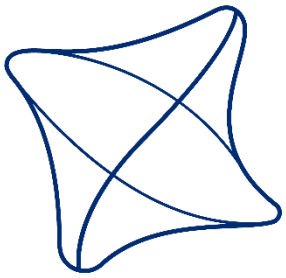
Žilinská univerzita v Žiline
Fakulta riadenia a informatiky



Semestrálna práca S1

Algoritmy a údajové štruktúry 2

Bc. Dominik Ježík

2024

Obsah

1 Použité údajové štruktúry

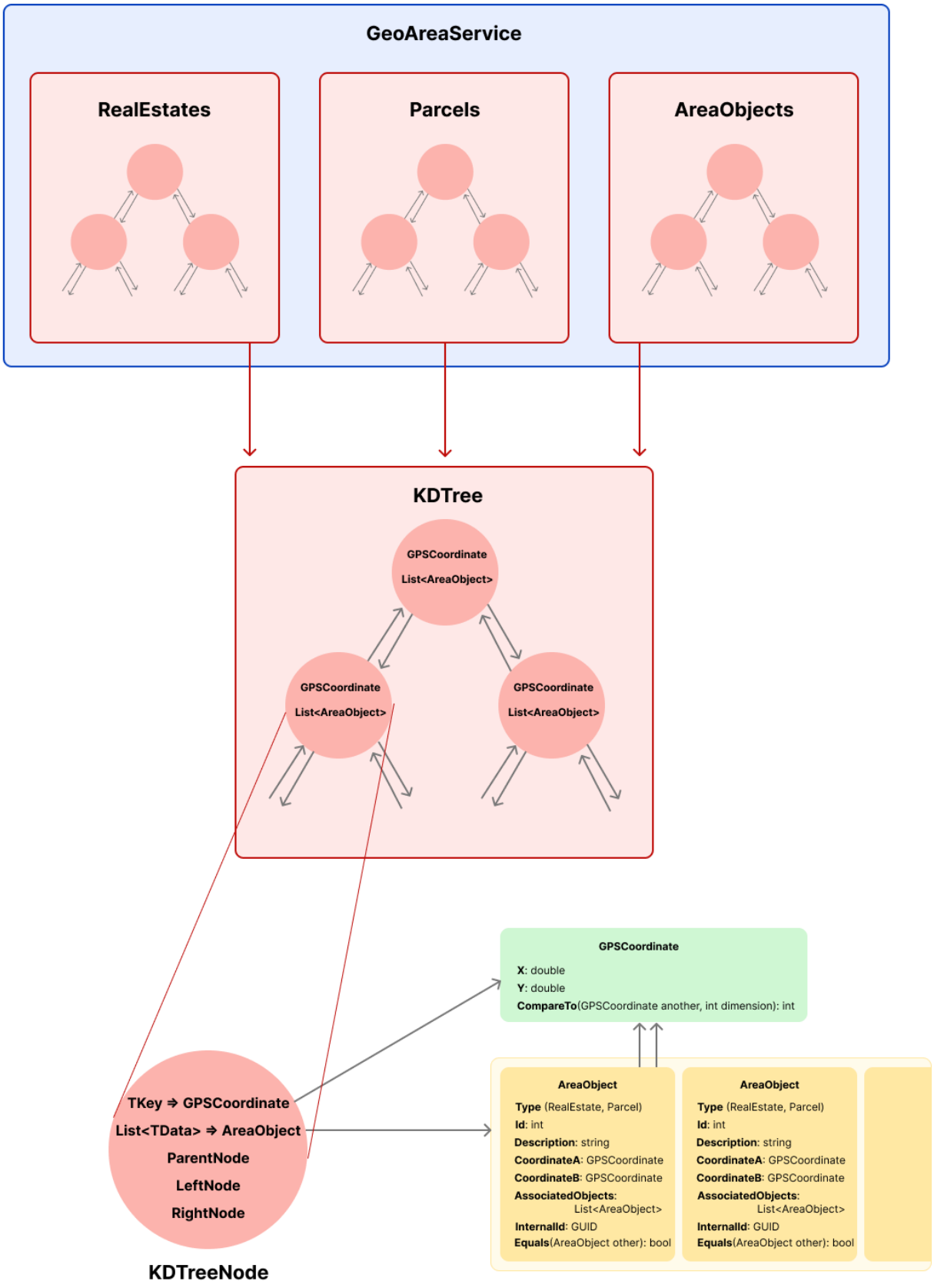
Základom jadra našej aplikácie sú 3 inštancie k-d stromu – v prvej inštancií ukladáme evidované nehnuteľnosti, v druhej inštancií parcely a v tretej sú aj nehnuteľnosti aj parcely. Kľúčom týchto stromov je GPS súradnica (trieda *GPSCoordinate*) s dvoma zložkami X a Y typu double. Ide teda o 2 dimenzionálne k-d stromy. Keďže dáta, ktoré v strome uchovávame majú rovnaké atribúty, v systéme reprezentuje nehnuteľnosti a parcely iba jedna spoločná trieda *AreaObject*. Duplicity podľa všetkých zložiek kľúča sú riešené zoznamom umiestneným vo vrchole.

Do každého stromu vkladáme parcelu alebo nehnuteľnosť 2 krát, keďže každý takýto objekt je definovaný dvoma súradnicami. Pre každý objekt (nehnuteľnosť, parcela) vytvárame iba jednu inštanciu a tú využívame pri viac násobnom vkladaní. Tretí strom obsahujúci oba typy objektu tak isto využíva iba referencie na už existujúce inštancie objektov. Tento strom využívame pri vyhľadávaní ľubovoľného objektu pri zadanej súradnici (prípadne dvoch súradniciach) ale aj pri výpise všetkých objektov evidovaných v systéme. Tento strom nezaberá významné miesto v operačnej pamäti keďže v ňom uchovávame iba referencie na už existujúce objekty ale vďaka jeho využití zvýšime výkon pri spomínaných operáciach.

Priemerne rádové zložitosti základných operácií využitých k-d stromov:

	Strom RealEstates	Strom parcels	Strom AreaObjects
Find	$O(log_2(n))$	$O(log_2(m))$	$O(log_2(n+m))$
Insert	$O(log_2(n))$	$O(log_2(m))$	$O(log_2(n+m))$
Delete	$O(log_2(n))$	$O(log_2(m))$	$O(log_2(n+m))$

kde n je počet prvkov v strome nehnuteľností, m je počet prvkov v strome parciel a $n+m$ je počet prvkov v strome oboch typov objektov (strom AreaObjects).

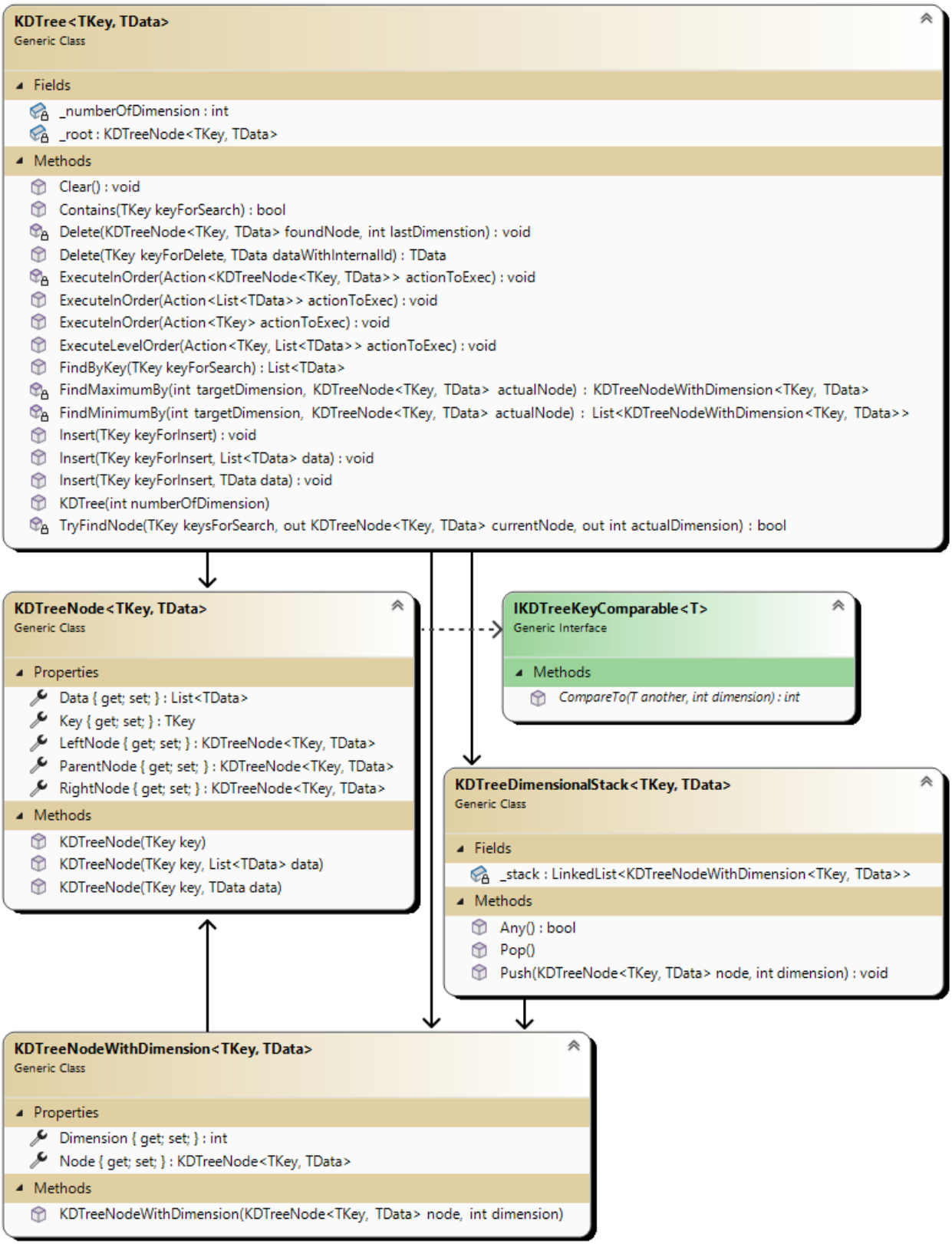

GeoAreaService
RealEstates
Parcels
AreaObjects
KDTree
GPSCoordinate
List<AreaObject>
GPSCoordinate
List<AreaObject>
GPSCoordinate
List<AreaObject>
GPSCoordinate
X: double
Y: double
CompareTo(GPSCoordinate another, int dimension): int
TKey ⇒ GPSCoordinate
List<TData> ⇒ AreaObject
ParentNode
LeftNode
RightNode
KDTreeNode
AreaObject
Type (RealEstate, Parcel)
Id: int
Description: string
CoordinateA: GPSCoordinate
CoordinateB: GPSCoordinate
AssociatedObjects: List<AreaObject>
InternalId: GUID
Equals(AreaObject other): bool
AreaObject
Type (RealEstate, Parcel)
Id: int
Description: string
CoordinateA: GPSCoordinate
CoordinateB: GPSCoordinate
AssociatedObjects: List<AreaObject>
InternalId: GUID
Equals(AreaObject other): bool

Zjednodušený nákres zobrazuje základné väzby medzi k-d stromami, ich vrcholmi, kľúčmi a vkladanými dátami. Stromy sú využívané vo vrstve *GeoArea* hlavným servisom *GeoAreaService*.

Vkladaný kľúč musí implementovať náš vlastný komparátor IKDTreeKeyComparable<T>, do ktorého vstupuje porovnávaný druhý kľúč a dimenzia.

V implementácií k-d stromu používame navyše pomocné štruktúry v podobe lineárnych zoznamov, a tiež jednoduchú triedu pre uchovanie vrcholu a dimenzie (využívame pri operácií *Delete*). Dodatočná trieda KDTreeDimensionalStack je tiež pomocná interná štruktúra, ktorá v sebe využíva lineárny zoznam a poskytuje rozhranie ako zásasobník (tiež využívame pri operácií *Delete*).

V UML diagrame sú zobrazené triedy jadra systému – časť knižnica KDTree.


KDTree<TKey, TData>
Generic Class
Fields
_numberOfDimension : int
_root : KDTreeNode<TKey, TData>
Methods
Clear() : void
Contains(TKey keyForSearch) : bool
Delete(KDTreeNode<TKey, TData> foundNode, int lastDimenstion) : void
Delete(TKey keyForDelete, TData dataWithInternalId) : TData
ExecuteInOrder(Action<KDTreeNode<TKey, TData>> actionToExec) : void
ExecuteInOrder(Action<List<TData>> actionToExec) : void
ExecuteInOrder(Action<TKey> actionToExec) : void
ExecuteLevelOrder(Action<TKey, List<TData>> actionToExec) : void
FindByKey(TKey keyForSearch) : List<TData>
FindMaximumBy(int targetDimension, KDTreeNode<TKey, TData> actualNode) : KDTreeNodeWithDimension<TKey, TData>
FindMinimumBy(int targetDimension, KDTreeNode<TKey, TData> actualNode) : List<KDTreeNodeWithDimension<TKey, TData>>
Insert(TKey keyForInsert) : void
Insert(TKey keyForInsert, List<TData> data) : void
Insert(TKey keyForInsert, TData data) : void
KDTree(int numberOfDimension)
TryFindNode(TKey keysForSearch, out KDTreeNode<TKey, TData> currentNode, out int actualDimension) : bool
KDTreeNode<TKey, TData>
Generic Class
Properties
Data { get; set; } : List<TData>
Key { get; set; } : TKey
LeftNode { get; set; } : KDTreeNode<TKey, TData>
ParentNode { get; set; } : KDTreeNode<TKey, TData>
RightNode { get; set; } : KDTreeNode<TKey, TData>
Methods
KDTreeNode(TKey key)
KDTreeNode(TKey key, List<TData> data)
KDTreeNode(TKey key, TData data)
IKDTreeKeyComparable<T>
Generic Interface
Methods
CompareTo(T another, int dimension) : int
KDTreeDimensionalStack<TKey, TData>
Generic Class
Fields
_stack : LinkedList<KDTreeNodeWithDimension<TKey, TData>>
Methods
Any() : bool
Pop()
Push(KDTreeNode<TKey, TData> node, int dimension) : void
KDTreeNodeWithDimension<TKey, TData>
Generic Class
Properties
Dimension { get; set; } : int
Node { get; set; } : KDTreeNode<TKey, TData>
Methods
KDTreeNodeWithDimension(KDTreeNode<TKey, TData> node, int dimension)

2 Operácie k-d stromu

Implementácie základných operácií nad k-d stromom – vyhľadanie podľa kľúča, vloženie a odstránenie sú podrobne popísané v nasledujúcich podkapitolách.

2.1 Operácia FindByKey

Rádová zložitosť operácie je $O(log_2(n))$, kde n je počet prvkov uložených v strome.

Operácia *FindByKey* vyhľadá na základe vstupného parametra (kľúča typu *TKey*) všetky vložené dáta v strome (typu *TData*) a vráti ich v podobe zoznamu.

Implementačne využíva pomocnú *private* metódu *TryFindNode*, ktorej výstupom je hodnota typu *bool*, vyjadrujúca úspech alebo neúspech nájdenia hľadaného vrcholu v strome. V prípade úspešného nájdenia je vrchol a jeho dimenzia k dispozícií vo výstupných parametroch tejto metódy. Z vrcholu získame zoznam uložených referencií na dáta a vrátime kópiu tohto zoznamu dát. V prípade neúspechu je vo výstupnom parametri posledný vrchol, na ktorom bol algoritmus vyhľadávania ukončený (túto vlastnosť pomocnej metódy sme využili pri operácií Insert). V tom prípade vrátime z metódy FindByKey prázdny zoznam.

2.1.1 Algoritmus vyhľadávania

Algoritmus začne v koreni stromu pričom si inicializuje pomocnú premennú pre uchovávanie aktuálnej dimenzie prehľadávania. V cykle kontroluje, či sa kľúč aktuálneho vrcholu zhoduje s hľadaným kľúčom (pomocou metódy *Equals*). V prípade zhody metóda vráti hodnotu *true*.

V opačnom prípade je potrebné rozhodnúť o vetvení smerom doprava alebo doľava, a to pomocou volania metódy *CompareTo* nad kľúčom aktuálneho vrcholu – vlastný komparátor. Vstupnom metódy je hľadaný kľúč a aktuálna dimenzia. Výstupom sú v tomto prípade hodnoty -1 alebo 1, na základe ktorých sa v algoritme presunieme do ľavého alebo pravého syna aktuálneho vrcholu. V prípade, že tohto syna aktuálny vrchol nemá (hodnota je *null*) ukončíme algoritmus a metóda vráti hodnotu *false* (hľadaný prvok pre zadaný kľúč sme nenašli). V prípade pokračovania cyklu (prechod na syna) aktualizujeme aktuálnu dimenziu – pripočítame 1 a spravíme operáciu zvyšok po delení maximálnou dimenziou stromu.

2.2 Operácia Insert

Rádová zložitosť operácie je $O(log_2(n))$, kde n je počet prvkov uložených v strome.

Operácia *Insert* vloží do stromu parameter dáta (typu *TData*) podľa zadaného kľúča (typu *TKey*).

Táto operácia použije už spomínanú pomocnú *private* metódu *TryFindNode*. Pomocou tejto metódy sa pokúsime nájsť vrchol podľa zadaného kľúča. Ak sa nám to podarí (vrátila hodnotu *true*), znamená to, že v strome sa už nachádza prvok (alebo viac prvkov) so zadaným kľúčom. V tom prípade stačí zobrať z výstupného parametra nájdený vrchol a do jeho dátovej časti, ktorá je realizovaná zoznamom dát, iba pridať nové dáta zo vstupu.

Ak sa nám neporadilo nájsť vrchol (pomocná metóda vrátila hodnotu *false*), v tom prípade algoritmus hľadania skončil v liste, do ktorého je potrebné ako nového syna umiestniť nové dáta. Je však potrebné rozhodnúť či sa má vytvoriť nový vrchol ako ľavý alebo pravý syn. Preto dodatočne porovnáme kľúč nájdeného vrcholu s kľúčom vkladaných dát cez metódu *CompareTo*. Aby táto metóda vedela porovnať kľúče, potrebuje na vstupe aj dimenziu, pre ktorú kľúče porovnáva. Preto dodatočným výstupným parametrom pomocnej metódy *TryFindNode* je aj dimenzia nájdeného vrcholu. Po porovnaní vytvoríme nový vrchol a spojíme ho s nájdeným vrcholom do príslušnej strany (ľavý / pravý syn).

2.3 Operácia Delete

Rádová zložitosť operácie je $O(log_2(n))$, kde n je počet prvkov uložených v strome.

Operácia *Delete* odoberie už vložené dáta zo stromu podľa zadaného kľúča (TKey) a podľa inštancie triedy typu TData, ktorá referenciou nie je vložená v strome ale obsahuje metódu *Equals*, slúžiacu na jednoznačnú identifikáciu mazaného prvku v strome (napríklad interné ID).

Na začiatku sa pokúsime nájsť vrchol podľa zadaného kľúča. V prípade, že sa v nájdenom vrchole v zozname dát nachádza viac položiek (položky, ktoré boli vložené pod rovnakým kľúčom), ide o triviálnu situáciu. Z tohto zoznamu identifikujeme hľadaný prvok mazania pomocou metódy *Equals* a vstupného parametra typu *TData*. Odstránený dátový prvok vrátime ako výstup tejto metódy. V prípade, že sa vo vrchole v zozname dát nachádza iba jeden prvok, je nutné spustiť zložitejší algoritmus pre odobratie vrcholu zo stromu. Tento algoritmus je oddelený do pomocnej *private* metódy pre lepšiu prehľadnosť. Do vstupných parametrov je potrebné vložiť referenciu na vrchol, ktorý chceme odstrániť a jeho dimenziu (tú máme tiež k dispozícii ako výstupný parameter metódy *TryFindNode*).

Algoritmus začne inicializáciou dvoch zoznamov:

- *nodesToDelete* – vrcholy, ktoré je nutné zo stromu odobrať a opätovne pridať, pretože pri hľadaní minima danej zložky kľúča boli zistené už existujúce duplicity.
- *nodesToAdd* – vrcholy, z predchádzajúceho zoznamu, ktoré je nutné do stromu znova vložiť (až na konci algoritmu).

Spustíme cyklus, ktorý sa vykoná vždy minimálne jeden krát. V prvej iterácií pre nájdený mazaný vrchol a “kaskádu“ vrcholov až pol list. V ďalší iteráciách bude bežať pokiaľ máme nejaké prvky v zozname na odobratie (*nodesToDelete*).

V prvej iterácií cyklu zistíme či nejde o triviálnu situáciu, kedy je mazaný vrchol listom stromu – v tom prípade stačí iba odobrať vrchol zo stromu (nastavenie null referencie rodičovi na daného syna).

V prípade, že mazaný vrchol nie je listom stromu, je potrebné nájsť vhodného náhradníka. Pri hľadaní preferujeme ľavý podstrom a hľadanie maxima, pretože nie je nutné v prípade duplicity podľa hľadanej zložky kľúča odoberať všetky takého vrcholy a opätovne ich vkladať.

Pre vyhľadanie maxima používame pomocnú metódu *FindMaximumBy*, kde na vstupe zadáme dimenziu, pre ktorú hľadáme maximum a tiež vrchol, ktorý definuje podstrom, kde je maximum hľadané. Návratová hodnota je vrchol reprezentujúci nájdené maximum. Algoritmus funguje na základe modifikovanej reverznej inorder prehliadky nad zadaným podstromom. V implementácií nie je použitá rekurzia, preto sa využíva zásobník vrcholov. V implementácií prechádzame vždy v podstrome čo najviac doprava (preto reverzná), pričom kontrolujeme či sme nenašli nové maximum. Pri každom prechode na inú úroveň je potrebné udržiavať informáciu o aktuálnej dimenzií. Vďaka nej vieme prehliadku ohraničiť a optimalizovať vyhľadávanie, pretože dokážeme vylúčiť vetvy, o ktorých vieme, že hľadané maximum neobsahujú. To znamená, že ak sa aktuálne nachádzame na vrchole, ktorého dimenzia sa rovná dimenzií zadanej ako parameter (pre ňu hľadáme maximum) dokážeme z vyhľadávania vylúčiť ľavú vetvu aktuálneho vrcholu. Keďže vždy potrebujeme informáciu o aktuálnej dimenzií, na ktorej sa nachádzame vznikal problém, kedy sme prechádzali stromom čo najviac doprava, následne nejaký počet vrcholov doľava a potrebovali sme sa spätne vrátiť a pokračovať v prehliadke. Cez zásobník sa dokážeme efektívne vrátiť ale nemali sme prístup k dimenzií pre daný vrchol v zásobníku. Z tohto dôvodu k vrcholom v zásobníku evidujeme aj ich príslušnú dimenziu.

Algoritmus vyhľadávania minima v pravom podstrome funguje analogicky s rozdielom, že miesto reverznej modifikovanej inorder prehliadky používame modifikovanú inorder prehliadku (presúvame sa čo najviac doľava). Pravidlo ohraničenia v tomto prípade znamená, že vylúčime pravú vetvu ak sa dimenzia aktuálneho vrcholu rovná zadanej dimenzie. Navyše návratová hodnota pomocnej metódy *FindMinimumBy* je zoznam nájdených miním (čiže obsahuje duplicity podľa hľadanej zložky minima). Potrebujeme tento zoznam, keďže prvky je potrebné zo stromu odobrať a opätovne vložiť.

Nájdeného náhradníka respektíve prvý prvok zo zoznamu možných náhradníkov (zároveň zoznam duplicít podľa zložky) vyberieme a do mazaného vrcholu premiestnime kľúč z tohto náhradníka

a aj jeho dáta. V zozname možných náhradníkov aktualizujeme referenciu tohto zvoleného náhradníka na nové premiestnené miesto. Ak sa zvolený náhradník nachádza v zozname *nodesToDelete* je tiež potrebné aktualizovať referenciu na toto nové premiestnené miesto vrátane jeho novej dimenzie. Ak sa náhradníkov našlo viacero znamená to, že máme duplicity podľa zložky a tieto vrcholy umiestnime do zoznamu *nodesToDelete* (dávame pri tom pozor aby tam vkladaný vrchol bol najviac jeden krát).

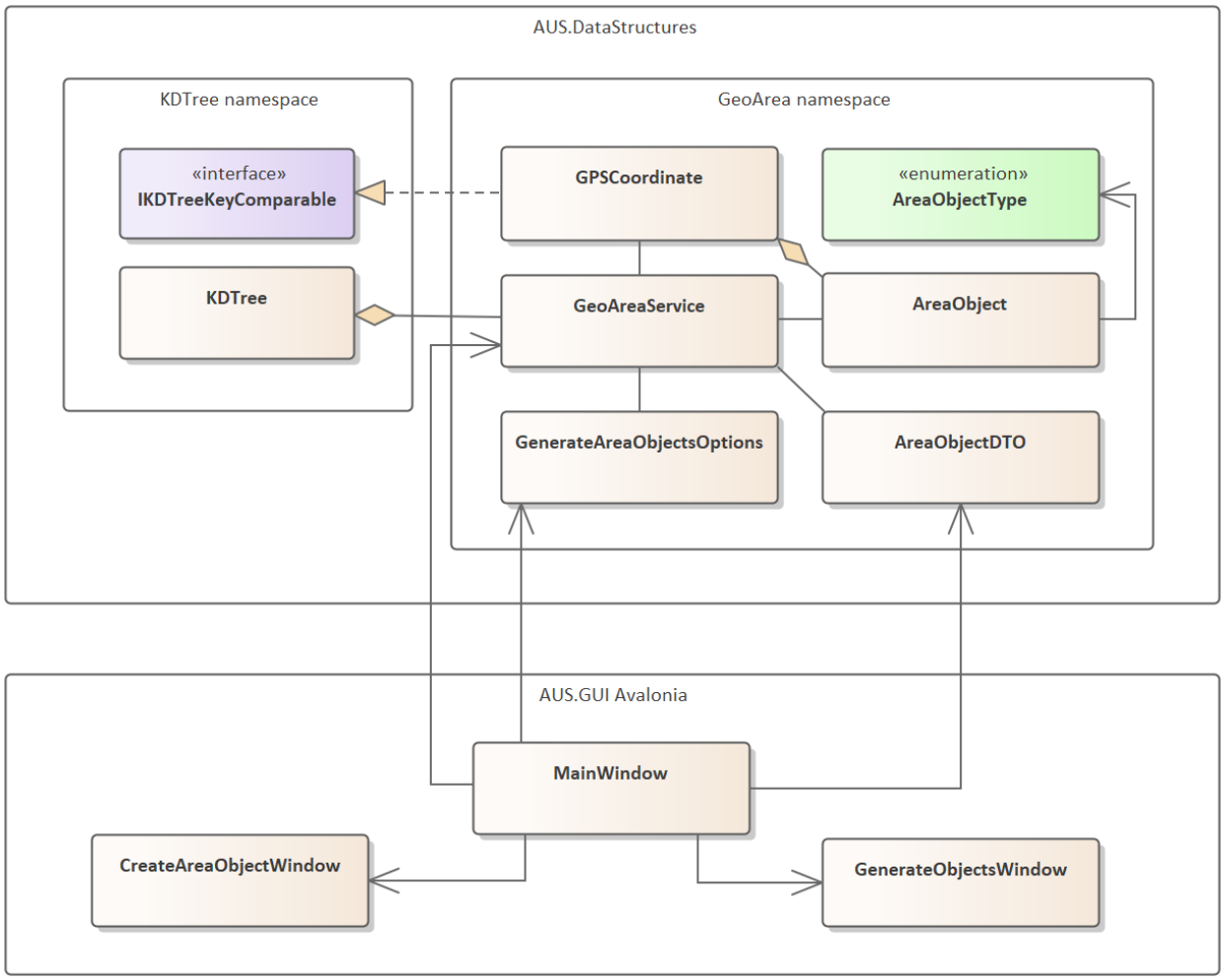
Ak je náhradníkom list, odoberieme ho zo stromu. V opačnom prípade, potrebujeme "rekurzívne/kaskádovito" hľadať ďalšieho náhradníka na miesto, kde sme už sme práve náhradníka našli – vzniklo tu prázdne miesto "diera". Cyklus sa preto zopakuje pre toto prázdne miesto s aktualizovanou dimenziou daného miesta/vrcholu. Cyklus opakujeme pokiaľ sa nedostaneme až k listu, kde ho môžeme zo stromu odobrať. Zároveň v tomto procese pokiaľ je to potrebné dopĺňame zoznam *nodesToDelete*.

Potom ako sme sa dostali až po list a odstránili ho, kontrolujeme vždy zoznam *nodesToDelete*. Ak sa v ňom nachádza nejaký vrchol tak ho poznačíme do druhého zoznamu *nodesToAdd*, a tiež ho nastavíme ako mazaný vrchol a proces mazania sa v cykle opakuje. Cyklus končí keď proces mazania zvolené prvku skončil (odstránili sme list) a zoznam *nodesToDelete* je prázdny.

Na konci vložíme späť do stromu všetky záznamy zo zoznamu *nodesToAdd*, čím sme ukončili operáciu Delete.

3 Architektúra aplikácie

Architektúra aplikácie je rozčlenená do troch častí. V rámci knižnice *AUS.DataStructures*, ktorá predstavuje jadro aplikácie sú dva menné priestory – *KDTree* a *GeoArea*. Z časti *KDTree* sú zámerne v diagrame zobrazené iba 2 objekty – *KDTree* a *IKDTreeKeyComparable*. Iné objekty z tejto časti neposkytujeme na verejné použitie, keďže slúžia iba na interné účely a sú označené aj prístupovým modifikátorom *internal* (tieto objekty sú zobrazené v diagrame v kapitole 1).



V časti *GeoArea* používame k-d strom (3 inštancie v servis triede). Celú logiku tejto časti jadra aplikácie riadi *GeoAreaService*, ktorý poskytuje základné a rozširujúce funkcie aplikácie. Tento servis na GUI nikdy nevracia objekty typu *AreaObject* ale ich namapovanú verziu *AreaObjectDTO*

(DTO = Data Transfer Object). Preto je zabezpečené mapovanie z oboch strán medzi týmito objektami.

Vrstva *AUS.GUI* je vrstva používateľského rozhrania, ktoré je implementované v technológií Avalonia. Používame tu navyše objekty typu ViewModel pre two-way data binding formulárov a dát na používateľskom rozhraní. Dáta sú vždy premapované na *AreaObjectDTO* (z tohto dôvodu a pre prehľadnosť nie sú zobrazené v diagrame).

4 Funkcie aplikácie a ich zložitosti

Podľa zadania semestrálnej práce je v aplikácií možné realizovať týchto 9 základných funkcií s uvedenými zložitosťami:

1. Vyhľadanie nehnuteľností – podľa zadanej GPS pozície sa nájdu všetky nehnuteľnosti, ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS.
   Zložitosť: $O(log_2(2n))$, kde n je počet nehnuteľností vložených v strome nehnuteľností.
   Trieda zložitosti: $O(log\ n)$

2. Vyhľadanie parciel – podľa zadanej GPS pozície sa nájdu všetky parcely, ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS.
   Zložitosť: $O(log_2(2m))$, kde m je počet parciel vložených v strome parciel.
   Trieda zložitosti: $O(log\ m)$

3. 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.
   Zložitosť: $O(log_2(2k))$, kde k je počet objektov (nehnuteľnosti + parcely) vložených v strome objektov.
   Trieda zložitosti: $O(log\ k)$

4. 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 avšak pre zjednodušenie budeme nehnuteľnosť považovať za stojacu na parcele ak s ňou má spoločný aspoň jeden z dvoch bodov.
   Zložitosť: $O(2 * log_2(2 * n) + 2 * log_2(2 * k))$, kde n je počet nehnuteľností vložených v strome nehnuteľností a k je počet objektov (nehnuteľnosti + parcely) vložených v strome objektov.
   Trieda zložitosti: $O(log\ k)$

5. 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 avšak pre zjednodušenie budeme nehnuteľnosť považovať za stojacu na parcele ak s ňou má spoločný aspoň jeden z dvoch bodov.

   Zložitosť: $O(2 * log_2(2 * m) + 2 * log_2(2 * k))$, kde m je počet parciel vložených v strome parciel a k je počet objektov (nehnuteľnosti + parcely) vložených v strome objektov.

   Trieda zložitosti: $O(log\ k)$

6. 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.

   Zložitosť pokiaľ používateľ nezmenil súradnice nehnuteľnosti:

   $O(log_2(2n))$, kde n je počet nehnuteľností vložených v strome nehnuteľností.

   Zložitosť pokiaľ používateľ zmenil súradnice (vykoná sa *delete* a *insert*):

   $O(4 * log_2(2 * n) + 4 * log_2(2 * k))$, kde n je počet nehnuteľností vložených v strome nehnuteľností a k je počet objektov (nehnuteľnosti + parcely) vložených v strome objektov.

   Trieda zložitosti: $O(log\ k)$

7. 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.
   Zložitosť pokiaľ používateľ nezmenil súradnice parcely:
   $O(log_2(2m))$, kde m je počet parciel vložených v strome parciel.
   Zložitosť pokiaľ používateľ zmenil súradnice (vykoná sa *delete* a *insert*):
   $O(4 * log_2(2 * m) + 4 * log_2(2 * k))$, kde m je počet parciel vložených v strome parciel a k je počet objektov (nehnuteľnosti + parcely) vložených v strome objektov.
   Trieda zložitosti: $O(\log k)$

8. Vyradenie nehnuteľnosti – podľa zadanej GPS pozície sa nájdu všetky nehnuteľnosti (ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS), užívateľ zvolí, ktorú chce vymazať.
   Zložitosť: $O(2 * log_2(2 * n) + 2 * log_2(2 * k))$, kde n je počet nehnuteľností vložených v strome nehnuteľností a k je počet objektov (nehnuteľnosti + parcely) vložených v strome objektov.
   Trieda zložitosti: $O(\log k)$

9. Vyradenie parcely– podľa zadanej GPS pozície sa nájdu všetky parcely (ktorých niektorý krajný bod sa zhoduje s vyhľadávanou GPS), užívateľ zvolí, ktorú chce vymazať.
   Zložitosť: $O(2 * log_2(2 * m) + 2 * log_2(2 * k))$, kde m je počet parciel vložených v strome parciel a k je počet objektov (nehnuteľnosti + parcely) vložených v strome objektov.
   Trieda zložitosti: $O(log\ k)$

5 Uloženie a načítanie stavu aplikácie

Uloženie a opätovné načítanie je realizované pomocou dvoch csv súborov – jeden pre nehnuteľnosti a jeden pre parcely. Zvolený formát týchto súborov minimalizuje ich výslednú veľkosť a ich vlastnosťou je aj zachovanie štruktúry/tvaru dvoch k-d stromov pre nehnuteľnosti a parcely. Tvar tretieho spoločného stromu nie je zachovaný a pri opätovnom načítaní môže byť odlišný ako pri uložení. Každý riadok csv súboru reprezentuje jeden vrchol daného stromu a začína jeho kľúčom – čiže súradnicou *A* so zložkami X a Y. V jednom vrchole môže byť uložených ale viacero dátových objektov. Tie ukladáme v riadku hneď za kľúčom, pričom vynecháme prvú súradnicu (A), keďže tá je kľúčom a bola by uvedená duplicitne.

Zjednodušený tvar riadku csv súboru je následovný:

KľúčVrcholu; DátaVloženéhoObjektu_1; DátaVloženéhoObjektu_2; ...; DátaVloženéhoObjektu_n

Konkrétnejší tvar riadku, kde *A* je 1. súradnica, ktorá je zároveň kľúčom a *B* je 2. súradnica objektu n-tého objektu:

AX ; AY ; BX_1; BY_1; Id_1; Description_1; BX_2; BY_2; Id_2; Description_2; ...; BX_n; BY_n; Id_n; Description_n

5.1 Uloženie do súboru

Keďže v strome je uložená referencia na každý objekt 2 krát – raz pre súradnicu *A* a raz pre súradnicu *B*, bolo potrebné implementovať nasledovné pravidlo aby tento spôsob fungoval:

Ak sa súradnica *A* zapisovaného objektu rovná kľúču, v tom prípade objekt zapíšeme do riadku. V opačnom prípade objekt nezapíšeme – čiže vo výslednom súbore nebude zbytočne zaberať miesto 2 krát ale iba raz. Z toho vyplýva, že v súbore môžu existovať aj riadky iba so súradnicou *A*, bez objektov, čiže pri načítaní dostaneme presný tvar ako pred uložením.

5.2 Načítanie zo súboru

Pri načítaní v cykle prechádzame každý riadok CSV súboru. V prípade, že riadok obsahuje iba 2 segmenty, znamená to, že k súradnici neevidujeme žiadne dátové objekty (také, že kľúč sa rovná ich súradnici *A*). V tom prípade k-d strom obsahuje metódu na vloženie iba vrcholu a zoznam dátových objektov bude prázdny.

V prípade že riadok obsahuje viac segmentov ako 2, pre ich počet bude vždy platiť:

*počet segmentov = 2 + (4 * n)*

kde n je počet vložených dátových objektov.

Podľa pravidla namapujeme segmenty do n objektov, vložíme ich do pomocného zoznamu a tiež do k-d stromov (strom pre daných typ objektu a do spoločného stromu) podľa súradnice *A*. Po prejdení všetkých riadkov CSV súboru, prejdeme všetky dátové objekty z pomocného zoznamu a opätovne ich vložíme do oboch k-d stromov ale podľa súradnice B.

Uloženie aj načítanie sa realizuje pre oba typy objektov – pre oba súbory.