ŽILINSKÁ UNIVERZITA V ŽILINE

Fakulta riadenia a informatiky

Semestrálna práca AaUŠ 2

Michael Holý

Systém pre evidovanie nehnuteľností a parciel

Cvičiaci : Ing. Peter Jankovič, Phd.

Žilina, 2024

Obsah

1 Zadanie semestrálnej práce

Cieľom semestrálnej práce bolo vytvoriť demonštračnú verziu softvéru informačného systému na evidenciu nehnuteľností a parciel, pričom systém mal byť implementovaný s dôrazom na rýchlosť, spoľahlivosť a nízku pamäťovú náročnosť.

V rámci práce bolo potrebné:

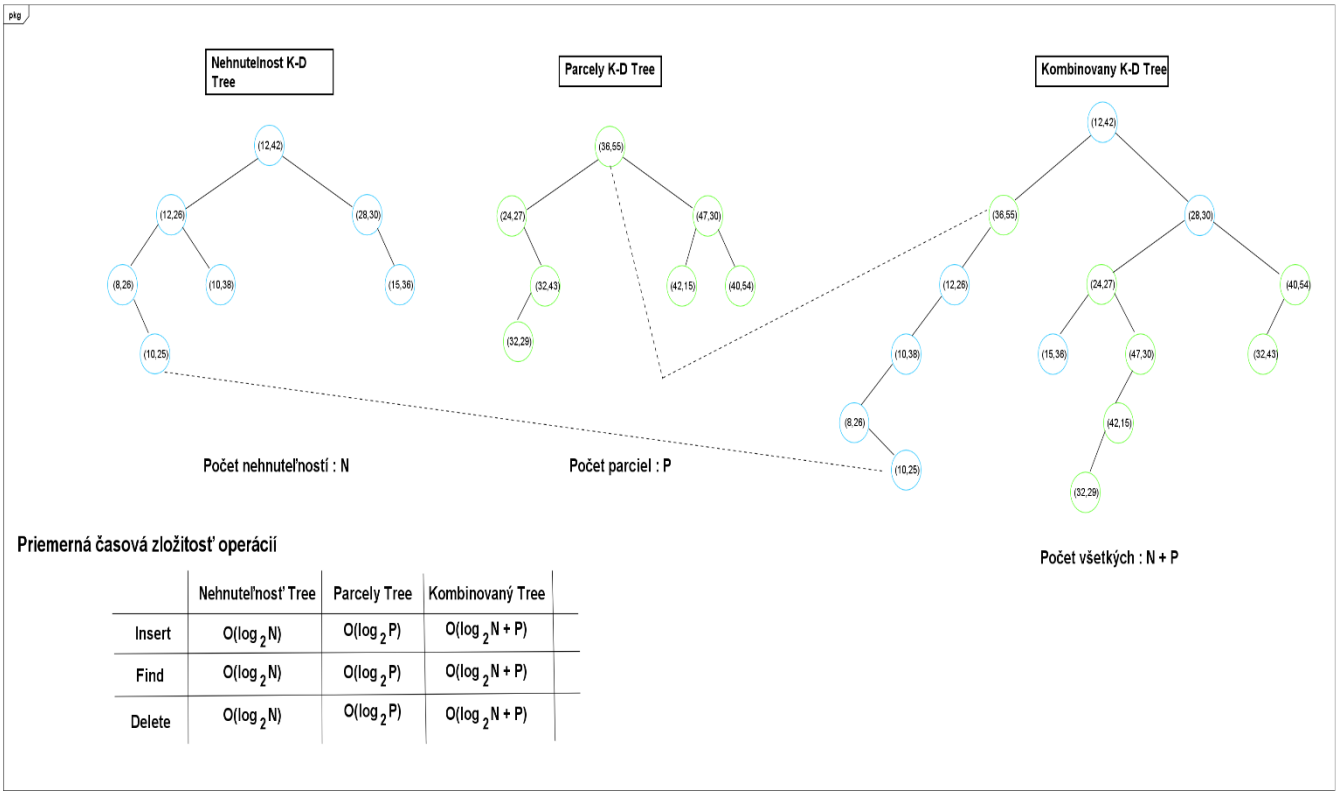
- Implementovať všeobecný K-D strom, ktorý efektívne podporuje vyhľadávanie podľa GPS súradníc a v prípade možnosti zvláda aj duplicitné kľúče, čo zaisťuje rýchly prístup k potrebným údajom.
- Vytvoriť aplikáciu umožňujúcu pridávanie, vyhľadávanie, úpravu a vymazávanie parciel a nehnuteľností v závislosti od GPS súradníc, ako aj evidenciu vzťahov medzi nehnuteľnosťami a parcelami na základe spoločných bodov.
- Zabezpečiť uloženie dát v CSV formáte a ich jednoduché načítanie, pričom dôležitým aspektom bola minimalizácia veľkosti súborov a efektívna správa pamäte.

Implementácia systému zahŕňala návrh dátovej štruktúry s dôrazom na optimalizáciu zložitosti jednotlivých operácií a vyhýbanie sa rekurzii.

2 Návrh systému a údajových štruktúr

Pre optimalizáciu operácií v systéme bolo potrebné navrhnúť tri samostatné K-D stromy: jeden na uchovávanie údajov o nehnuteľnostiach, druhý pre údaje o parcelách a tretí zmiešaný strom, ktorý obsahuje oba typy objektov.

Táto štruktúra umožňuje efektívne vykonávať vyhľadávacie a manipulačné operácie nad každým typom objektu, čím sa znižuje časová náročnosť pri spracovávaní dát, keďže v prípade operácií iba nad jedným typom objektov sa spúšťajú operácie na strome prislúchajúce daným objektom.

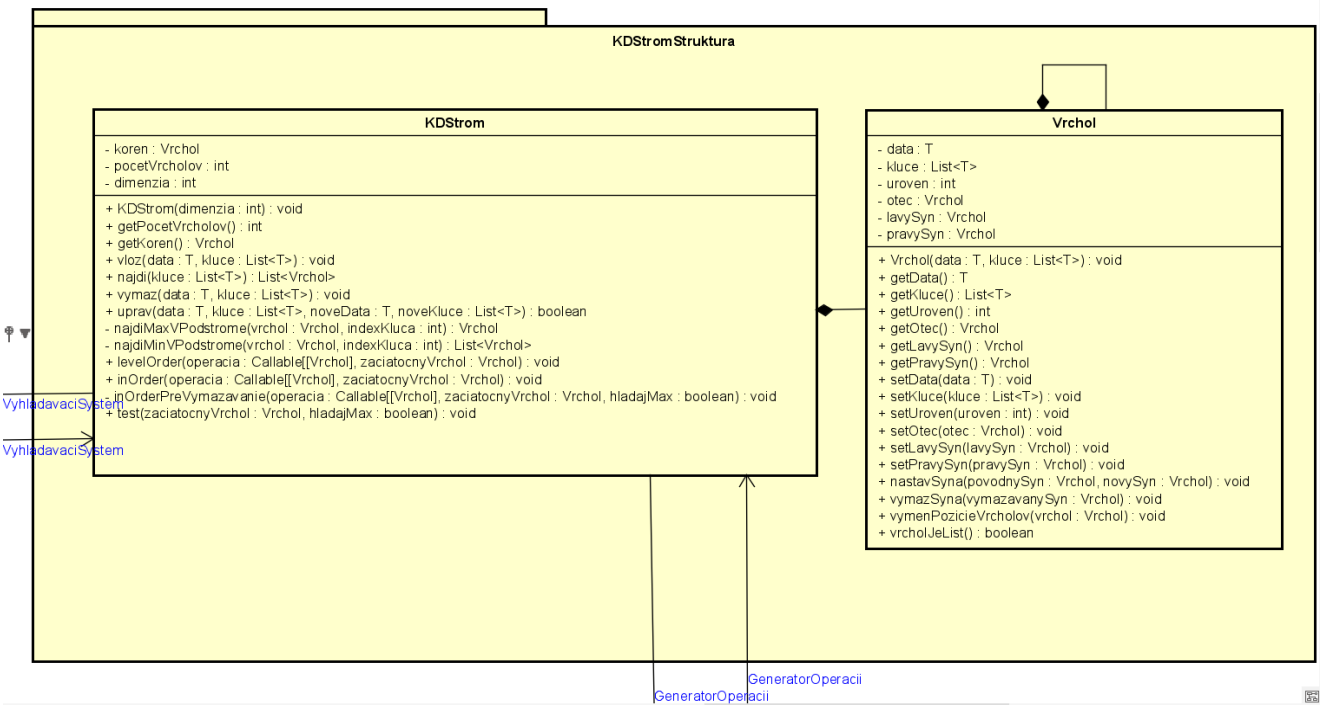


Priemerná časová zložitosť operácií

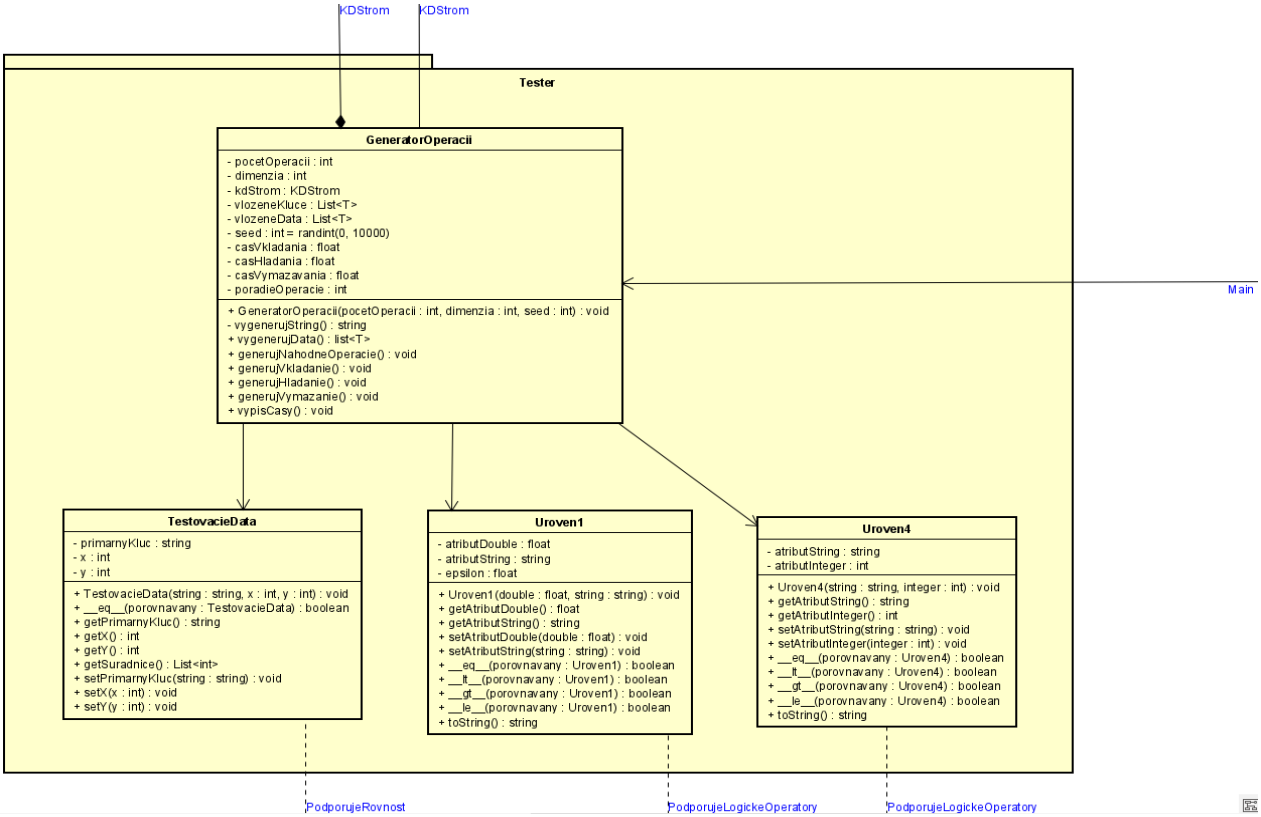
	Nehnuteľnosť Tree	Parcely Tree	Kombinovaný Tree
Insert	$O(\log_2 N)$	$O(\log_2 P)$	$O(\log_2 N + P)$
Find	$O(\log_2 N)$	$O(\log_2 P)$	$O(\log_2 N + P)$
Delete	$O(\log_2 N)$	$O(\log_2 P)$	$O(\log_2 N + P)$

3 Návrh UML diagramu tried

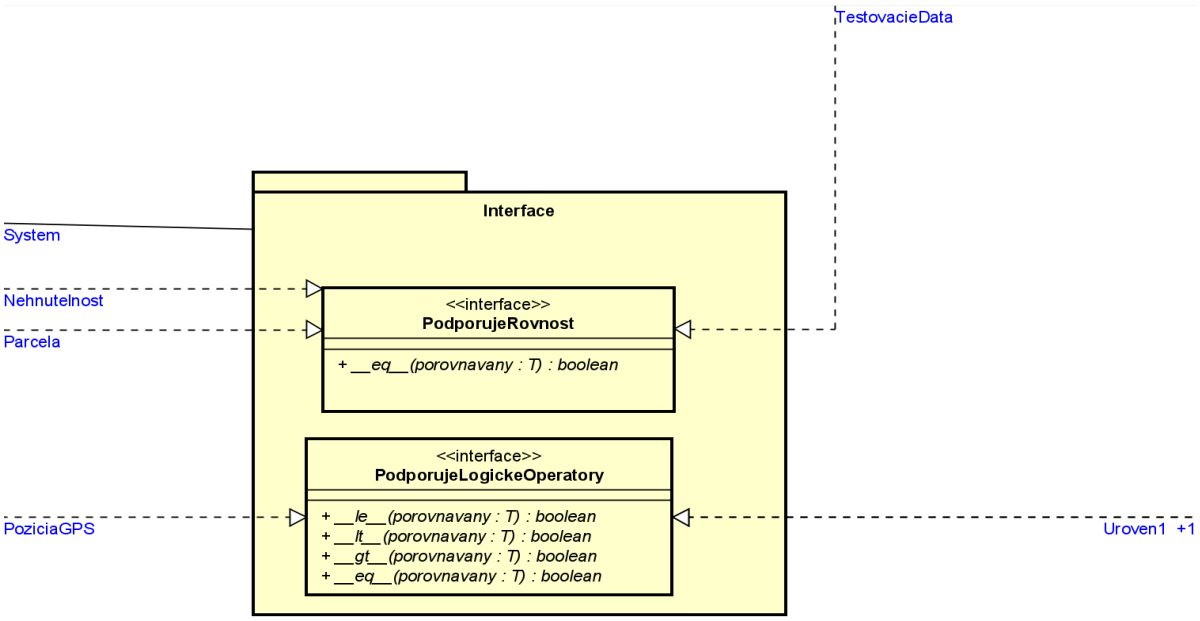
3.1 Balíček štruktúry K-D Stromu



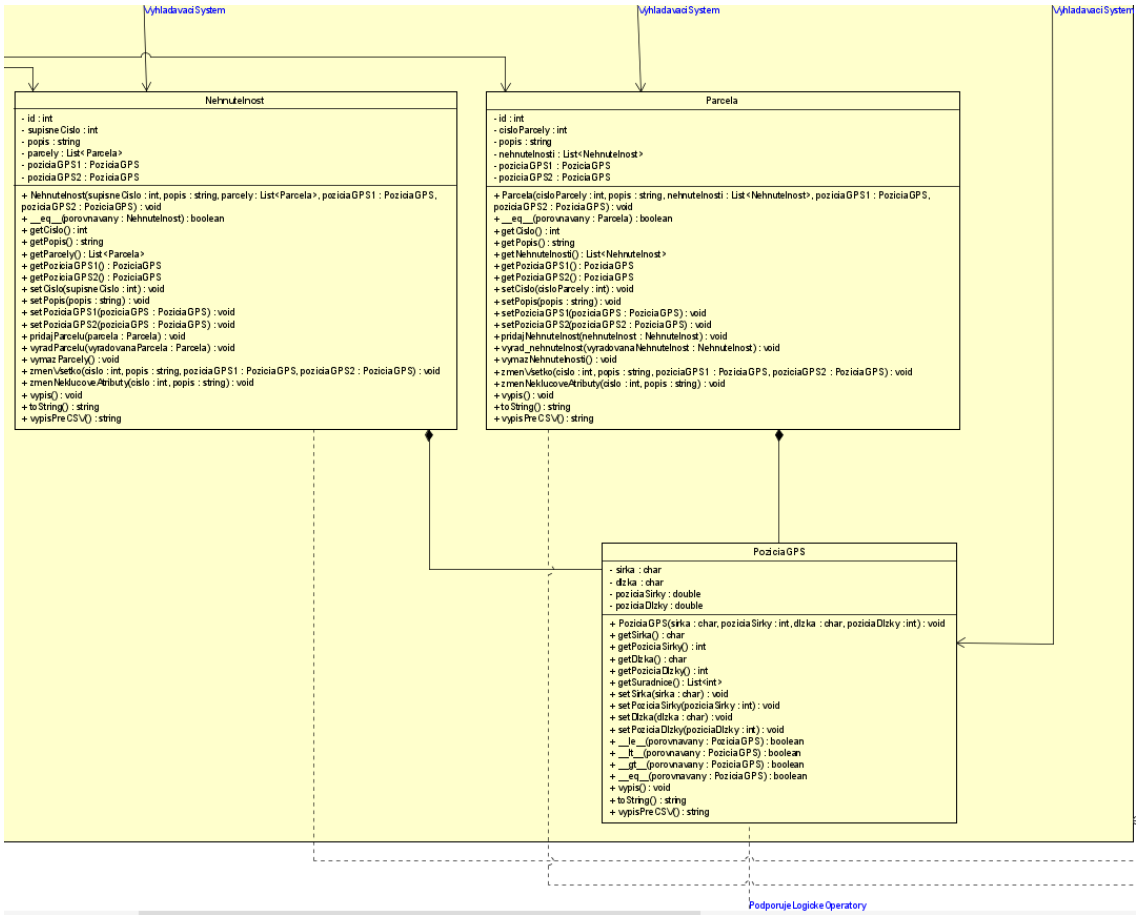
3.2 Balíček testera údajovej štruktúry



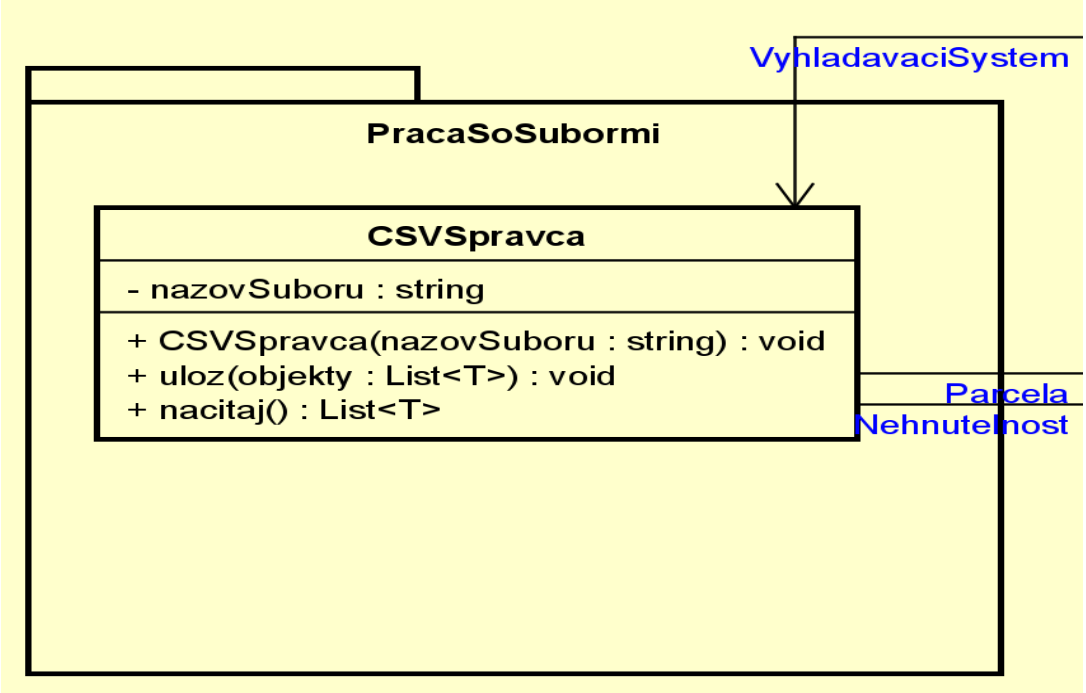
3.3 Balíček interfacov



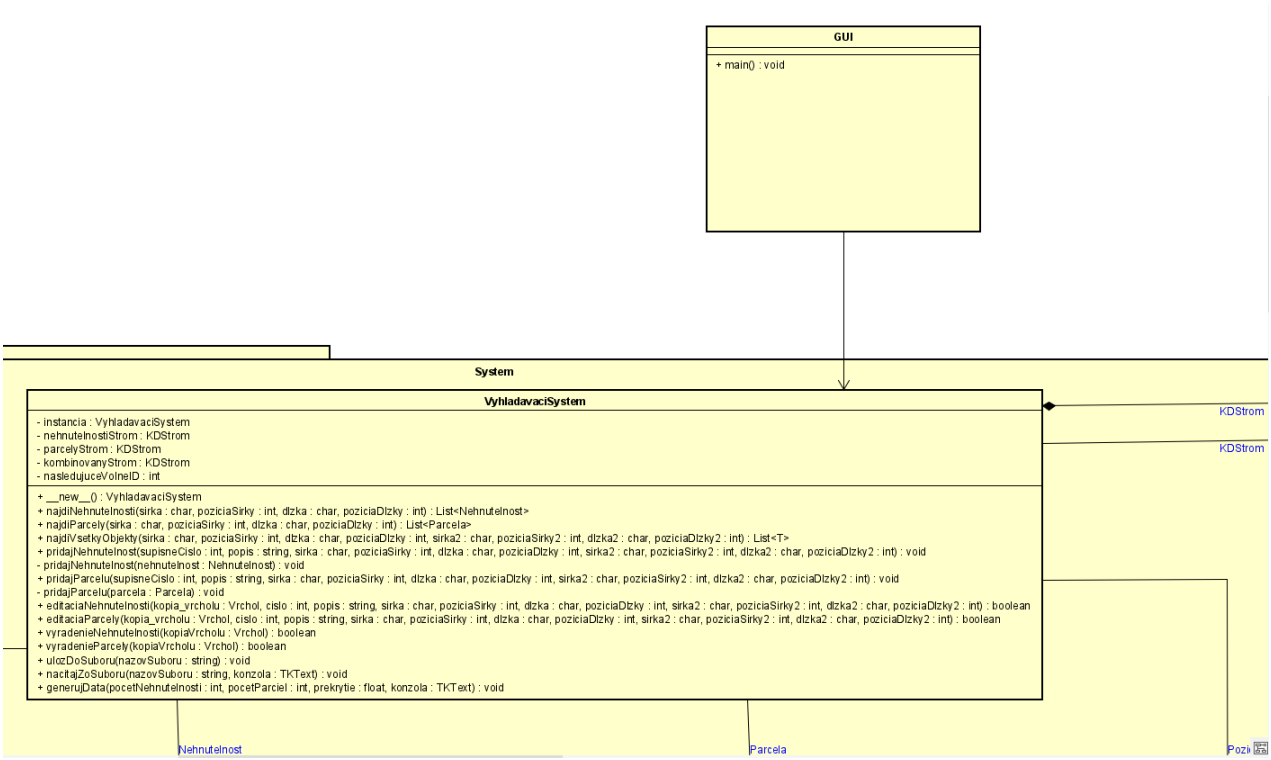
3.4 Balíček systém – triedy Nehnuteľnosť, Parcela a PoziciaGPS



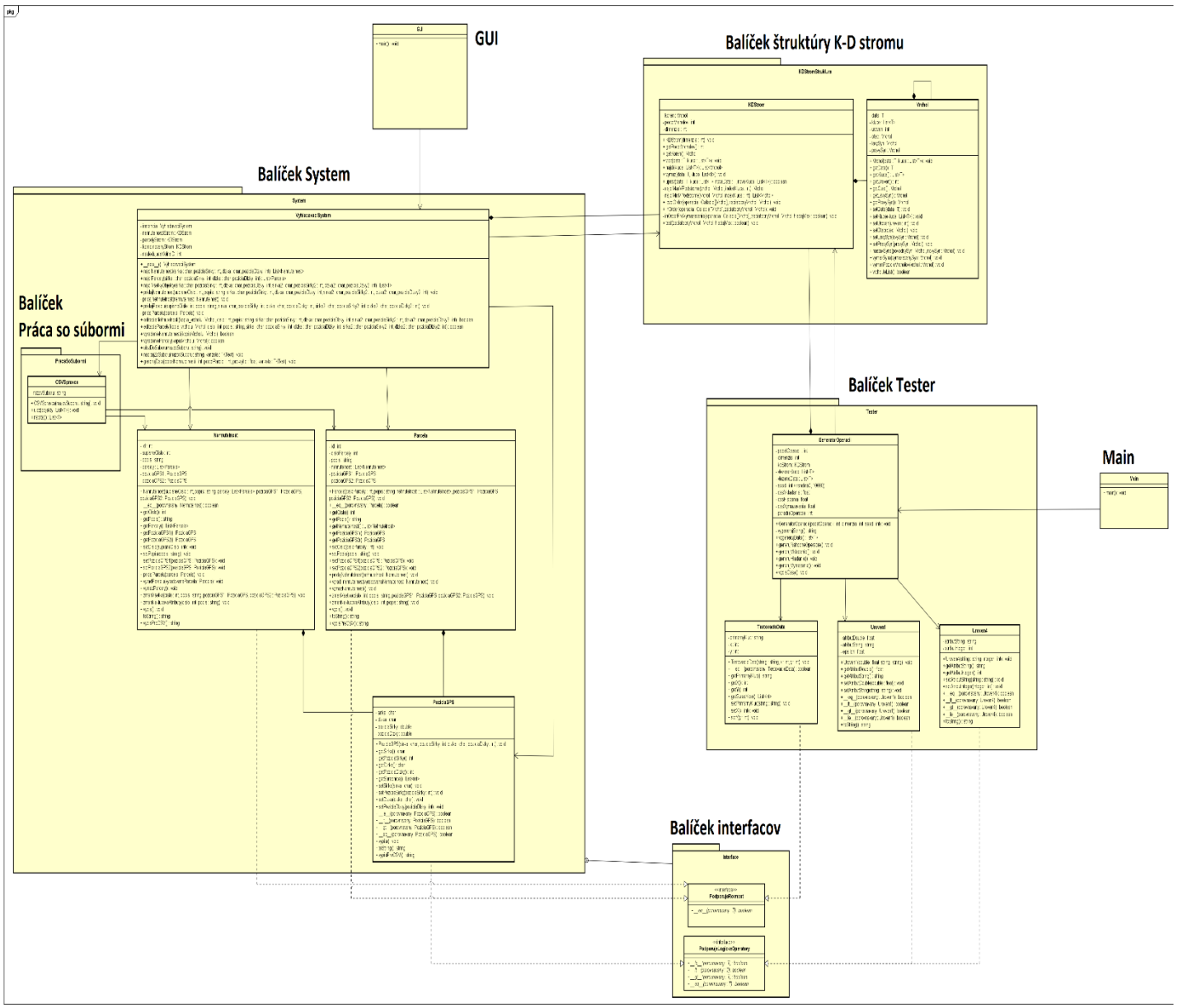
3.5 Balíček práca so súbormi



3.6 Balíček systém – trieda VyhladavaciSystem



3.7 Celkový UML diagram



4 Rozbor návrhu štruktúry programu

4.1 Balíček štruktúry K-D Stromu

Tento balíček reprezentuje dátovú štruktúru K-D stromu a všetky komponenty potrebné pre jeho fungovanie. Balíček slúži na efektívne usporiadanie dát, pričom bolo cieľom implementácie, aby implementovaný K-D Strom umožňoval ukladať ľubovoľné typy dát a kľúčov. Keďže bola práca programovaná v programovacom jazyku Python, jazyk nám umožňoval už defaultne ukladať do štruktúry ľubovoľné typy dát, no pre kompatibilitu s operáciami štruktúry bolo zadefinované pravidlo, že všetky typy ukladaných dát a kľúčov v štruktúre musia implementovať interface PodporujeRovnosť v prípade ukladaných dát a interface PodporujeLogickeOperatory v prípade ukladaných kľúčov.

Tento balíček obsahuje triedy:

- KDStrom – implementácia údajovej štruktúry.
- Vrchol – trieda predstavujúca jednotlivé uzly v K-D Strome.

4.2 Balíček interfacov

Tento balíček definuje základné funkcie, ktoré musia implementovať všetky typy dát a kľúčov vkladané do vrcholov K-D stromu. Tento návrh zaručuje jednotné spracovanie údajov a podporuje základné operácie nevyhnutné pre fungovanie K-D stromu. Programovanie bolo realizované v programovacom jazyku Python, kde interfacy nie sú podporované, preto sa jedná iba o pseudointerfacy, ktoré sú implementované pomocou dedičnosti a abstraktných metód.

Tento balíček obsahuje interfacy:

- PodporujeRovnost – operátor ==.
- PodporujeLogickeOperatory – operátory <=, <, > ,==.

4.3 Balíček testera údajovej štruktúry

Tento balíček obsahuje komponenty pre testovanie údajovej štruktúry K-D Stromu, pričom umožňuje voľbu medzi testovaním štruktúry pre dimenziu kľúčov 2 alebo 4 jednoduchou zmenou parametra v konštruktore generátora operácií.

Tento balíček obsahuje triedy:

- GeneratorOperacii – táto trieda umožňuje generovať sadu operácií podľa zadefinovaného počtu týchto operácií, najprv generátor vygeneruje vkladanie pre 20000 náhodných prvkov jedinečných dát a náhodných alebo duplicitných kľúčov a následne generuje náhodné operácie vkladania, hľadania alebo vymazávania zo štruktúry podľa zadaného počtu operácií, pričom po každej z týchto operácií vykonáva celkovú prehliadku stromu pre kontrolu správnosti operácie.
- TestovacieData – Dátová trieda určená pre testovanie K-D stromu s dimenziou 2.
- Uroven1 a Uroven4 – Dátové triedy určené pre testovanie K-D stromu s dimenziou 4.

4.4 Balíček systém

Tento balíček predstavuje hlavné rozhranie pre správu a manipuláciu s údajmi o nehnuteľnostiach a parcelách prostredníctvom triedy VyhladavaciSystem a taktiež definuje základné údaje a metódy potrebné pre správu nehnuteľností a parciel v systéme prostredníctvom tried Nehnutelnost, Parcela a PoziciaGPS.

Okrem toho obsahuje aj balíček Práca so súbormi, čím zabezpečuje možnosť exportu a importu dát v systéme prostredníctvom súborov CSV.

Tento balíček obsahuje triedy:

- VyhladavaciSystem – táto trieda obsahuje základné metódy na vyhľadávanie, pridávanie, editáciu a mazanie nehnuteľností a parciel podľa GPS pozícií. Taktiež zahŕňa aj funkcie na generovanie údajov do systému, vytváranie CSV výstupov a spätné načítanie údajov z CSV súborov do systému.

- Nehnutelnost - Táto trieda reprezentuje nehnuteľnosť evidovanú v systéme.
- Parcela – Táto trieda reprezentuje parcelu evidovanú v systéme.
- PoziciaGPS - Táto trieda poskytuje metódy pre prácu so súradnicami a porovnávanie GPS bodov.

4.5 Balíček práca so súbormi

Tento balíček obsahuje triedu CSVSpravca, ktorá sa zameriava na prácu s CSV súbormi. Táto trieda umožňuje uloženie a načítanie údajov o nehnuteľnostiach a parcelách do/z CSV formátu.

4.6 GUI

Trieda GUI slúži ako používateľské rozhranie pre využívanie vyhľadávacieho systému. Obsahuje prvky používateľského rozhrania, ako sú tlačidlá, textové polia, menu a podobne.

Umožňuje používateľovi prístup k operáciám definovaným v triede VyhladavaciSystem ako napríklad vyhľadávanie, pridávanie, editovanie a odstraňovanie nehnuteľností a parciel na základe zadaných GPS pozícií.

Vstupy od používateľa odovzdáva ďalej do systému a na základe výstupov systému zobrazuje používateľovi výsledky operácií prostredníctvom konzoly.

Aby sa zabezpečilo, že používateľ nebude mať priamy prístup k údajom uloženým v K-D strome, a tým pádom nemohol priamo modifikovať originálne dáta uložené v systéme, pracuje táto trieda výlučne s hlbokými kópiami vrcholov čím sa zabezpečí práca s identickými dátami a zároveň sa zabráni úmyselným či neúmyselným zmenám v pôvodných dátach štruktúry.

5 Rozbor implementácie K-D Stromu

Ako už bolo spomenuté vyššie, pre kompatibilitu s operáciami štruktúry bolo zadefinované pravidlo, že všetky typy ukladaných dát a kľúčov v štruktúre musia implementovať interface PodporujeRovnosť a interface PodporujeLogickeOperatory.

Toto je zabezpečené priamo v triede Vrchol, pri konštrukcii objektu, kde sa vykonávajú kontroly, či vkladané dáta a kľúče sú inštanciou týchto pseudo interfacov, čiže z nich dedia.

```
class Vrchol:

    def __init__(self, data, kluce):
        """
        Inicializuje vrchol s danými dátami a kľúčmi.
        """

        #ak sú dáta napr. samostatny objekt a nie su iterovatelne
        if not isinstance(data, Iterable):

            if type(data) not in (int, float, bool, str, bytes, tuple, list, dict, set, frozenset, complex):
                if not isinstance(data, PodporujeRovnost):
                    raise TypeError('Data musia podporovat logicky operator ==')
        else:
        #ak su iterovatelne, znamena to ze ich mozeme prechadzat napr. ako zoznam
            for objekt in data:
                if type(objekt) not in (int, float, bool, str, bytes, tuple, list, dict, set, frozenset, complex):
                    if not isinstance(objekt, PodporujeRovnost):
                        raise TypeError('Data musia podporovat logicky operator ==')

        if not isinstance(kluce, Iterable):
            raise TypeError('Kluce musia byt iterovatelne')
        else:
            for kluc in kluce:
                if type(kluc) not in (int, float, bool, str, bytes, tuple, list, dict, set, frozenset, complex):
                    if not isinstance(kluc, PodporujeLogickeOperatory):
                        raise TypeError('Kluce musia podporovat logicke operatory <=, <, > a ==')
```

5.1 Implementácie operácie vloženia

Implementácia operácie vloženia spočíva v úvodných kontrolách toho, či je strom prázdny alebo už obsahuje nejaké vrcholy pomocou interného atribútu pocetVrcholov.

V prípade, že strom ešte neobsahuje žiadne vrcholy, sa vkladané dáta a kľúče vložia do koreňa stromu, v opačnom prípade sa spustí cyklus, ktorý hľadá správne umiestnenie nového vrcholu na základe jeho kľúčov a prehľadávať sa začne od koreňa stromu.

Na začiatku cyklu sa vždy vypočíta, podľa ktorého kľúča sa má porovnávať s nasledujúcim vrcholom. Toto je vypočítané pomocou vnútorného atribútu aktuálnej úrovne vrcholu a dimenzia daného K-D stromu.

```
def vloz(self, data, kluce):
    """
    Vloží nový vrchol do stromu s danými dátami a kľúčmi.
    """

    if self.__koren is None:
        self.__koren = Vrchol(data, kluce)
        self.__pocet_vrcholov += 1
    else:
        novyVrchol = Vrchol(data, kluce)
        aktualny_vrchol = self.__koren
        aktualna_uroven = 0


        while True:
            index_kluca = aktualna_uroven % self.__dimenzia
            porovnavany_kluc = kluce[index_kluca]
            kluc_aktualneho_vrcholu = aktualny_vrchol.get_kluce()[index_kluca]
```

Na základe daného kľúča sa potom rozhoduje či sa má umiestnenie nového vrcholu ďalej hľadať v ľavom alebo pravom pod strome aktuálneho vrcholu.

V tomto prípade je **podpora duplicít** zabezpečená už spomenutými logickými operátormi na porovnávanie ukladaných kľúčov.

V prípade, že je kľúč vkladaného vrcholu, podľa ktorého sa na danej úrovni porovnáva väčší ako kľúč aktuálneho vrcholu, hľadanie umiestnenie pokračuje ďalej v pravom pod strome.

V prípade, že je kľúč vkladaného vrcholu menší alebo rovný, hľadanie umiestnenia pokračuje v ľavom pod strome, čím sa zabezpečuje, že všetky duplicitné kľúče na konkrétnej úrovni budú ukladané v ľavom pod strome.

Tento cyklus hľadania sa ukončí, keď sa v danom pod strome nájde nasledujúci vrchol (ľavý alebo pravý syn v závislosti od toho, do ktorého pod stromu pokračuje hľadanie), ktorý je prázdny, v takom prípade sa na dané miesto vloží nový vrchol a cyklus sa ukončí, inak sa do aktuálneho vrcholu načíta nasledujúci vrchol, zvýši sa aktuálna úroveň a cyklus sa opakuje.

```
class KDStrom:
    def vloz(self, data, kluce):
        while True:
            index_kluca = aktualna_uroven % self.__dimenzia
            porovnavany_kluc = kluce[index_kluca]
            kluc_aktualneho_vrcholu = aktualny_vrchol.get_kluce()[index_kluca]
            lavy = False

            if isinstance(porovnavany_kluc, float) and isinstance(kluc_aktualneho_vrcholu, float):
                if porovnavany_kluc < kluc_aktualneho_vrcholu or abs(porovnavany_kluc - kluc_aktualneho_vrcholu) < 1e-7:
                    nasledujuci_vrchol = aktualny_vrchol.get_lavy_syn()
                    lavy = True
                else:
                    nasledujuci_vrchol = aktualny_vrchol.get_pravy_syn()

            else:
                if porovnavany_kluc <= kluc_aktualneho_vrcholu: #pokial su porovnavane kluce rovnake, prvok sa vlozi do laveho podstromu
                    nasledujuci_vrchol = aktualny_vrchol.get_lavy_syn()
                    lavy = True
                else:
                    nasledujuci_vrchol = aktualny_vrchol.get_pravy_syn()

            if nasledujuci_vrchol is None and lavy:
                novyVrchol.set_uroven(aktualna_uroven + 1)
                novyVrchol.set_otec(aktualny_vrchol)
                aktualny_vrchol.set_lavy_syn(novyVrchol)
                self.__pocet_vrcholov += 1
                break
            elif nasledujuci_vrchol is None and not lavy:
                novyVrchol.set_uroven(aktualna_uroven + 1)
                novyVrchol.set_otec(aktualny_vrchol)
                aktualny_vrchol.set_pravy_syn(novyVrchol)
                self.__pocet_vrcholov += 1
                break

            aktualny_vrchol = nasledujuci_vrchol
            aktualna_uroven += 1
```

5.2 Implementácia operácie hľadania

Implementácia operácie hľadania taktiež spočíva v úvodných kontrolách, či je strom prázdny a v prípade že je, vráti prázdny zoznam, keďže v strome nie sú žiadne vrcholy na hľadanie.

Následne začne prehľadávanie stromu od koreňa prostredníctvom cyklu, v ktorom sa na začiatku porovnávajú všetky dimenzie kľúča aktuálne prehľadávaného vrcholu s vyhľadávanými kľúčmi.

V prípade, že sa všetky kľúče rovnajú, tento aktuálny vrchol sa pridá do zoznamu vrcholov, ktorý sa na konci cyklu a metódy vracia.

```
class KDStrom:
    def najdi(self, kluce):
        """
        Nájde vrcholy v strome s danými kľúčmi.
        """
        vratene_vrcholy = []
        aktualna_uroven = 0

        if self.__pocet_vrcholov == 0:
            return vratene_vrcholy

        else:
            aktualny_vrchol = self.__koren

            epsilon = 1e-7
            while True:
                kluce_sa_rovnaju = True

                for i in range(self.__dimenzia):

                    if type(kluce[i]) == float and type(aktualny_vrchol.get_kluce()[i]) == float:
                        #ak je absolutna hodnota rozdielu dvoch klucov vacsia ako epsilon - nerovnaju sa
                        if abs(kluce[i] - aktualny_vrchol.get_kluce()[i]) > epsilon:
                            kluce_sa_rovnaju = False
                            break

                    elif kluce[i] != aktualny_vrchol.get_kluce()[i]:
                        kluce_sa_rovnaju = False
                        break

                if kluce_sa_rovnaju: # vrati najdeny vrchol
                    vratene_vrcholy.append(aktualny_vrchol)
```

V prípade, že sa tieto kľúče nerovnajú, pokračuje cyklus v prehľadávaní buď ľavého alebo pravého pod stromu rovnako ako to bolo v prípade vkladania, keďže pri **podpore duplicít** môžu byť ďalšie vrcholy s duplicitnými kľúčmi uložené nižšie v pod stromoch.

Cyklus končí v prípade, že aktuálne prehľadávaný vrchol je listom a tým pádom už nemá žiadne vrcholy, ktoré by mohli nasledovať alebo že vrchol, ktorý mal nasledovať počas prehľadávania neexistuje a tým pádom už v strome, nemôžu byť ďalšie vrcholy s hodnotou kľúčov rovnakou, aká je vyhľadávaná a preto sa vráti aktuálny zoznam nájdených vrcholov.

```
class KDStrom:
    def najdi(self, kluce):
                if aktualny_vrchol.vrchol_je_list():
                    return vratene_vrcholy

                else:
                    index_kluca = aktualna_uroven % self.__dimenzia
                    porovnavany_kluc = kluce[index_kluca]

                    if isinstance(porovnavany_kluc, float) and isinstance(aktualny_vrchol.get_kluce()[index_kluca], float):
                        if porovnavany_kluc < aktualny_vrchol.get_kluce()[index_kluca] or abs(porovnavany_kluc - aktualny_vrchol.get_kluce()[index_k
                            aktualny_vrchol = aktualny_vrchol.get_lavy_syn()
                            if aktualny_vrchol is None:
                                return vratene_vrcholy

                        else:
                            aktualny_vrchol = aktualny_vrchol.get_pravy_syn()
                            if aktualny_vrchol is None:
                                return vratene_vrcholy

                    else:

                        if porovnavany_kluc <= aktualny_vrchol.get_kluce()[index_kluca]:
                            aktualny_vrchol = aktualny_vrchol.get_lavy_syn()
                            if aktualny_vrchol is None:
                                return vratene_vrcholy

                        else:
                            aktualny_vrchol = aktualny_vrchol.get_pravy_syn()
                            if aktualny_vrchol is None:
                                return vratene_vrcholy

                    aktualna_uroven += 1
```

5.3 Implementácia operácie vymaž

Implementácia operácie vymaž začína volaním už implementovanej operácie nájdi, ktorá nájde na základe vyhľadávaných kľúčov všetky vrcholy s identickými kľúčmi.

Tieto nájdené vrcholy následne porovnáva so zadanými dátami, pomocou ktorých odlíši ostatné vrcholy s duplicitnými kľúčmi práve od toho konkrétneho vrcholu, ktorý je potrebné vymazať.

```
class KDStrom:

    def vymaz(self, data, kluce):
        """
        Vymaže vrchol s danými dátami a kľúčmi zo stromu.
        """
        vymazavane_vrcholy = self.najdi(kluce)
        vymazavany_vrchol = None
        vysledok_mazania = True
        vymazavane_duplicitne_vrcholy = []
        naspat_vkladane_vrcholy = []
        vymazavanie_duplicit = False
        v_datach_je_float = False
        #prehľadáva či je v dátach dátový typ float
        if isinstance(data, Iterable) and not isinstance(data, (str, bytes)):
            for i in range(len(data)):
                if isinstance(data[i], float):
                    v_datach_je_float = True
                    break
        #pokiaľ sa v dátach nachádza float, porovnávajú sa podľa absolútnej hodnoty ich rozdielu s epsilonom
        if v_datach_je_float: ···
        #v opačnom prípade sa porovnávajú priamo dáta
        else:

            if len(vymazavane_vrcholy) == 1 and vymazavane_vrcholy[0].get_data() == data:
                vymazavany_vrchol = vymazavane_vrcholy[0]

            elif len(vymazavane_vrcholy) > 1:
                for vrchol in vymazavane_vrcholy:
                    if vrchol.get_data() == data:
                        vymazavany_vrchol = vrchol
                        break
```

Následne už máme v premennej vymazavy_vrchol uloženú referenciu na konkrétny vrchol, ktorý sa má vymazať.

Nasledujú dva cykly, v ktorých je zahrnutá logika celého vymazávania.

5.3.1 Vnútorný cyklus

Vo vnútornom cykly sa vymazavany_vrchol premiestňuje s náhradnými vrcholmi, ktoré sú nájdené pomocou pomocných metód až dokým sa vymazávaný vrchol nestane listom:

najdi_max_v_podstrome – pokiaľ existuje ľavý syn na aktuálnej pozícii vymazávaného vrcholu, nájde sa v tomto ľavom pod strome vrchol s maximálnou hodnotou kľúča v dimenzii, podľa ktorej sa na danej úrovni porovnáva

- **najdi_min_v_podstrome** – pokiaľ existuje pravý syn na aktuálnej pozícii vymazávaného vrcholu, nájdu sa v tomto pravom pod strome všetky vrcholy s minimálnou hodnotou kľúča v dimenzii, podľa ktorej sa na danej úrovni porovnáva

V prípade oboch metód sa ale prioritne vyhľadáva náhradný vrchol v ľavom pod strome pokiaľ to je možné, keďže v prípade vyhľadávania v pravom pod strome sú potrebné ďalšie operácie pre zachovanie správnej štruktúry stromu.

Po nájdení sa pozícia vymazávaného vrcholu vymení s pozíciou náhradného vrcholu.

```
class KDStrom:
    def vymaz(self, data, kluce):



        while True:
            if not vymazavane_vrcholy or vymazavany_vrchol is None:
                vysledok_mazania = False
                break

            elif len(vymazavane_duplicitne_vrcholy) > 0:
                vymazavany_vrchol = vymazavane_duplicitne_vrcholy.pop()
                if vymazavany_vrchol not in naspat_vkladane_vrcholy:
                    naspat_vkladane_vrcholy.append(vymazavany_vrchol)
            #pokial nie je listom, nahrad ho najvacsim v lavom/ najmensim v pravom podstrome na zaklade kluca Ki
            while vymazavany_vrchol.vrchol_je_list() == False:

                index_kluca = vymazavany_vrchol.get_uroven() % self.__dimenzia
                lavy_syn = vymazavany_vrchol.get_lavy_syn()
                pravy_syn = vymazavany_vrchol.get_pravy_syn()
                if lavy_syn is not None: #ak ma laveho syna, vyberie max z lava
                    nahradny_vrchol = self.__najdi_max_v_podstrome(lavy_syn, index_kluca)
                    vymazavany_vrchol.vymen_pozicie_vrcholov(nahradny_vrchol)

                elif pravy_syn is not None: #ak ma len praveho syna, vyberie min z prava
                    vymazavane_duplicitne_vrcholy += self.__najdi_min_v_podstrome(pravy_syn, index_kluca)


                    if len(vymazavane_duplicitne_vrcholy) > 1:
                        vymazavanie_duplicit = True
                    nahradny_vrchol = vymazavane_duplicitne_vrcholy.pop()
                    vymazavany_vrchol.vymen_pozicie_vrcholov(nahradny_vrchol)
```

Metódy najdi_max / najdi_min v pod strome

V prípade oboch metód sa na vrchole, nad ktorým bola spustená táto metóda zavolá upravená **in order prehliadka pre vymazávanie**, ktorá na základe toho, či sa hľadá maximum alebo minimum naplní do pomocného listu vrcholy, s potenciálnym minimom alebo maximom.

Následne sa naplnené listy prehľadávajú a v prípade maxima, sa vráti jeden konkrétny vrchol, u ktorého bola daná hodnota maxima zaznamenaná ako prvá, v prípade minima sa najprv prejde celý pomocný list, nájde sa hodnota minima a následne je prehľadaný znovu pričom sa vrátia všetky vrcholy s touto minimálnou hodnotou.

```
class KDStrom:
    def __najdi_max_v_podstrome(self, vrchol, index_kluca):
        """
        Nájde vrchol s maximálnou hodnotou kľúča v ľavom podstrome.
        """
        aktualny_vrchol = vrchol
        if aktualny_vrchol is None:
            return None
        else:
            vybrane_vrcholy = []
            self.__in_order_pre_vymazavanie(vybrane_vrcholy.append, aktualny_vrchol, True)
            max_vrchol = vybrane_vrcholy[0]

            for vrchol in vybrane_vrcholy:
                if vrchol.get_kluce()[index_kluca] > max_vrchol.get_kluce()[index_kluca]:
                    max_vrchol = vrchol

            return max_vrchol

    def __najdi_min_v_podstrome(self, vrchol, index_kluca):
        """
        Nájde vrcholy s minimálnou hodnotou kľúča v pravom podstrome.
        """
        aktualny_vrchol = vrchol
        if aktualny_vrchol is None:
            return None
        else:
            vybrane_vrcholy = []
            self.__in_order_pre_vymazavanie(vybrane_vrcholy.append, aktualny_vrchol)
            min_vrchol = vybrane_vrcholy[0]

            for vrchol in vybrane_vrcholy:
                if vrchol.get_kluce()[index_kluca] < min_vrchol.get_kluce()[index_kluca]:
                    min_vrchol = vrchol

            vsetky_min_vrcholy = []
            minimalna_hodnota = min_vrchol.get_kluce()[index_kluca]

            for vrchol in vybrane_vrcholy:
                if type(minimalna_hodnota) == float and type(vrchol.get_kluce()[index_kluca]) == float:
                    if abs(vrchol.get_kluce()[index_kluca] - minimalna_hodnota) < 1e-7:
                        vsetky_min_vrcholy.append(vrchol)

                else:
                    if vrchol.get_kluce()[index_kluca] == minimalna_hodnota:
                        vsetky_min_vrcholy.append(vrchol)

            return vsetky_min_vrcholy
```

Upravená in order prehliadka

Princíp tejto upravenej in order prehliadky spočíva v tom, že sa každý spracovaný vrchol ukladá do pomocného listu, ktorý využívame ako zásobník pri spätnom vyhľadávaní.

Hlavný cyklus sa opakuje, pokiaľ existuje vrchol načítaný do aktuálneho vrcholu alebo zásobník nie je prázdny a postupne sa daný strom prechádza do hĺbky.

Následne sa opakuje vnútorný cyklus, v ktorom sa zásobník napĺňa vhodnými vrcholmi podľa úrovne na ktorej sa nachádza , tento cyklus sa opakuje dokým existuje vrchol načítaný v aktuálnom vrchole.

V prípade voľby nasledujúceho vrcholu , ktorý sa pridá do zásobníka sa rozhoduje podľa toho či aktuálny vrchol je na úrovni, na ktorej sa vyhľadáva minimum/maximum a toho, čo z daných dvoch možností sa hľadá:

- **Je na hľadanej úrovni a hľadá sa maximum** – prejde sa do pravého pod stromu
- **Je na hľadanej úrovni a hľadá sa minimum –** prejde sa do ľavého pod stromu
- **Nie je na hľadanej úrovni a hľadá sa maximum** – prejde sa do ľavého pod stromu
- **Nie je na hľadanej úrovni a hľadá sa minimum –** prejde sa do pravého pod stromu

V prípade špecifickej situácie, kedy **je na hľadanej úrovni a hľadá sa maximum** a neexistuje pravý pod strom , tam sa vnútorný cyklus končí, keďže v ľavom pod strome by boli prvky iba <= ako hodnota v aktuálnom vrchole.

```
class KDStrom:
    def __in_order_pre_vymazavanie(self, operacia, zaciatocny_vrchol, hladaj_max = False):
        """
        Prejde strom v upravenom in-order poradí pre účely vymazávania a vykoná operáciu na každom vrchole.
        """

        aktualny_vrchol = zaciatocny_vrchol
        #zaciatocny_vrchol je synom vrcholu, ktory sa vymazava, cize sa min/max hlada podla urovne otca
        hladana_uroven = zaciatocny_vrchol.get_uroven() - 1
        vrcholy = []

        while aktualny_vrchol is not None or vrcholy:
            while aktualny_vrchol is not None:
                vrcholy.append(aktualny_vrchol)
        #pokial sa hlada max, prejde sa pravy podstrom, ak sme na spravnej urovni a pravy podstrom uz nie je, v lavom podstrome budu uz iba <=
                if (aktualny_vrchol.get_uroven() % self.__dimenzia ) == hladana_uroven:
                    if hladaj_max:
                        if aktualny_vrchol.get_pravy_syn() is not None:
                            aktualny_vrchol = aktualny_vrchol.get_pravy_syn()
                        else:
                            break
                    else:
                        aktualny_vrchol = aktualny_vrchol.get_lavy_syn()
                else:
                    if hladaj_max:
                        aktualny_vrchol = aktualny_vrchol.get_lavy_syn()
                    else:
                        aktualny_vrchol = aktualny_vrchol.get_pravy_syn()
```

Po ukončení vnútorného cyklu sa spätne prehľadávajú vrcholy naplnené v zásobníku.

Následne sa vykoná zadaná operácia nad daným vrcholom a potom sa rozhodne, či sa bude prehľadávať aj druhý pod strom aktuálneho vrcholu, to závisí od toho, či sa aktuálny vrchol nachádza na hľadanej úrovni:

- **Nie je na hľadanej úrovni a hľadá sa maximum** – prejde do pravého pod stromu
- **Nie je na hľadanej úrovni a hľadá sa minimum** – prejde do ľavého pod stromu
- **Je na hľadanej úrovni –** aktuálny vrchol sa nastaví na neexistujúceho, čim sa buď vyberie zo zásobníka predchádzajúci vrchol alebo sa ukončí prehliadka, pokiaľ v zásobníku už nie sú vrcholy.

```
            if vrcholy:
                aktualny_vrchol = vrcholy.pop()
                operacia(aktualny_vrchol)

#ak spätne prehladavam strom, nenachadzam sa na hladanej urovni, prejdem aj druhy podstrom, ak sa nachadzam na spravnej urovni, prejdem naspat
                if (aktualny_vrchol.get_uroven() % self.__dimenzia) != hladana_uroven:
                    if hladaj_max:
                        aktualny_vrchol = aktualny_vrchol.get_pravy_syn()
                    else:
                        aktualny_vrchol = aktualny_vrchol.get_lavy_syn()
                else:
                    aktualny_vrchol = None
            else:
                aktualny_vrchol = None
```

Už ako bolo spomenuté, po nájdení náhradného/náhradných vrcholov sa pozície vymazávaného vrcholu a náhradného vymenia pomocou metódy **vymen_pozicie_vrcholov,** priamo v triede Vrchol.

Metóda vymen_pozicie_vrcholov

V tejto metóde sa najprv všetky potrebné hodnoty a referencie náhradného vrcholu uložia do lokálnych premenných.

```
class Vrchol:
    def vymen_pozicie_vrcholov(self, vrchol):
        """
        Vymení pozície dvoch vrcholov.
        """
        povodna_uroven = vrchol.get_uroven()
        povodny_otec = vrchol.get_otec()
        povodny_lavy_syn = vrchol.get_lavy_syn()
        povodny_pravy_syn = vrchol.get_pravy_syn()
```

Následne sa náhradnému vrcholu nastaví pozícia vymazávaného vrcholu pomocou všetkých dostupných referencií na otca a synov vymazávaného vrcholu.

```
# nastavenie nahradneho vrcholu na poziciu vymazavaneho vrcholu
if self.__otec is not None:
    self.__otec.nastav_syna(self, vrchol)
vrchol.set_uroven(self.__uroven)
vrchol.set_otec(self.__otec)
if self.__lavy_syn is vrchol:
    vrchol.set_lavy_syn(self)
else:
    vrchol.set_lavy_syn(self.__lavy_syn)

if self.__pravy_syn is vrchol:
    vrchol.set_pravy_syn(self)
else:
    vrchol.set_pravy_syn(self.__pravy_syn)

if self.__lavy_syn is not None and self.__lavy_syn is not vrchol:
    self.__lavy_syn.set_otec(vrchol)
if self.__pravy_syn is not None and self.__pravy_syn is not vrchol:
    self.__pravy_syn.set_otec(vrchol)
```

Ako posledné sa nastaví pozícia vymazávaného vrcholu na pozíciu náhradného vrcholu cez hodnoty a referencie uložené v lokálnych premenných.

```
# nastavenie vymazavaneho vrcholu na poziciu nahradneho vrcholu
if povodny_otec is self:
    self.set_otec(vrchol)
else:
    self.set_otec(povodny_otec)
    if povodny_otec is not None:
        povodny_otec.nastav_syna(vrchol, self)
self.set_uroven(povodna_uroven)
self.set_lavy_syn(povodny_lavy_syn)
self.set_pravy_syn(povodny_pravy_syn)

if povodny_lavy_syn is not None:
    povodny_lavy_syn.set_otec(self)
if povodny_pravy_syn is not None:
    povodny_pravy_syn.set_otec(self)
```

Po výmene vymazávaného vrcholu s vrcholom z ľavého pod stromu sa nič dodatočné neudeje, cyklus sa opakuje dokým vymazávaný vrchol nie je listom, v prípade ale, že sa vrchol vymieňal s náhradným vrcholom z pravého pod stromu sú stále v liste vymazavane_duplicitne_vrcholy ešte zvyšné vrcholy, ktoré bude potrebné po mazaní dodatočne zmazať a znovu vložiť do stromu aby sa zachovala jeho správna štruktúra.

Cyklus každopádne ďalej pokračuje kým sa vymazávaný vrchol nestane listom a následne sa cyklus ukončí a daný vrchol sa vymaže – čo znamená, že sa na neho odstránia všetky referencie.

```
            elif pravy_syn is not None: #ak ma len praveho syna, vyberie min z prava
                vymazavane_duplicitne_vrcholy += self.__najdi_min_v_podstrome(pravy_syn, index_kluca)


                if len(vymazavane_duplicitne_vrcholy) > 1:
                    vymazavanie_duplicit = True
                nahradny_vrchol = vymazavane_duplicitne_vrcholy.pop()
                vymazavany_vrchol.vymen_pozicie_vrcholov(nahradny_vrchol)

            if vymazavany_vrchol is self.__koren:
                self.__koren = nahradny_vrchol

        if self.__pocet_vrcholov == 1:
            self.__koren = None
        else:
            otec = vymazavany_vrchol.get_otec()
            if otec is not None:
                otec.vymaz_syna(vymazavany_vrchol)
                vymazavany_vrchol.set_otec(None)
```

5.3.2 Vonkajší cyklus

Po vymazaní vrcholu sa vonkajší cyklus automaticky ukončí, pokiaľ sa pri prehľadávaní pravého pod stromu počas behu vnútorného cyklu nenašlo viac ako 1 vrcholov s minimálnou hodnotou.

Ak sa ale takýchto vrcholov našlo viac, vonkajší cyklus celú logiku vymazávania spúšťa znovu a ako ďalší vymazávaný vrchol nastaví posledný vrchol z listu obsahujúci všetky vrcholy nájdené z pravého pod stromu. Tento ďalší vymazávaný vrchol sa taktiež pridá do listu, ktorý obsahuje vrcholy, ktoré je potrebné na konci vymazávania vložiť znovu do stromu.

```
while True:
    if not vymazavane_vrcholy or vymazavany_vrchol is None:
        vysledok_mazania = False
        break

    elif len(vymazavane_duplicitne_vrcholy) > 0:
        vymazavany_vrchol = vymazavane_duplicitne_vrcholy.pop()
        if vymazavany_vrchol not in naspat_vkladane_vrcholy:
            naspat_vkladane_vrcholy.append(vymazavany_vrchol)
    #pokial nie je listom, nahrad ho najvacsim v lavom/ najmensim v pravom podstrome na zaklade kluca Ki
    while vymazavany_vrchol.vrchol_je_list() == False:
```

Následne sa opakuje celý proces vymazávania znovu, dokým sa vymazávaný vrchol opäť nestane listom.

V prípade, že sa konkrétny vrchol načíta do duplicitných vrcholov viackrát (napríklad v prípadoch, kedy je na spodku stromu a jeho hodnota minima sa nájde tým pádom viackrát počas vyberania náhradných vrcholov z pravého pod stromu) sa na konci procesu vymazávania nezníži aktuálny počet vrcholov uložený v strome, keďže tento vrchol bude ešte raz určite vymazávaný a tento počet sa zníži až vtedy, keď sa vymaže posledná duplicita tohto vrcholu.

```
#ak sa nachadza v duplicitach, bude urcite este raz vymazany, pocet sa znizi az po vymazani poslednej duplicity daneho vrchola
if vymazavany_vrchol not in vymazavane_duplicitne_vrcholy:
    self.__pocet_vrcholov -= 1
```

Po opätovnom vymazaní všetkých duplicitných minimálnych vrcholov sa z vyššie spomenutého listu vyberú vrcholy a opäť sa vložia do stromu a na konci metódy sa vráti hodnota boolean podľa toho, či vymazávanie prebehlo úspešne alebo nie.

```
            if vymazavanie_duplicit == True and len(vymazavane_duplicitne_vrcholy) == 0:
                for vrchol in naspat_vkladane_vrcholy:
                    vkladany = Vrchol(vrchol.get_data(), vrchol.get_kluce())
                    self.vloz(vkladany.get_data(), vkladany.get_kluce())
                break
            elif vymazavanie_duplicit == False:
                break

        return vysledok_mazania
```

5.4 Implementácia operácie uprav

Implementácia operácie uprav spočíva v tom, že sa najprv porovnajú staré kľúče aktuálne editovaného objektu so zadanými „novými“ kľúčmi.

Podobne sa skontrolujú aj staré a „nové“ dáta a na základe výsledkov týchto porovnaní sa načítajú hodnoty do boolean premenných pre úpravu kľúčových alebo nekľúčových atribútov.

```
class KDStrom:

    def uprav(self, data, kluce, nove_data, nove_kluce):
        """
        Upraví vrchol s danými dátami a kľúčmi na základe zadaných nových dát a kľúčov.
        """
        vysledok_upravy = False
        uprava_klucovych_atributov = False
        uprava_neklucovych_atributov = False

        for i in range (self.__dimenzia):
            if isinstance(kluce[i], float) and isinstance(nove_kluce[i], float):
                if abs(kluce[i] - nove_kluce[i]) > 1e-7:
                    uprava_klucovych_atributov = True
                    break
            else:
                if kluce[i] != nove_kluce[i]:
                    uprava_klucovych_atributov = True
                    break

        if data != nove_data:
            uprava_neklucovych_atributov = True
```

Pokiaľ sa majú meniť kľúčové atribúty, je potrebné daný objekt so starými kľúčmi najprv zo stromu odstrániť a pokiaľ toto prebehlo úspešne, opäť sa do stromu vloží s novými dátami aj kľúčmi.

V prípade ale, že sa majú meniť iba nekľúčové atribúty, daný vrchol sa vyhľadá v strome pomocou operácie nájdi a porovnaním dát s dátami nájdených vrcholov.

Tomuto nájdenému vrcholu sa už iba nastaví nová hodnota dát.

Na konci metódy sa ešte vráti boolean hodnota signalizujúca či editácia prebehla úspešne alebo neúspešne.

```
if uprava_klucovych_atributov:
    vysledok_vymazania = self.vymaz(data, kluce)
    if vysledok_vymazania:
        self.vloz(nove_data, nove_kluce)
        vysledok_upravy = True

elif uprava_neklucovych_atributov:
    najdene_vrcholy = self.najdi(kluce)
    if najdene_vrcholy:
        for vrchol in najdene_vrcholy:
            if vrchol.get_data() == data:
                vrchol.set_data(nove_data)
                vysledok_upravy = True
                break

return vysledok_upravy
```

6 Vyčíslenie zložitosti jednotlivých operácií v systéme

Pri vyčíslení zložitostí pre jednotlivé operácie vyhľadávacieho systému budeme vychádzať z tejto tabuľky.

Operácia	Nehnuteľnosti Tree	Parcely Tree	Kombinovaný Tree
Insert(vlož)	O (log2 N)	O (log2 P)	O (log2 N + P)
Find(nájdi)	O (log2 N)	O (log2 P)	O (log2 N + P)
Delete(vymaž)	O (log2 N)	O (log2 P)	O (log2 N + P)

Pričom „N“ reprezentuje počet nehnutelností v prislúchajúcom strome a „P“ počet parciel v prislúchajúcom strome.

6.1 Nájdi nehnuteľnosti

„Vyhľadanie nehnuteľností – podľa zadanej GPS pozície sa nájdu všetky nehnuteľnosti, ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS“

V tejto operácii sa volá 1x operácia nájdi zo stromu nehnuteľností podľa zadanej súradnice a následne sa vracia hlboká kópia listu vráteného touto operáciou.

Priemerná časová zložitosť = O ($\log_2$ N)

6.2 Nájdi parcely

„Vyhľadanie parciel – podľa zadanej GPS pozície sa nájdu všetky parcely, ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS.“

Rovnaký postup ako pri hľadaní nehnuteľností , len na strome s parcelami.

Priemerná časová zložitosť = O ($\log_2$ P)

6.3 Vyhľadanie všetkých objektov

„Vyhľadanie všetkých objektov – podľa dvoch zadaných GPS pozícií (tieto definujú obdĺžnik) sa nájdu všetky evidované parcely aj nehnuteľnosti, ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS.“

V tejto operácii je potrebné vyhľadať všetky objekty v kombinovanom strome najprv raz s prvou súradnicou a potom druhýkrát s druhou, v prípade, že sa daný objekt nachádza súčasne na oboch zadaných pozíciách by sa našiel znovu, preto je potrebné takéto objekty vyfiltrovať.

Priemerná časová zložitosť = $O\ 2.(\log_2 N + P)$

6.4 Pridanie nehnuteľnosti

„Pridanie nehnuteľnosti – na základe vstupných údajov (súpisné číslo, popis, zoznam pozícií GPS ohraničujúcich nehnuteľnosť) sa pridá nehnuteľnosť do evidencie. Zoznam referencií na parcely, na ktorých stojí naplní systém automaticky“

V tejto operácii je potrebné najprv vložiť nehnuteľnosť 2x do stromu s nehnuteľnosťami (najprv s jednou súradnicou, potom s druhou) a tak isto 2x aj do stromu so všetkými objektami. Následne je pre vybudovanie referencií pre prekryté objekty potrebné vyhľadanie v strome s parcelami všetky parcely, ktoré sa nachádzajú na prvej alebo druhej súradnici pridanej nehnuteľnosti.

Priemerná zložitosť = $O\ 2.(\log_2 N) + O\ 2.(\log_2 N + P) + O\ 2.(\log_2 P)$

6.5 Pridanie parcely

„Pridanie parcely – na základe vstupných údajov (číslo parcely, popis, zoznam pozícií GPS ohraničujúcich parcelu) sa pridá parcela do evidencie. Zoznam referencií na nehnuteľnosti, ktoré sa na nej nachádzajú naplní systém automaticky“

Priebeh tejto operácie bude rovnaký ako v prípade pridávania nehnuteľnosti, len sa budú operácie spúšťať nad inými stromami

Priemerná zložitosť = $O\ 2.(\log_2 P) + O\ 2.(\log_2 N + P) + O\ 2.(\log_2 N)$

6.6 Editácia nehnuteľnosti

„Editácia nehnuteľnosti – podľa zadanej GPS pozície sa nájdu všetky nehnuteľnosti (ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS), užívateľ zvolí, ktorú chce editovať. Následne program umožní zmeniť evidované údaje vrátane GPS súradníc ohraničujúcich bodov.“

V tejto operácii je potrebné najprv nájsť nehnuteľnosti na základe zadanej súradnice a následne priemerná časová zložitosť závisí od toho, či sa budú editovať kľúčové alebo nekľúčové atribúty. V prípade zmeny kľúčových atribútov je potrebné nehnuteľnosť vyradiť zo stromu nehnuteľností a zmiešaného stromu a následne opäť vložiť naspäť, taktiež je potrebné odstrániť neplatné referencie a doplniť nové. V prípade zmeny nekľúčových atribútov je potrebné danú nehnuteľnosť nájsť a upraviť dané atribúty.

Kľúčové atr. Priemerná časová zložitosť – $O\ 5.(\log_2 N) + O\ 4.(\log_2 N + P)$

Nekľúčové atr. Priemerná časová zložitosť - $O\ 3.(\log_2 N) + O\ 2.(\log_2 N + P)$

6.7 Editácia parcely

„Editácia parcely – podľa zadanej GPS pozície sa nájdu všetky parcely (ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS), užívateľ zvolí, ktorú chce editovať. Následne program umožní zmeniť evidované údaje vrátane GPS súradníc ohraničujúcich bodov.“

Priebeh tejto operácie bude rovnaký ako v prípade vyradenia nehnuteľnosti, len sa budú operácie spúšťať nad inými stromami

Kľúčové atr. Priemerná časová zložitosť – $O\ 5.(\log_2 N) + O\ 4.(\log_2 N + P)$

Nekľúčové atr. Priemerná časová zložitosť - $O\ 3.(\log_2 N) + O\ 2.(\log_2 N + P)$

6.8 Vyradenie nehnuteľnosti

„Vyradenie nehnuteľnosti – podľa zadanej GPS pozície sa nájdu všetky nehnuteľnosti (ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS), užívateľ zvolí, ktorú chce vymazať.“

V tejto operácii je potrebné vyhľadať nehnuteľnosť podľa zadanej súradnice, vymazať 2x nehnuteľnosť zo stromu nehnuteľností (najprv podľa prvej súradnice a potom podľa druhej) a taktiež je potrebné túto nehnuteľnosť vymazať 2x z kombinovaného stromu

Priemerná časová zložitosť = $O\ 3.(\log_2 N) + O\ 2.(\log_2 N + P)$

6.9 Vyradenie parcely

„Vyradenie parcely– podľa zadanej GPS pozície sa nájdu všetky parcely (ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS), užívateľ zvolí, ktorú chce vymazať.“

Priebeh tejto operácie bude rovnaký ako v prípade vyradenia nehnuteľnosti, len sa budú operácie spúšťať nad inými stromami

Priemerná časová zložitosť = $O\ 3.(\log_2 P) + O\ 2.(\log_2 N + P)$

7 Záver

V tejto semestrálnej práci sa mi podarilo úspešne implementovať všeobecný k-d strom, ktorý slúži ako efektívna dátová štruktúra pre spracovanie a vyhľadávanie viacrozmerných údajov. Na základe tejto implementácie som vyvinul aj jednoduchú aplikáciu s grafickým používateľským rozhraním, ktorá umožňuje používateľom pohodlné vyhľadávanie v dátach. Aplikácia zároveň obsahuje funkcie na ukladanie údajov do súboru, čím zabezpečuje ich trvalé uchovanie a umožňuje ich opätovné načítanie pri ďalšom použití.