanjem moramo voditi računa o korekciji indexa početka i kraja podniza koji se šalju pri pozivu funkcije. Tako da ukoliko je vrijednost elementa na srednjem indeksu manja od vrijednosti koju tražimo, uzimamo desni podniz za daljnje razmatranje, s jednom bitnom promjenom, a to je da je sada početni indeks jednak indeksu sredine prethodnog niza uvećan za 1, analogno tome se obavlja traženje u lijevom podnizu gdje se korigira krajnji indeks.

Slika na kojoj se prikazuje tekst, Font, snimka zaslona, broj

Opis je automatski generiran

1. **Zadatak: Smart Arrays**

Programski kod (klasa SmartArray):

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SmartArray

{

class SmartArray

{

public int size;

public int last = -1;

int[] array;

public SmartArray(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;

SmartArray smarty;

public SmartEnumerator(SmartArray 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]; }

}

}

}

}

Klasa SmartArray predstavlja pametan niz koji ima mogučnost dinamičkog dodavanja, uklanjanja i iteraciju kroz elemente. Klasa sadrži konstruktor koji prima veličinu niza i služi za njegovu inicijalizaciju. Implementiran je također i indekser koji sadrži get i set metode koje služe za dohvaćanje vrijednosti niza s određenog indexa i za postavljanje elementa na određenom indeksu. Metoda lenght služi za dohvaćanje duljine niza.

Pomoću metode Add dodajemo element item u niz. U slučaju da je posljednji element na indeksu koji odgovara veličini niza umanjen za jedan zada moramo povećati veličinu niza jer novi element ne stane u postojeći niz. Zato kreiramo niz koji je duplo veći od trenutnoga, postavi se veličina na novu veličinu, kopira trenutni niz u taj novi, zatim postavljamo element na idući prazan indeks.

Metoda Remove prima element koji je potrebno ukloniti iz niza. For petljom prolazi kroz niz i pronalazi se element jednak onome kojeg želimo ukloniti. Zatim metodom Copy kopiramo niz u drugi niz, metoda Copy prima originalni niz, početni indeks od kojeg se vrši kopiranje (indeks nakon indeksa elementa kojeg izbacujemo), odredišni niz to jest niz u koji želimo kopirati dio niza nakon elementa kojeg izbacujemo (na to mjesto upisujemo početni niz jer želimo zadržati dio početnog niza do elementa koji izbacujemo), indeks u odredišnom nizu od kojeg započinjemo kopiranje (to je element s indeksom za jedan manjim od onog elementa koji izbacujemo, jer sada se kopirani niz dodaje to jest kopira kao nastavak na dio originalnog niza koji počinje od početka i ide do elementa koji je izbačen). Zatim se last smanjuje za jedan, jer se duljina niza umanjila za jedan element.

Unutar klase je definirana nova klasa SmartEnumerator koja služi kao iterator za šetanje kroz niz. Klasu realiziramo pomoću interfacea IEnumerator (šetanje) i IDisposable (oslobađanje memorije kod objekata koji se više ne koriste). Implementirane je metoda MoveNext koja služi za pomicanje iteratora po nizu na način da mu uvećava indeks za jedan i vrati true ako je indeks elementa manji od duljine niza. Metoda Current vraća vrijednost elementa u nizu na koji iterator pokazuje.

Programski kod (Main program):

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SmartArray

{

class Program

{

static void Main(string[] args)

{

SmartArray smarty = new SmartArray(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();

}

}

}

}

U mainu je kreiran objekt smarty u kojeg dodajemo 8 elemenata. Kreiran je iterator enumterator tom objektu kako bi mogli šetati po nizu. Za slučaj kada metoda enumatatora koju smo nazvali MoveNext vraća true ispisuju se svi elemente pomoću metode enumeratora koju smo nazvali Current. Nakon toga metodom Remove izbacujemo elemente niza jedan po jedan i svaki put nakon izbačenog elementa ispišemo niz koji je preostao.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font, dizajn

Opis je automatski generiran

**Vježba 3: Sorting**

1. **Zadatak: Selection Sort**

Programski kod (klasa Selection):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Selection

{

class Selection

{

public static void Sort(int[] arr, int startIndex)

{

for (int i = startIndex; i < arr.Length - 1; i++)

{

int minIndex = i;

for (int j = i + 1; j < arr.Length; j++)

{

if (arr[j] < arr[minIndex])

{

minIndex = j;

}

}

if (minIndex != i)

{

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

}

}

}

Prvo definiramo metodu za sortiranje niza cijelih brojeva koja prima niz cijelih brojeva koje treba sortirati i indeks od kojeg sortiranje treba vršiti. Zatim petlja ide kroz niz počevši od startIndex indeksa do predzadnjeg elementa, zatim taj indeks postavlja kao minIndex to jest odabire se on kao minimalni element. Petlja opet iterira kroz nesortirani dio niza s početkom na kraju niza i ide do minimalnog elementa u nesortiranom dijelu. Kada petlja naiđe na element koji je manji od elementa koji je trenutno u minIndex varijabli onda se minIndex postavlja na taj element. Nakon pronalaska minimalnog elementa, ako je njegov indeks različit od trenutnog indeksa „i“, radi se zamjena elemenata na indeksu „i“ i pronađenog minimalnog elementa. Tako se najmanji element u nesortiranom dijelu prebacuje na točnu poziciju kojoj pripada u tom nizu.

Programski kod (Main program):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Selection

{

class Program

{

static void Main(string[] args)

{

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

Console.WriteLine("Original array:");

PrintArray(array);

Selection.Sort(array, 0);

Console.WriteLine("\nSorted array:");

PrintArray(array);

}

static void PrintArray(int[] arr)

{

foreach (var item in arr)

{

Console.Write(item + " ");

}

Console.WriteLine();

}

}

}

Definirali smo niz cijelih brojeva kao što je rečeno u zadatku i zatim smo ga ispisali. Pozivamo funkciju Sort koja sortira niz i zatim ga ponovno ispisujemo. Napisana je i metoda za ispis niza u konzolu koju smo objasnili već prije.

Slika na kojoj se prikazuje tekst, Font, snimka zaslona, broj

Opis je automatski generiran

1. **Zadatak: Quick Sort**

Programski kod (klasa Student):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Quick\_Sort

{

class Student

{

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 static bool CompareName(object a, object b)

{

if (a is Student studentA && b is Student studentB)

{

return String.Compare(studentA.name, studentB.name, StringComparison.Ordinal) < 0;

}

return false;

}

public static bool CompareGrade(object a, object b)

{

if (a is Student studentA && b is Student studentB)

{

return studentA.grade > studentB.grade;

}

return false;

}

}

}

U klasi student su definirani parametri name i grade te su u konstruktoru postavljeni na poslane vrijednosti. Klasa također sadrži CompareName i CompareGrade metode koje koriste CompareTo metodu kako bi usporedili studente. S obzirom da tu to bool funkcije one vračaju True ili False ovisno o usporedbi dvaju studenata, a to smo već objasnili na prijašnjim primjerima.

Programski kod (klasa Quick):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Quick\_Sort

{

class Quick

{

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

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);

}

public static int Partition(object[] array, int left, int right, Comparison cmp)

{

object pivot = array[right];

int i = left - 1;

for (int j = left; j < right; j++)

{

if (cmp(array[j], pivot))

{

i++;

Swap(array, i, j);

}

}

Swap(array, i + 1, right);

return i + 1;

}

public static void Swap(object[] array, int first, int second)

{

object temp = array[first];

array[first] = array[second];

array[second] = temp;

}

}

}

Na početku smo deklarirali delegat koji kao argument prima dva objekta. On nam koristi da pomoću njega šaljemo kriterij prema kojem sortiramo, metodama Sort i Partition.

Metoda Sort koristi Partition rekurzivno to jest poziva ga za desni i lijevi podniz oko pivota. Odabere se pivot putem varijable last pozivanjem metode Partition. Zatim Sort sam sebe poziva rekurzivno za desni i lijevi podniz oko pivota last.

Metodom Partition odabiremo pivot niza, a niz podijelimo na desni i lijevi podniz gledajući od pivota. Pivot postavljamo na bilo koji element niza. Taj pivot uspoređujemo s ostalim elementima niza, a uspoređujemo objekte tipa student pa je potrebno pomoću cmp proslijediti kriterij prema kojem sortiramo. Vršimo Swap metodu nad objektima studenata.

Metoda Swap zamjenjuje dva elementa u nizu.

Programski kod (Main program):

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Quick\_Sort

{

class Program

{

static void Main()

{

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),

};

Console.WriteLine("Početni niz studenata:");

DisplayArray(students);

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

Console.WriteLine("\nSortirani niz studenata po imenu:");

DisplayArray(students);

Quick.Sort(students, 0, students.Length - 1, Student.CompareName);

Console.WriteLine("\nSortirani niz studenata po imenu:");

DisplayArray(students);

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

Console.WriteLine("\nSortirani niz studenata po ocjeni:");

DisplayArray(students);

}

static void DisplayArray(Student[] array)

{

foreach (var student in array)

{

Console.WriteLine(student);

}

Console.WriteLine();

}

}

}

U Main dijelu programa definirani su objekti klase Student s pripadajućim argumentima imena i ocijene. Zatim se ispisao početni niz, pa provjerile funkcionalnosti metoda za obavljanje Quick Sort algoritma na način da su pozvane pa ponovno ispisan niz studenata nakon svake metode.

Slika na kojoj se prikazuje tekst, snimka zaslona

Opis je automatski generiran

1. **Zadatak: Priority Queue**

Programski kod (klasa Heap):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Priority\_Queue

{

class Heap

{

private int size;

private int last;

private int[] priorities;

public Heap(int size)

{

this.size = size;

priorities = new int[size + 1];

last = 0;

}

public void Insert(int priority)

{

if (last >= size)

{

throw new Exception("Queue is full.");

}

last++;

priorities[last] = priority;

BubbleUp(last);

}

public int Retrieve()

{

if (last == 0)

{

throw new Exception("Queue is empty.");

}

int priority = priorities[1];

priorities[1] = priorities[last];

last--;

BubbleDown(1);

return priority;

}

public void BubbleUp(int i)

{

if (i == 1) return;

int parent = i / 2;

if (priorities[i] > priorities[parent])

{

Swap(i, parent);

BubbleUp(parent);

}

}

public void BubbleDown(int i)

{

int leftChild = 2 \* i;

int rightChild = 2 \* i + 1;

int maxIndex = i;

if (leftChild <= last && priorities[leftChild] > priorities[maxIndex])

{

maxIndex = leftChild;

}

if (rightChild <= last && priorities[rightChild] > priorities[maxIndex])

{

maxIndex = rightChild;

}

if (maxIndex != i)

{

Swap(i, maxIndex);

BubbleDown(maxIndex);

}

}

public void Swap(int i, int j)

{

int temp = priorities[i];

priorities[i] = priorities[j];

priorities[j] = temp;

}

public void Display()

{

for (int i = 1; i <= last; i++)

{

Console.Write(priorities[i] + " ");

}

Console.WriteLine();

}

}

}

U klasi Heap realiziramo Partially Ordered Tree, a on je realiziran tako da postoje roditelji i njihova djeca, odnosno čvorovi i nasljedni elementi tog čvora. Svaki čvor ima dva djeteta, s tim da su djeca manja od roditelja. Veličina Heapa se definira varijablom size, niz priorities sadrži elemente Heapa varijabla last je indeks posljednjeg popunjenog heapa.

Metode Insert i Retrieve koriste BubbleUp i BubbleDown metode kako bi ubacili ili izbacili element iz Heapa. Prvo se provjeri postoji li mogućnost za to tako što se provjeri da li je niz možda pun ili prazan. Insert metoda ubacuje neki novi element na indeks s prvim slobodnim mjestom u nizu, te uz pomoć BubbleUp metode ga namještamo na određenu poziciju. Retrieve metoda funkcionira tako da u priority varijablu spremimo prvi element niza, onda na mjesto prvog upišemo vrijednost posljednjeg, a indeks posljednjeg elementa umanjimo za jedan jer nam se sada duljina niza smanjuje zbog izbacivanja elementa. Na kraju je potrebno obaviti BubbleDown metodu za prvi element jer smo ga zamijenili pa Heap moramo sortirati.

BubbleUp metoda prima indeks i element koji se postavlja na višu poziciju u stablu (POT). Pronalazimo indeks roditelja dijeljenjem indeksa i, zato jer su indeksi djece uvijek na indeksu duplo većem i duplo većem plus jedan od indeksa roditelja. U slučaju da je vrijednost djeteta veća od vrijednosti tada mijenjamo mjesta djetetu i roditelju. Pa se nakon toga rekurzivno poziva BubbleUp za roditelja u slučaju da opet treba napraviti zamjenu. BubbleDown radi istu stvar, ali u suprotnom smjeru.

Programski kod (Main program):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Priority\_Queue;

namespace Priority\_Queue

{

class Program

{

static void Main(string[] args)

{

Heap heap = new Heap(10);

try

{

for (int i = 0; i < 11; i++)

heap.Insert(i);

}

catch (Exception x)

{

Console.WriteLine(x.Message);

}

heap.Display();

try

{

while (true)

{

int priority = heap.Retrieve();

heap.Display();

}

}

catch (Exception x)

{

Console.WriteLine(x.Message);

}

}

}

}

U Main dijelu programa stvaramo Heap od 10 članova. Upisujemo brojeve pomoću Insert metode (član 0 nije dio stabla, zato se broji do 11 u for petlji), a zatim izbacujemo pomoću Retrieve metode.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font, broj

Opis je automatski generiran

1. **Zadatak: Heap Sort**

Programski kod (klasa Heap):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Heap\_Sort

{

class Heap

{

public static void Sort(int[] array)

{

Heapify(array);

Arrange(array);

}

public static void Heapify(int[] array)

{

int n = array.Length;

for (int i = n / 2 - 1; i >= 0; i--)

{

BubbleDown(array, i, n);

}

}

public static void Arrange(int[] array)

{

int n = array.Length;

for (int i = n - 1; i > 0; i--)

{

Swap(array, 0, i);

BubbleDown(array, 0, i);

}

}

public static void BubbleDown(int[] array, int i, int last)

{

int leftChild = 2 \* i + 1;

int rightChild = 2 \* i + 2;

int maxIndex = i;

if (leftChild < last && array[leftChild] > array[maxIndex])

{

maxIndex = leftChild;

}

if (rightChild < last && array[rightChild] > array[maxIndex])

{

maxIndex = rightChild;

}

if (maxIndex != i)

{

Swap(array, i, maxIndex);

BubbleDown(array, maxIndex, last);

}

}

public static void Swap(int[] array, int i, int j)

{

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

}

Klasa Heap započinje s definiranjem metoda Heapify i Arrange koje nam služe kod sortiranja Heapa preko metode Sort.

Heapify metoda prima niz i obavlja BubbleDown unutar niza. BubbleDown se poziva onoliko puta koliko je polovina duljine niza, a on kreira Heap od proslijeđenog niza.

Metoda Arrange služi za sortiranje elemenata Heapa tako da koristi Swap i BubbleDown metode. Prvi i zadnji element zamijene mjesta pa se poziva BubbleDown metoda koja gura taj odabrani element prema dolje ako je on manji od onoga ispod.

BubbleDown metoda je slična onoj iz prethodnog zadatka, ona slaže Heap od vrha prema dnu ovisno o vrijednostima elemenata, tako što primi niz unutar kojeg su elementi i element koji treba postavit na pravo mjesto.

Programski kod (Main program):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Heap\_Sort

{

class Program

{

static void Main(string[] args)

{

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

Console.WriteLine("Original array:");

PrintArray(array);

Heap.Sort(array);

Console.WriteLine("\nSorted array:");

PrintArray(array);

}

static void PrintArray(int[] arr)

{

foreach (var item in arr)

{

Console.Write(item + " ");

}

Console.WriteLine();

}

}

}

Unutar Main dijela programa je prvo stvoren niz koji smo ispisali, a zatim ispitalu funkcionalnost metoda Heapa i opet ga ispisali sortiranog.

Slika na kojoj se prikazuje tekst, Font, snimka zaslona, broj

Opis je automatski generiran

**Vježba 4: Lists**

1. **Zadatak: Single Linked Lists**

Programski kod (klasa Node):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Single\_List

{

class Node

{

public object Element { get; private set; }

public Node Next { get; set; }

public Node(object data)

{

Element = data;

Next = null;

}

}

}

Kao što samo ime klase Node kaže, u klasi je kreiran čvor koji sadrži objekt Element to jest element koji se nalazi na tom čvoru, također sadrži objekt tipa Node koji je sljedeću u toj vezanoj listi. Pomoću konstruktora se postavljaju vrijednosti.

Programski kod (klasa List):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Single\_List

{

public class List

{

private Node head;

private Node tail;

public List()

{

head = null;

tail = null;

}

public bool IsEmpty()

{

return head == null;

}

public void InsertFront(object data)

{

Node newNode = new Node(data);

if (head == null)

{

head = newNode;

tail = newNode;

}

else

{

newNode.Next = head;

head = newNode;

}

}

public void InsertEnd(object data)

{

Node newNode = new Node(data);

if (head == null)

{

head = newNode;

tail = newNode;

}

else

{

tail.Next = newNode;

tail = newNode;

}

}

public object RemoveFront()

{

if (IsEmpty())

{

throw new Exception("List is empty");

}

object data = head.Element;

head = head.Next;

if (head == null)

{

tail = null;

}

return data;

}

public object RemoveEnd()

{

if (IsEmpty())

{

throw new Exception("List is empty");

}

object data = tail.Element;

if (head == tail)

{

head = null;

tail = null;

return data;

}

Node current = head;

while (current.Next != tail)

{

current = current.Next;

}

tail = current;

tail.Next = null;

return data;

}

public void Display()

{

Node current = head;

while (current != null)

{

Console.WriteLine(current.Element);

current = current.Next;

}

}

}

}

Na početku klase List su definirana dva čvora, head i tail, koji predstavljaju prvi i zadnji čvor liste, te se oni u konstruktoru postavljaju na null. Funkcija IsEmpty tipa bool gleda da li je prvi element liste prazan, što samim time povlači da je i lista prazna.

Metoda InsertFront ubacuje novi čvor na početak liste. Ako je lista prazna onda i head i tail postavimo na taj novi čvor. U slučaju da lista nije prazna onda se postavi da je idući čvor tog novog čvor na koji head trenutno pokazuje odnosno onaj koji je prvi u vezanoj listi, a zatim head postaviti na naš novi čvor koji tada postaje prvi element liste dok prijašnji prvi je sada drugi jer Next novog pokazuje na njega i na njemu više nije head.

Metoda InsertEnd ubacuje novi čvor na kraj liste. Ako je lista prazna onda head i tail postavimo na taj novi čvor. U slučaju da lista nije prazna onda se zadnji element nadoveže na naš novi čvor s Next, a zatim tail postavimo na novi čvor, time naš novi čvor postaje zadnji element vezane liste, a ujedno i idući element prošlom zadnjem čvoru tako da lista bude ispravno vezana.

Metoda RemoveFront izbacuje čvor s početka liste. Ako je lista prazna nema se što uklonit, pa porukom u konzoli javljamo da je lista prazna. Ukoliko lista nije prazna kreira se objekt data kojeg postavimo na vrijednost di je head tako da ne izgubimo početak liste kada se makne prvi element, head zatim postavimo na vrijednost idućeg čvora i samim time smo izbacili prvi element liste. U slučaju da je sada head null to znači da imamo samo jedan element liste, a samim time postavimo i tail na null.

Metoda RemoveEnd izbacuje čvor s kraja liste. Ako je lista prazna nema se što uklonit, pa porukom u konzoli javljamo da je lista prazna. U ovom slučaju također kreiramo objekt kojeg postavimo na vrijednost elementa na kojeg pokazuje tail, tako da kasnije možemo vratiti element čvora kojeg ćemo ukloniti. Ako su head i tail jednaki to znači da imamo jedan element i onda oboje možemo postaviti na null jer brišemo taj jedini element. Kreira se novi čvor zvan current koji postavljamo na početak liste tako što ga izjednačimo s head, jer nam on služi da idemo kroz listu i dođemo do predzadnjeg elementa liste to jest onoga kojem je current.Next jedan tailu, onda tail postavimo na taj predzadnji element liste i njegov Next postavimo na null jer je to kraj liste.

Metoda Display nam služi za ispis vezane liste, a ispisuje na način da kreiramo čvor zvan current i postavimo na head i zatim njime šetamo kroz listu i ispisujemo element svakog čvora dok ne dođe do null vrijednosti jer to znači da je došao na kraj liste, jer Next zadnjeg elementa je null.

Programski kod (Main program):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Single\_List

{

class Program

{

static void Main(string[] args)

{

List list = new List();

list.InsertFront("Ivo");

list.InsertEnd(1);

list.InsertEnd(3.14);

list.InsertFront(true);

list.InsertEnd(100m);

Console.WriteLine("Lista:");

list.Display();

try

{

list.RemoveEnd();

list.RemoveFront();

Console.WriteLine("\nLista nakon micanja prvog i zadnjeg elementa:");

list.Display();

list.RemoveEnd();

list.RemoveFront();

list.RemoveFront();

}

catch (Exception x)

{

Console.WriteLine(x.Message);

}

Console.WriteLine("\nLista nakon micanja preostalih elemenata:");

list.Display();

}

}

}

U main dijelu programa smo isprobali svaku od metoda koje smo definirali, tako što smo dodali nove elemente koji su zadani u zadatku u listu i na početak i na kraj, a zatim i micali elementa i s početka i s kraja liste, te sve to popratili ispisivanjem liste u konzolu.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font

Opis je automatski generiran

1. **Zadatak: Double Linked Lists**

Programski kod (klasa Node):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Double\_List

{

class Node

{

public object Element { get; private set; }

public Node Next { get; set; }

public Node Prev { get; set; }

public Node()

{

}

public Node(object data, Node nextNode, Node prevNode)

{

Element = data;

Next = nextNode;

Prev = prevNode;

}

}

}

Klasa Node je drugačija od one u prethodnom zadatku po tome što kada je dvostruko vezana lista, onda čvorovi pokazuju na idući i na prethodni element te liste, a definirali smo ih sa Next i Prev.

Programski kod (klasa BllList):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Double\_List

{

public class BllList

{

private Node head;

private int count;

public int Count { get { return count; } }

public BllList()

{

head = new Node();

head.Next = head;

head.Prev = head;

}

private bool IsEmpty()

{

return count == 0;

}

public void Append(object data)

{

Node newNode = new Node(data, head, head.Prev);

head.Prev.Next = newNode;

head.Prev = newNode;

count++;

}

private Node Find(object data)

{

Node current = head.Next;

while (current != head)

{

if (current.Element.Equals(data))

{

return current;

}

current = current.Next;

}

return null;

}

public void Remove(object data)

{

Node nodeToRemove = Find(data);

if (nodeToRemove == null)

{

throw new Exception("Element not found in the list.");

}

nodeToRemove.Prev.Next = nodeToRemove.Next;

nodeToRemove.Next.Prev = nodeToRemove.Prev;

count--;

}

public bool Search(object data)

{

return Find(data) != null;

}

public void Display()

{

Node current = head.Next;

while (current != head)

{

Console.WriteLine(current.Element);

current = current.Next;

}

}

}

}

Klasa BllList nam služi da kreiramo dvostruko vezanu listu, tako što ima varijablu za početak liste head i za broj elemenata liste count. Konstruktor uvijek postavlja head na novi čvor koji smo stvorili i njegov pokazivač na prethodni i slijedeći također postavlja na isti čvor.

Metoda IsEmpty pomoću count varijable gleda da li je lista prazna.

Metoda Append stvara novi čvor u listi s elementom koji je primljen preko dana. Postavlja se da je idući čvor tog novog čvora onaj koji je trenutni head, a prethodni na head.Prev. Onda head.Prev.Next postavimo na novi čvor i head.Prev postavljamo na novi čvor. Na kraju se count uveća za jedan, jer smo dodali element u listu.

Za napraviti ispravnu funkcionalnost metode Find prvo smo kreirali čvor current koji nam služi za šetanje listom, a taj current se postavi na drugi element liste. Ako čvor postoji vrača se taj čvor, a ako nije pronađen nijedan vrača se null. Metoda Search se nadovezuje na ovu metodu jer je koristi i vrača true ili false ovisno o tome postoji li element ili ne.

Metoda Remove prvo koristi Find metodu za pronalazak onog elementa koji treba ukloniti. Next prethodnog elementa se postavlja na onaj element nakon toga koji želimo ukloniti, a Prev idućeg elementa od tog elementa kojeg želimo ukloniti se postavi na onaj element prije tog elementa kojeg želimo ukloniti. Count smanjimo za jedan jer je iz liste izbačen jedan element.

Programski kod (klasa Student):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Double\_List

{

class Student

{

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 override bool Equals(object obj)

{

if (obj == null) return false;

if (this.GetType() != obj.GetType())

return false;

Student other = obj as Student;

return name == other.name && grade == other.grade;

}

public override int GetHashCode()

{

return name.GetHashCode() | grade.GetHashCode();

}

public static bool operator ==(Student a, Student b)

{

if ((object)a == null)

if ((object)b == null) return true;

else return false;

return (a.Equals(b));

}

public static bool operator !=(Student a, Student b)

{

return !(a == b);

}

}

}

Klasa Student za promjenu od prethodnih ima override metodu Equals koja provjerava studente po ocjeni i imenu, metoda GetHashCode, operator == koji vrača true ili false ovisno o usporedbi dvaju studenata po imenu i ocjeni i operator !=.

Programski kod (Main program):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Double\_List

{

class Program

{

static void Main(string[] args)

{

BllList list = new BllList();

Student a = new Student("Tom", 4.7);

Student b = new Student("Kim", 4.3);

if (a == b)

Console.WriteLine("Objects are EQUAL");

else

Console.WriteLine("Objects are NOT EQUAL");

list.Append(a);

list.Append(b);

list.Append(new Student("Mia", 4.2));

list.Append(new Student("Tea", 4.4));

list.Append(new Student("Lea", 4.1));

Student c = new Student("Kim", 4.3);

list.Append(c);

list.Append(new Student("Bob", 4.5));

list.Append(new Student("Ena", 4.9));

Console.WriteLine("Count: " + list.Count);

list.Display();

if (list.Search(c))

Console.WriteLine("Element is in the list.");

else

Console.WriteLine("Element is not in the list.");

try

{

list.Remove(a);

list.Remove(c);

}

catch (Exception x)

{

Console.WriteLine(x.Message);

}

Console.WriteLine("Count: " + list.Count);

list.Display();

}

}

}

U main dijelu programa smo dodali studente skupa s njihovim imenom i ocjenama u dvostruko vezanu listu metodom Append, te zatim ispitali funkcionalnost metode Search i Remove.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font, dizajn

Opis je automatski generiran

**Vježba 5: Stacks i Queues**

1. **Zadatak: Stacks**

U ovom zadatku se prvi put susrećemo s korištenjem delegata. Dodajemo referencu na projekt Single List koji smo već napisali, u naš trenutni projekt Stack.

Programski kod (klasa Stack):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Single\_List;

namespace Single\_List

{

class Stack

{

private List list;

public Stack()

{

list = new List();

}

public void Push(object data)

{

list.InsertFront(data);

}

public object Pop()

{

object data = list.RemoveFront();

return data;

}

public bool IsEmpty()

{

return list.IsEmpty();

}

public void Display()

{

list.Display();

}

}

}

Klasa Stack koristi već prethodno napisanu jednostruku listu i sve njene metode. Metode za izbacivanje i dodavanje elemenata u stog smo nazvali Push i Pop, jer i izbacujemo i dodajemo elemente isključivo s početka (zbog toga što izbjegavamo složene metode, odnosno draže su nam one kojima me složenost O(1) nego O(n)-Remove.End). Za implementaciju Push metode koristi se InsertFront metoda vezane liste, a za implementaciju Pop metode se koristi RemoveFront metoda vezane liste.

Programski kod (Main program):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Single\_List

{

class Program

{

static void Main(string[] args)

{

Stack stack = new Stack();

try

{

stack.Push("Ivo");

stack.Push("Ana");

stack.Push("Bob");

stack.Push("Iva");

stack.Push("Kim");

Console.WriteLine("Elementi u stogu:");

stack.Display();

while (!stack.IsEmpty())

{

Console.WriteLine("\nStog nakon izbacivanja:");

stack.Pop();

stack.Display();

}

Console.WriteLine("\nStog je prazan");

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

}

}

Kao i u svakom projektu, kod izrade Main dijela programa, prema uputama iz skripte definiramo elemente i s njima provodimo ispitivanje funkcionalnosti napisanih metoda, u ovom slučaju su to Push i Pop metoda te također IsEmpty metoda.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font, izbornik

Opis je automatski generiran

1. **Zadatak: Queues**

Za ovaj projekt implementiramo red elemenata koristeći prethodno implementiranu jednostruko vezanu listu, to vršimo kao i u prethodnom zadatku.

Programski kod (klasa Queue):

using Single\_List;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Queue

{

class Queue

{

private List list;

public Queue()

{

list = new List();

}

public void Enqueue(object data)

{

list.InsertEnd(data);

}

public object Dequeue()

{

object data = list.RemoveFront();

return data;

}

public bool IsEmpty()

{

return list.IsEmpty();

}

public void Display()

{

list.Display();

}

}

}

U ovom zadatku implementiramo metode za dodavanje na kraj reda i micanje s početka reda. Metoda Enqueue koristi InsertEnd metodu za dodavanje novog elementa u red. Metoda Dequeue koristi RemoveFront metodu za uklanjanje elementa iz reda. IsEmpty metoda služi za provjeru da li je Queue prazan.

Programski kod (Main program):

using Single\_List;

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();

try

{

queue.Enqueue("Ivo");

queue.Enqueue("Ana");

queue.Enqueue("Bob");

queue.Enqueue("Iva");

queue.Enqueue("Leo");

Console.WriteLine("Elementi u redu:");

queue.Display();

while (!queue.IsEmpty())

{

Console.WriteLine("\nRed nakon izbacivanja:");

queue.Dequeue();

queue.Display();

}

Console.WriteLine("\nQueue is empty");

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

}

}

Kao i u svakom projektu, kod izrade Main dijela programa, prema uputama iz skripte definiramo elemente i s njima provodimo ispitivanje funkcionalnosti napisanih metoda, u ovom slučaju su to Enqueue i Dequeue metoda te također IsEmpty metoda.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font, izbornik

Opis je automatski generiran

**Vježba 7: Hash Tables**

1. **Zadatak: Defining a Hash Table**

Programski kod (klasa Node):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Hash\_Table

{

public 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;

}

}

}

U klasu Node implementirane su varijable Name, Value i Next, te su definirane u konstruktoru.

Programski kod (klasa HashTable):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Hash\_Table

{

public class HashTable

{

private Node[] buckets;

private int length;

public HashTable(int length)

{

this.length = length;

buckets = new Node[length];

}

public void Display()

{

for (int i = 0; i < length; i++)

{

Node current = buckets[i];

Console.Write("Bucket "+i+": ");

while (current != null)

{

Console.Write("["+current.Name+", "+current.Value+"]");

current = current.Next;

}

Console.WriteLine("null");

}

}

private int Hash(string str)

{

int hash = 0;

foreach (char c in str)

{

hash += c;

}

return hash % length;

}

}

}

Hash tablica je podatkovna struktura koja određenim ključevima pridjeljuje njihove vrijednosti. Klasa HashTable se sastoji od čvorova u kojima se pohranjuje ključ i vrijednost, u našem slučaju Name i Value. Lenght sprema veličinu niza tih čvorova koje smo nazvali buckets.

Display metoda ispisuje sve elemente iz Hash tablice, ispisuje ih na način da za svaki bucket se ispiše Ime i Vrijednost i tako redom od početka do kraja.

Metoda Hash prima string kojemu se svako slovo zbroji kao integer i cjelobrojno podijeli sa dužinom stringa. Tako dobivamo indeks elementa na kojem se nalazi taj element u nizu.

Programski kod (Main program):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Hash\_Table

{

class Program

{

static void Main(string[] args)

{

HashTable hashTable = new HashTable(10);

hashTable.Display();

}

}

}

U ovom dijelu main programa definira se Hash tablica sa 10 čvorova i ispisuje.

1. **Zadatak: Ubserting Name-Value Pairs**

Programski kod (HashTable):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Hash\_Table

{

public class HashTable

{

private Node[] buckets;

private int length;

public HashTable(int length)

{

this.length = length;

buckets = new Node[length];

}

public void Display()

{

for (int i = 0; i < length; i++)

{

Node current = buckets[i];

Console.Write("Bucket "+i+": ");

while (current != null)

{

Console.Write("["+current.Name+", "+current.Value+"]");

current = current.Next;

}

Console.WriteLine("null");

}

}

private int Hash(string str)

{

int hash = 0;

foreach (char c in str)

{

hash += c;

}

return hash % length;

}

public void Insert(string name, int value)

{

int index = Hash(name);

Node newNode = new Node(name, value, null);

if (buckets[index] == null)

{

buckets[index] = newNode;

}

else

{

Node current = buckets[index];

while (current.Next != null)

{

current = current.Next;

}

current.Next = newNode;

}

}

}

}

U drugom zadatku smo trebali implementirati metodu Insert koja će ubaciti element u Hash tablicu. Zbog toga je navedena samo klasa koja je izmijenjena, ostale klase kada ne navedem znači da su ostale identične jer se zadaci nadovezuju na isti projekt.

Metoda Insert poziva Hash metodu kako bi izračunala odgovarajuću vrijednost za string, a ta vrijednost će postati indeks u Hash tablici, a na mjestu pod tim indeksom se taj string ubacuje kao novi element odnosno bucket. Na taj izračunati indeks se u tablici umetne novi čvor s pripadajućim vrijednostima. Njegov Next postavimo na njega to jest pokazuje sam na sebe, a njegov prethodnik će pokazivati na njega.

1. **Zadatak: Searching for Names**

Programski kod (klasa HashTable):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Hash\_Table

{

public class HashTable

{

private Node[] buckets;

private int length;

public HashTable(int length)

{

this.length = length;

buckets = new Node[length];

}

public void Display()

{

for (int i = 0; i < length; i++)

{

Node current = buckets[i];

Console.Write("Bucket " + i + ": ");

while (current != null)

{

Console.Write("[" + current.Name + ", " + current.Value + "]");

current = current.Next;

}

Console.WriteLine("null");

}

}

private int Hash(string str)

{

int hash = 0;

foreach (char c in str)

{

hash += c;

}

return hash % length;

}

public void Insert(string name, int value)

{

int index = Hash(name);

Node newNode = new Node(name, value, null);

if (buckets[index] == null)

{

buckets[index] = newNode;

}

else

{

Node current = buckets[index];

while (current.Next != null)

{

current = current.Next;

}

current.Next = newNode;

}

}

public int Search(string name)

{

int index = Hash(name);

Node current = buckets[index];

while (current != null)

{

if (current.Name == name)

{

return current.Value;

}

current = current.Next;

}

throw new InvalidOperationException("Node not found in the hash table.");

}

}

}

U trećem zadatku smo implementirali metodu Search. Ona prima ime elementa kojeg tražimo, to ime iskoristimo u Hash metodi kako bi pronašli indeks na kojem je taj element smješten u nizu buckets. Kreiramo čvor current koji postavljamo na taj element kojem je indeks izračunat. Ovim čvorom šetamo kroz niz kako bi pronašli element kojem je ime jednako imenu našeg pomoćnog čvora. Ako nađemo element onda se vrača njegova vrijednost, a ako ne nađemo onda se javlja greška da tog elementa nema.

1. **Zadatak: Deleting Name-Value Pairs**

Programski kod (klasa HashTable):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Hash\_Table

{

public class HashTable

{

private Node[] buckets;

private int length;

public HashTable(int length)

{

this.length = length;

buckets = new Node[length];

}

public void Display()

{

for (int i = 0; i < length; i++)

{

Node current = buckets[i];

Console.Write("Bucket " + i + ": ");

while (current != null)

{

Console.Write("[" + current.Name + ", " + current.Value + "]");

current = current.Next;

}

Console.WriteLine("null");

}

}

private int Hash(string str)

{

int hash = 0;

foreach (char c in str)

{

hash += c;

}

return hash % length;

}

public void Insert(string name, int value)

{

int index = Hash(name);

Node newNode = new Node(name, value, null);

if (buckets[index] == null)

{

buckets[index] = newNode;

}

else

{

Node current = buckets[index];

while (current.Next != null)

{

current = current.Next;

}

current.Next = newNode;

}

}

public int Search(string name)

{

int index = Hash(name);

Node current = buckets[index];

while (current != null)

{

if (current.Name == name)

{

return current.Value;

}

current = current.Next;

}

throw new InvalidOperationException("Node not found in the hash table.");

}

public void Delete(string name)

{

int index = Hash(name);

if (buckets[index] == null)

{

throw new InvalidOperationException("Node not found in the hash table.");

}

if (buckets[index].Name == name)

{

buckets[index] = buckets[index].Next;

return;

}

Node prev = buckets[index];

Node current = buckets[index].Next;

while (current != null && current.Name != name)

{

prev = current;

current = current.Next;

}

if (current == null)

{

throw new InvalidOperationException("Node not found in the hash table.");

}

prev.Next = current.Next;

}

}

}

U četvrtom zadatku smo implementirali metodu Delete koja briše Name-Value par iz Hash tablice. Metoda prima ime elementa koji se briše, te se opet preko metode Hash izračuna indeks na kojem je taj element u tablici. Kreiramo

1. **Zadatak: Testing Hash Tables**

Programski kod (Main program):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Hash\_Table

{

class Program

{

static void Main(string[] args)

{

HashTable hashTable = new HashTable(3);

hashTable.Insert("Boris", 33);

hashTable.Insert("Grgo", 25);

hashTable.Insert("Ivana", 22);

hashTable.Insert("Antonia", 11);

Console.WriteLine("Hash Table Contents:");

hashTable.Display();

string searchKey = "Boris";

try

{

int searchResult = hashTable.Search(searchKey);

Console.WriteLine($"Value for key '{searchKey}' found: {searchResult}");

}

catch (InvalidOperationException ex)

{

Console.WriteLine(ex.Message);

}

string[] keysToDelete = { "Grgo", "Ivana" };

foreach (string key in keysToDelete)

{

try

{

hashTable.Delete(key);

Console.WriteLine($"Key '{key}' deleted successfully.");

}

catch (InvalidOperationException ex)

{

Console.WriteLine(ex.Message);

}

}

Console.WriteLine("\nUpdated Hash Table Contents:");

hashTable.Display();

}

}

}

U main dijelu programa smo ispitali funkcionalnost medota: Insert, Display, Search, Delete. Za imena sam koristio ima svojih članova obitelji.

Slika na kojoj se prikazuje tekst, snimka zaslona, Font

Opis je automatski generiran

**Vježba 8: Trees**

1. **Zadatak: Defining Binary Search Trees**

Programski kod (klasa Node):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Binary\_Tree

{

public class Node

{

public int Data { get; set; }

public Node Left { get; set; }

public Node Right { get; set; }

public Node(int data)

{

Data = data;

Left = Right = null;

}

public void Insert(int value)

{

if (value < Data)

{

if (Left == null)

{

Left = new Node(value);

}

else

{

Left.Insert(value);

}

}

else if (value > Data)

{

if (Right == null)

{

Right = new Node(value);

}

else

{

Right.Insert(value);

}

}

}

}

}

U prvome zadatku smo napisali konstruktor klase Node koji postavlja Data na primljeni argument, a Left i Right postavlja na null automatski. Također smo napisali metodu Insert koja nam služi za umetanje čvorova u stablo, na način da se od gleda od čvora koji je roditelj tako da se manji element upisuje na lijevi „pokazivač“ (dijete), a veći element se upisuje na desni „pokazivač“, u slučaju da nema tog čvora, u koji se treba novi element upisati, onda se umeće novi element na to mjesto, a i slučaju da je na tom mjestu već neki element onda se metoda Insert mora pozvati rekurzivno za to podstablo dok ne dođemo do slobodnog mjesta za upis tog elementa.

1. **Zadatak: Inserting in BSTs**

Programski kod (klasa Tree):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Binary\_Tree

{

public class Tree

{

private Node root;

public Tree()

{

root = null;

}

public void Insert(int value)

{

if (root == null)

{

root = new Node(value);

}

else

{

root.Insert(value);

}

}

}

}

U drugom zadatku smo kreirali klasu Tree u kojoj smo u konstruktoru root postavili na null i implementirali metodu Insert. Insert metoda koristi Node metodu za umetanje. U slučaju da je stablo prazno novi čvor će biti zapravo korijen stabla. A u slučaju da stablo nije prazno koristi se Node metoda za ubacivanje novog elementa.

1. **Zadatak: Searching in BSTs**

Programski kod (klasa Tree):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Binary\_Tree

{

public class Tree

{

private Node root;

public Tree()

{

root = null;

}

public void Insert(int value)

{

if (root == null)

{

root = new Node(value);

}

else

{

root.Insert(value);

}

}

public bool Search(int value)

{

Node node = Search(root, value);

return node != null;

}

private Node Search(Node current, int value)

{

if (current == null || current.Data == value)

{

return current;

}

if (value < current.Data)

{

return Search(current.Left, value);

}

else

{

return Search(current.Right, value);

}

}

}

}

U trećem zadatku je implementirana metoda Search u klasi Tree i ta metoda pronalazi vrijednost elementa u stablu i vrača taj čvor. Također je implementirana metoda Search, ali je bool pa vrača true ili false ovisno da li je element pronađen ili ne.

Metoda Search prima čvor koji nazovemo current i vrijednost koju tražimo. U slučaju da vrijednost odgovara vrijednosti na čvoru current onda vračamo taj čvor, a ako nije onda je potrebno dalje tražiti u lijevom ili desnom podstablu ovisno dali je tražena vrijednost manja ili veća od one koju sadrži naš čvor current i tada se metoda Search rekurzivno poziva dok se ne pronađe traženi element.

1. **Zadatak: Deleting in BSTs**

Programski kod (klasa Tree):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Binary\_Tree

{

public class Tree

{

private Node root;

public Tree()

{

root = null;

}

public void Insert(int value)

{

if (root == null)

{

root = new Node(value);

}

else

{

root.Insert(value);

}

}

public bool Search(int value)

{

Node node = Search(root, value);

return node != null;

}

private Node Search(Node current, int value)

{

if (current == null || current.Data == value)

{

return current;

}

if (value < current.Data)

{

return Search(current.Left, value);

}

else

{

return Search(current.Right, value);

}

}

public void Delete(int value)

{

Delete(ref root, value);

}

private void Delete(ref Node current, int value)

{

if (current == null)

{

return;

}

if (value < current.Data)

{

Node left = current.Left;

Delete(ref left, value);

current.Left = left;

}

else if (value > current.Data)

{

Node right = current.Right;

Delete(ref right, value);

current.Right = right;

}

else

{

if (current.Left == null)

{

current = current.Right;

}

else if (current.Right == null)

{

current = current.Left;

}

else

{

int successorValue = DeleteSuccessor(current.Right, current);

current.Data = successorValue;

}

}

}

private int DeleteSuccessor(Node current, Node parent)

{

while (current.Left != null)

{

parent = current;

current = current.Left;

}

int smallestValue = current.Data;

if (parent.Left == current)

{

parent.Left = current.Right;

}

else

{

parent.Right = current.Right;

}

return smallestValue;

}

}

}

U četvrtom zadatku smo trebali implementirati metodu koja briše čvor iz stabla, jedna metoda prima kao argument vrijednost elementa na tom čvoru i poziva istoimenu metodu koju implementiramo.

Metoda putem reference prima čvor od kojeg kreće sa šetanjem po stablu, a to je korijenski čvor i prima vrijednost koja se traži. Kada se dođe do krajnjeg lijevog elementa njegova vrijednost se zapisuje u varijablu successorValue. Taj zadnji čvor izbrišemo tako što current postavimo na current.right. Tako smo izbrisali čvor, a metoda se rekurzivno opet poziva i šalje se vrijednost također. Metoda provjeri da li je stablo prazno, a ako nije onda brišemo element. Ukoliko to nije element koji trebamo obrisati šetamo dalje po stablu kao u prethodnom zadatku jer su elementi isto organizirani lijevo desno kao manji veći. A kada nađemo točno element koji treba obrisati poziva se metoda DeleteSuccessor koja briše element, a toj metodi prosljeđujemo desni čvor od traženog jer funkcija briše zapravo taj čvor.

1. **Zadatak: Traversing BSTs**

Programski kod (klasa Tree):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Binary\_Tree

{

public enum TraversalType { PreOrder, InOrder, PostOrder }

public class Tree

{

private Node root;

public Tree()

{

root = null;

}

public void Insert(int value)

{

if (root == null)

{

root = new Node(value);

}

else

{

root.Insert(value);

}

}

public bool Search(int value)

{

Node node = Search(root, value);

return node != null;

}

private Node Search(Node current, int value)

{

if (current == null || current.Data == value)

{

return current;

}

if (value < current.Data)

{

return Search(current.Left, value);

}

else

{

return Search(current.Right, value);

}

}

public void Delete(int value)

{

Delete(ref root, value);

}

private void Delete(ref Node current, int value)

{

if (current == null)

{

return;

}

if (value < current.Data)

{

Node left = current.Left;

Delete(ref left, value);

current.Left = left;

}

else if (value > current.Data)

{

Node right = current.Right;

Delete(ref right, value);

current.Right = right;

}

else

{

if (current.Left == null)

{

current = current.Right;

}

else if (current.Right == null)

{

current = current.Left;

}

else

{

int successorValue = DeleteSuccessor(current.Right, current);

current.Data = successorValue;

}

}

}

private int DeleteSuccessor(Node current, Node parent)

{

while (current.Left != null)

{

parent = current;

current = current.Left;

}

int smallestValue = current.Data;

if (parent.Left == current)

{

parent.Left = current.Right;

}

else

{

parent.Right = current.Right;

}

return smallestValue;

}

public void Traverse(TraversalType traverse)

{

switch (traverse)

{

case TraversalType.PreOrder:

TraversePreorder(root);

break;

case TraversalType.InOrder:

TraverseInorder(root);

break;

case TraversalType.PostOrder:

TraversePostorder(root);

break;

default:

Console.WriteLine("Invalid traversal type!");

break;

}

}

private void TraversePreorder(Node node)

{

if (node == null)

return;

Console.Write(node.Data + " ");

TraversePreorder(node.Left);

TraversePreorder(node.Right);

}

private void TraverseInorder(Node node)

{

if (node == null)

return;

TraverseInorder(node.Left);

Console.Write(node.Data + " ");

TraverseInorder(node.Right);

}

private void TraversePostorder(Node node)

{

if (node == null)

return;

TraversePostorder(node.Left);

TraversePostorder(node.Right);

Console.Write(node.Data + " ");

}

}

}

U petom zadatku smo implementirali metode za šetanje po stablu. Za razliku od drugih struktura podataka koje imaju jedan način za šetanje sada šetamo na drugačiji način jer smo dodali enumeraciju TraversalType u datoteku Tree.cs koja definira tri načina prolaska kroz binarno pretraživačko stablo: PreOrder, InOrder i PostOrder. Zatim se unutar klase Tree dodaju metode Traverse koje omogućuju izvođenje odabrane vrste prolaska kroz stablo. Metoda Traverse prima parametar tipa TraversalType koji određuje način prolaska i prema tome poziva odgovarajuću privatnu metodu (TraversePreorder, TraverseInorder ili TraversePostorder) s početnim čvorom (root). Svaka od privatnih metoda implementira odgovarajući algoritam za rekurzivni prolazak kroz stablo u odabranom redoslijedu.

1. **Zadatak: Testing BSTs**

Programski kod (Main program):

using Binary\_Tree;

using System;

class Program

{

static void Main(string[] args)

{

Tree tree = new Tree();

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

foreach (int i in array)

tree.Insert(i);

tree.Traverse(TraversalType.InOrder);

int value = 4;

if (tree.Search(value))

Console.WriteLine("\nValue {0:d} is found in the tree.", value);

else

Console.WriteLine("Value {0:d} is not found in the tree.", value);

tree.Delete(4);

tree.Traverse(TraversalType.InOrder);

}

}

U šestom zadatku smo testirali rade li napisane metode na ispravan način. Prema uputama smo napravili taj dio programa i u konzolu je ispisano točno što se tražilo.

Slika na kojoj se prikazuje tekst, Font, snimka zaslona, broj

Opis je automatski generiran

**Vježba 9: Graphs**

1. **Zadatak: Defining an Edge**

Programski kod (klasa Edge):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Shortest\_Path

{

public class Edge

{

public int Destination { get; set; }

public double Cost { get; set; }

public Edge(int destination, double cost)

{

Destination = destination;

Cost = cost;

}

public override string ToString()

{

return $"[{Destination}, {Cost}]";

}

}

}

U prvom zadatku smo dodali klasu Edge koja opisuje bridove grafa, sadrži Destination i Cost argumente. U konstruktoru se te varijable definiraju i postavljaju na primljene argumente. Također sadrži ToString metodu koja ispisuje objekt u formatu koji je lakši za razumijevanje.

1. **Zadatak: Defining a Vertex**

Programski kod (klasa Vertex):

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Shortest\_Path

{

public class Vertex

{

public int Source { get; set; }

public double Distance { get; set; }

public int Index { get; set; }

public List<Edge> Neighbors { get; }

public Vertex(int source)

{

Source = source;

Distance = double.MaxValue;

Neighbors = new List<Edge>();

}

public void AddEdge(int destination, double cost)

{

Neighbors.Add(new Edge(destination, cost));

}

public override string ToString()

{

StringBuilder builder = new StringBuilder();

string distance = string.Format("{0,2}", (Distance == double.MaxValue) ? -1.0 : Distance);

builder.Append("Q" + Index + " N" + Source + "(" + distance + ") --> ");

foreach (var edge in Neighbors)

{

builder.Append(edge);

}

return builder.ToString();

}

}

}

U drugom zadatku smo kreirali klasu Vertex koja ima Source Distance i Indeks varijable. Također deklarira se lista susjednih bridova. Konstruktor služi za postavljanje Sourcea na određenu primljenu vrijednost. Metoda AddEdge dodaje brid u listu susjednih bridova sa svojim vrijednostima. A to ToString ispisuje u formatu lakšeg razumijevanja tog i susjednih bridova, tako se čvorovi grafa opisuju na jednostavan način.

1. **Zadatak: Grapf as an Adjacenty list**

Programski kod (klasa Graph):

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Shortest\_Path

{

public enum DirectionType

{

Directed,

Undirected

}

public class Graph : IEnumerable

{

internal Vertex[] vertices;

PartiallyOrderedTree pot;

public Graph(int[] nodes)

{

vertices = new Vertex[nodes.Length];

pot = new PartiallyOrderedTree(this);

for (int i = 0; i < vertices.Length; i++)

{

this.vertices[i] = new Vertex(nodes[i]);

vertices[i].Index = i + 1;

}

}

public void AddEdge(int source, int destination, double cost, DirectionType direction)

{

if (direction == DirectionType.Undirected)

{

vertices[source - 1].AddEdge(destination, cost);

vertices[destination - 1].AddEdge(source, cost);

}

else

{

vertices[source - 1].AddEdge(destination, cost);

}

}

IEnumerator IEnumerable.GetEnumerator()

{

foreach (var vertex in vertices)

{

yield return vertex;

}

}

public void Display()

{

foreach (var o in this)

{

Console.WriteLine(o);

}

pot.Display();

Console.ReadLine();

}

}

}

U trećem zadatku dodajemo klasu Graph koja se implementira kao IEnumerable jer se koristi enumeracija DirectionTYpe koja ima članove Directed i Undirected. Konstruktor prima niz čvorova i inicijalizira polje vertices, a zatim kreira polje kao PartiallyOrderedTree s referencom na graf. AddEdge metoda služi za dodavanje novog brida. Metoda GetEnumerator šeće kroz grafičke čvorove, a Display ih ispisuje.

1. **Zadatak: Partially Order Tree**

Programski kod (klasa PartiallyOrderedTree):

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Shortest\_Path

{

public class PartiallyOrderedTree : IEnumerable

{

internal int[] queue;

Graph graph;

public int last;

public PartiallyOrderedTree(Graph graph)

{

this.graph = graph;

last = graph.vertices.Length;

queue = new int[last + 1];

queue[0] = -1;

for (int i = 0; i < last; i++)

{

queue[i + 1] = i;

}

}

public void Swap(int a, int b)

{

int temp = queue[a];

queue[a] = queue[b];

queue[b] = temp;

}

double GetPriority(int a)

{

return graph.vertices[queue[a]].Distance;

}

internal void BubbleUp(int index)

{

while (index > 1 && GetPriority(index / 2) > GetPriority(index))

{

Swap(index, index / 2);

index /= 2;

}

}

internal void BubbleDown(int index)

{

while (2 \* index <= last)

{

int childIndex = 2 \* index;

if (childIndex < last && GetPriority(childIndex + 1) < GetPriority(childIndex))

{

childIndex++;

}

if (GetPriority(index) <= GetPriority(childIndex))

{

break;

}

Swap(index, childIndex);

index = childIndex;

}

}

IEnumerator IEnumerable.GetEnumerator()

{

foreach (var priority in queue)

{

yield return priority;

}

}

public void Display()

{

Console.Write("Queued: ");

for (int i = 1; i <= last; i++)

{

Console.Write(queue[i] + " ");

}

Console.WriteLine();

Console.Write("Settled: ");

for (int i = last + 1; i < queue.Length; i++)

{

Console.Write(queue[i] + " ");

}

Console.WriteLine();

}

}

}

U četvrtom zadatku unutar ove klase se deklariraju interna polja queue koja predstavljaju red prioriteta i Graph. Konstruktor prima graf kao paramerar i postavlja polje graph na tu vrijednost. Swap metoda zamjenjuje elemente u polju queue sa primljenih indeksima. GetPriority vrača prioritet čvora sa određenog indeksa pomoću distance. BubbleUp i BubbleDown služe za reorganizaciju prioriteta u redu koristeći GetPriority i Swap. Na kraju GetEnumerator i Display na isti naćin ispisuju.

1. **Zadatak: Finding Shortest Path**

Programski kod (klasa FindShorthestPath):

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Shortest\_Path

{

public class FindShorthestPath : IEnumerable

{

internal Vertex[] vertices;

PartiallyOrderedTree pot;

public Graph(int[] nodes)

{

vertices = new Vertex[nodes.Length];

pot = new PartiallyOrderedTree(this);

for (int i = 0; i < vertices.Length; i++)

{

this.vertices[i] = new Vertex(nodes[i]);

vertices[i].Index = i + 1;

}

}

public void AddEdge(int source, int destination, double cost, DirectionType direction)

{

if (direction == DirectionType.Undirected)

{

vertices[source - 1].AddEdge(destination, cost);

vertices[destination - 1].AddEdge(source, cost);

}

else

{

vertices[source - 1].AddEdge(destination, cost);

}

}

public void FindShortestPath(int source)

{

vertices[source - 1].Distance = 0;

pot.BubbleUp(vertices[source - 1].Index);

while (pot.last > 0)

{

int u = pot.queue[1];

pot.Swap(1, pot.last);

pot.last--;

pot.BubbleDown(1);

foreach (Edge edge in vertices[u].Neighbors)

{

int v = edge.Destination;

double alt = vertices[u].Distance + edge.Cost;

if (alt < vertices[v - 1].Distance)

{

vertices[v - 1].Distance = alt;

pot.BubbleUp(vertices[v - 1].Index);

}

}

}

}

}

}

U petom zadatku sam napravio ipak novu klasu jer u klasi Graph ne funkcionira. Metoda FindShorthestPath implementira algoritam kako bi se pronašao najkraći put do određenog čvora u grafu. Prolazi kroz sve čvorove i ažurira se sam.

1. **Zadatak: Testing the Graph**

Programski kod (Main program):

using Shortest\_Path;

using System;

class Program

{

static void Main(string[] args)

{

Graph undirectedGraph = BuildUndirectedGraph();

Console.WriteLine("Undirected Graph:");

undirectedGraph.FindShortestPath();

undirectedGraph.Display();

Console.WriteLine();

Graph directedGraph = BuildDirectedGraph();

Console.WriteLine("Directed Graph:");

directedGraph.FindShortestPath();

directedGraph.Display();

}

public static Graph BuildUndirectedGraph()

{

int[] nodes = { 0, 1, 2, 3, 4, 5 };

Graph graph = new Graph(nodes);

graph.AddEdge(0, 1, 24.0, DirectionType.Undirected);

graph.AddEdge(0, 5, 28.0, DirectionType.Undirected);

graph.AddEdge(1, 2, 11.0, DirectionType.Undirected);

graph.AddEdge(2, 3, 13.0, DirectionType.Undirected);

graph.AddEdge(3, 4, 20.0, DirectionType.Undirected);

graph.AddEdge(3, 5, 12.0, DirectionType.Undirected);

graph.AddEdge(4, 5, 15.0, DirectionType.Undirected);

return graph;

}

public static Graph BuildDirectedGraph()

{

int[] nodes = { 0, 1, 2, 3, 4, 5, 6, 7 };

Graph graph = new Graph(nodes);

graph.AddEdge(0, 1, 9.0, DirectionType.Directed);

graph.AddEdge(0, 5, 14.0, DirectionType.Directed);

graph.AddEdge(0, 6, 15.0, DirectionType.Directed);

graph.AddEdge(1, 2, 24.0, DirectionType.Directed);

graph.AddEdge(2, 4, 2.0, DirectionType.Directed);

graph.AddEdge(2, 7, 19.0, DirectionType.Directed);

graph.AddEdge(3, 7, 6.0, DirectionType.Directed);

graph.AddEdge(3, 2, 6.0, DirectionType.Directed);

graph.AddEdge(4, 3, 11.0, DirectionType.Directed);

graph.AddEdge(4, 7, 16.0, DirectionType.Directed);

graph.AddEdge(5, 4, 30.0, DirectionType.Directed);

graph.AddEdge(5, 6, 5.0, DirectionType.Directed);

graph.AddEdge(5, 2, 18.0, DirectionType.Directed);

graph.AddEdge(6, 4, 20.0, DirectionType.Directed);

graph.AddEdge(6, 7, 44.0, DirectionType.Directed);

return graph;

}

}

U main dijelu programa smo koristili instance za stvaranje klase graph i konstruirali bridove. Nakon toga smo pronašli najkraći put te time ispitali funkcionalnost.