VYSOKÁ ŠKOLA POLYTECHNICKÁ JIHLAVA

Katedra elektrotechniky a informatiky

Obor Počítačové systémy

Porovnání výkonosti polí, kolekcí a nástrojů pro práci s kolekcemi v prostředí .NET/C#

bakalářská práce

Autor: Tomáš Zadražil

Vedoucí práce: Ing. Marek Musil

Jihlava 2015

Vysoká škola polytechnická Jihlava

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Autor práce:	**Tomáš Zadražil**
Studijní program:	Elektrotechnika a informatika
Obor:	Počítačové systémy
Název práce:	**Porovnání výkonosti polí, kolekcí a nástrojů pro práci s kolekcemi v prostředí .NET/C#**
Cíl práce:	Porovnání výkonosti polí, kolekcí a nástrojů pro práci s kolekcemi v prostředí .NET/C#. Cílem práce je porovnat časy potřebné na provedení operací nad kolekcemi, dále pak porovnat časy potřebné pro vyhledání a řazení pomocí definovaných nástrojů. Součástí práce může být porovnání známých algoritmů řazení a vyhledávání s nástroji dostupnými v .NET/C# a proveden odhad vnitřní implementace dostupných nástrojů.

Ing. Marek Musil
vedoucí bakalářské práce

doc. Ing. Bc. Michal Vopálenský, Ph.D.
vedoucí katedry
Katedra elektrotechniky a informatiky

Abstrakt

Tato práce řeší rychlost kolekcí, které jsou implementovány v prostředí .NET/C#. Budou zkoumány metody vkládání, mazání, řazení a hledání prvků. Součástí práce je také měření času vlastní implementace pole pomocí klasického pole nebo spojovaného seznamu, a jejich následné porovnání již implementovaný kolekcí v .NET/C#. Ze zjištěných výsledků bude odhadnut algoritmus, kterým jsou implementovány zkoumané metody kolekce.

Klíčová slova

.NET, C#, kolekce, rychlost, řazení, vkládání, vyhledávání, mazání, implementace, složitost algoritmu

Abstract

This work solves the speed of collections, which are implemented in .NET/C#. They will be investigated methods of inserting, deleting, sorting and searching features. In this work is also included timekeeping own implementation of field used the classical field or linked list, and their follow comparison already implemented collections in .NET/C#. From the observed results will be estimated algorithm, which are implemented investigated methods of collection.

Key words

.NET, C#, collection, speed, sort, insert, find, delete, implementation, comparison,

algorithm complexity

Prohlašuji, že předložená bakalářská práce je původní a zpracoval/a jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem v práci neporušil/a autorská práva (ve smyslu zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů, v platném znění, dále též „**AZ**“).

Souhlasím s umístěním bakalářské práce v knihovně VŠPJ a s jejím užitím k výuce nebo k vlastní vnitřní potřebě VŠPJ.

Byl/a jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje **AZ**, zejména § 60 (školní dílo).

Beru na vědomí, že VŠPJ má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom/a toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem VŠPJ, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených vysokou školou na vytvoření díla (až do jejich skutečné výše), z výdělku dosaženého v souvislosti s užitím díla či poskytnutí licence.

V Jihlavě dne

.....

Podpis

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu práce Ing. Marku Musilovi za poskytnutí tématu a možnost vytvářet ho pod jeho vedením.

Obsah

1 Úvod a motivace

V dnešní době rozvoj elektrotechniky a informatiky dosáhl již takové úrovně, o které se nám ještě před pár lety ani nesnilo. Začátkem 90. let se např. paměť u stolního počítače měřila v kB (kilobytech), maximálně v MB (megabytech). Pokud člověk měl v té době stolní počítač, jednalo se o 8 bitový počítač (počítač to byla tehdy poměrně finančně nákladnou záležitostí). Samozřejmě na západě vývoj v tomto odvětví byl dál a k nám se tato výpočetní technika dostávala se zpožděním. V té době dokonce ani disketová mechanika nebyla samozřejmostí a pro nahrávání programu do počítače se používal často magnetofonový pásek. V polovině 90. let jsme mohli doma sedět už u 32 bitového stolního počítače s na tu dobu solidním operačním systémem Windows 95. Velikost pamětí u PC se začala měřit v jednotkách GB (gigabytech) a TB (terabytech).

Velké množství paměti a výkonný mikroprocesor schopný zpracovávat enormní množství dat za sekundu se často využívají k obrovskému a rychlému zpracování dat v mnoha odvětví. Následoval obrovský rozmach elektrotechniky a informatiky, díky čemuž se přešlo či přechází od 32 bitové architektury k 64 bitové.

Výkonná výpočetní technika umožňuje zpracovávat obrovské množství informací za poměrně krátkou dobu. Nahrazuje řadu lidských a pracovních činností. Přestože jsou dnešní prostředky výpočetní techniky vysoce výkonné a rychlé, rychlost zpracování dat a využití paměti stále hraje roli. U algoritmů se stále měří paměťová a časová náročnost. Problém rychlosti zpracování dat se projeví při velkém množství dat, např. při zpracování záznamů velké databáze (evidence osob, bankovní účty, atd.). Dalo by se říci, že počítače už ovlivňují každodenní život každého z nás. Setkáváme se s nimi ve školách, průmyslu, administrativní správě a v mnoha dalších odvětvích.

Právě rychlost zpracování dat je tématem tohoto semestrálního projektu. Do rychlosti zpracování dat se značně promítá vhodný návrh algoritmu, který může zredukovat čas potřebný výpočet nebo pro třídění dat i o desítky procent.

V posledních době se zajímám o platformu .NET a programování v jazyce C# [:sí šárp]. Ve srovnání s jazyky C a C++, jazyk C# nabízí možnost využití datových kontejnerů (kolekcí), které umožňují třídit a vyhledávat data uložené v kolekci. Tyto objekty mají

přímo implementované metody pro vyhledávání a řazení dat. Přepokládá se, že při velkém množství dat bude docházet k časovým prodlevám.

Budoucím snažením je porovnat časovou složitost prováděných operací nad kolekcemi a to při různých klíčových situacích. Součástí je také pomocí testů nad vlastní implementací známých algoritmů odhadnout, jaký algoritmus vyhledávání nebo řazení je v metodách těchto kolekcí implementován. Nebo alespoň porovnat výsledky známých algoritmů s výsledky metod implementovaných v kolekcích.

2 Současný stav

Algoritmus se posuzuje z hlediska paměťové a časové složitosti. Přestože počítače jsou v dnešní době výkonné a mají k dispozici velké množství paměti, při zpracování velkého množství dat se může projevit časová náročnost.

Složitost je rozdílná pro pole s přímým přístupem a se sekvenčním přístupem. U přímého přístupu přistupujeme k jednotlivým položkám pole přímo, aniž bychom museli před tím, postupně procházet všechny předchozí prvky v poli. U sekvenčního přístupu přistupujeme k jednotlivým položkám pole sekvenčně, což znamená, že musíme postupně procházet všechny předchozí položky pole až k danému prvku.

V prostředí .NET je implementována řada kolekcí – datových kontejnerů. Předpokládáme, že se liší vnitřní implementací a tedy budou vykazovat rozdílný čas potřebný pro provedení operace. Nenašel jsem žádnou publikaci, která by se touto problematikou zabývala.

Metoda pro vyhledávání prvku v poli metodou půlení intervalu je všeobecně rychlejší, než sekvenční vyhledávání.

Pro řazení prvků v poli se používá celá řada známých algoritmů řazení jako například SelectionSort, QuickSort, InsertionSort, BubbleSort, HeapSort a další. Hlavní dvě vlastnosti, které se posuzují u těchto algoritmů řazení, jsou paměťová náročnost a časová složitost. Právě u algoritmů řazení, které jsou implementovány pomocí rekurzivního volání, značně stoupají paměťové nároky, protože před zavoláním rekurze se musí na zásobník uložit všechny potřebné proměné. Ze složitosti, která znamená kolik operací je potřeba provést nad seřazovaným polem, aby se stalo seřazeným, můžeme zase odhadnout časovou náročnost algoritmu. Každý z těchto algoritmů má své výhody i nevýhody a může rozhodovat i povaha seřazovaných prvků. Právě tyto otázky by měla zodpovědět tato práce.

3 Analýza problému

V této kapitole jsou probrány všechny kolekce v prostředí .NET a jejich klíčové metody. Nejpoužívanějšími metodami v kolekci jsou vkládání, vyhledávání, řazení a odstraňování.

3.1 Souhrn tříd kolekcí v .NET

V prostředí .NET je implementováno mnoho kolekcí které jsou diskutovány v následujícím textu. Každá kolekce se může odlišovat svou vnitřní implementací, ale navenek mají stejné metody. Jedná se o vkládání prvku do kolekce, mazání prvku v kolekci, vyhledávání prvku v kolekci, řazení kolekce atd. Důležitá vlastnost každé této třídy je, že má společného rodiče. Jinými slovy, každá třída je poděděna ze společné třídy typu *Object*. Další zajímavá vlastnost je, že tyto třídy implementují společné rozhraní. Kolekce se mohou třídit podle několika hledisek. Můžeme je dělit podle způsobu přistupu k jednotlivým prvkům kolekce, nebo podle toho jestli daný prvek je typový nebo beztypový, popř. podle způsobu implementace kolekce.

Kolekce (třídy) sekvenční a asociativní

Každou kolekci můžeme rozdělit na sekvenční a asociativní. U sekvenční kolekce (posloupnosti) záleží na pořadí v jakém jsou jednotlivé prvky do kolekce přidávány. Jako příklad sekvenční kolekce lze uvést spojovaný seznam. [2]

Do asociativních kolekcí (slovníků) je vždy ukládáno ke každému prvku (hodnotě) i klíč. V asociativní kolekci nezáleží v jakém pořadí jednotlivé prvky byly přidávány do kolekce. U asociativních kolekcí vyhledáváme prvky podle klíče. Jako příklad asociativní kolekce lze uvést slovník.

Kolekce (třídy) typové a beztypové

Jiné dělení podle kterého můžeme rozdělovat kolekce je podle typu ukládaného prvku (hodnoty) do kolekce na beztypové (negenerické), kde všechny prvky kolekce jsou obecného typu *object*, a na typové (generické) kde musíme uvést datový typ s jakým daná kolekce pracuje. Každý sám musí danou položku získanou z kolekce explicitně přetypovat. Výhoda generické kolekce je v tom, že není nutné se o explicitní přetypování vůbec zajímat. Další výhoda použití generické kolekce je, že pokud by

jsme někde v programu použili špatný datový typ, chyba by byla hned odhalena, a ne až za běhu programu, jak je to možné u negenerické kolekce. [2]

Kolekce s přímým přístupem a se sekvenčním

Toto dělení rozděluje kolekce podle toho jak přistupujeme k jednotlivým prvkům kolekce. Kolekce s přímým přístupem mohou přistupovat ke svým položkám přímo aniž bychom procházeli jednotlivé předchozí položky. Typickým příkladem přímého přístupu k jednotlivým položkám je klasické pole. U sekvenčních kolekcí musíme projít všechny předchozí prvky než se dostaneme na požadovaný prvek v kolekci. Zde jako příklad můžeme uvést spojový seznam.

3.2 Klasické pole

U klasické pole můžeme zdůraznit tu přednost, že je poměrně jednoduché, jednotlivé prvky v tomto poli jsou v paměti ukládány po sobě, tedy záleží na pořadí v jakém jsou tyto prvky do pole přidávány, rychle se ke každému prvku přistupuje a v paměti prvky tohoto pole zabírají stejnou velikost. Ke každému prvku v tomto poli lze přistupovat pomocí indexu, které označuje pozici prvku v poli.

V prostředí .NET je implementována třída Array ve které jsou definovány metody pro manipulaci s tímto polem. Statická metoda *Sort* seřadí pole. V každém prvku v tomto poli musí být implementováno rozhraní IComparable, které umožní prostřenictvím metody *CompareTo* porovnávat tyto prvky mezi sebou.

Ukázka implementace metody *CompareTo* ve třídě Color

```
public int CompareTo(object obj)
{
     if (obj is Color) {
         Color o1 = obj as Color;
         if (o1 == null)
             return 0;
         else
         {
             if (this.barva > o1.barva) return 1;
             if (this.barva < o1.barva) return -1;
             return 0;
         }
     }
     else
     {
         return 0;
     }
}
```

Následující ukázka programu ukazuje zacházení s tímto polem:

```
// klasické jednorozměrné pole
Color[] pole = new Color[10];
pole[1].R = 128;
pole[1].G = 34;
pole[1].B = 88;
byte R = pole[1].R;
Array.Sort(pole);     // pozor, máme vlastní datový typ
Color col = new Color(33, 22, 56);
pole.SetValue(col, 5);
pole[3] = col;

Array pole2 = Array.CreateInstance(typeof(Color), 10);
pole2.SetValue(col, 5);

Color color1 = new Color(19, 33, 45);
Color color2 = new Color(190, 44, 49);
Color[] pole3 = {color1, color2};
byte red = pole3[0].R;

// klasické dvojrozměrné pole
Color[,] pole3 = new Color[10, 10];
pole.SetValue(col, 4, 4);
pole3[2, 3] = col;

// vícerozměrné pole s nepravidelnými rozměry dimenzí
Color[][] pole4 = new Color[2][];
pole4[0] = new Color [4]; // jedna dimenze má 4 prvky
pole4[1] = new Color [6]; // druhá dimenze má 6 prvků
```

3.3 Beztypové (negenerické) kolekce

Beztypové kolekce jsou definovány v pracovním prostoru *System.Collections*. Po vyjmuti prvku z kolekce si musí pak tento prvek explicitně přetypovat na původní typ. případná chyba přetypování je objevena až při spuštění programu. S využitím klíčového slova *is* a klíčového slova *as* se dá otestovat tento prvek v kolekci na konkrétní datový typ a pak ho následně přetypovat.

ArrayList – velikost se dynamicky zvětšuje podle potřeby za běhu program vždy na dvojnásobnou velikost. Ke každému prvku z této kolekce lze přistupovat přímo.

Queue – funguje jako fronta na principu „první dovnitř, první ven“. Anglicky first in, first out neboli FIFO.

Stack – funguje jako zásobník na principu poslední dovnitř, první ven. Anglicky last in, first out neboli LIFO.

Stack a Queue se řadí mezi sekvenční kolekce.

Hashtable – je kolekce, která používá hešovací tabulku. V této tabulce je vždy uložen vedle hodnoty též klíč. Pokud tedy do tabulky chceme přidat prvek, je nejdřív metodou *GetHashCode()* vrácen hešový kód klíče a podle něj spočítána poloha, kam se uloží prvek.

SortedList – funguje na principu že ukládá vždy ke každé hodnotě i klíč. Již při vkládání prvku do kolekce se nejdřív zjistí jestli v kolekci není již prvek se stejným klíčem a pokud ne, uloží prvek do kolekce tak, aby byla kolekce potom již opět seřazená. Ke každému prvku v této kolekci lze přistupovat přes klíč nebo přes index. Je to vlastně kombinace pole a hešové tabulky.

Do kolekce lze uložit i prvky atomického datového typu. V takovém případě probíhá proces tzv. *balení* a *vybalení*. Každý takový prvek atomického datového typu se zabalí a vytvoří se instance objektu ve kterém je konkrétní atomický datový typ uložen. Pokud pak chceme takový prvek přečíst postup je přesně obrácený, probíhá vybalování. Tento proces je časově náročný, protože se vytváří instance nového objektu, do kterého se ukládá zabalovaná hodnota, a pak ještě tento odkaz na objekt je potřeba uložit do konkrétní kolekce. Při čtení takového datového typu z kolekce je nutné aby proběhlo tzv. *vybalování*, musí být přečtena hodnota z objektu ve kterém je uložena a musí být přiřazena do proměnné stejného datového typu. [2]

3.4 Typové (generické) kolekce

Typové kolekce jsou definovány v pracovním prostoru *System.Collections.Generic*. Tyto typové třídy (kolekce) jsou tzv. *typově zabezpečené*. Protože každá taková kolekce pracuje vždy s nějakým konkrétním datovým typem, programátor není nucen explicitně přetypovávat prvky v kolekci na příslušný datový typ, jak tomu bylo u beztypové

kolekce. Často jsou tyto typové kolekce označovány jako bezpečnější a řazeny do typové skupiny tzv. *šablonové*.

List<T> - je podobná kolekci ArrayList, ale jedná se o typovou třídu. Je to jedna z nejvíce používaných typových tříd. Jedná se o pole. Ke každému prveku v této kolekci je možno přistupovat pomocí indexu. Pokud je kapacita pole vyčerpána, je automaticky přealokována a zvětšena na dvojnásobek.

Dictionary<T> je to typová obdoba hešové tabulky. [2]

SortedDictionary<T> - funguje podobně jako SortedList. To znamená, že opět ukládá dvojici klíč a hodnota. Prvek do kolekce ukládá na místo podle hodnoty klíče, tak aby pole bylo pořád seřazené. Vyhledávání v této kolekci je velmi rychlé, protože se hledá podle klíče a binárním vyhledáváním.

Queue<T> – je typ typové kolekce fungující jako fronta.

Stack<T> – funguje jako zásobník.

3.5 Procházení kolekcí

Enumerátory slouží k procházení prvků v kolekci. Při tomto procházení kolekcí se nemusíme starat o konkrétní uložení dat v kolekci. Enumerátory jsou implementovány v každé kolekci díky implementovanému rozhraní *IEnumerable* Jsou definované jako typové a beztypové. Navíc každý enumerátor implementuje tzv *iterátor*, což je klasický návrhový vzor. [2] S ním pak je možné od sebe oddělit definici algoritmů od definice kolekce. Jak ukazuje následující příklad, při definici *GetEnumerátoru* je nutné, aby třída dědila rozhraní *IEnumerable*.

```
class Class1 : IEnumerable
{
    object[] pole1 = null;
    public IEnumerator GetEnumerator()
    {
        foreach (object o in pole1)
        {
            if (o == null) break;
            yield return o;
        }
    }
}
```

Potom můžeme takovou kolekci, která má definici *GetEnumerátoru* procházet užitím enumerátoru jak ukazuje následující příklad.

```
IEnumerator e1 = pole.GetEnumerator();
while (e1.MoveNext())
    Console.Write("{0}", e1.Current);
```

Další možnost je použití příkazu foreach nebo operátoru [].Pro procházení kolekcemi můžeme použít tři způsoby. Můžeme využít foreach, enumerator nebo klasický přístup k poli pomocí indexu. Použijemeli foreach nebo enumerator nemusíme se starat o typ kolekce.

Procházení beztypovými (negenerickými) kolekcemi

```
foreach (object o in pole)
    Console.Write("{0}", o);
```

```
IEnumerator e1 = pole.GetEnumerator();
    while (e1.MoveNext())
        Console.Write("{0}", e1.Current);
```

Procházení typovými (generickými) kolekcemi [2]

```
foreach (Color color in pole)
    Console.Write("{0}", color);

IEnumerator<int> e2 = ((IEnumerable<int>)pole).GetEnumerator();
    while (e2.MoveNext())
        Console.Write("{0}", e2.Current);
```

3.6 Práce s beztypovými (negenerickými) kolekcemi

Zde uvedu způsoby užití nejčastěji používaných metod kolekce.

Při vytváření kolekce není nutno předem zadat velikost vytvářené kolekce.

ArrayList pole = new ArrayList();

.Add(object value) - vložení prvku do pole na první neobsazenou pozici.

.Remove(object value) - odstranění prvku z pole o konkrétní hodnotě.

Pokud se v kolekci vyskytnou duplicity, což znamená že se v ní vyskytují dvě a více stejné hodnoty, je odstraněn prvek s prvním výskytem.

pole.RemoveAt(int index) - odtranění prvku z kolekce podle zadané pozice

Vzhledem k tomu, že kolekce implementují rozhraní *IList*, můžeme k jednotlivým prvkům kolekce přistupovat pomocí operátoru[].[2]

int hodnota = (int) pole[5]; // nesmíme zapomenout přetypovat

Pokud v kolekci provádíme operaci vyhledávání a prvek není nalezen, metoda vrací hodnotu -1. V opačném případě metoda *IndexOf* vrací pozici, kde se prvek v kolekci vyskytuje. Pokud je v kolekci více hodnot stejné velikosti, vrací metoda pozici prvního výskytu.

int IndexOf(object value) - vyhledá prvek o konkrétní hodnotě

Metoda *Sort* řadí prvky v kolekci podle implicitního kritéria. V případě že do kolekce ukládáme vlastní datový typ, přesněji řečeno vlastní třídu, nesmíme zapomenout také nadefinovat vlastní komparátor řazení, jináč by metoda *Sort* nevěděla podle jakých kritérií řadit.

.*Sort* - seřadí kolekci

3.7 Práce s typovými (generickými) kolekcemi

Při vytváření kolekce nemusíme uvést velikost s jakou se má kolekce vytvořit. Nesmíme však zapomenout uvést datový typ s jakým bude kolekce pracovat.

List<int> cisla = new List<int>();

Přidávání prvků do typové kolekce je analogické jako u beztypové kolekce, nesmíme však zapomenout přidávat konkrétní datový typ. Prvky jsou opět v kolekci řazeny podle toho v jakém pořadí byly do kolekce přidávány.

cisla.Add(int value) - přidá prvek o konkrétní hodnotě

cisla.Remove(int value) - odstraní prvek o konkrétní hodnotě

cisla.RemoveAt(int index) - odstraní prvek z konkrétní pozice

V případě zadání indexu mimo rozsah, to jest mimo velikost kolekce, je generována vyjímka *ArgumentOutOfRangeException*.

Analogicky jako u beztypových kolekcí tak i u typových kolekcí, které implementují rozhraní *IList<T>*, můžeme k jednotlivým prvkům kolekce přistupovat pomocí operátoru []. Ovšem zde se nemusíme starat o přetypování.

int hodnota = cisla.IndexOf(4)

int IndexOf(int value) - vyhledání prvku o konkrétní hodnotě

Tato metoda nám vrátí pozici prvního výskytu v kolekci uvedené hodnoty *value*.

3.8 Způsoby řazení kolekce

Metoda Sortseřazuje prvky v kolekci podle implicitního kritéria, které je definovány pro hodnotové datové typy. Pokud tedy do kolekce uložíme vlastní datový typ, nebo chceme řadit podle jiných kritérií, musíme nadefinovat vlastní komparátor. Komparátor můžeme nadefinovat dvěmi způsoby. Buď vytvoříme třídu *Comparer*, která bude implementovat rozhraní *IComparer* a bude v ní definována metoda *Compare*. Druhou možností je, aby námi vytvořená vlastní třída pro kterou chceme vytvořit komparátor, implementovala rozhraní *IComparable* a v této třídě ještě nadefinovat metodu *CompareTo*. Pokud bychom totiž pro vlastní třídu kterou ukládáme do kolekce nenadefinovali vlastní komparátor, a zavolali metodu *Sort*, byla by vyvolána vyjímka, protože kolekce by nevěděla podle čeho řadit.

Způsoby řazení negenerické (beztypové) kolekce [5]

a) vlastní definice třídy `Compare(object x, object y)`

```
class Comparer1 : IComparer
{
    public int Compare(object x, object y)
    {
        if (x > y) return 1;
        if (x < y) return -1;
        else return 0;
    }
}
pole2.Sort(new Comparer1());
```

b) vlastní definice třídy `CompareTo(object obj)` ve třídě `Color` [5]

```
class Color : IComparable
{
    public int CompareTo(object obj)
    {
        if (this.object > obj) return 1;
        if (this.object < obj) return -1;
        else return 0;
    }
}
pole2.Sort();
```

Způsoby řazení generické (typové) kolekce

a) vlastní definice třídy `Compare(Color x, Color y)`

```
class Comparer2 : IComparer<Color>
{
    public int Compare(Color x, Color y)
    {
        if (x > y) return 1;
        if (x < y) return -1;
        else return 0;
    }
}
pole2.Sort(new Comparer2());
```

b) vlastní definice třídy `CompareTo(Color color)` ve třídě `Color`

```
class Color : IComparable<Color>
{
    public int CompareTo(Color color)
    {
        if (this.color > color) return 1;
        if (this.color < color) return -1;
        else return 0;
    }
}
pole2.Sort();
```

3.9 Způsoby vyhledávání v kolekci

Používá se metoda *IndexOf* a *Contains*.V případě že v kolekci chceme vyhledat prvek a opět v kolekci máme uloženou vlastní třídu neboli vlastní datový typ, musíme v této třídě překrýt virtuální metodu *Equals*.

Způsob vyhledávání v negenerické (beztypové) kolekci

```
public override bool Equals(object obj) {
    //return base.Equals(obj);
    if (this.object == obj) return true;
    else return false;
}
```

Způsob vyhledávání v generické (typové) kolekci

```
public override bool Equals(Color color)
{
    //return base.Equals(color);
    if (this.color == color) return true;
    else return false;
}
```

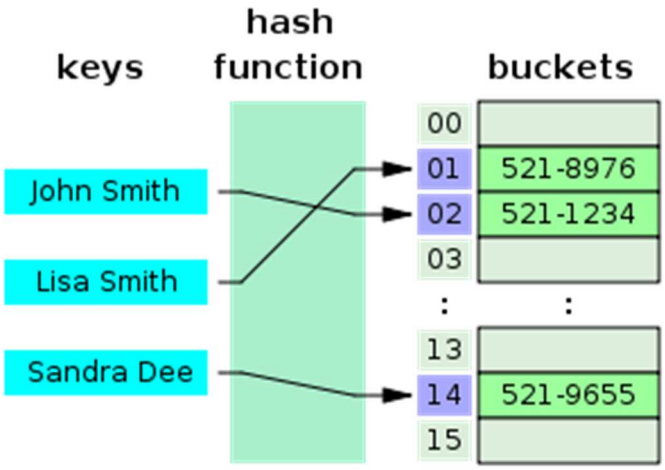
3.10 Indexery

Pokud ve třídě nadefinujeme indexer, můžou být pak instance této třídy indexovány jako pole. Ke každé instanci této třídy lze potom přistupovat pomocí operátoru []. [5]

```
class Color1
{
    private Color[] pole1;
    public Color this[int index]
    {
          get { return pole1[index]; }
          set { pole1[index] = value; }
    }
}
```

3.11 Třída Hashtable a metoda GetHashCode

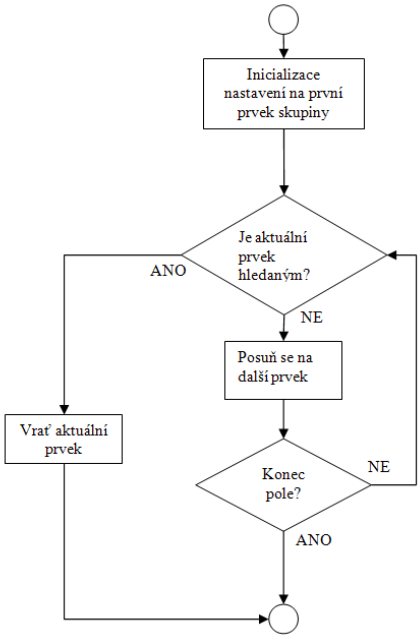
Třída *Hashtable* slouží k velmi rychlému vyhledávání prvků v kolekci podle klíče. Hashová tabulka se někdy také nazývá jako asociativní pole. Základem hashové tabulky je tzv. *hashovací funkce*, která vygeneruje celočíselnou hodnotu. Tento klíč je vygenerován pro každou instanci třídy. Způsob jakým metoda vytvoří klíč je takový, že se do tohoto klíče zhustí obsah objektu. Zároveň se nevylučuje duplicita hashového kódu, ale pravděpodobnost je velmi nízká. Důvod užití hashového kódu je, aby se při hledání konkrétního prvku v hashové tabulce velmi zredukovalo množství prohledávaných prvků. Princip je takový, že při hledání konkrétního prvku se nejdříve zjistí hashový kód prvku a pak se již prohledávají prvky jen ty, které mají stejný hashový kód. Protože často prvků s identickým hashovým kódem je málo, prohledávání je často velice rychlé. [5]



Obrázek 1: Telefonní seznam jako hashová tabulka [9] [10]

3.12 Způsoby vyhledávání

U způsobu vyhledávání prvku v kolekci musíme rozlišovat dva případy kolekce. Buď jestli operaci vyhledávání provádíme na již seřazené kolekci, nebo na neseřazené kolekci.



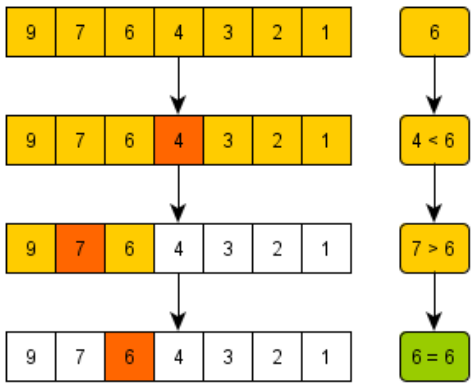
Obrázek 2: Vývojový diagram sekvenčního vyhledávání [11]

Sekvenční vyhledávání

Tento způsob vyhledávání je někdy označovaný jako lineární vyhledávání, pracuje na metodě postupného procházení všech objektů v kolekci. Sekvenční vyhledávání má časovou složitost O(n). Sekvenční vyhledávání má velkou výhodu, že ho lze použít na neseřazenou kolekci.

Binární vyhledávání

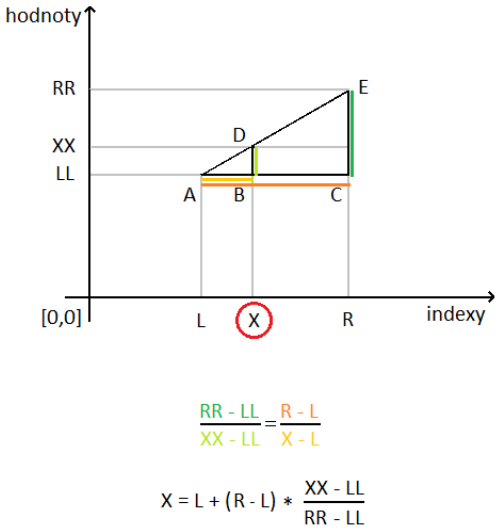
Pokud máme kolekci již seřazenou, můžeme místo sekvenčního vyhledávání, provést binární vyhledávání. Tím velmi zrychlíme proces vyhledávání. Binární vyhledávání funguje tak, že najde prostřední prvek v poli (medián), ten porovná s hledanou hodnotou, a podle toho, jestli hledaný prvek je větší nebo menší než medián, hledá v horní nebo dolní polovině pole. Celý postup se následně opakuje. Algoritmus binárního vyhledávání je často označovaný jako typ rozděl a panuj. Při vyhledávání, v nejhorším případě u binárního vyhledávání je složitost $O(\log_2 n)$.



Obrázek 3: Princip binárního vyhledávání [8]

Interpolační vyhledávání

Rychlost vyhledávání můžeme ještě zvýšit, když u původního binárního vyhledávání místo metody půlení intervalu, použijeme metodu interpolační. Pokud tedy hledaná hodnota je blíže k levé krajní hodnotě v hledaném poli, než k pravé krajní hodnotě, můžeme místo volby prostředního prvku v poli zvolit poměrově hodnotu bližší k levé krajní hodnotě. Samozřejmě i tuto operaci, stejně jako u binárního vyhledávání, můžeme provádět pouze na seřazené kolekci. Tuto metodu prohledávání lze přirovnat k metodě, jako když hledáme ve slovníku.



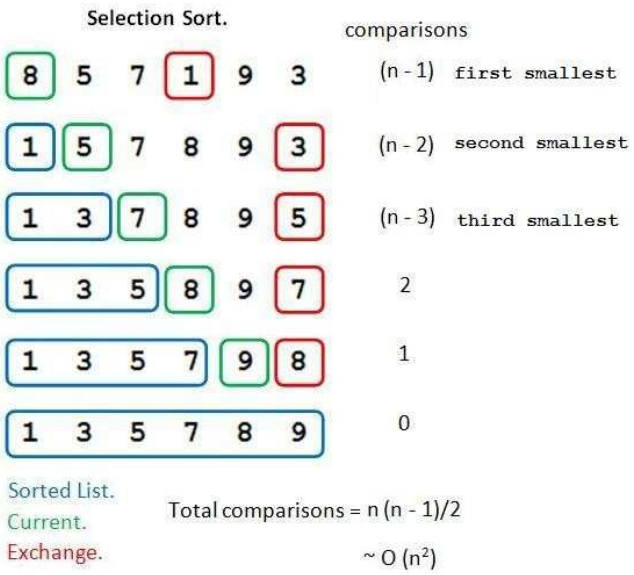
Obrázek 4: Princip interpolačního vyhledávání [12]

3.13 Způsoby řazení

V této práci se mimo jiné budeme zabývat různýma metodama řazení prvků v kolekci. Jedním z hlavních úkolů této práce je, porovnat jednotlivé metody řazení podle rychlosti, s jakou danou operaci provedou.

SelectionSort

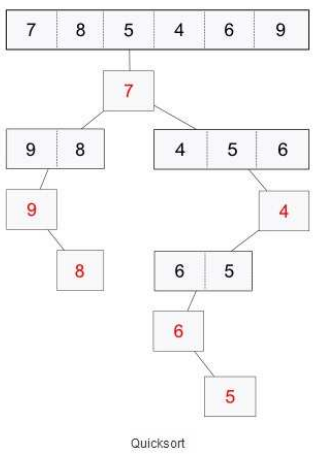
Tento algoritmus řazení pracuje na principu, že prochází sekvenčně pole a hledá v něm největší prvek. Až pole projde a má tedy najitý největší prvek, tak ho prohodí s krajním prvkem. Potom pole začne znovu prohledávat a hledat v něm opět největší prvek, ovšem toto pole už nezahrnuje krajní prvek, je tedy o jeden prvek menší. Celý algoritmus se opakuje, dokud prohledávané pole nemá velikost jedna. Složitost je $O(n^2)$. [5]



Obrázek 5: Princip SelectionSort řazení [13]

QuickSort

QuickSort funguje tak, že v řazené kolekci (poli) vybereme jakýkoliv prvek, můžeme ho pojmenovat třeba pivot, a pole uspořádáme tak, že prvky pole, které jsou menší než pivot dáme do levé části od pivota, a prvky větší než pivot dáme do pravé části od pivota. Tuto operaci potom rekurzivně opakujeme na obě rozdělené části pole, kromě pivota, dokud každé pole nemá velikost jednoho prvku. V takovém případě už je pole seřazené.

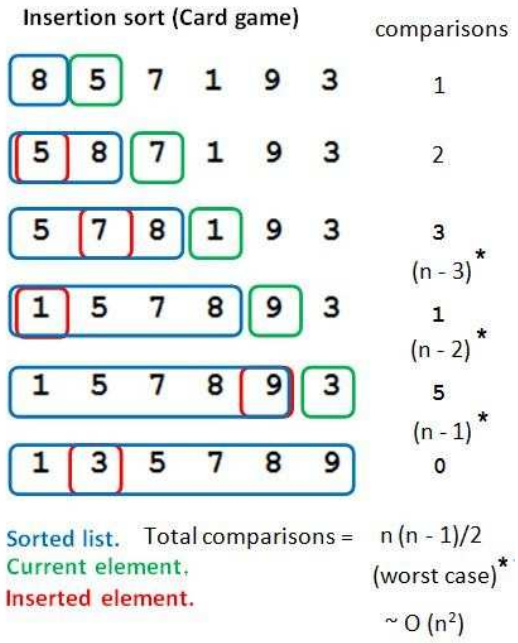


Obrázek 6: Princip QuickSort řazení [14]

Rychlost řazení u QuickSortu je hlavě určená dobrou volbou pivotu. Pokud je pivot zvolen vhodně, tedy ideálně, složitost je $O(n.\log_2 n)$. [5] Ovšem pokud pivota zvolíme nešťastně, tedy největší nebo nejmenší prvek, složitost je $O(n^2)$.

InsertionSort

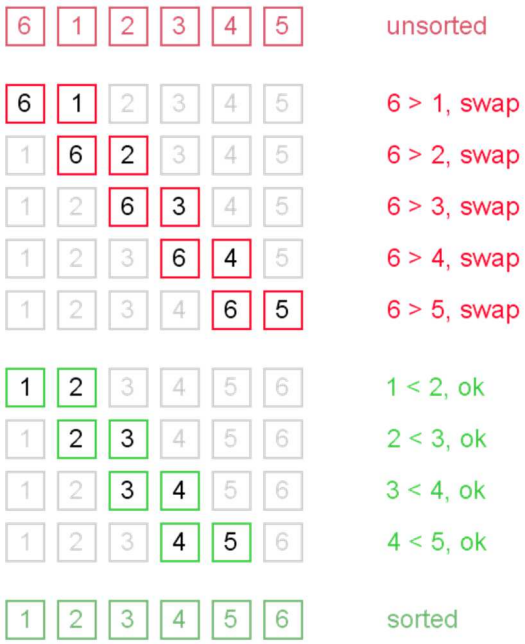
Princip tohoto řadícího algoritmu je postaven na myšlence, že pokud máme pole o jednom prvku, je automaticky seřazené. Potom do takto seřazeného pole přidáme další prvek na již správné místo, aby pole zůstalo seřazené. Tento postup opakujeme dokud do takovéhoto pole nezařadíme všechny prvky. Složitost je $O(n^2)$. [5]



Obrázek 7: Princip InsertionSort řazení [13]

BubbleSort

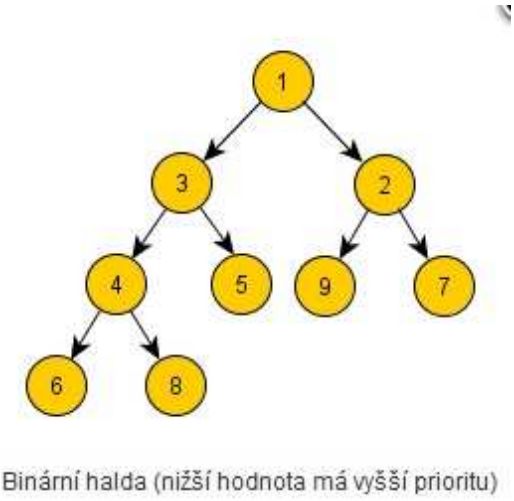
Tento algoritmus řazení funguje tak, že procházíme postupně pole, které chceme řadit, a vždy porovnáváme dva sousední prvky. Pokud je prvek s nižší hodnotou vlevo, než prvek v pravo, prvky prohodíme. Takovýmto postupem prvek s nejmenší hodnotou v poli probublá na konec pole. Celý postup opakujeme od začátku, ovšem na pole, kde už neuvažujeme ten nejmenší prvek, který nám probublal na konec, ten má již správnou pozici. Složitost je $O(n^2)$. [5]



Obrázek 8: Princip BubbleSort řazení [15]

HeapSort

Je považován za jeden z nejúčinnějších algoritmů řazení. Jeho složitost se blíží k $O(n.\log_2 n)$. [5] Základem tohoto algoritmu je binární halda, proto zde uvedu nejprve obrázek a popis co binární halda je.



Obrázek 9: Binární halda [16]

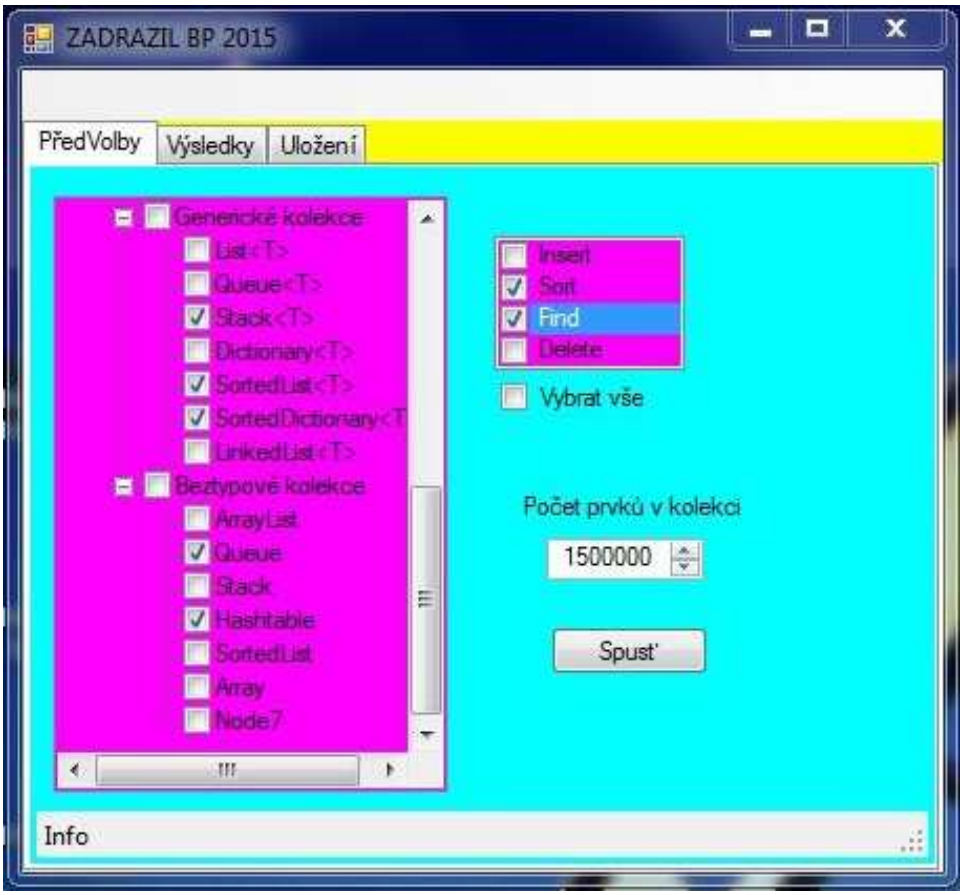
V binární haldě platí pravidlo, že každý otec má pokaždé větší hodnotu než oba jeho potomci. Dále zde platí pravidlo, že i každý podstrom binární haldy je také binární halda. Binární haldu můžeme implementovat polem tak, že pro každého otce s indexem *i*, který je uložen v poli platí, že má potomky v poli uložený na indexu $2i+1$ a $2i+2$, za předpokladu, že indexujeme od nuly.

Algoritmus *HeapSort* používá metodu *RepairTop*, která spravuje binární haldu, to znamená, pokud je porušeno pravidlo, že otec má větší hodnotu než oba jeho potomci, tak otce prohodí s potomkem, který má nejvyšší hodnotu. Samozřejmě při této operaci se může přenést tato nerovnováha o úroveň níž, tedy je nutné případně opravit i jeho podstromy. Tento postup končí v případě, že otec už nemá žádné potomky, nebo že oba potomci mají nižší hodnotu. Samotný algoritmus *HeapSort* tedy začíná voláním metody *RepairTop*, kterou opraví všechny binární podhaldy. Nyní může začít samotný algoritmus řazení. Vezmeme prvek z vrcholu binární haldy, což je prvek s největší hodnotou a prohodíme ho s posledním prvkem binární haldy. Nyní binární haldu zmenšíme o *1*, protože na konci už máme seřazený prvek s největší hodnotou. Potom zavoláme metodu *RepairTop*, protože je nyní po prohození prvku nutné binární haldu opravit. Opět vezmeme prvek z vrcholu binární haldy a celý postup opakujeme, dokud řazená binární halda nemá velikost jednoho prvku.

4 Řešení

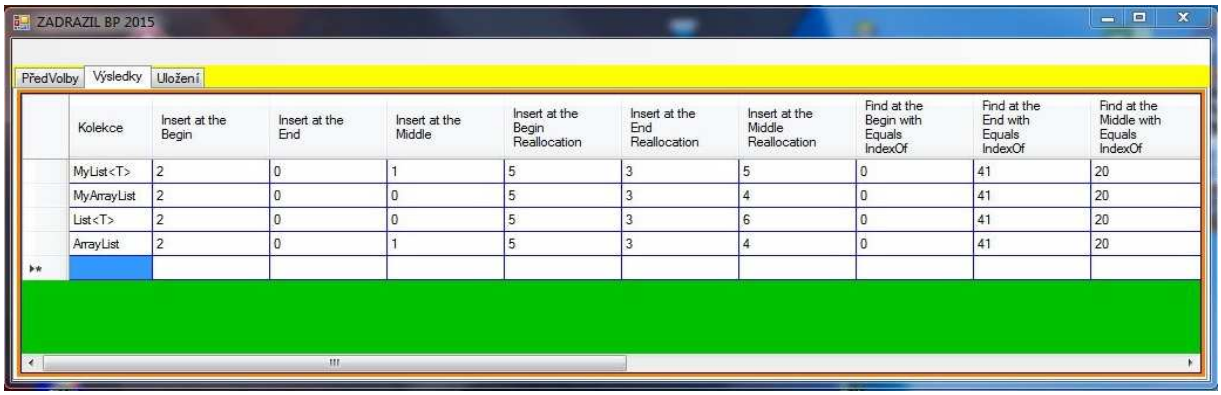
4.1 Testovací aplikace

Pro měření rychlostí kolekcí byla vyvinuta testovací aplikace a to v prostředí .NET/C# s využitím Microsoft Visual C# 2010 Express. Aplikace umožňuje zvolit typ testu (testovací případy), výsledky zapisuje do souboru csv a zobrazuje v gridu.



Obrázek 10: Testovací aplikace - PředVolby

Na obrázku je patrné, že na záložce PředVolby můžeme v komponentě *TreeView* zaškrtnout, všechny kolekce, které chceme testovat a zároveň můžeme v komponentě *CheckedListBox* zaškrtnout, které operace na vybraných kolekcí chceme provádět. Na této záložce je take komponenta *NumericUpDown*, do které můžeme zadat počet prvků se kterými kolekce mají pracovat. Komponenta *ToolStripStatusLabel* nám signalizuje průběh a konec výpočtu. Po skončení výpočtu se můžeme přepnout na druhou záložku Výsledky, kde v komponentě *DataGridView* budou zobrazeny výsledky.



Kolekce	Insert at the Begin	Insert at the End	Insert at the Middle	Insert at the Begin Reallocation	Insert at the End Reallocation	Insert at the Middle Reallocation	Find at the Begin with Equals IndexOf	Find at the End with Equals IndexOf	Find at the Middle with Equals IndexOf
MyList<T>	2	0	1	5	3	5	0	41	20
MyArrayList	2	0	0	5	3	4	0	41	20
List<T>	2	0	0	5	3	6	0	41	20
ArrayList	2	0	1	5	3	4	0	41	20

Obrázek 11: Testovací aplikace - Výsledky

Na tomto obrázku, na záložce Výsledky, jsou zobrazený výsledky s jakou rychlostí každá kolekce provedla svůj úkol. V levém sloupečku jsou názvy kolekcí, které byly testovány, v záhlaví každého sloupečku je popisek jaká operace byla provedena.

4.2 Zvolený způsob řešení

Základní třídou v projektu, která bude řazena a bude ukládaná do kolekcí je vytvořená třída Color. V této třídě jsme mimo jiné, přetížili metodu *Equals* a *CompareTo*. Kód je následující.

```
public override bool Equals(object obj)
{
    //return base.Equals(obj);
    if (obj is Color)
    {
        Color o1 = obj as Color;
        if (o1 == null)
            return false;
        else
        {
            return ((this.r == o1.R) && (this.g == o1.G) && (this.b == o1.B));
        }
    }
    else
    {
        return false;
    }
}
```

```
public int CompareTo(object obj)
{
    if (obj is Color)
    {
        Color o1 = obj as Color;
        if (o1 == null)
            return 0;
        else
        {
            if (this.barva > o1.barva) return 1;
            if (this.barva < o1.barva) return -1;
            return 0;
        }
    }
    else
    {
        return 0;
    }
}
```

Tuto třídu při vytváření objektu plníme bajtovými položkami R, G, B. Data, která používáme pro přiřazení těmto položkám čteme ze souboru Picture1200x1250.BMP. Tuto volbu řešení jsme zvolili z důvodu, že potřebujeme naplnit kolekce náhodnými daty, na kterých budeme následně provádět operace řazení, vyhledávání, vkládání a mazaní. Zmíněný obrázek typu BMP nám vlastně poslouží jako generátor náhodných dat.

4.3 Třída pro měření času

Pro měření času používám třídu *Stopwatch*. Před zahájením měřené operace zavolán metodu *Start*, a po skončení operace zavolám metodu *Stop*. Výsledek v milisekundách mi poskytne metoda *ElapsedMilliseconds*.

```
Stopwatch sw = new Stopwatch();
sw.Start();
// měřený proces
sw.Stop();
sw.ElapsedMilliseconds.ToString();
```

4.4 Přehled testovaných kolekcí a testovaných operací

Kolekce	Vkládání	Řazení	Vyhledávání	Mazání
MyList<T>	A	A	A	A
My2List<T>	A	A	A	A
MyLinkedList<T>	A	A	A	A
MyArrayList	A	A	A	A
My2ArrayList	A	A	A	A
MySortedList	A	N	A	A
My2SortedList	A	N	A	A
List<T>	A	A	A	A
Queue<T>	A	N	A	A
Stack<T>	A	N	A	A
Dictionary<T>	A	A	A	A
SortedList<T>	A	N	A	A
SortedDictionary<T>	A	N	A	A
LinkedList<T>	A	N	A	A
ArrayList	A	A	A	A
Queue	A	N	A	A
Stack	A	N	A	A
Hashtable	A	N	A	A
SortedList	A	N	A	A
Array	A	A	A	A

Tabulka 1: Testovací scénař

Vkládání	Na začátek pole	Bez realokace
	Na konec pole	S realokací
	Do prostřed pole	
Řazení	Neseřazeného pole	
	Seřazeného pole	
Vyhledávání	Na začátku pole	
	Na konci pole	
	Uprostřed pole	
Mazání	Na začátku pole	
	Na konci pole	
	Uprostřed pole	

Tabulka 2: Testovací scénař - pokračování

Operace vkládání (Insert)

Při operaci vkládání (Insert) jsme nejprve při vytváření kolekce naalokovali kolekci o konkrétní kapacitě, například 900 000 prvků, kterou jsme potom naplnili prvky typu *Color* (počet prvků se rovná kapacita minus jedna), tedy v tomto případě počtem prvků 899 999. Na takto připravený případ jsme testovali dobu vkládání prvku na začátek kolekce, na konec kolekce a doprostřed kolekce.

Dále jsme testovali případ, kdy kolekce už má plně využitou svou kapacitu, například kdy má kapacitu 900 000 prvků a je také naplněna 900 000 prvkama. Na takto připravený případ jsme opět testovali dobu vkládání prvku na začátek kolekce, na konec kolekce a doprostřed.

Operace řazení (Sort)

Pro řazení námi vytvořených objektů třídy *Color* byl nadefinován vlastní komparátor. Ve třídě *Color* jsme implementovali rozhraní *IComparable* a přetížili virtuální metodu *CompareTo*.

Ukázka přetížení virtuální metody *CompareTo* ve třídě *Color*.

```
public int CompareTo(object obj)
{
    if (obj is Color)
    {
        Color o1 = obj as Color;
        if (o1 == null)
            return 0;
        else
        {
            if (this.barva > o1.barva) return 1;
            if (this.barva < o1.barva) return -1;
            return 0;
        }
    }
    else
    {
        return 0;
    }
}
```

Druhý způsob jak implementovat vlastní komparátor pro řazení je vytvořit vlastní třídu *Comparer* a v ní vytvořit vlastní metodu *Compare*. Nesmíme zapomenout implementovat rozhraní *IComparer* pro třídu *Comparer*, jak ukazuje následující ukázka.

```
class Comparer1 : IComparer
{
    public int Compare(object x, object y)
    {
        if ((x is Color) && (y is Color))
        {
            Color o1 = x as Color;
            Color o2 = y as Color;
            if ((o1 == null) || (o2 == null))
                return 0;
            else
            {
                if (o1.Barva > o2.Barva) return 1;
                if (o1.Barva < o2.Barva) return -1;
                return 0;
            }
        }
        else
        {
            return 0;
        }
    }
}
```

Dále jsme prováděli případ, kdy už řadíme již seřazené pole, a testovali jak se změní doba potřebná k provedení této operace.

Operace hledání (Find)

Pokud chceme v kolekci provést operaci hledání a máme v ní uložený vlastní datový typ, v našem případě *Color*, musíme ve třídě *Color* také přetížit virtuální metodu *Equals*.

```
public override bool Equals(object obj)
{
    //return base.Equals(obj);
    if (obj is Color)
    {
        Color o1 = obj as Color;
        if (o1 == null)
            return false;
        else
        {
            return ((this.r == o1.R) && (this.g == o1.G) && (this.b == o1.B));
        }
    }
    else
    {
            return false;
    }
}
```

Některé kolekce vedle sekvenčního vyhledávání metodou *IndexOf* nebo *Contains* umožňují i binární vyhledávání metodou *BinarySearch*. Je samozřejmé že binární vyhledávání můžeme provést jen na již seřazené kolekci.

Testovali jsme opět tři případy, kdy je hledaný prvek na začátku kolekce, na konci kolekce a nebo uprostřed kolekce.

Operace mazaní (Delete)

Poslední operace kterou jsme prováděli nad kolekcemi je operace mazání (Delete), tedy odstraňování prvku z kolekce. Opět jsme testovali tři případy, a to kdy je odstraňovaný prvek na začátku kolekce, na konci kolekce a nebo uprostřed kolekce.

4.5 Vlastní implementace kolekcí

Kolekce v .NET/C# jsou implementovány jako třídy. Některé kolekce jako například *List*, *ArrayList* však používají k některým operacím třídu *Array*.

Jako první vlastní implementaci jsme udělali implementaci typové kolekce *List* a nazvali jí *MyList*. I tato kolekce volá k některým svým operacím třídu *Array*. Kolekce implementuje rozhraní *IList*. Pokud je kolekce již plná a je potřeba provést realokaci, tedy zvětšení kapacity provádíme to následovně.

```
if ((value > vychozi_kapacita) && (value > velikost))
{
     T[] NewPolozky = new T[value];
     if (velikost > 0)
     {
          Array.Copy(polozky, 0, NewPolozky, 0, velikost);
     }
     polozky = NewPolozky;
}
```

Po vytvoření nového pole o velikosti value, pak samotné kopírování obsahu starého pole do nového provádíme metodou *Copy* ze třídy *Array*.

Další metodu kterou jsme implementovali do této kolekce je metoda *BinarySearch*.

```
public int BinarySearch(int index, int count, T polozka, IComparer<T> comparer)
{
     return Array.BinarySearch<T>(polozky, index, count, polozka, comparer);
}
```

I zde pro samotné binární vyhledávání voláme metodu *BinarySearch* třídy *Array*, jako to provádí standardní kolekce *List* v .NET/C#. Jako pátý parametr nesmíme zapomenout předat komparátor, protože pracujeme s vlastním datovým typem *Color* a pokud bychom předávali jenom defaultní, metoda by neměla podle čeho porovnávat.

Pro zjištění zda v kolekci již je konkrétní prvek jsme implementovali metodu *Contains*.

```
public bool Contains(T polozka)
{
    if ((T)polozka == null)
    {
        return false;
    }
    else
    {
        for (int i = 0; i < velikost; i++)
        {
            if ((polozky[i] != null) && (polozky[i].Equals(polozka)))
            return true;
        }
        return false;
    }
}
```

Pokud je prvek v kolekci nalezen, vrací se hodnota true, v opačném případě false. Jedná se o sekvenční vyhledávání. Samotné porovnávání prvků se provádí metodou *Equals*, což je důvod proč bylo důležité tuto virtuální metodu v naší třídě *Color* přetížit.

Pro zjištění indexu dané položky v kolekci jsme implementovali do naší kolekce *MyList* metodu *IndexOf*.

```
public int IndexOf(T polozka)
{
    return Array.IndexOf(polozky, polozka, 0, velikost);
}
```

Samotné vyhledávání indexu položky v kolekci se provádí metodou *IndexOf* třídy *Array*.

Pro vkládání prvku do kolekce jsme implementovali metodu *Insert*.

```
public void Insert(int index, T polozka)
{
    if (velikost == polozky.Length) ZajistitCapacitu(velikost + 1);
    if (index < velikost)
             Array.Copy(polozky, index, polozky, index+1, velikost-index);
    polozky[index] = polozka;
    velikost++;
}
```

Na pozici o hodnotě index je potřeba udělat místo pro vkládanou položku, proto při volání metody *Copy* třídy *Array* udáváme do parametru pozici index+1, kde se má nakopírovat zbytek pole. Pak už jenom přiřadíme vkládaný prvek na pozici index. Tady tedy také při manipulaci s polem voláme metodu třídy *Array*.

Pro případ, že chceme odstranit prvek z kolekce jsme implementovali metodu *RemoveAt*, která odstraní prvek z kolekce s patřičným indexem.

```
public void RemoveAt(int index)
{
    velikost--;
    if (index < velikost)
            Array.Copy(polozky, index + 1, polozky, index, velikost - index);
    polozky[velikost] = default(T);
}
```

Jelikož se prvek odstraňuje, zbytek pole kopírujeme na pozici index. Ovšem už bez odstraňovaného prvku, proto je parametr index+1. Na konci pole nám potom vznikl volný prvek, kterému přiřadíme defaultní hodnotu.

Poslední metoda, která stojí za povšimnutí, a kterou jsme implementovali do naší kolekce *MyList* je metoda *Sort*.

```
public void Sort(int index, int count, IComparer<T> comparer)
{
    Array.Sort<T>(polozky, index, count, comparer);
}
```

Zde je opět vidět, že pro samotné řazení kolekce se využívá metoda *Sort* třídy *Array*.

Jak již bylo řečeno, kolekce *List* v .NET/C# a naše implementace této kolekce *MyList* používá k některým svým operacím metody z třídy *Array*. Naše další snažení bylo, že jsme se pokusili nahradit v kolekci všechny tyto metody z třídy *Array* vlastními. Proto jsme vytvořili kolekci *My2List*, která už nepoužívá metody z třídy *Array*, ale tyto metody nahrazuje metodama z námi napsané třídy *AlgoritmyTYP*.

Tato kolekce *My2List* je napsaná analogicky podle kolekce *MyList*, opět pro názornost ukážu v čem se přesně liší.

Pokud je kolekce *My2List* již plná a je potřeba provést realokaci, tedy zvětšení kapacity provádíme to následovně.

```
if ((value > vychozi_kapacita) && (value > velikost))
{
    T[] NewPolozky = new T[value];
    if (velikost > 0)
    {
            AlgoritmyTYP<T>.Copy(polozky, 0, NewPolozky, 0, velikost);
    }
    polozky = NewPolozky;
}
```

Zde stojí za povšimnutí, že pro operaci kopírování obsahu pole již nevoláme metodu *Copy* z třídy *Array*, ale vlastní metodu *Copy* z třídy *AlgoritmyTYP*.

Další metodu kterou jsme implementovali do této kolekce je metoda *BinarySearch*.

```
public int BinarySearch(int index, int count, T polozka, IComparer<T> comparer)
{
   return AlgoritmyTYP<T>.BinarySearch(polozky, index, count, polozka, comparer);
}
```

Opět pro binární vyhledávání voláme již metodu *BinarySearch* z třídy *AlgoritmyTYP*, místo metody volané z třídy *Array*. Všechny parametry však typově zůstaly stejné, stejně jako název metody.

Metoda *Contains* je naprosto stejná významově i vnitřní implementací jako u předešlé třídy *MyList*, protože pro své operace nepoužívá žádnou metodu z třídy *Array*. Proto ji zde již nebudu vypisovat.

Pro zjištění indexu dané položky v kolekci jsme implementovali do naší kolekce *My2List* metodu *IndexOf*.

```
public int IndexOf(T polozka)
{
    return AlgoritmyTYP<T>.IndexOf(polozky, polozka, 0, velikost);
}
```

I zde stojí za povšimnutí, že samotné vyhledávání indexu položky v kolekci se provádí metodou *IndexOf* třídy *AlgoritmyTYP* a ne již metodou z třídy *Array*.

Pro vkládání prvku do kolekce jsme implementovali metodu *Insert*.

```
public void Insert(int index, T polozka)
{
    if (velikost == polozky.Length) ZajistitCapacitu(velikost + 1);
    if (index < velikost)
      AlgoritmyTYP<T>.Copy(polozky, index, polozky, index + 1, velikost - index);
    polozky[index] = polozka;
    velikost++;
}
```

Na pozici o hodnotě index je potřeba udělat místo pro vkládanou položku, proto při volání metody *Copy* naší třídy *AlgoritmyTYP* udáváme do parametru pozici index+1, kde se má nakopírovat zbytek pole. Pak už jenom přiřadíme vkládaný prvek na pozici index. Tady tedy také při manipulaci s polem voláme metodu třídy *AlgoritmyTYP* místo volání metody z třídy *Array*.

Pro případ, že chceme odstranit prvek z kolekce jsme implementovali metodu *RemoveAt*, která odstraní prvek z kolekce s patřičným indexem.

```
public void RemoveAt(int index)
{
    velikost--;
    if (index < velikost)
      AlgoritmyTYP<T>.Copy(polozky, index + 1, polozky, index, velikost - index);
    polozky[velikost] = default(T);
}
```

Jelikož se prvek odstraňuje, zbytek pole kopírujeme na pozici index. Ovšem už bez odstraňovaného prvku, proto je parametr index+1. Na konci pole nám potom vznikl volný prvek, kterému přiřadíme defaultní hodnotu.

Poslední metoda, která stojí za povšimnutí, a kterou jsme implementovali do naší kolekce *My2List* je metoda *Sort*.

```
public void Sort(int index, int count, IComparer<T> comparer)
{
    if (vybertrideni == 1)
          AlgoritmyTYP<T>.BubbleSort(polozky, index, count, comparer);
    if (vybertrideni == 2)
          AlgoritmyTYP<T>.SelectionSort(polozky, index, count, comparer);
    if (vybertrideni == 3)
          AlgoritmyTYP<T>.InsertionSort(polozky, index, count, comparer);
    if (vybertrideni == 4)
          AlgoritmyTYP<T>.HeapSort(polozky, index, count, comparer);
    if (vybertrideni == 5)
          AlgoritmyTYP<T>.QuickSort(polozky, index, count, comparer);
}
```

V této metodě pro operaci řazení opět voláme metody z třídy *AlgoritmyTYP*, ovšem je zde na výběr z pěti možných algoritmů řazení, které jsme také implementovali do naší třídy *AlgoritmyTYP*.

Porovnávání rychlosti neboli výkonosti jednotlivých těchto algoritmů řazení je jeden z hlavních cílů této práce.

Jako další vlastní implementaci jsem provedli implementaci kolekce *LinkedList* a nazvali jí *MyLinkedList*. Jedná se o typovou kolekci, která implementuje rozhraní *ICollection*. Tato naše kolekce *MyLinkedList*, a ani kolekce *LinkedList* v .NET/C# nepoužívá k některým svým operacím třídu *Array*, tak jak to bylo například u kolekce *List*. Je to obousměrně spojový seznam, kdy z každého uzlu můžeme jít nejen na následující, ale i na předchozí. To znamená mimo jiné, že z posledního uzlu můžeme jít na první, který je vlastně následující. I když kolekce *LinkedList* v .NET/C# neobsahuje metody pro řazení, naše vlastní implementace této kolekce je obohacena o algoritmy řazení.

Hlavní částí této kolekce je implementace třídy *SpojovanyListNode*.

```
public class SpojovanyListNode<T>
{
    MyLinkedList<T> list;
    SpojovanyListNode<T> pristi;
    SpojovanyListNode<T> predchozi;
    T polozka;

    public SpojovanyListNode(T value)
    {
        this.polozka = value;
    }
    internal SpojovanyListNode(MyLinkedList<T> list, T value)
    {
        this.list = list;
        this.polozka = value;
    }
    public SpojovanyListNode<T> Pristi
    {
        get
        {
            return pristi;
        }
    }
    public SpojovanyListNode<T> Predchozi
    {
        get
        {
            return predchozi;
        }
    }
    public T Value
    {
        get
        {
            return polozka;
        }
        set
        {
            polozka = value;
        }
    }
}
```

Tato třída *SpojovanyListNode* je tedy jako uzel spojového seznamu, ve kterém kromě hodnoty (*T polozka*) je také uložen odkaz na příští uzel (*SpojovanyListNode<T> pristi*) a na předchozí (*SpojovanyListNode<T> predchozi*).

Jedna z hlavních metod této kolekce, kterou jsme implementovali je metoda *VlozitUzelPred*, která vkládá uzel do spojového seznamu, před zadaný uzel.

```
private void VlozitUzelPred(SpojovanyListNode<T> Uzel,
                            SpojovanyListNode<T> NovyUzel)
{
    NovyUzel.pristi = Uzel;
    NovyUzel.predchozi = Uzel.predchozi;
    Uzel.predchozi.pristi = NovyUzel;
    Uzel.predchozi = NovyUzel;
    count++;
}
```

Další důležitá metoda, kterou jsme implementovali, a která používá tuto metodu *VložitUzelPred*, je metoda *AddBefore*. Tato metoda před zavoláním metody *VlozitUzelPred* nejprve uzel vytvoří a přes parametr konstruktorem mu předá hodnotu.

```
public SpojovanyListNode<T> AddBefore(SpojovanyListNode<T> uzel, T value)
{
    SpojovanyListNode<T> vysledek = new SpojovanyListNode<T>(uzel.list, value);
    VlozitUzelPred(uzel, vysledek);
    if (uzel == hlava)
    {
         hlava = vysledek;
    }
         return vysledek;
}
```

Pokud chceme vyhledat uzel ve spojovém seznamu o konkrétní hodnotě, můžeme to udělat metodou *Find*.

```
public SpojovanyListNode<T> Find(T value)
{
    SpojovanyListNode<T> uzel = hlava;
    if (uzel != null)
    {
         if (value != null)
         {
             do
             {
                   if ((uzel.polozka != null) && (uzel.polozka.Equals(value)))
                        return uzel;
                   if (uzel.pristi == null) return null;
                   uzel = uzel.pristi;
             } while (uzel != hlava);
         }
         else
         {
             return null;
         }
    }
    return null;
}
```

Jedná se o sekvenční prohledávání, kdy přecházíme z jednoho uzlu na následující a porovnáváme hodnotu uloženou v příslušném uzlu. Zde je vidět, že pro porovnávání používáme metodu *Equals*, kterou jsme přetížili ve třídě *Color*.

Pro zjištění, zdali se vyskytuje v seznamu uzel s konkrétní hodnotou, aniž by jsme chtěli vrátit odkaz na tento uzel, jsme implementovali metodu *Contains*. Tato metoda používá předešlou metodu *Find*.

```
public bool Contains(T value)
{
    return (Find(value) != null);
}
```

Pro odstranění samotného uzlu z kolekce jsme implementovali vnitřní metodu *OdstranUzel*, které předáváme jako parameter odstraňovaný uzel.

```
internal void OdstranUzel(SpojovanyListNode<T> uzel)
{
    if (uzel.pristi == uzel)
    {
        hlava = null;
    }
    else
    {
        uzel.pristi.predchozi = uzel.predchozi;
        uzel.predchozi.pristi = uzel.pristi;
        if (hlava == uzel)
        {
            hlava = uzel.pristi;
        }
    }
    uzel.Zrusit();
    count--;
}
```

Tuto metodu *OdstranUzel* a také metodu *Find* používá další naše implementace metody *Remove*. Tato metoda odstraňuje uzel ze spojového seznamu o konkrétní hodnotě.

```
public bool Remove(T value)
{
    SpojovanyListNode<T> uzel = Find(value);
    if (uzel != null)
    {
        OdstranUzel(uzel);
        return true;
    }
    return false;
}
```

V případě, že prvek nebyl ze seznamu odstraněn vrací metoda hodnotu false.

Poslední metoda která stojí za zmíňku, a kterou jsme implementovali do spojového seznamu je metoda *Sort*. Jak jsem již uvedl, kolekce *LinkedList* v.NET/C# nemá implementovanou metodu *Sort* a naše implementace metody *Sort* je takový nadstandard, který používá metody řazení implementované též v této kolekci *MyLinkedList*.

```
public void Sort(int index, int count, IComparer<T> comparer)
{
    if (vybertrideni == 1) BubbleSort(hlava, index, count, comparer);
    if (vybertrideni == 2) SelectionSort(hlava, index, count, comparer);
}
```

Nesmíme zapomenout na třetí parametr, komparátor, aby naše implementace řazení mohla prvky mezi sebou porovnávat.

Další naše vlastní implementace kolekce je implementace beztypové kolekce *ArrayList*, kterou jsme nazvali *MyArrayList*. I tato kolekce volá k některým svým operacím třídu *Array*. Implementace této kolekce *MyArrayList* je velmi podobná implementaci kolekce *MyList*, až na jeden zásadní rozdíl. Jedná se o beztypovou kolekci. Proto nemá smysl tady vypisovat znovu všechny metody, které byly uvedeny v předchozích odstavcích kolekce *MyList*, ale upozorním jen na rozdíly.

Např. při vytváření pole už vytváříme beztypové pole, což znamená že každý element pole je typu *Object*, stejně tak jako parametr metodám předáváme položku typu *Object*.

```
Object[] NewPolozky = new Object[value];
```

Jako další vlastní implementaci kolekce jsme implementovali kolekci *ArrayList* a nazvali jí *My2ArrayList*, která se ovšem liší od kolekce *ArrayList* v .NET/C# a naší vlastní implementace kolekce *MyArrayList* tím, že nepoužívá k některým svým operacím metody třídy *Array*, ale používá k takovým operacím metody z naší třídy *Algoritmy*. Je velmi podobná implementaci *My2List*, jen s tím rozdílem že se jedná o beztypovou kolekci. Proto je zde stejný rozdíl jako v předešlém odstavci.

Při vytváření pole vytváříme beztypové pole.

```
Object[] NewPolozky = new Object[value];
```

A při volání metod, např. *Add*, kde se dosazuje za parameter hodnota, se používá datový typ též *Object*.

```
public virtual int Add(Object value)
{
}
```

MySortedList je další vlastní implementace beztypové kolekce, která má mít podobnou funkci jako beztypová kolekce *SortedList* v .NET/C#. Každý prvek v této kolekci má kromě své hodnoty i klíč, podle kterého je v kolekci řazen. Při vkládání prvku do této kolekce zařazujeme tedy prvek podle klíče do již seřazené kolekce, tak aby kolekce

zůstala stále seřazená. Tato kolekce implementuje rozhraní *IDictionary*. I tato kolekce pro některé své operace volá některé metody ze třídy *Array*.

V této kolekci, na rozdíl od předešlých kolekcí typu *List* a *ArrayList*, musíme mít ve třídě definované dvě pole, jedno pro uložení klíče a druhé pro uložení hodnoty.

```
private Object[] klice;
private Object[] hodnoty;
```

Pro přidání prvku do kolekce s konkrétním klíčem a konkrétní hodnotou jsme implementovali soukromou metodu *Vloz*.

```
private void Vloz(int index, Object klic, Object hodnota)
{
    if (velikost == klice.Length) ZajistitCapacitu(velikost + 1);
    if (index < velikost)
    {
        Array.Copy(klice, index, klice, index + 1, velikost - index);
        Array.Copy(hodnoty, index, hodnoty, index + 1, velikost - index);
    }
    klice[index] = klic;
    hodnoty[index] = hodnota;
    velikost++;
}
```

Je podobná stejnojmenné metodě v kolekci *List* a *ArrayList* s tím rozdílem, že tato metoda pracuje s dvěmi poli. Dále je vhodné si všimnout, že parametry metody klic i hodnota je datového typu *Object*. Pro to, aby mohl být prvek na pozici index přidán, je nutné všechny prvky v obou polí od indexu výš posunout o jedna.

Pokud při vkládání nového prvku je potřeba zajistit kapacitu, tedy zvýšit velikost obou polí udělá se to následovně.

```
if ((value > vychozi_kapacita) && (value > velikost))
{
    Object[] NewKlice = new Object[value];
    Object[] NewHodnoty = new Object[value];
    if (velikost > 0)
    {
        Array.Copy(klice, 0, NewKlice, 0, velikost);
        Array.Copy(hodnoty, 0, NewHodnoty, 0, velikost);
    }
    klice = NewKlice;
    hodnoty = NewHodnoty;
}
```

Tedy vytvoříme dvě nové pole s požadovanou kapacitou a všechny prvky ze starých polí do něj nakopírujeme.

Dále jsme nadefinovali veřejnou metodu *Add*, která přidá prvek do kolekce. Tato metoda ovšem před zavoláním metody *Vloz*, zavolá metodu *BinarySearch* a zjistí jestli už není prvek se stejným klíčem vložen v kolekci.

```
public virtual void Add(Object klic, Object hodnota)
{
    int index = Array.BinarySearch(klice, 0, velikost, klic, comparer);
    if (index >= 0) return;  // prvek již je v SortedListu
    Vloz(~index, klic, hodnota);
}
```

Pokud je vrácena nula nebo kladná hodnota, prvek již v kolekci je. Pokud je však vrácena záporná hodnota na kterou je aplikován operátor ~ , tak v této proměné máme uloženou pozici kam do kolekce zařadit prvek. Zbývá podotknout, že metoda *BinarySearch* je volaná ze třídy *Array*.

Pro hledání prvku v kolekci podle klíče jsme implementovali veřejnou metodu *IndexOfKey*.

```
public virtual int IndexOfKey(Object klic)
{
    int ret = Array.BinarySearch(klice, 0, velikost, klic, comparer);
    if (ret >= 0) return (ret);
    return (-1);
}
```

Stojí za povšimnutí, že pracujeme se seřazenou kolekcí, na kterou můžeme z tohoto důvodu aplikovat binární vyhledávání. Pokud metoda *BinarySearch* volaná ze třídy *Array* vrátí nulu nebo kladné číslo, znamená to, že vrací pozici v kolekci hledaného prvku.

Pro případ, že by jsme chtěli v kolekci hledat podle hodnoty uloženého prvku, implementovali jsme metodu *IndexOfValue*.

```
public virtual int IndexOfValue(Object hodnota)
{
    return Array.IndexOf(hodnoty, hodnota, 0, velikost);
}
```

Zde je volaná metoda *IndexOf* třídy *Array*.

Pro odstranění prvku z kolekce o konkrétní pozici jsme implementovali veřejnou metodu *RemoveAt*. Od pozice index+1 do konce pole, musíme všechny prvky posunout o jedna do leva. Pro kopírování pole opět využíváme metodu *Copy* ze třídy *Array*.

```
public virtual void RemoveAt(int index)
{
    velikost--;
    if (index < velikost)
    {
        Array.Copy(klice, index + 1, klice, index, velikost - index);
        Array.Copy(hodnoty, index + 1, hodnoty, index, velikost - index);
    }
    klice[velikost] = null;
    hodnoty[velikost] = null;
}
```

Vzhledem k tomu, že máme již nadefinovanou metodu *RemoveAt* a metodu *IndexOfKey*, mohli jsme také nadefinovat metodu *Remove*, která odstraní prvek z kolekce o konkrétním klíči.

```
public virtual void Remove(Object klic)
{
    int index = IndexOfKey(klic);
    if (index >= 0) RemoveAt(index);
}
```

Naše vlastní implementace kolekce *MySortedList*, která má plnit stejnou funkci jako kolekce *SortedList* v .NET/C# pro některé své operace volá metody jako *BinarySearch* a *Copy* ze třídy *Array*. Proto jsme napsali upravenou verzi *MySortedListu* a nazvali tuto kolekci *My2SortedList*, která již nevolá metody z třídy *Array*, ale místo toho volá metody z naší třídy *Algoritmy*. Nemá smysl tady znovu psát celou implementaci, která je podobná implementaci kolekce *MySortedListu*, proto uvedu jen rozdíly.

Pro operaci kopírování místo volání metody

```
Array.Copy(klice, 0, NewKlice, 0, velikost);
```

volám metodu

```
Algoritmy.Copy(klice, 0, NewKlice, 0, velikost);
```

Místo volání metody

```
int index = Array.BinarySearch(klice, 0, velikost, klic, comparer);
```

volám metodu

```
int index = Algoritmy.BinarySearchKey(klice, 0, velikost, klic, comparer);
```

A místo volání metody

```
Array.IndexOf(hodnoty, hodnota, 0, velikost);
```

volám metodu

```
Algoritmy.IndexOf(hodnoty, hodnota, 0, velikost)
```

Třídu *AlgoritmyTYP*, kterou jsme vytvořily, je používána naší implementací typové kolekce *My2List*. Jedná se především o operace *Copy*, *BinarySearch*, *IndexOf*, *Sort*, pro které kolekce *MyList* volá metody z třídy *Array*. Tedy tato třída *AlgoritmyTYP* by měla suplovat třídu *Array*.

Implementace metody *Copy* ve třídě *AlgoritmyTYP*.

```
public static void Copy(T[] ZdrojPolozky, int ZdrojIndex, T[] CilPolozky, int
                        CilIndex, int delka)
{
    for (int i = 0; i < delka; i++)
    {
        CilPolozky[CilIndex + i] = ZdrojPolozky[ZdrojIndex + i];
    }
}
```

Implementace metody *BinarySearch*

Binární vyhledávání probíhá na již seřazeném poli tak, že se prohledávaný interval rozpůlí a podle velikosti hledané hodnoty se již hledá buď v horní nebo dolní půlce intervalu. Pro porovnání, jestli se již hledaná hodnota nalezla se používá metoda *Equals* ve třídě *Color*. Dále musíme mít definovaný vlastní komparátor, aby jsme měli podle čeho porovnávat náš datový typ *Color*. Pokud je hledaná hodnota nalezena, metoda vrací index položky v poli.

```
public static int BinarySearch(T[] polozky, int StartIndex, int Count, T polozka,
        IComparer<T> comparer)
{
    int left = StartIndex;
    int right = StartIndex + Count - 1;
    int center;
    while (left <= right)
    {
        center = (left + right) / 2;
        if (polozky[center].Equals(polozka)) return center;
        else
            if (comparer.Compare(polozky[center], polozka) == 1)
                right = center - 1;
            else
                left = center + 1;
    }
    return -1;
}
```

Implementace metody *IndexOf*

Jedná se o sekvenční prohledávání, kde používáme metodu *Equals* ze třídy *Color*.

```
public static int IndexOf(T[] polozky, T Hodnota, int StartIndex, int
        Count)
{
    for (int i = StartIndex; i < StartIndex + Count; i++)
    {
        if (polozky[i].Equals(Hodnota)) return i;
    }
    return -1;
}
```

Implementace metod řazení

Dále jsme do třídy *AlgoritmyTYP* implementovali metody řazení *BubbleSort*, *SelectionSort*, *InsertionSort*, *HeapSort*, *QuickSort*. Jeden z hlavních úkolů této práce je porovnat časy potřebné pro řazení mezi těmito metodami řazení.

Vlastní implementace třída *Algoritmy* je používaná našimi implementacemi beztypových kolekcí *My2ArrayList* a *My2SortedList*. Je velmi podobná třídě *AlgoritmyTYP*, proto zde uvedu jenom rozdíly v kterých se obě třídy liší. Opět tato třída *Algoritmy* by měla suplovat třídu *Array*.

Místo definice typového pole

```
T[] polozky
```

se používá definice beztypového pole

```
Object[] polozky
```

Analogicky pro komparátor místo definice typového komparátoru

```
IComparer<T> comparer
```

používáme definici beztypového komparátoru

```
IComparer comparer
```

Navíc jsem tuto třídu *Algoritmy* obohatil o metodu *BinarySearchKey*, která ve třídě *AlgoritmyTYP* není implementovaná. Tuto metodu používá pouze naše implementace kolekce *My2SortedList*.

Implementace metody *BinarySearchKey*

Je podobná metodě *BinarySearch* s tím rozdílem, že vrací v případě nenalezení prvku pozici kam lze prvek s tímto klíčem vložit, aby zůstalo pole stále seřazené podle klíče. Těsně před vrácením této hodnoty je ještě na tuto proměnou použit operátor ~.

```
public static int BinarySearchKey(Object[] polozky, int StartIndex, int Count,
        Object polozka, IComparer comparer)
{
    int left = StartIndex;
    int right = StartIndex + Count - 1;
    int center;
    while (left <= right)
    {
        center = (left + right) / 2;
        if ((int)polozky[center] == (int)polozka) return center;
        else
            if ((int)polozky[center] > (int)polozka)
                right = center - 1;
            else
                left = center + 1;
    }
    return ~left;    // tylda a levy !!!
}
```

Tím, že na proměnou byl použit operator ~, se vrací záporná hodnota, a tedy poznáme, že je vrácena pozice pro uložení prvku.

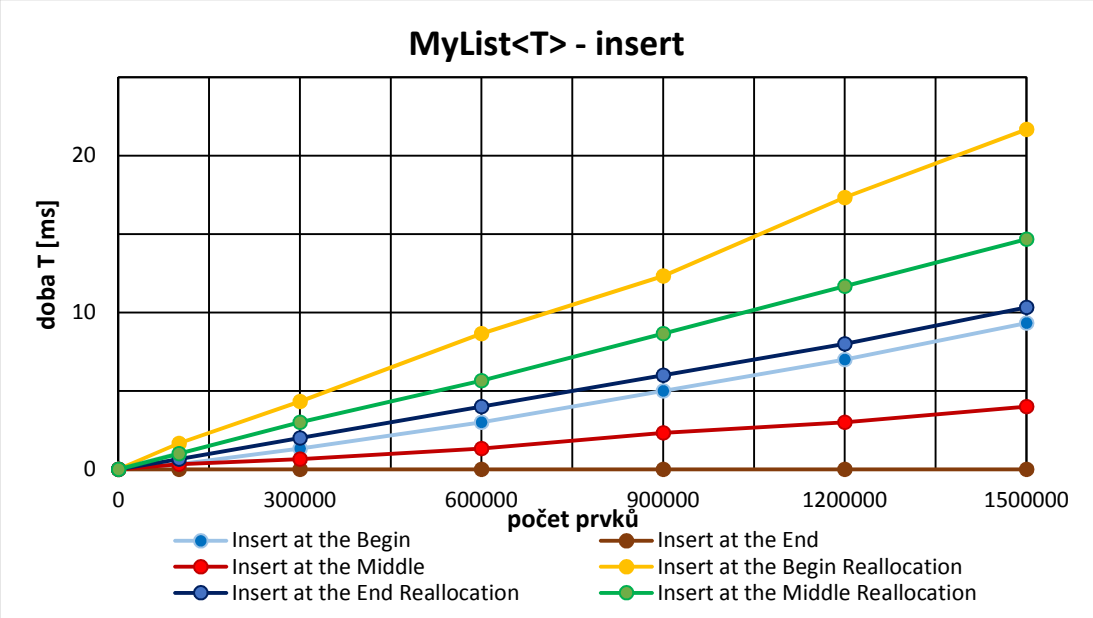
5 Výsledky a diskuse

5.1 Vlastní implementace

Vlastní implementace kolekce MyList<T>

Operace vkládání (Insert)

Na tomto grafu je vidět, že operace vkládání prvku na konec kolekce bez realokace je okamžitá, což potvrzuje způsob naší implementace. Naproti tomu, pokud vkládáme do prostřed kolekce, kdy musíme posunout polovinu pole o index 1, časová režie lineárně narůstá. Při vkládání prvku na začátek kolekce dokonce musíme posunout celé pole, z čehož plyne, že časová režie bude dvojnásobná a stále lineární, což graf potvrzuje. Při realokaci dochází k nárustu potřebného času, protože před operací vkládání se musí vytvořit nové pole požadované kapacity a ještě se do něj musí staré pole skopírovat. Dalo by se říct, že realokace přidává konstantní čas k případům vkládání prvků bez realokace. I tento předpoklad graf potvrzuje.

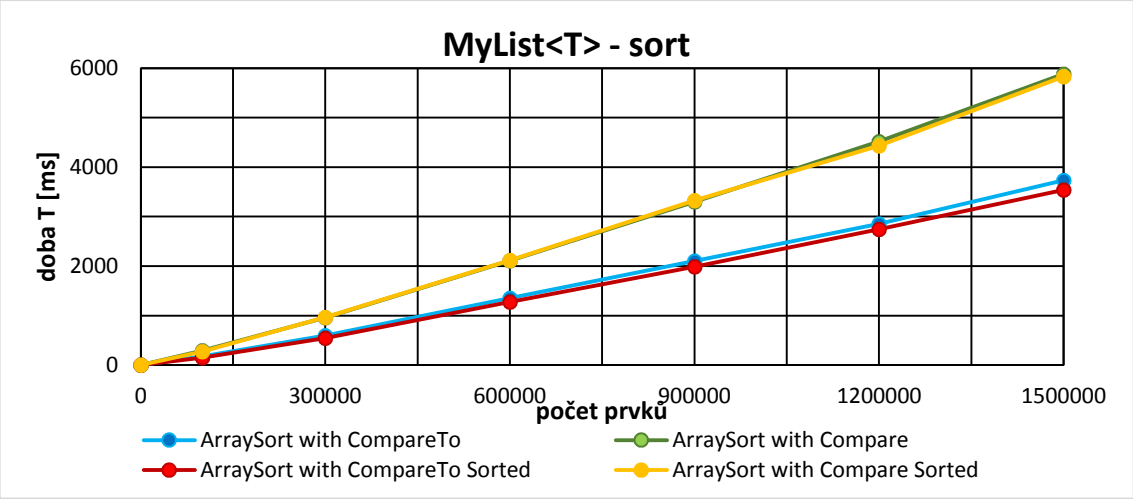


Graf 1: Kolekce MyList<T> - vkládání

Operace řazení (Sort)

Pro řazení je využita metoda *Sort* ze třídy *Array*, o které nevíme jak je implementována, neboli jakou metodu řazení používá. Této metodě jsme pouze nadefinovali komparátor,

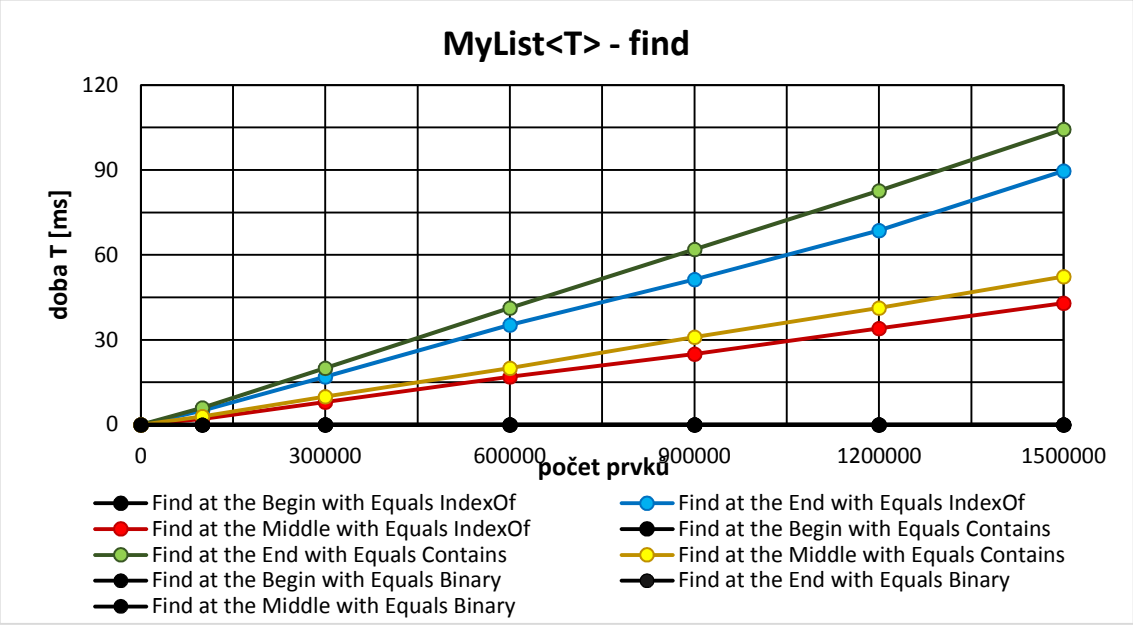
aby mohla prvky při řazení porovnávat. Na grafu je vidět, že pokud metodu řazení použijeme na již seřazenou kolekci, časová náročnost se v podstatě nemění a že časová náročnost narůstá lineárně s počtem prvků. Algoritmus pracuje vždy jako s neseřazenými prvky a je podobný metodě řazení *HeapSort*. Metoda *CompareTo* ve třídě *Color* řazení provádí o třetinu rychleji, vedle metody *Compare* ve třídě *Comparer*.



Graf 2: Kolekce MyList<T> - řazení

Operace hledání (Find)

Na tomto grafu je vidět, že operace binárního vyhledávání byla okamžitá, bez ohledu na to jestli hledaný prvek byl na začátku, na konci, nebo uprostřed kolekce. Dále je zde vidět, že pokud byl hledaný prvek na začátku kolekce tak byl také nalezen okamžitě.

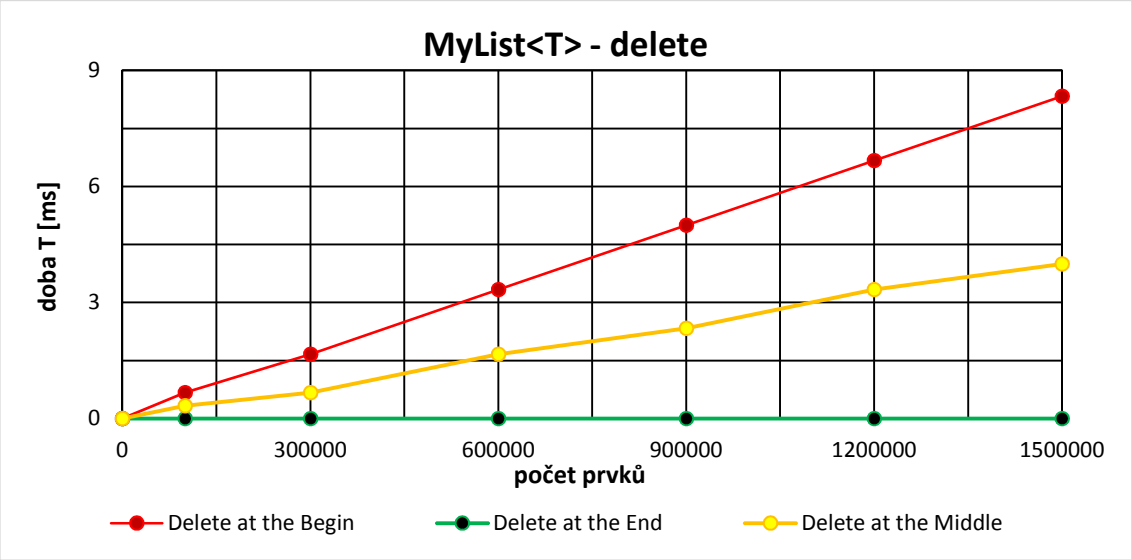


Graf 3: Kolekce MyList<T> - hledání

Hledání metodou *Contains*, která provádí sekvenční hledání je o trošku časově náročnější, než hledání metodou *IndexOf*, která k tomu používá metodu *IndexOf* třídy *Array*. Opět je zde vidět lineární závislost v závislosti na počtu prvků v kolekci.

Operace mazání (Delete)

Operace mazání, kdy mazaný prvek byl na konci kolekce byla okamžitá. Pokud byl mazaný prvek uprostřed kolekce, bylo nutné od mazaného prvku do konce pole, všechny prvky posunout o index 1. To samozřejmě se projevilo v časové náročnosti, která je lineární v závislosti na počtu prvků, jak ukazuje graf. Pokud mazaný prvek byl na začátku, bylo nutné od mazaného prvku do konce pole, všechny prvky pole posunout o index 1. Tedy se jednalo o dvojnásobný přesun dat, což odpovídá dvojnásobné časové náročnosti, jak je zřejmé z grafu.

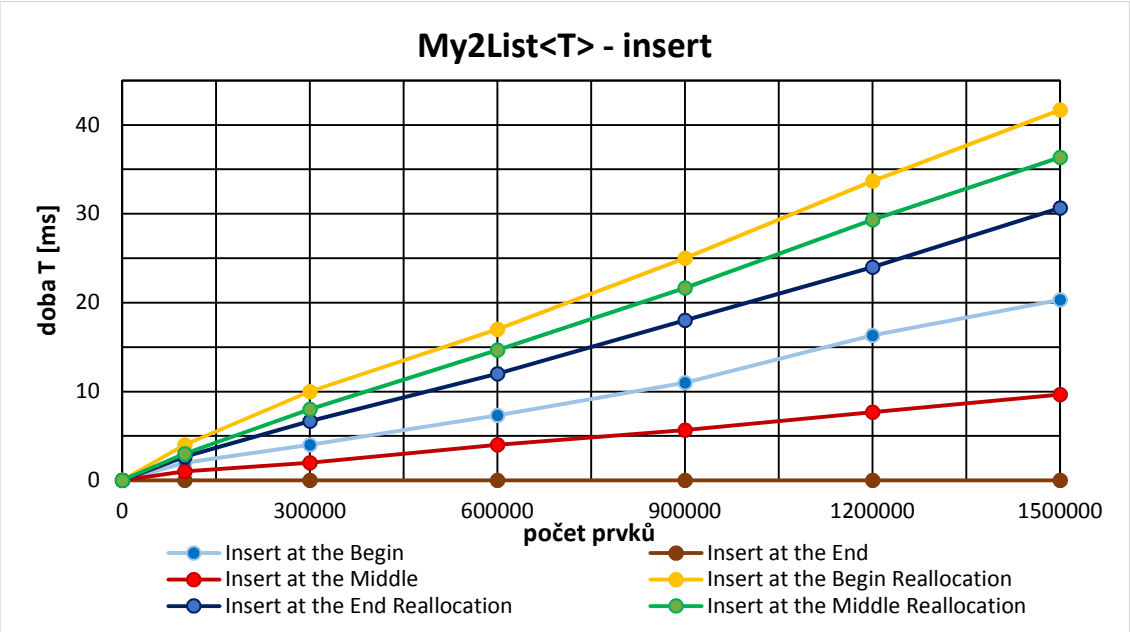


Graf 4: Kolekce MyList<T> - mazání

Vlastní implementace kolekce My2List<T>

Operace vkládání (Insert)

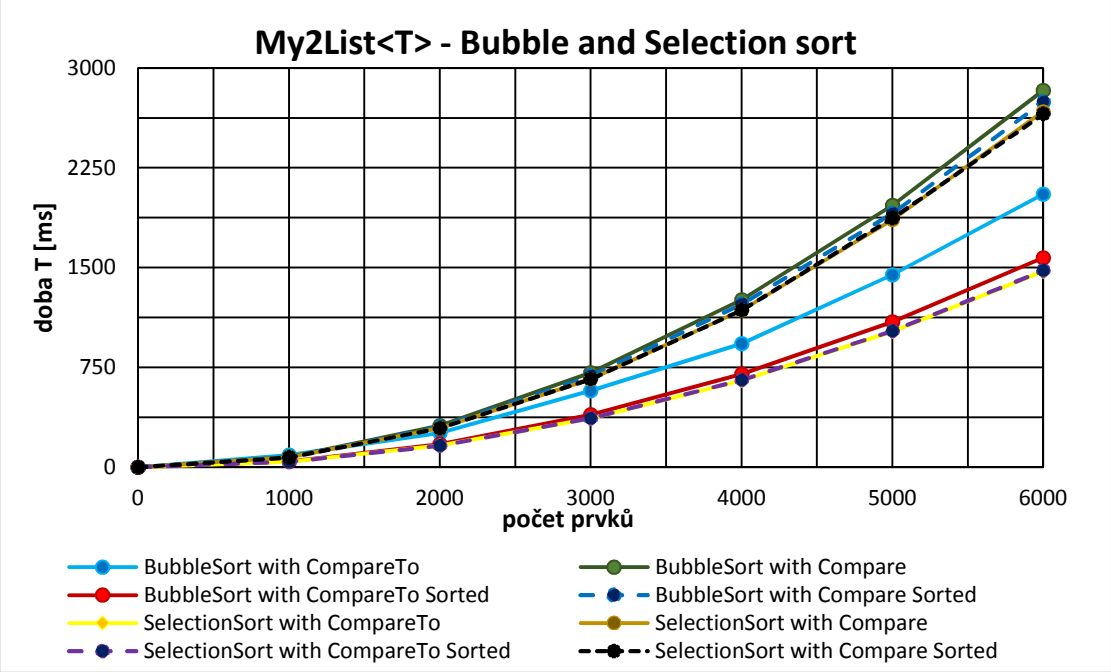
Tento graf je vhodné srovnat s grafem MyList<T> - insert. V kolekci *My2List* již místo metod volaných z třídy *Array*, voláme metody vlastní implementace z třídy *AlgoritmyTYP*. Předpokládáme že časová náročnost našich implementací bude větší. Jak ukazuje graf, tento předpoklad se potvrzuje, časová náročnost, všech operací vzrostla přibližně dvojnásobně.



Graf 5: Kolekce My2List<T> - vkládání

Operace řazení (Sort)

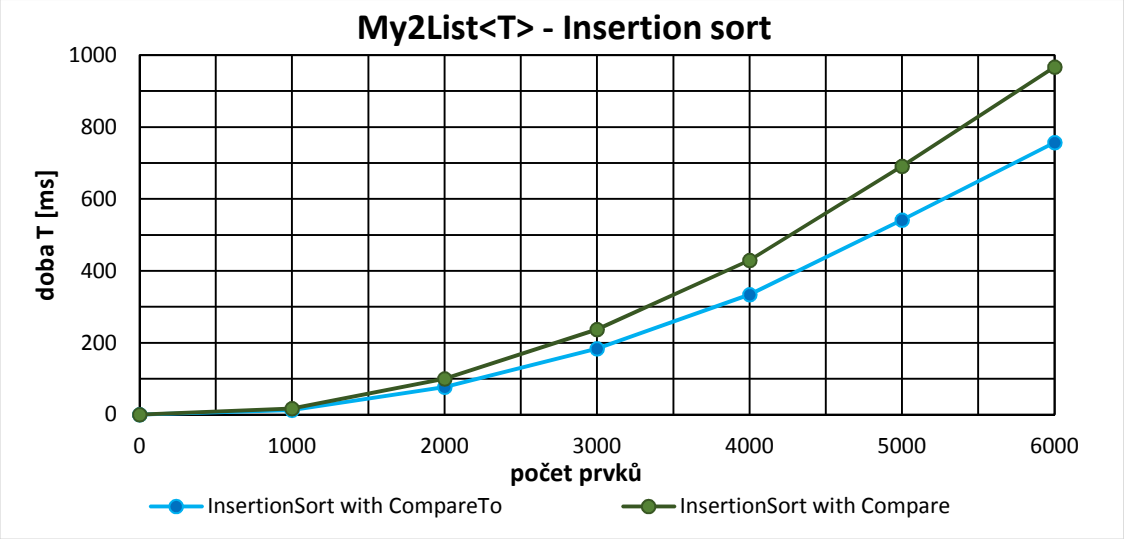
V naší implementaci kolekce *My2List* pro řazení voláme metody ze třídy *AlgoritmyTYP*, kde jsme naimplementovali několik algoritmů řazení. Jedním z hlavních úkolů této práce je porovnat tyto algoritmy řazení mezi sebou podle časové náročnosti. Tady už se úplně nedá hovořit o lineární závislosti, ale o závislosti, která se blíží kvadratické, v závislosti na počtu prvků.



Graf 6: Kolekce My2List<T> - řazení Bubble and Selection sort

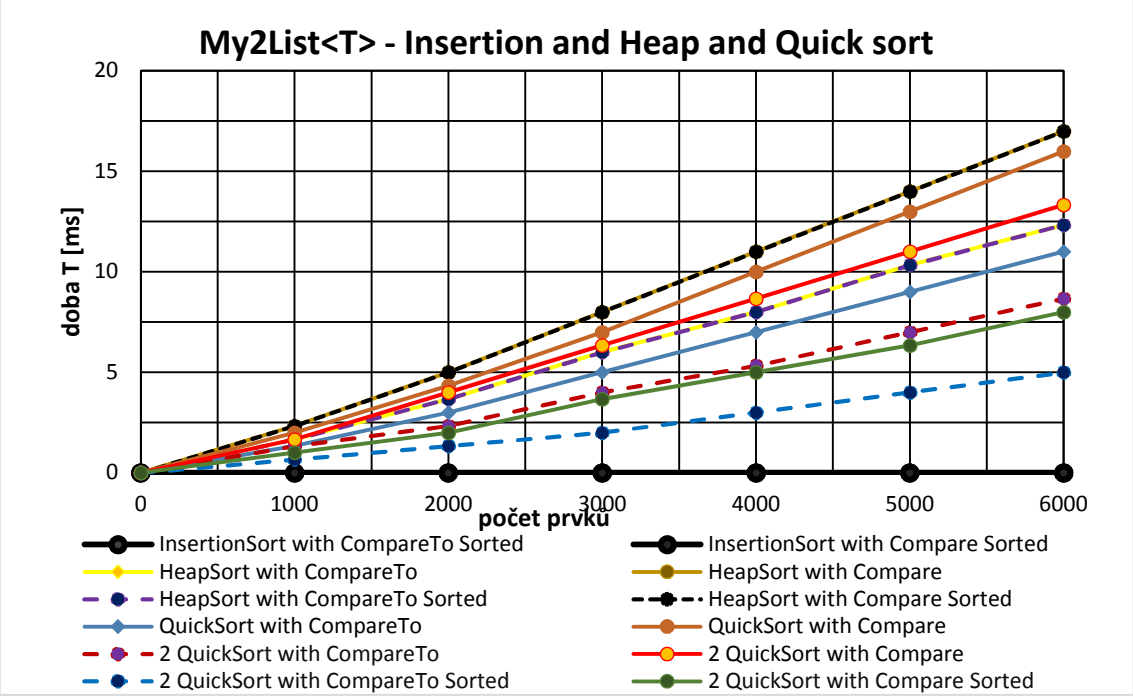
Dále je zde vidět, že algoritmus *SelectionSort*, který používá pro porovnávání metodu *CompareTo* definovanou ve třídě *Color*, pracuje nejrychleji. Dalo by se říci, že nezáleží na tom jestli už byla kolekce seřazená nebo ne.

Tento graf ukazuje, že algoritmus *InsertionSort* je ještě o poznání rychlejší než algoritmy *Bubble* a *Selection Sort*.



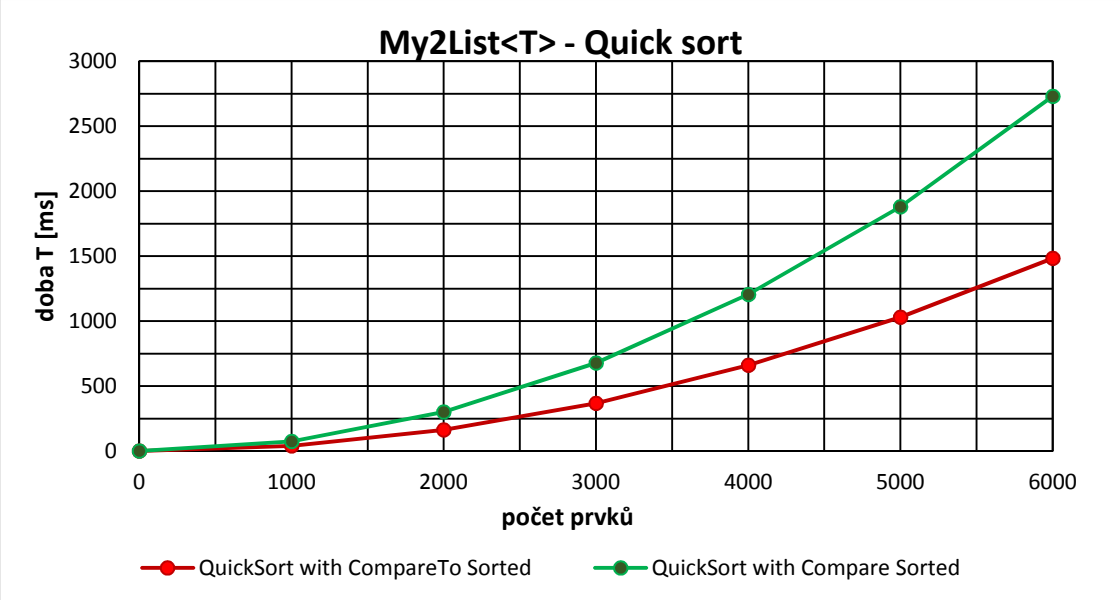
Graf 7: Kolekce My2List<T> - řazení Insertion sort

Na tomto grafu je vidět, že tyto algoritmy pracují ještě rychleji než ty předešlé. Dále je zde vidět, že se jedná o lineární závislost, v závislosti na počtu prvků.



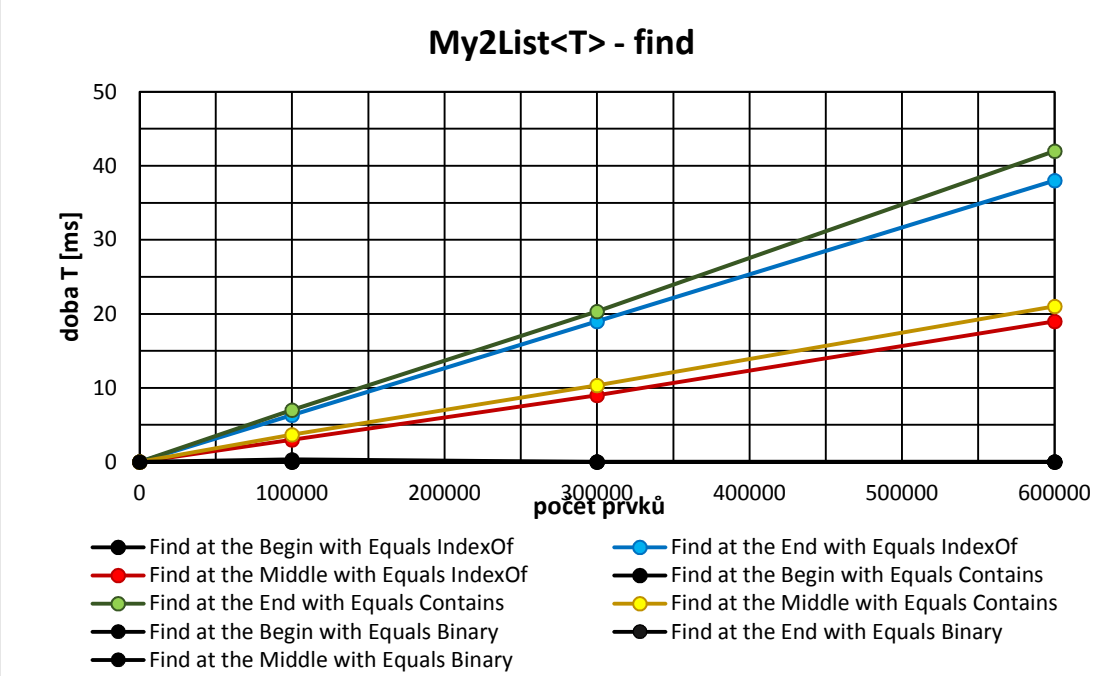
Graf 8: Kolekce My2List<T> - řazení Insertion and Heap and Quick sort

Na tomto grafu se projevil rozdíl mezi použitím komparátorů. Jeden byl definovaný jako metoda *CompareTo* ve třídě *Color*, druhý byl definovaný jako metoda *Compare* ve třídě *Comparer*. Dále zde vidíme, že došlo k obrovskému nárůstu časové režie u seřazené kolekce.



Graf 9: Kolekce My2List<T> - řazení Quick sort

Operace hledání (Find)



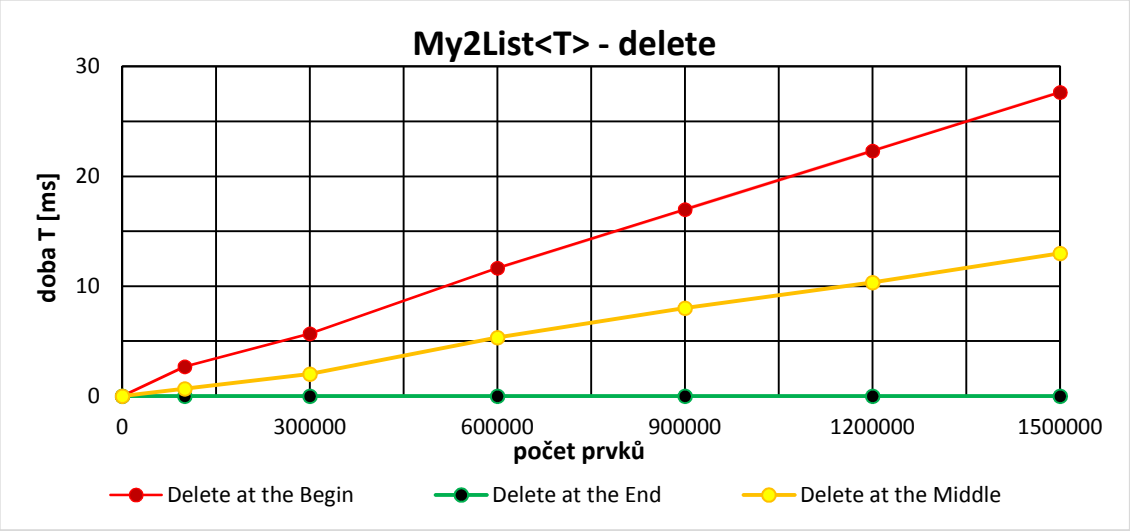
Graf 10: Kolekce My2List<T> - hledání

Na tomto grafu je vidět, že operace binárního vyhledávání byla okamžitá, bez ohledu na to jestli hledaný prvek byl na začátku, na konci, nebo uprostřed kolekce.

Dále je zde vidět, že pokud byl hledaný prvek na začátku kolekce tak byl také nalezen okamžitě. Hledání metodou *Contains*, která provádí sekvenční hledání je o trošku časově náročnější, než hledání metodou *IndexOf*. Opět je zde vidět lineární závislost v závislosti na počtu prvků v kolekci.

Operace mazání (Delete)

Operace mazání, kdy mazaný prvek byl na konci kolekce byla okamžitá. Pokud byl mazaný prvek uprostřed kolekce, bylo nutné od mazaného prvku do konce pole, všechny prvky posunout o index 1. To samozřejmě se projevilo v časové náročnosti, která je lineární v závislosti na počtu prvků, jak ukazuje graf. Pokud mazaný prvek byl na začátku, bylo nutné od mazaného prvku do konce pole, všechny prvky pole posunout o index 1. Tedy se jednalo o dvojnásobný přesun dat, což odpovídá dvojnásobné časové náročnosti, jak je zřejmé z grafu. Tento graf je vhodné porovnat s grafem MyList<T> - delete. Je vidět, že časová náročnost pro tuto operaci stoupla trojnásobně.



Graf 11: Kolekce My2List<T> - mazání

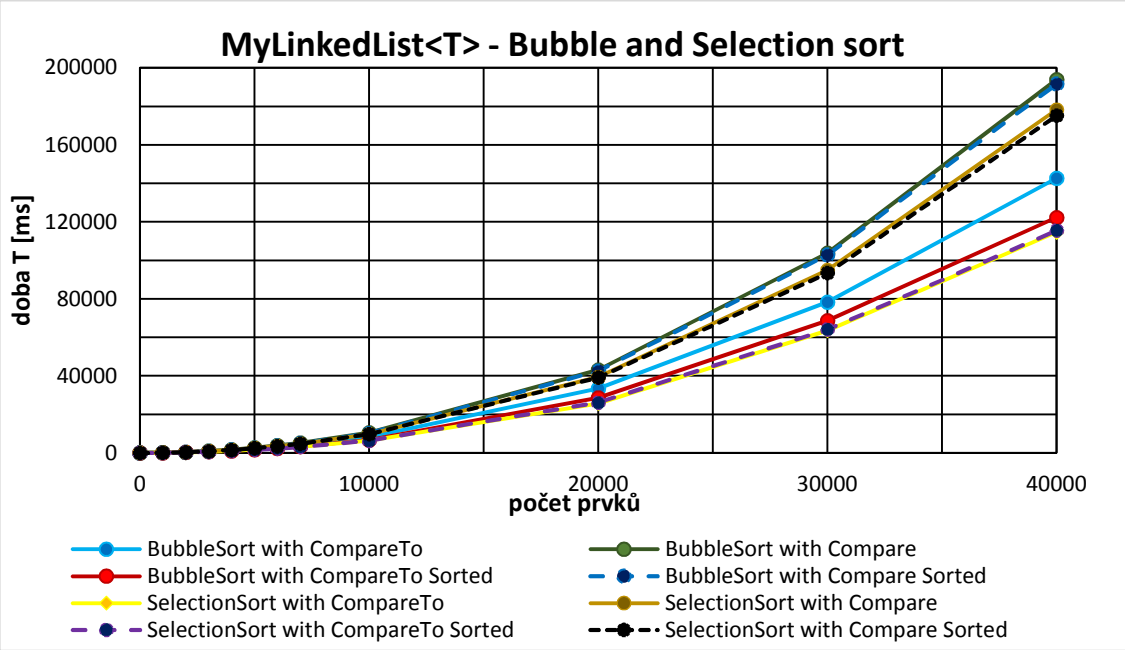
Vlastní implementace kolekce MyLinkedList<T>

Operace vkládání (Insert)

Operace vkládání uzlu do spojového seznamu na začátek , na konec a doprostřed byla okamžitá. Proto nebylo nutné časovou závislost vynášet do grafu.

Operace řazení (Sort)

Operace řazení spojového seznamu není v kolekci LinkedList v .NET/C# definovaná. Tyto algoritmy řazení jsem do naší implementace kolekce MyLinkedList přidal jako nadstandard.

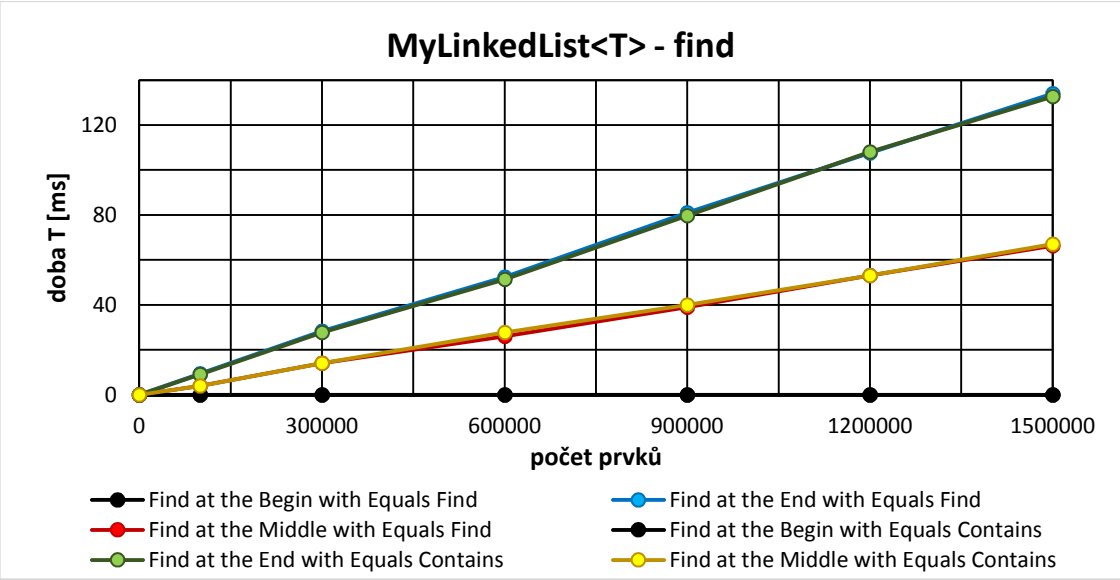


Graf 12: Kolekce MyLinkedList<T> - řazení

Tento graf je vhodné porovnat s grafem My2List<T> - Bubble and Selection sort. U obou grafů je vidět, že algoritmus *SelectionSort*, který používá pro porovnávání metodu *CompareTo* definovanou ve třídě *Color*, pracuje nejrychleji. Dalo by se říci, že nezáleží na tom jestli už byla kolekce seřazená nebo ne. Ovšem se ukázalo, že časová režie u spojového seznamu *MyLinkedList* je desetinásobná, než u kolekce *My2List*.

Operace hledání (Find)

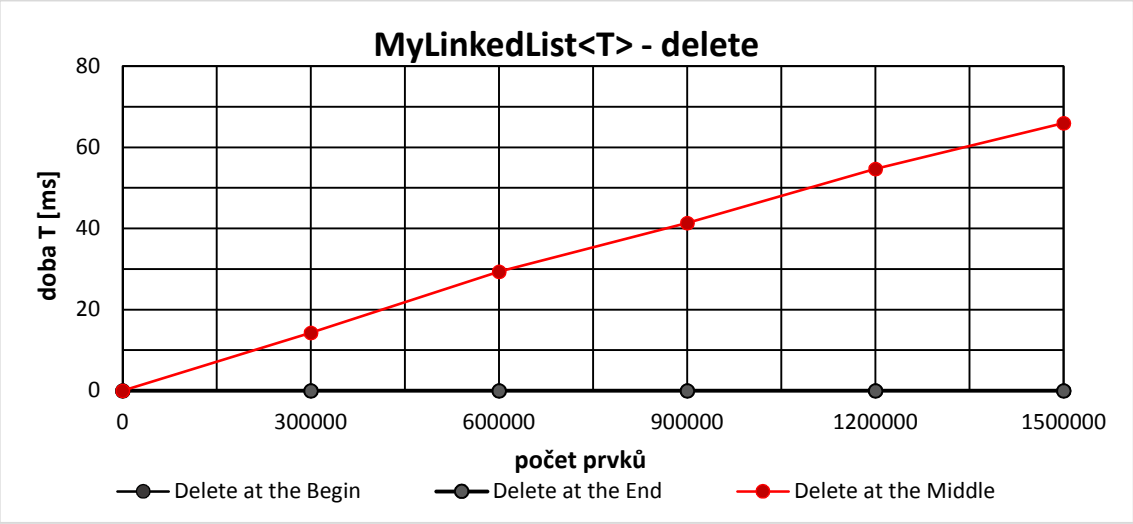
Na tomto grafu je vidět, že pokud byl hledaný uzel na začátku spojového seznamu tak byl nalezen okamžitě. Hledání metodou *Contains* a metodou *Find* je ekvivalentní, protože metoda *Contains* využívá metodu *Find*. Hledání uzlu probíhá sekvenční metodou, kdy přecházíme z jednoho uzlu na další a přitom porovnáváme hodnotu v konkrétním uzlu metodou *Equals* definovanou ve třídě *Color*. Opět je zde vidět lineární závislost v závislosti na počtu uzlů ve spojovém seznamu. Tento graf je vhodné porovnat s grafem My2List<T> - find. Zjistíme, že časová náročnost pro tuto operaci je trojnásobná.



Graf 13: Kolekce MyLinkedList<T> - hledání

Operace mazání (Delete)

Operace mazání, kdy mazaný uzel byl na konci a na začátku spojového seznamu byla okamžitá. Je to zdůvodu, že jsme implementovali obousměrný spojový seznam, který je navíc uzavřený. Pokud byl mazaný uzel uprostřed spojového seznamu, bylo nutné nejprve zavolat metodu *Find*, která nalezla uzel o konkrétní hodnotě (sekvenčně) a pak zavolat metodu *OdstranUzel*, která uzel ze spojového seznamu odstranila. Proto je vhodné tento graf porovnat s grafem MyLinkedList<T> - find a uvědomit si, že největší časová režie je na vyhledání. Samozřejmě časová náročnost zůstává lineární v závislosti na počtu uzlů.

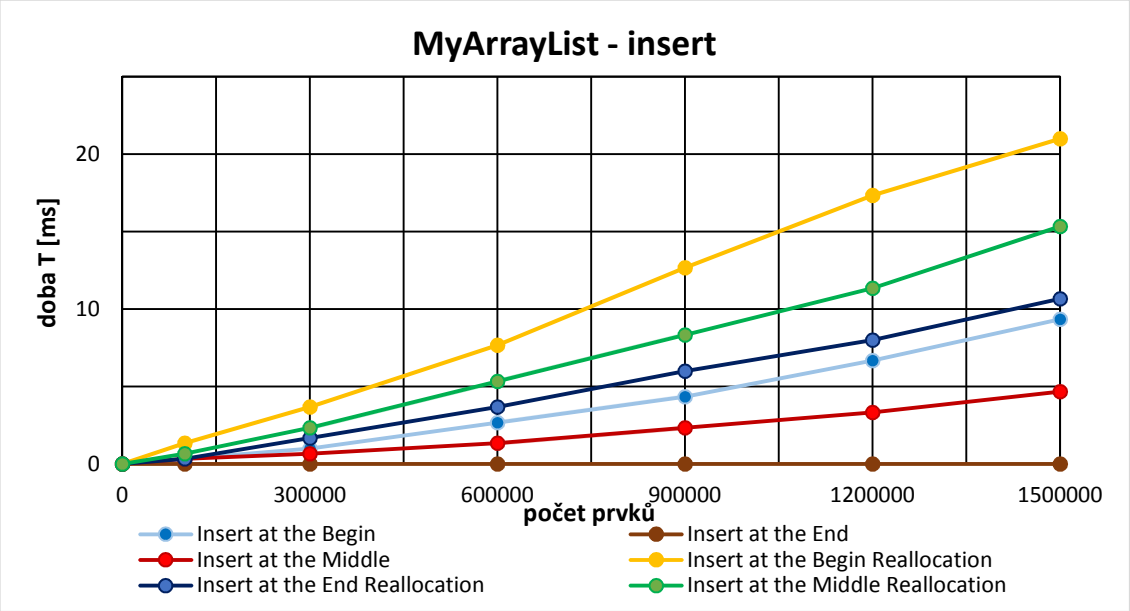


Graf 14: Kolekce MyLinkedList<T> - mazání

Vlastní implementace kolekce MyArrayList

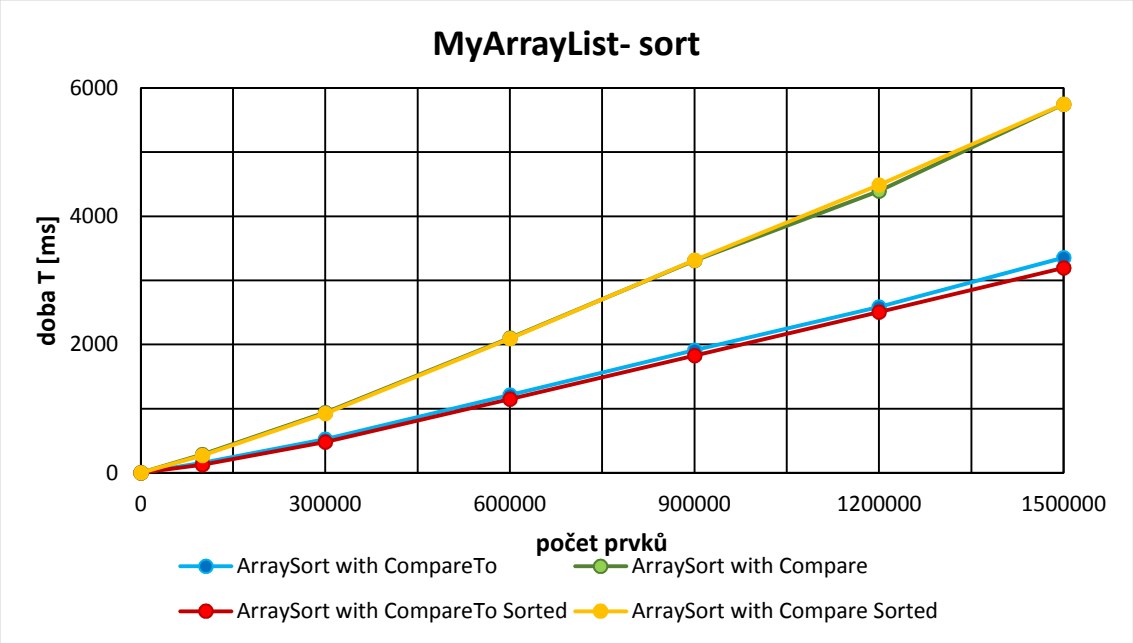
Pro tuto kolekci platí stejný komentář, který jsem psal pro kolekci MyList<T>, proto ho zde již nebudu znovu psát.

Operace vkládání (Insert)



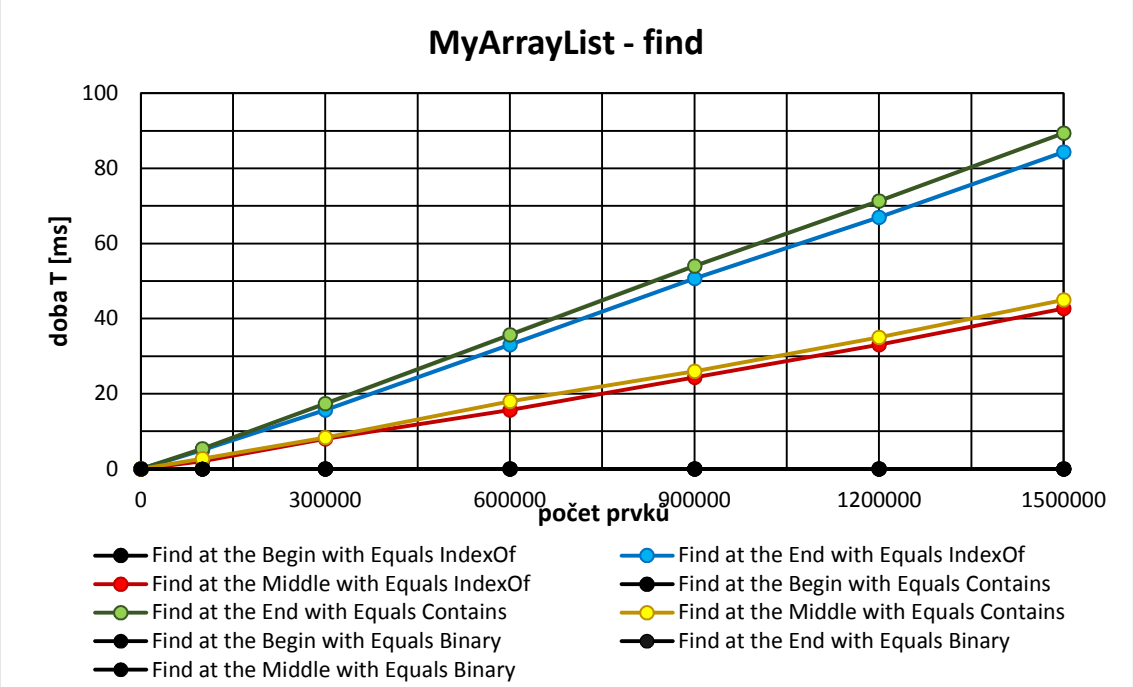
Graf 15: Kolekce MyArrayList – vkládání

Operace řazení (Sort)



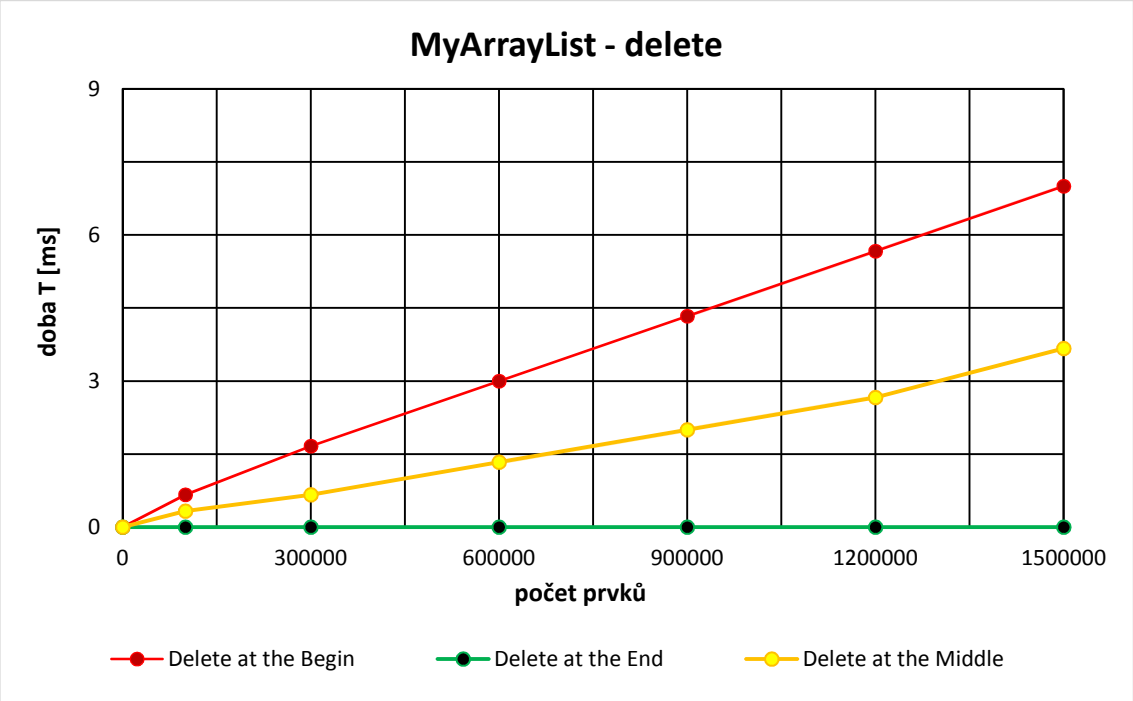
Graf 16: Kolekce MyArrayList – řazení

Operace hledání (Find)



Graf 17: Kolekce MyArrayList - hledání

Operace mazání (Delete)

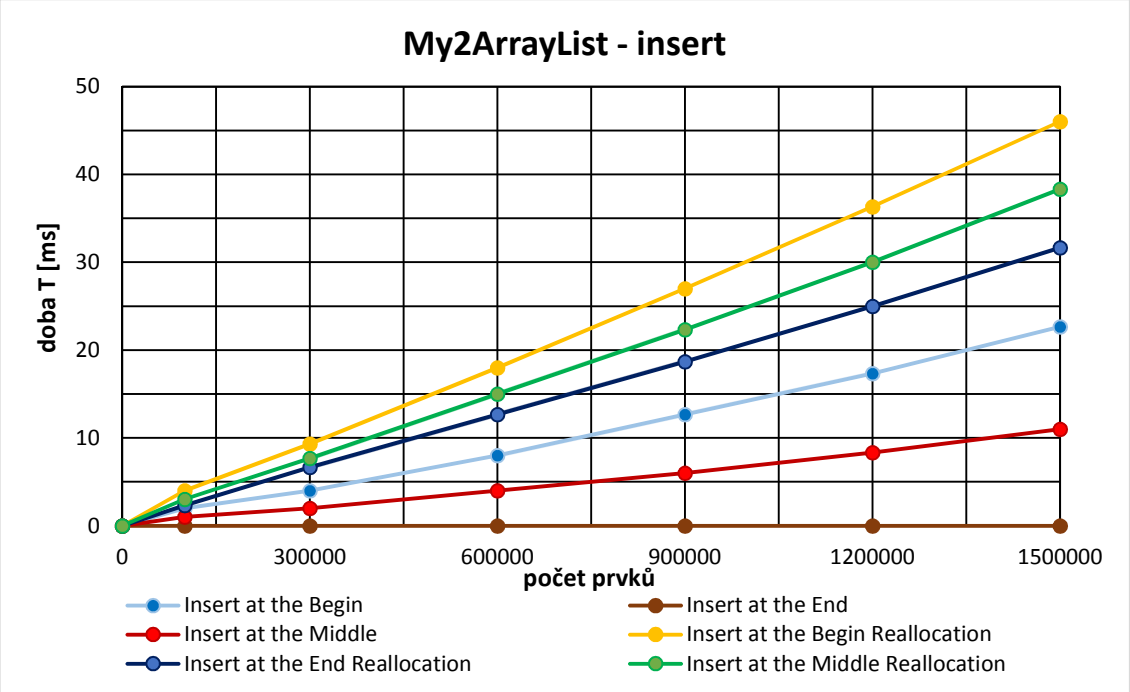


Graf 18: Kolekce MyArrayList - mazání

Vlastní implementace kolekce My2ArrayList

Operace vkládání (Insert)

Tento graf je vhodné srovnat s grafem MyArrayList - insert. V kolekci *My2ArrayList* již místo metod volaných z třídy *Array*, voláme metody vlastní implementace z třídy *Algoritmy*. Předpokládáme že časová náročnost našich implementací bude větší. Jak ukazuje graf, tento předpoklad se potvrzuje, časová náročnost, všech operací vzrostla přibližně dvojnásobně.

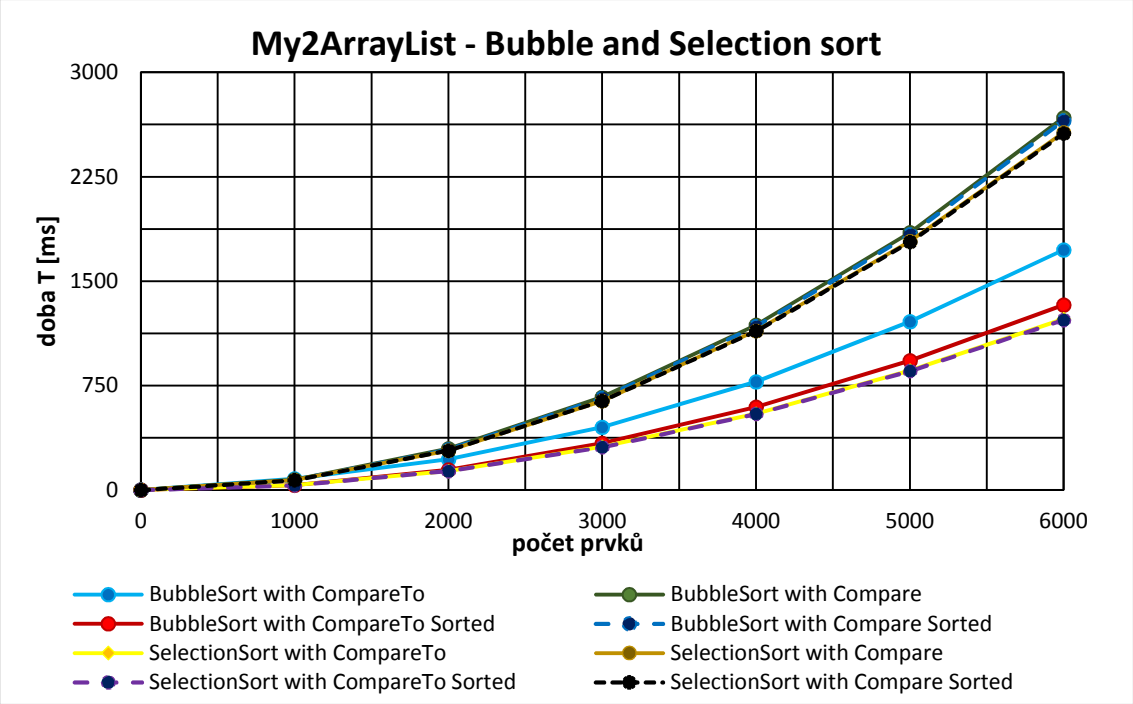


Graf 19: Kolekce My2ArrayList - vkládání

Operace řazení (Sort)

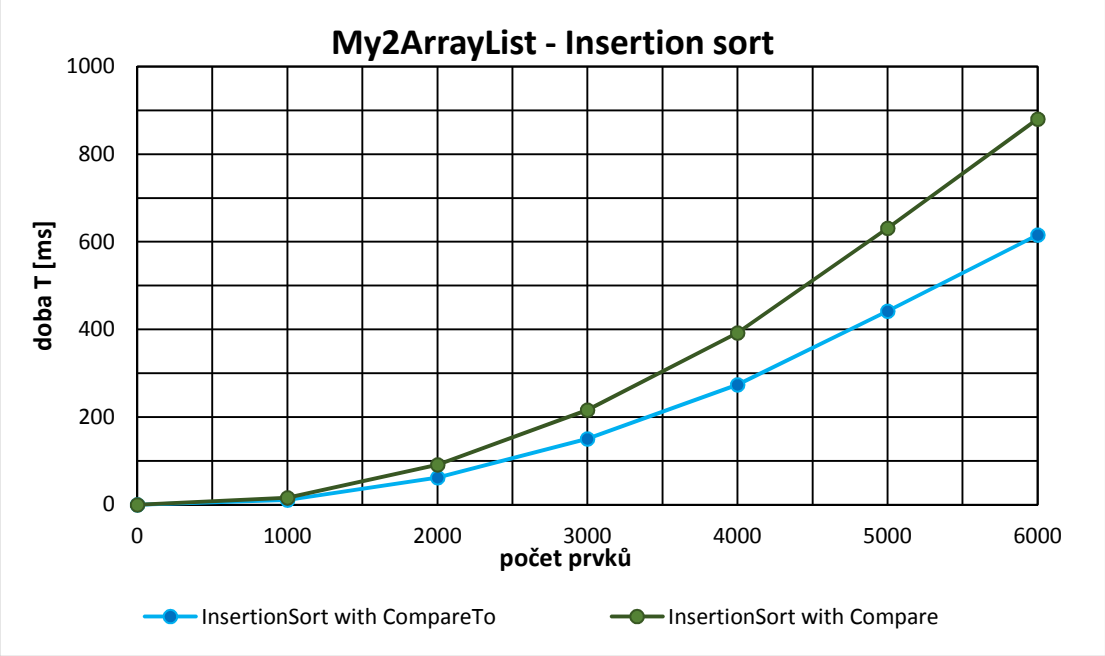
V naší implementaci kolekce *My2ArrayList* pro řazení voláme metody ze třídy *Algoritmy*, kde jsme naimplementovali několik algoritmů řazení. Jedním z hlavních úkolů této práce je porovnat tyto algoritmy řazení mezi sebou podle časové náročnosti. Tady už se úplně nedá hovořit o lineární závislosti, ale o závislosti, která se blíží kvadratické, v závislosti na počtu prvků. Všechny implementace algoritmů řazení ve třídě *Algoritmy* jsou analogické k implementacím algoritmů řazení ve třídě *AlgoritmyTYP*. Rozdíl je jen že algoritmy řazení ve třídě *AlgoritmyTYP* pracují s typovými kolekcemi. Proto komentáře k těmto grafům řazení jsou podobné a nebudu je zde znovu psát.

Tento graf je tedy velmi podobný grafu My2List<T> - Bubble and Selection sort



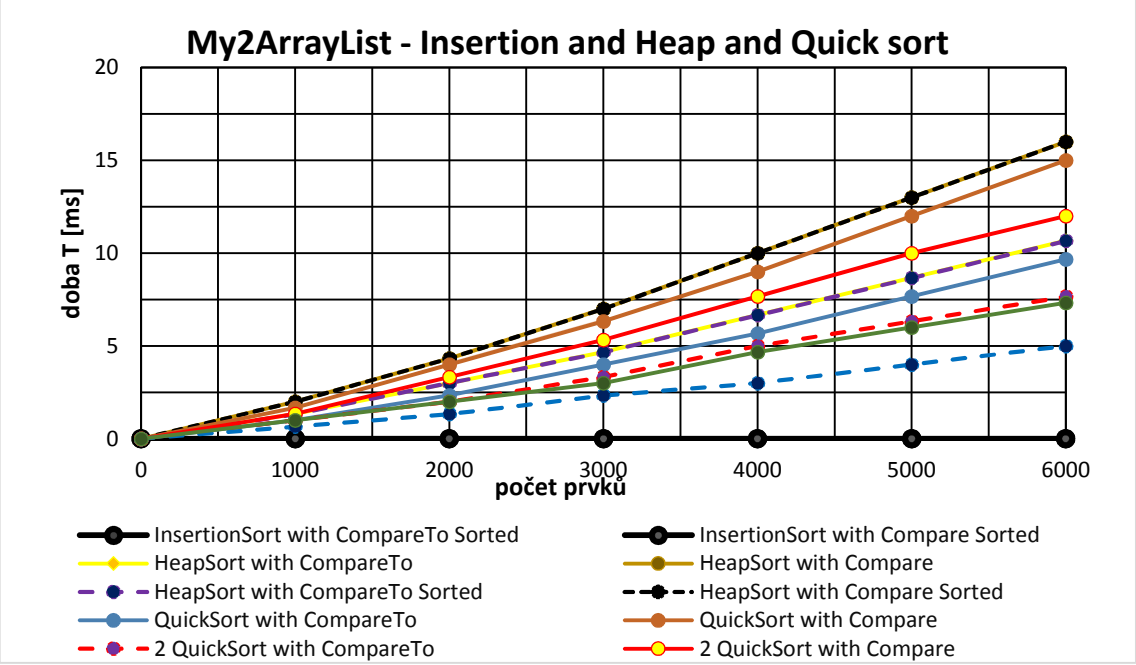
Graf 20: Kolekce My2ArrayList - řazení Bubble and Selection sort

U tohoto grafu ve srovnání s grafem My2List<T> - Insertion sort, vyšel průběh funkcí též podobný.



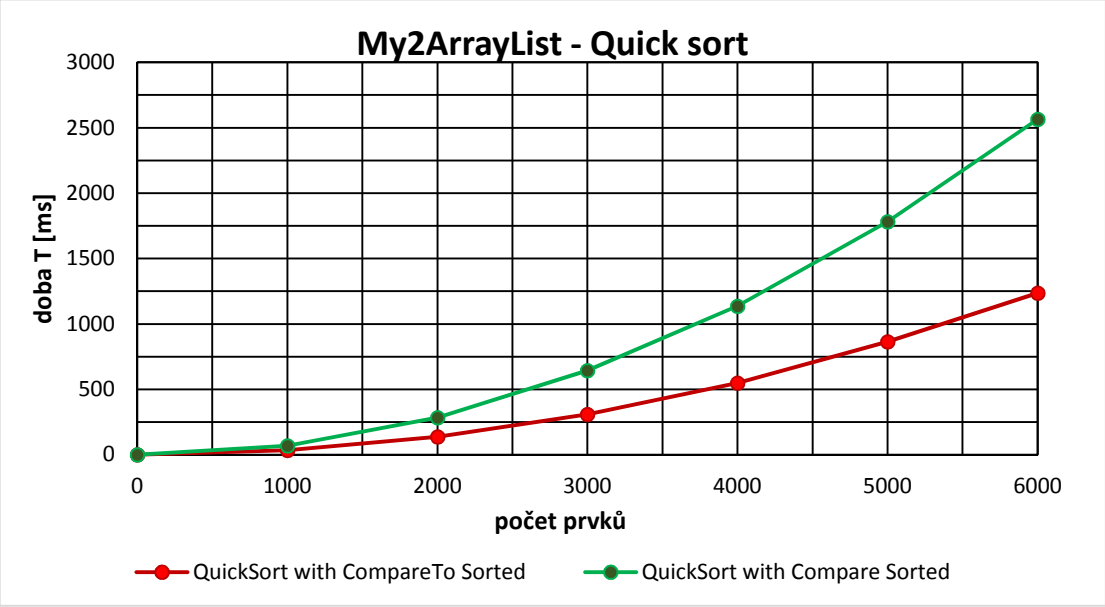
Graf 21: Kolekce My2ArrayList - řazení Insertion sort

Tento graf je zas velmi podobný z již uvedených důvodů grafu My2List<T> - Insertion and Heap and Quick sort.



Graf 22: Kolekce My2ArrayList - řazení Insertion and Heap and Quick sort

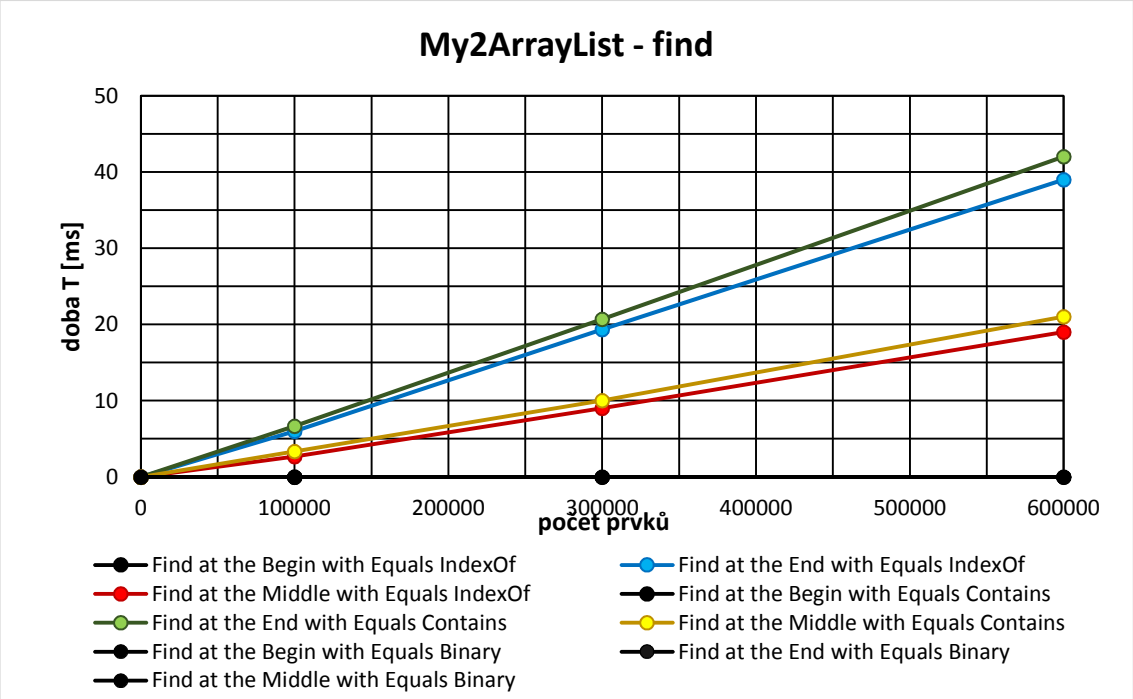
A poslední graf ve kterém je zobrazen algoritmus Quick sort je velmi podobný grafu My2List<T> - Quick sort.



Graf 23: Kolekce My2ArrayList - řazení Quick sort

Operace hledání (Find)

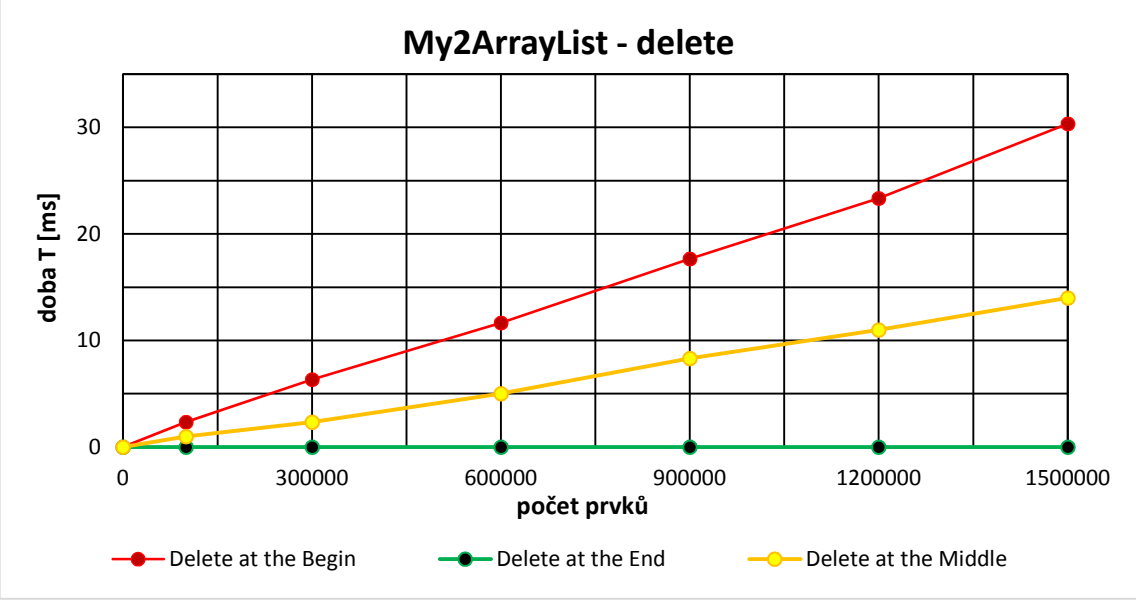
Opět zde je shoda s grafem My2List<T> - find.



Graf 24: Kolekce My2ArrayList - hledání

Operace mazání (Delete)

Jako poslední graf této kolekce srovnáme operaci mazání s grafem My2List<T> - delete. I zde je vidět shoda.

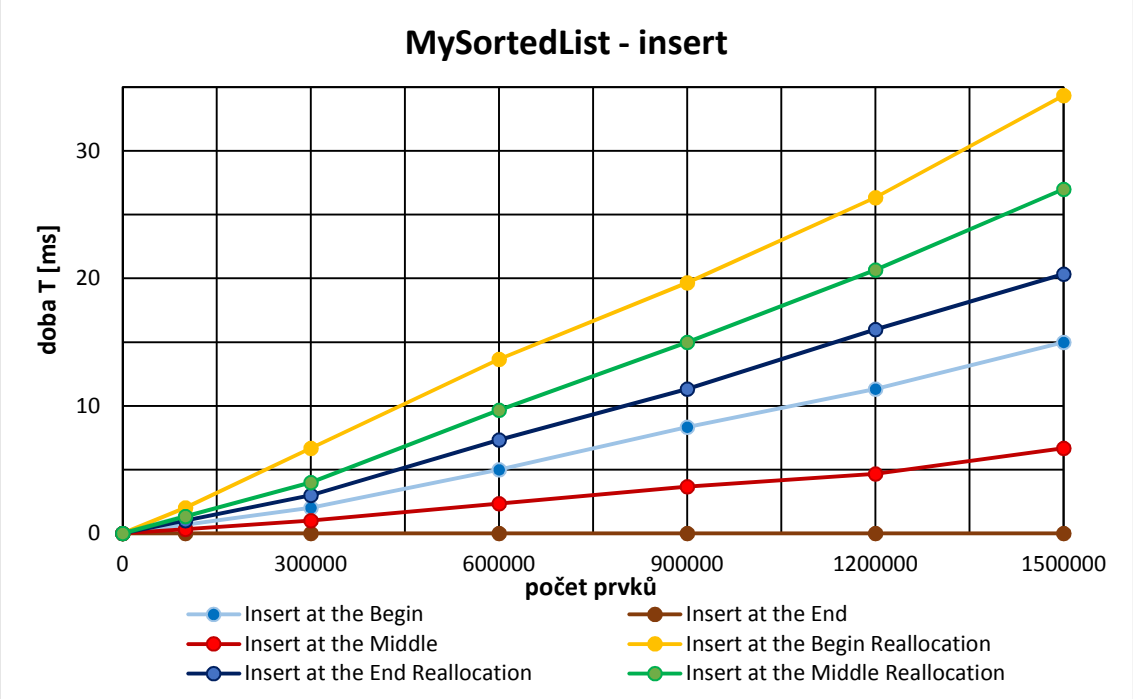


Graf 25: Kolekce My2ArrayList - mazání

Vlastní implementace kolekce MySortedList

Operace vkládání (Insert)

Na tomto grafu je vidět, že operace vkládání prvku na konec kolekce bez realokace je okamžitá, což potvrzuje způsob naší implementace. Naproti tomu, pokud vkládáme do prostřed kolekce, kdy musíme posunout polovinu pole o index 1, časová režie lineárně narůstá. Při vkládání prvku na začátek kolekce dokonce musíme posunout celé pole, z čehož plyne, že časová režie bude dvojnásobná a stále lineární, což graf potvrzuje. Při realokaci dochází k nárustu potřebného času, protože před operací vkládání se musí vytvořit nové pole požadované kapacity a ještě se do něj musí staré pole skopírovat. Dalo by se říct, že realokace přidává konstantní čas k případům vkládání prvků bez realokace. I tento předpoklad graf potvrzuje. Tento graf je vhodné srovnat s grafem MyArrayList - insert, a uvidíme, že došlo k nárůstu časové náročnosti přibližně o třetinu. To je způsobeno tím, že vkládáme do již seřazeného pole a musíme volat navíc metodu *BinarySearch* ze třídy *Array*, která vrátí index kam vložit do kolekce zařazovaný prvek. Dále je dobré si uvědomit, že zde pracujeme již s dvěmi poli v kolekci, jedno máme pro uložení klíče a druhé pro uložení hodnoty.



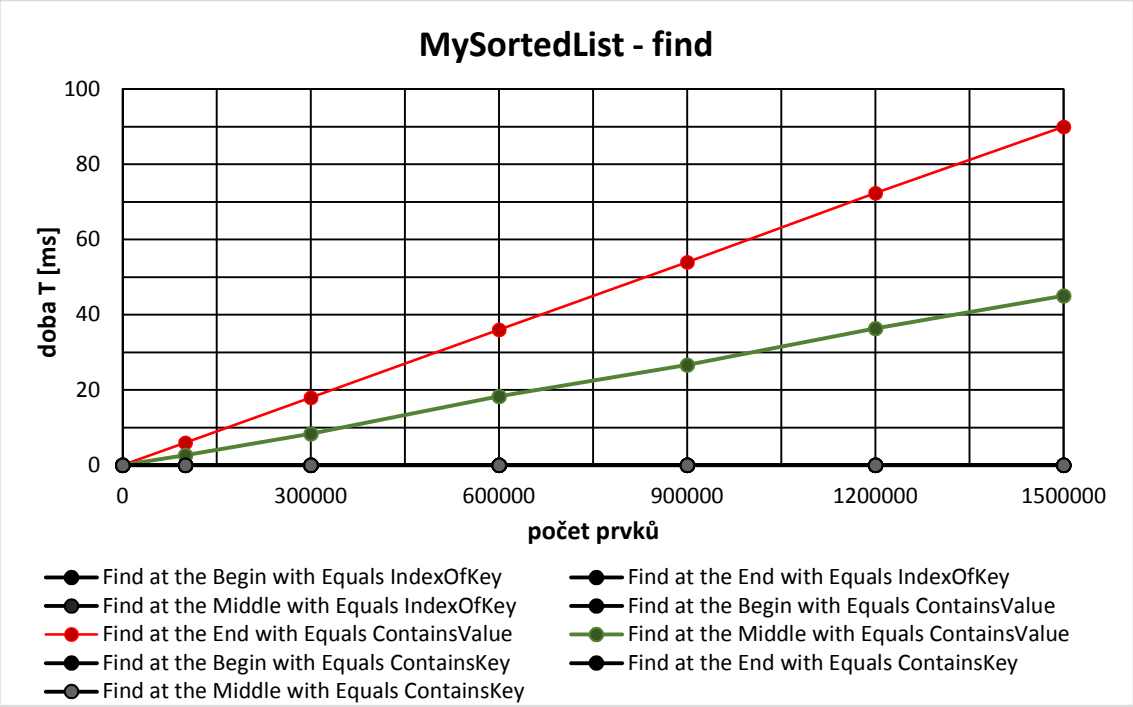
Graf 26: Kolekce MySortedList - vkládání

Operace řazení (Sort)

Jelikož vkládáme do již seřazené kolekce, operace řazení kolekce jako taková není potřeba.

Operace hledání (Find)

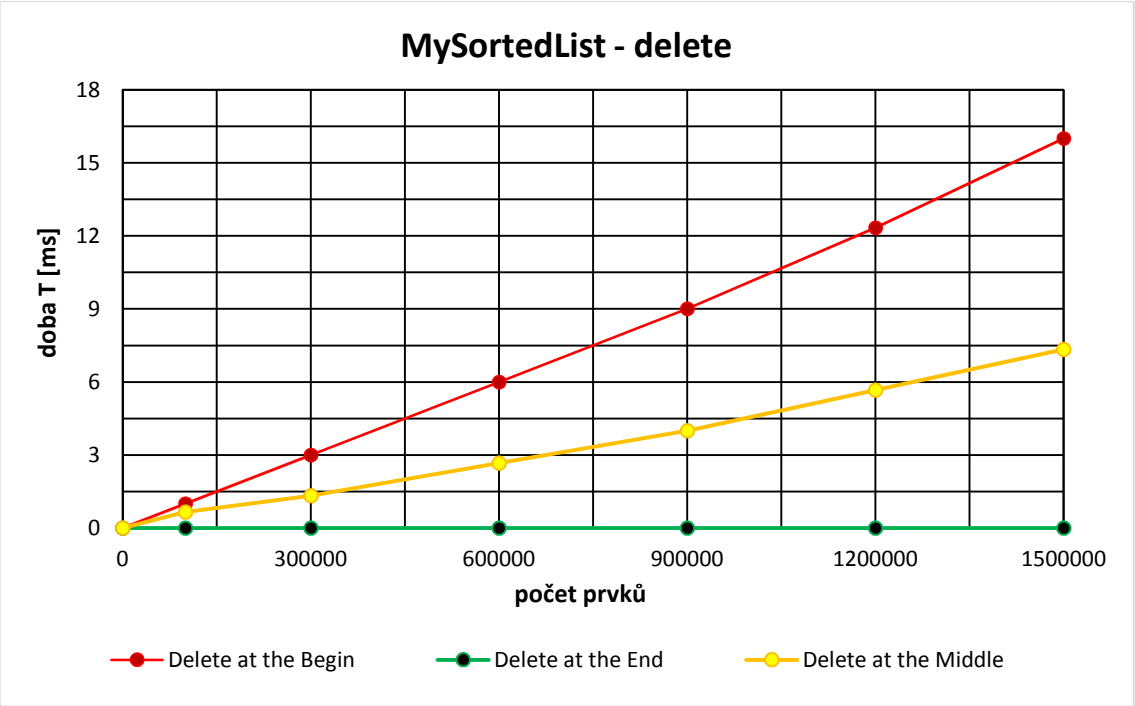
Zde je velmi zajímavé srovnat tento graf s grafem MyArrayList - find, a vidíme, že pokud hledáme prvek v kolekci podle klíče, který je jednoduchý datový typ, operace je okamžitá. Je to z důvodu, že tato operace probíhá binárním vyhledáváním. Pokud však prohledáváme kolekci podle hodnoty, dochází k nárustu časové režie podle počtu prvků. Tato operace probíhá metodou *IndexOf* volané ze třídy *Array*, kterou též používá kolekce *MyArrayList*, proto by měli být a jsou průběhy velmi podobné.



Graf 27: Kolekce MySortedList - hledání

Operace mazání (Delete)

Pokud bychom srovnali tento graf s grafem MyArrayList – delete, zjistíme že operace mazání prvku z kolekce *MySortedList* je přibližně dvakrát časově náročnější. Je to zdůvodu, že zde manipulujeme s dvěmi poli, s jedním ve kterém máme klíč a s druhým ve kterém máme hodnotu.

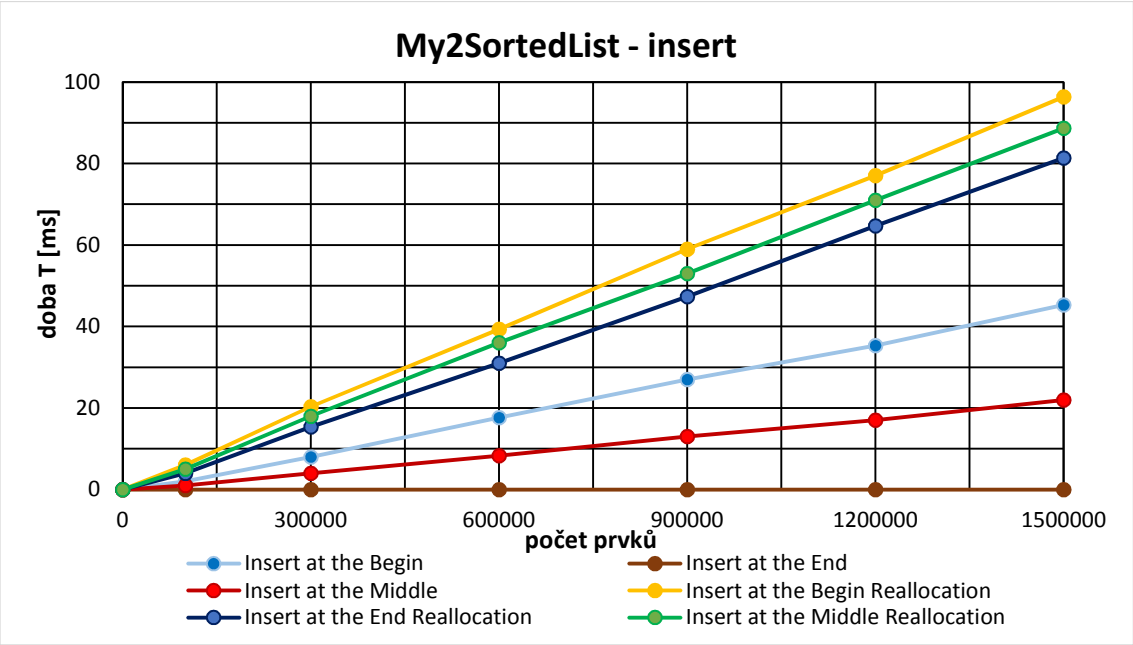


Graf 28: Kolekce MySortedList – mazání

Vlastní implementace kolekce My2SortedList

Operace vkládání (Insert)

Tento graf je vhodné porovnat s grafem MySortedList – insert. Vzhledem k tomu, že v této kolekci jsme nahradili všechny metody volané ze třídy *Array* vlastními, které voláme ze třídy *Algoritmy*, očekáváme značný nárůst časové režie pro jednotlivé operace. U kolekce *My2ArrayList* ve srovnání s kolekcí *MyArrayList* časová náročnost vzrostal přibližně dvakrát. Tady u kolekce *My2SortedList* ve srovnání s kolekcí *MySortedList* časová náročnost pro vkládání vzrostla dokonce třikrát. Je to způsobeno tím že jsme tam nahrazovali o jednu metodu navíc, tedy metodu *BinarySearch* ze třídy *Array*. Na toto místo by se hodili dále komentáře, které jsem psal u grafu MySortedList – insert, ale už bych se opakoval, proto je tady nebudu uvádět.



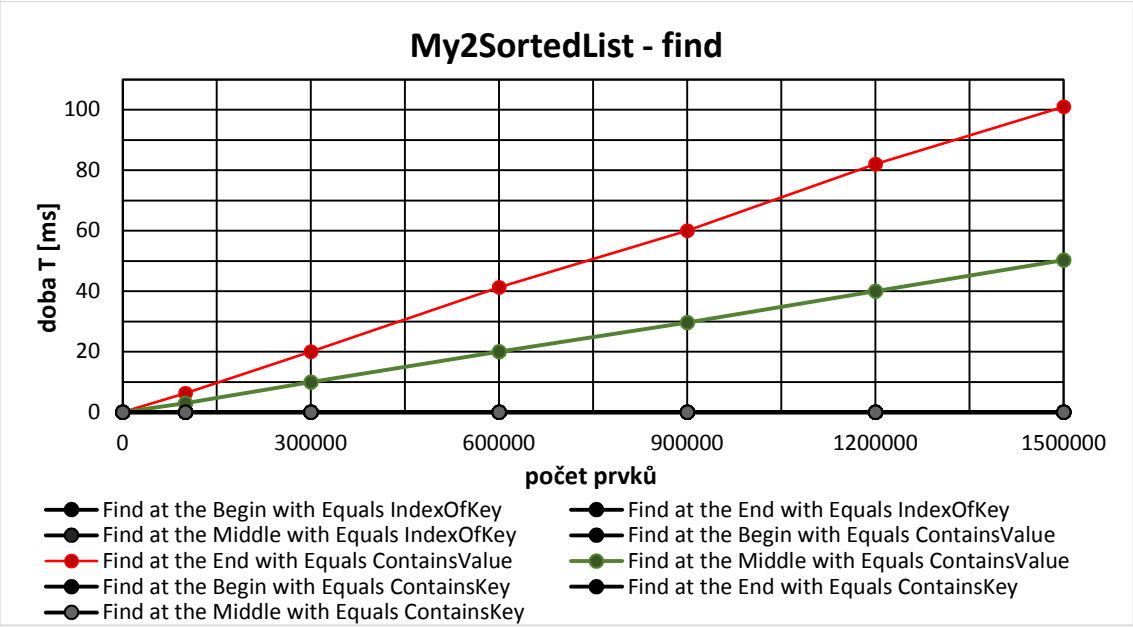
Graf 29: Kolekce My2SortedList – vkládání

Operace řazení (Sort)

Jelikož vkládáme do již seřazené kolekce, operace řazení kolekce jako taková není potřeba.

Operace hledání (Find)

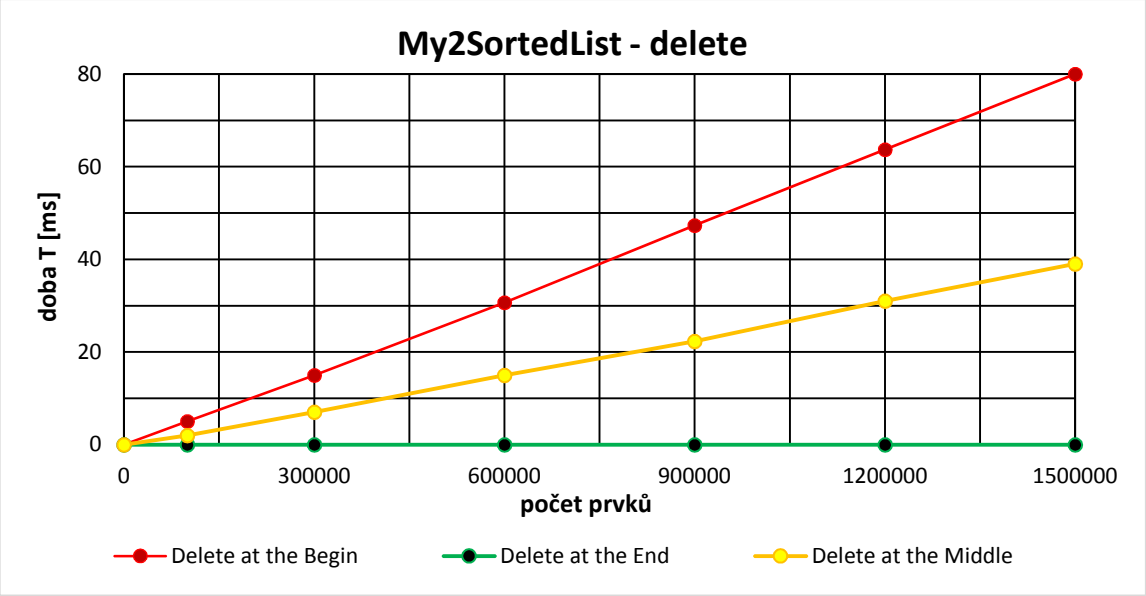
Zde je zajímavé srovnat tento graf s grafem MySortedList - find, a vidíme, že časová náročnost pro tuto operaci vzrostla jen velmi nepatrně. To svědčí, že jsme navrhli velmi podobnou implementaci.



Graf 30: Kolekce My2SortedList – hledání

Operace mazání (Delete)

Pokud bychom srovnali tento graf s grafem MySortedList – delete, zjistíme že operace mazání prvku z kolekce *My2SortedList* je přibližně čtyřikrát časově náročnější. Je to zdůvodu, že jsme zde naimplementovali vlastní metody pro manipulaci s polem, kde máme uložený klíče, a pro manipuli s polem, kde máme uloženy hodnoty. Dále zde používáme vlastní metodu pro vyhledání prvku v kolekci. Zřejmě tyto důvody vysvětlují proč tak narostla časová režie pro tuto operaci.



Graf 31: Kolekce My2SortedList – mazání

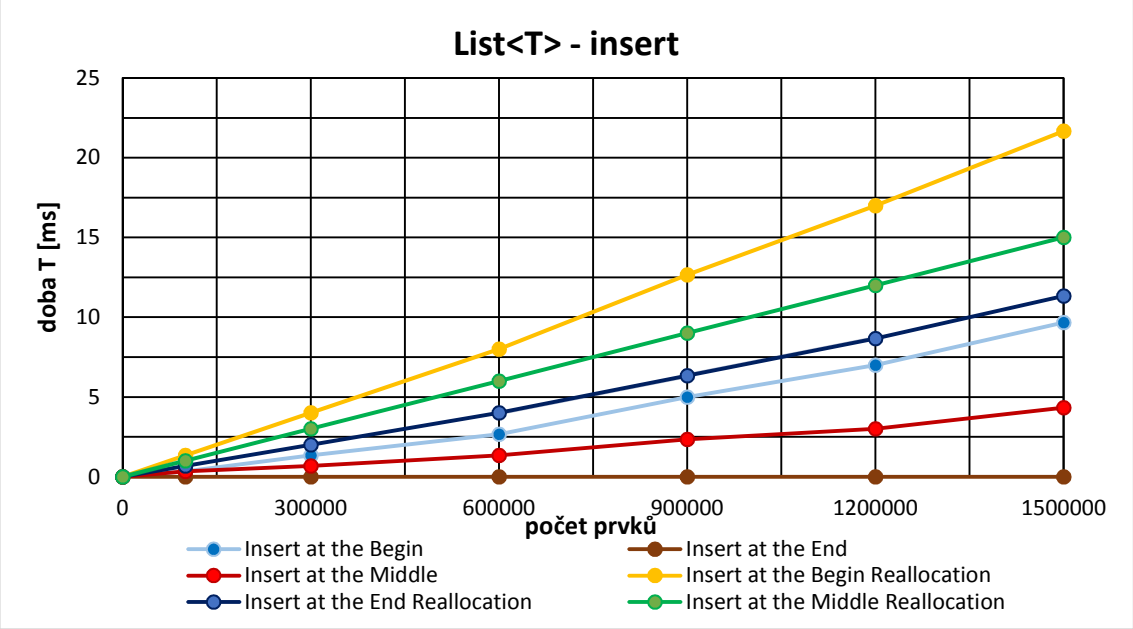
5.2 Typové kolekce

Kolekce List<T>

Zde jsme testovali standardní kolekci *List<T>*, která je součástí .NET/C#. Našim úkolem je srovnat časovou náročnost této kolekce s naší vlastní implementací kolekce stejného typu. Sice předpokládáme, že časová náročnost pro jednotlivé operace bude naší vlastní implementace kolekce větší, ale předpokládáme že proporcionálně grafy si budou odpovídat.

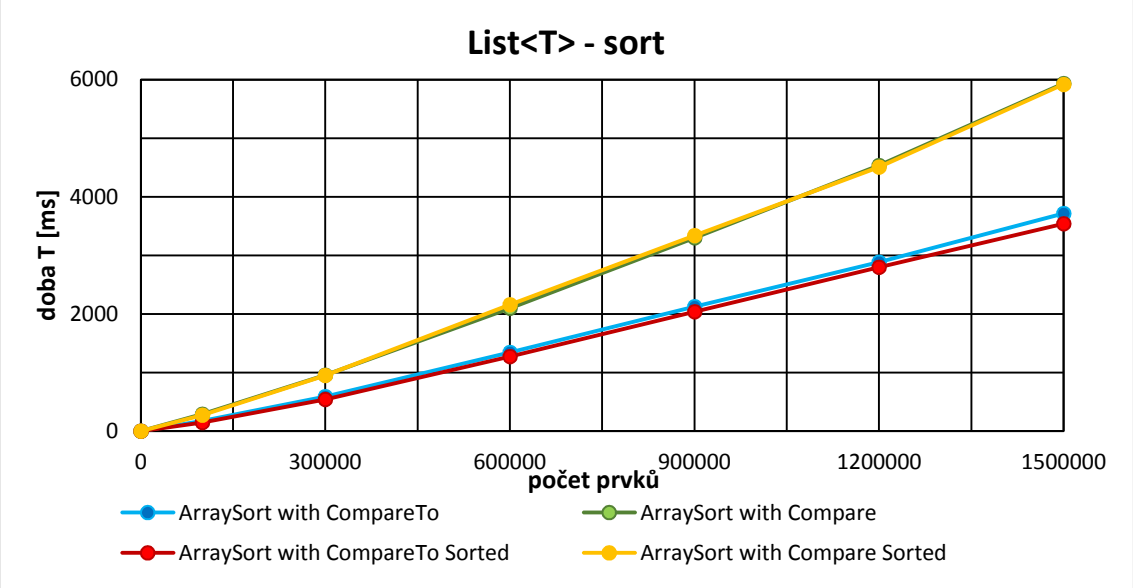
Operace vkládání (Insert)

Tento graf je vhodné srovnat s grafem MyList<T> - insert, a s grafem My2List<T> - insert. Vidíme, že s grafem MyList<T> - insert je tento graf velmi podobný, což svědčí o správné implementaci naší vlastní kolekce. Naproti tomu, srovnámeli tento graf s grafem My2List<T> - insert, kde jsme navíc nahradili všechny metody volané z třídy *Array* vlastními, vidímé v podstatě dvojnásobný nárůst časové režie u kolekce *My2List*.



Graf 32: Kolekce List<T> - vkládání

Operace řazení (Sort)

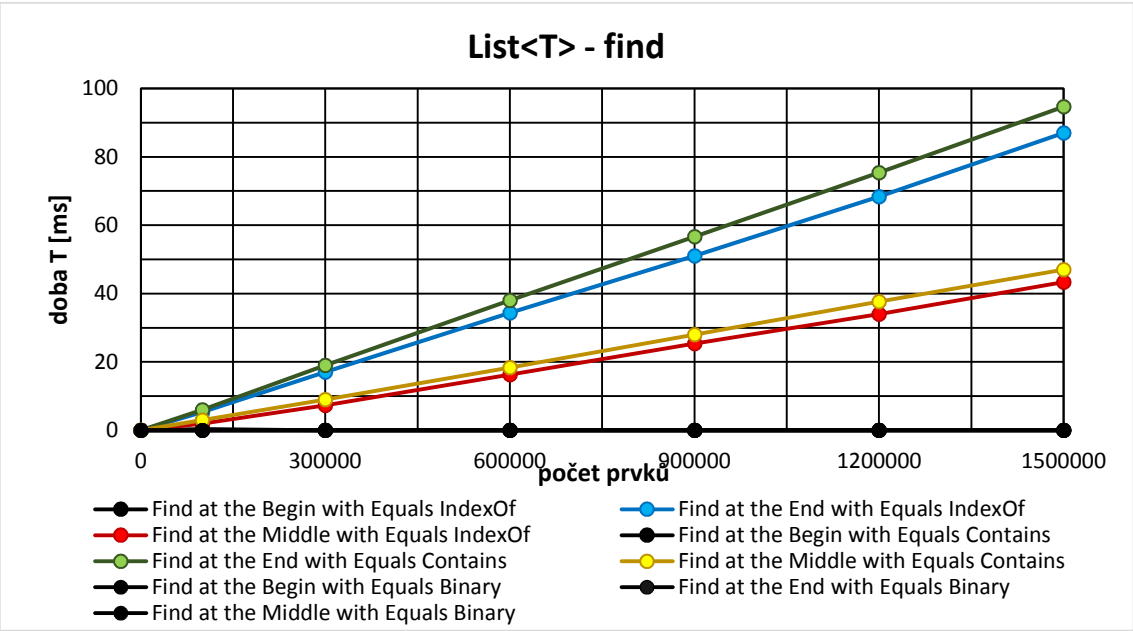


Graf 33: Kolekce List<T> - řazení

Tento graf List<T> - sort, pokud srovnáme s grafem MyList<T> zjistíme že jsou skoro stejné. Je to pochopitelné, protože obě kolekce používají metody volané ze třídy *Array*.

Operace hledání (Find)

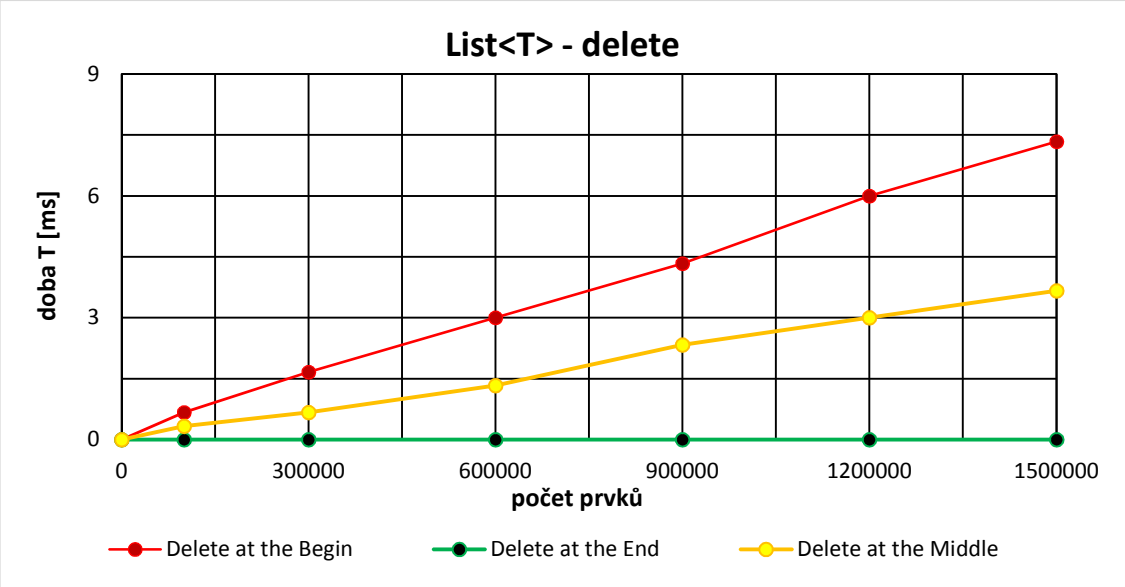
Pokud srovnáme tento graf s grafem MyList<T> - find, zjistíme, že jsou si hodně podobné. Obě kolekce opět používají metody ze třídy *Array*.



Graf 34: Kolekce List<T> - hledání

Operace mazání (Delete)

Tento graf je velmi podobný grafu MyList<T> - delete.

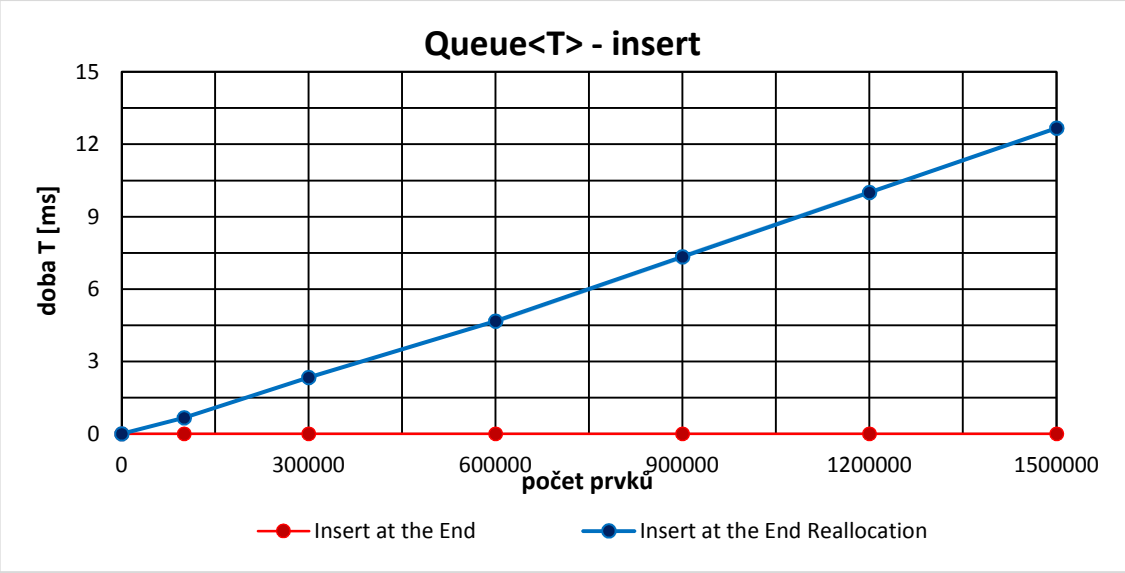


Graf 35: Kolekce List<T> - mazání

Kolekce Queue<T>

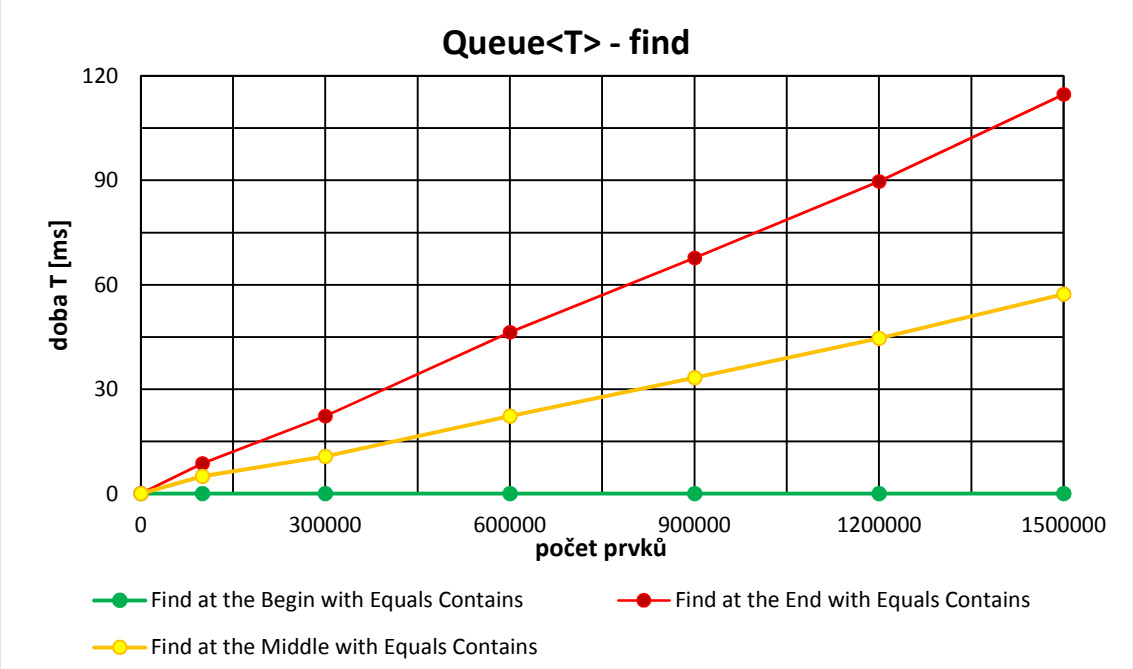
Operace vkládání (Insert)

Jelikož jsme nedělali vlastní implementaci této kolekce, můžeme jí pouze srovnat s jiným typem kolekcí. Například u kolekce List<T> - insert, vidíme značnou podobnost.



Graf 36: Kolekce Queue<T> - vkládání

Operace hledání (Find)



Graf 37: Kolekce Queue<T> - hledání

Operace mazání (Delete)

Pokud jsme odebírali prvek z fronty, tato operace byla okamžitá. Není třeba vynášet do grafu.

Operace řazení (Sort)

Operace řazení u fronty není vůbec pochopitelně implementována.

Kolekce Stack<T>

Operace řazení (Sort)

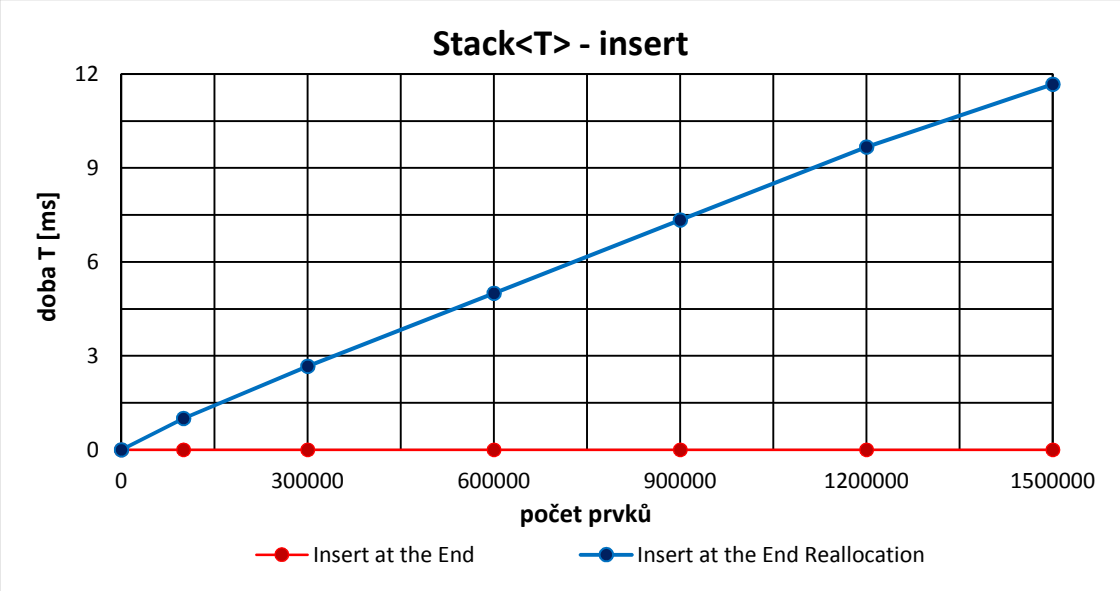
Operace řazení u zásobníku není vůbec pochopitelně implementována.

Operace mazání (Delete)

Pokud jsme vyndavali prvek ze zásobníku, tato operace byla okamžitá. Není třeba vynášet do grafu.

Operace vkládání (Insert)

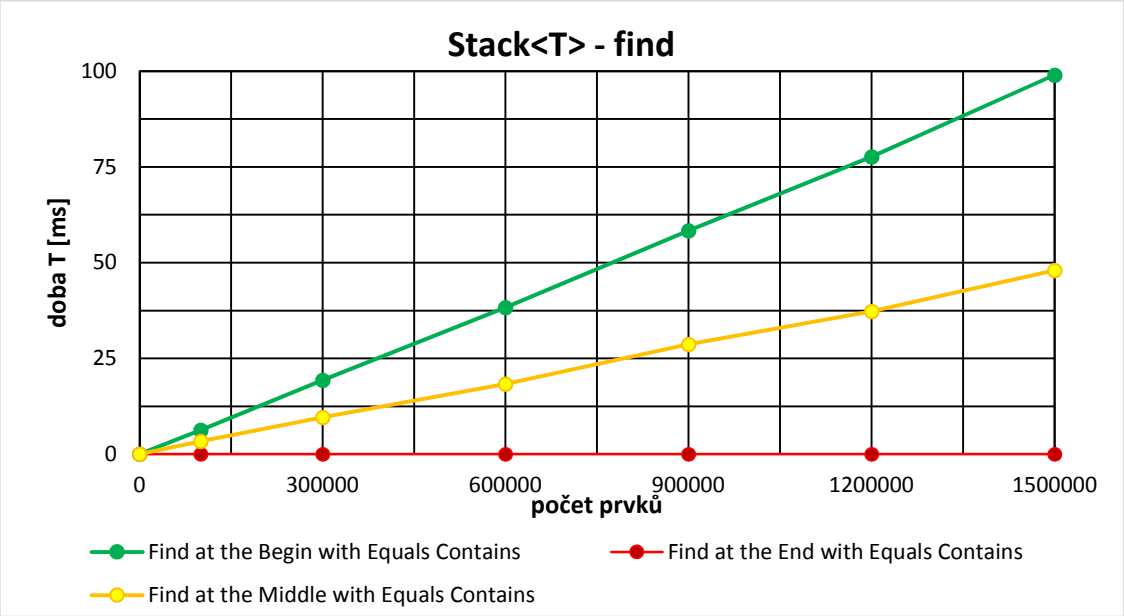
Operace vkládání do zásobníku, pokud to porovnáme s grafem Queue<T> - insert, je velmi podobná, co se týče časové náročnosti.



Graf 38: Kolekce Stack<T> - vkládání

Operace hledání (Find)

Pokud tento graf porovnáme s grafem Queue<T> - find, zjistíme podle časové náročnosti, že jsou si podobné.



Graf 39: Kolekce Stack<T> - hledání

Kolekce Dictionary<T>

Operace vkládání (Insert)

Operace vkládání do kolekce Dictionary<T> byla okamžitá. Z tohoto důvodu jsem jí vůbec nevynášel do grafu.

Operace řazení (Sort)

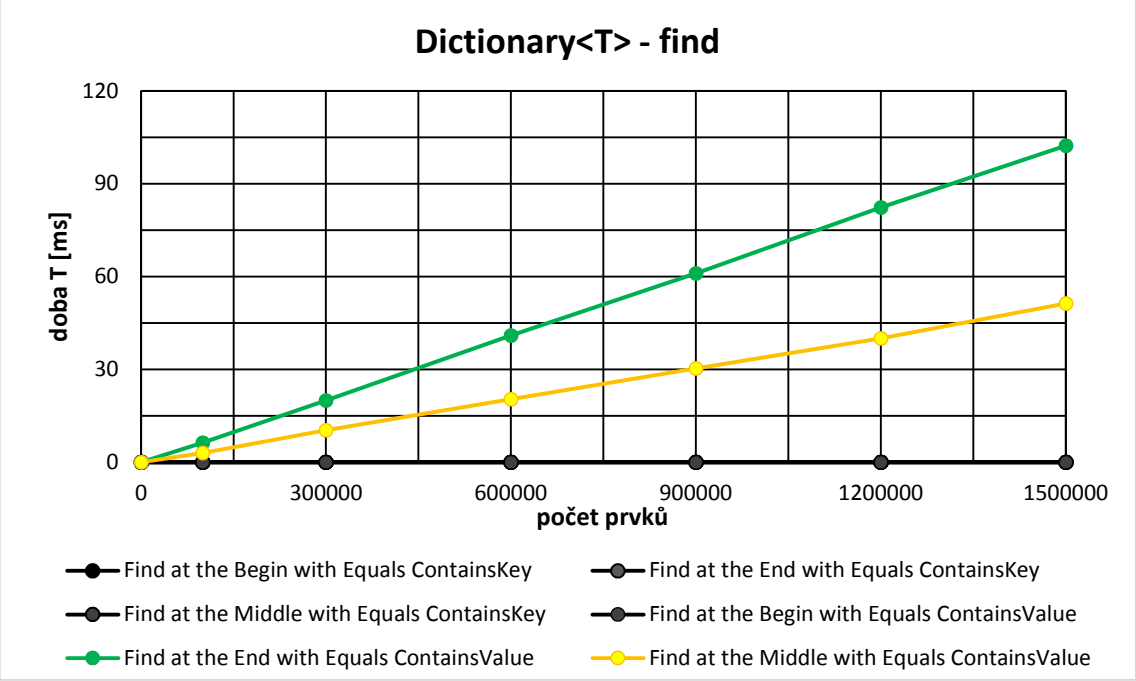
Operace řazení u slovníku není vůbec implementována.

Operace mazání (Delete)

Pokud jsme mazali prvek ze slovníku, tato operace proběhla okamžitě. Není třeba vynášet do grafu.

Operace hledání (Find)

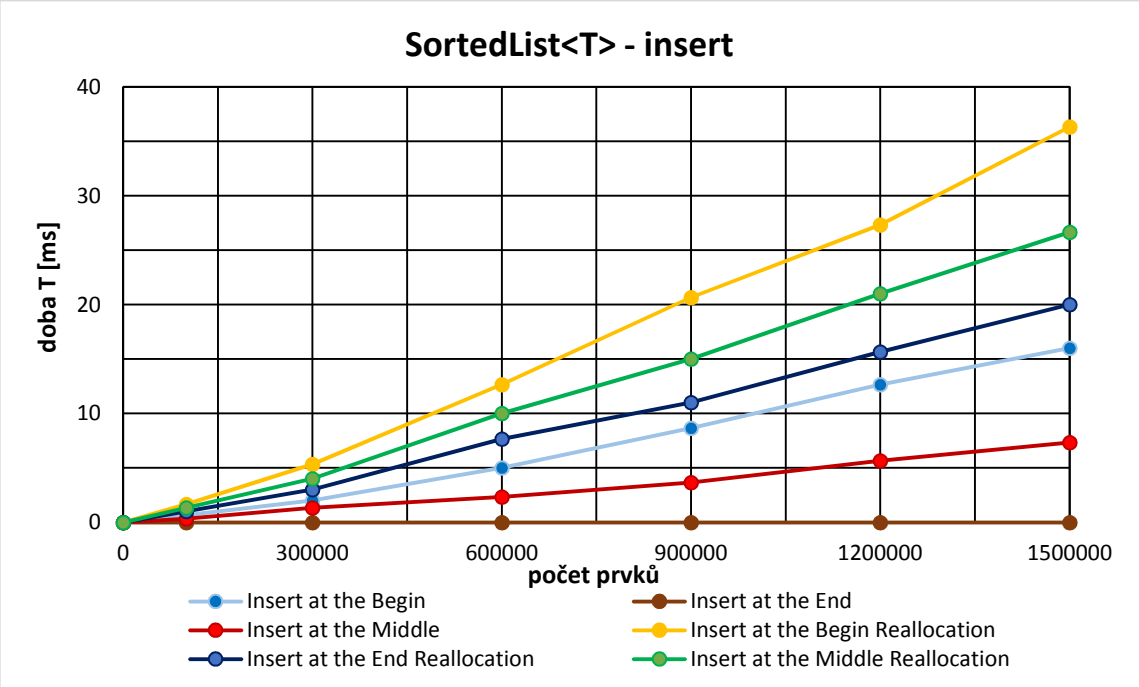
Pokud jsme slovník prohledávali podle klíče, tato operace proběhla okamžitě. Pokud jsme však slovník prohledávali podle hodnoty prvku, je vidět z grafu, že dochází k nárůstu časové režie podle počtu prvků. Tento graf bych eventuelně porovnal s grafem List<T> - find, kde hledání, podle časové náročnosti, zřejmě probíhá podobně.



Graf 40: Kolekce Dictionary<T> - hledání

Kolekce SortedList<T>

Operace vkládání (Insert)



Graf 41: Kolekce SortedList<T> - vkládání

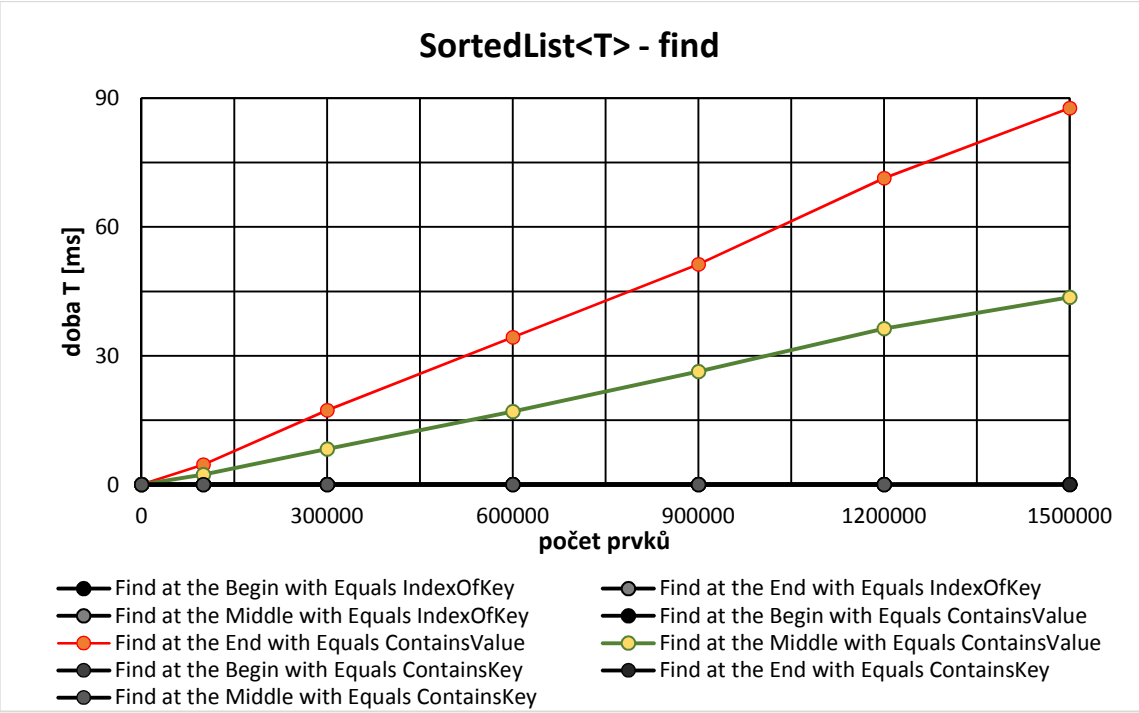
Tento graf bych doporučoval srovnat s grafem MySortedList – insert. Obě tyto kolekce volají metody z třídy *Array*. To opět značí to, že časová náročnost operace vkládání, podle počtu prvků v kolekci, bude u obou grafů velmi podobná. Mimo to to svědčí o tom, že jsme správně implementovali kolekci *MySortedList*.

Operace řazení (Sort)

Operace řazení u seřazeného seznamu pochopitelně není vůbec implementována.

Operace hledání (Find)

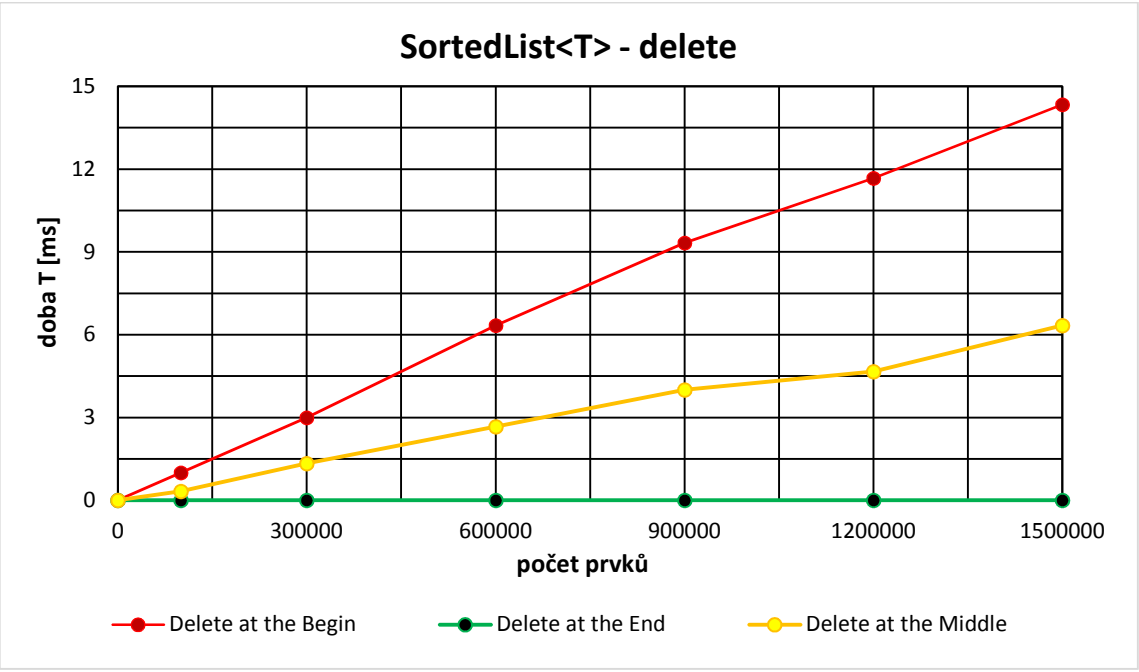
Pokud jsme seřazený seznam prohledávali podle klíče, tato operace proběhla okamžitě. Pokud jsme však seřazený seznam prohledávali podle hodnoty prvku, je vidět z grafu, že dochází k nárůstu časové režie podle počtu prvků. Tento graf bych srovnal s grafem MySortedList – find, a zjistíme že je velmi podobný. Obě kolekce používají metody z třídy *Array*.



Graf 42: Kolekce SortedList<T> - hledání

Operace mazání (Delete)

Pokud jsme mazali prvek ze seřazeného seznamu, je vidět z grafu, že dochází k nárůstu časové režie, podle počtu prvků v kolekci. Tento graf bych srovnal s grafem MySortedList – delete. Opět oba grafy jsou si podobný.



Graf 43: Kolekce SortedList<T> - mazání

Kolekce SortedDictionary<T>

Operace vkládání (Insert)

Operace vkládání do kolekce SortedDictionary<T> byla okamžitá. Z tohoto důvodu jsem jí vůbec nevynášel do grafu.

Operace řazení (Sort)

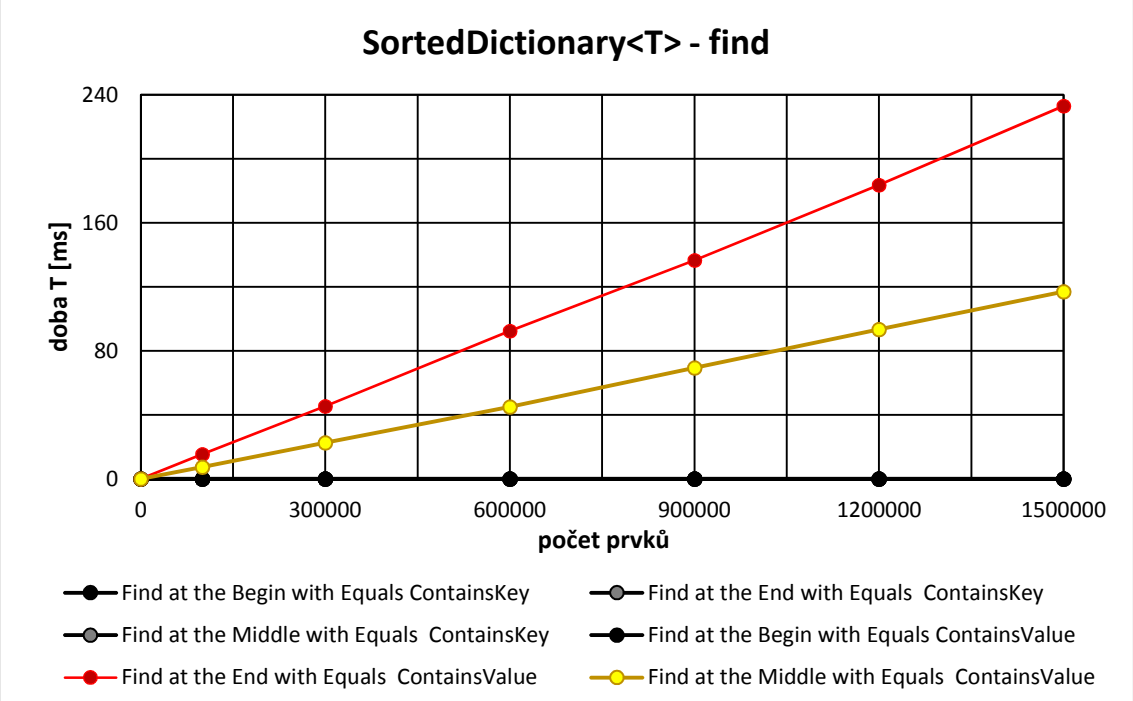
Operace řazení u seřazeného slovníku pochopitelně není vůbec implementována.

Operace mazání (Delete)

Pokud jsme mazali prvek ze seřazeného slovníku, tato operace proběhla okamžitě. Není třeba vynášet do grafu.

Operace hledání (Find)

Pokud jsme seřazený slovník prohledávali podle klíče, tato operace proběhla okamžitě. Pokud jsme však seřazený slovník prohledávali podle hodnoty, graf vykazuje jistou časovou režii v závislosti na počtu prvků v kolekci.



Graf 44: Kolekce SortedDictionary<T> - hledání

Kolekce LinkedList<T>

Operace vkládání (Insert)

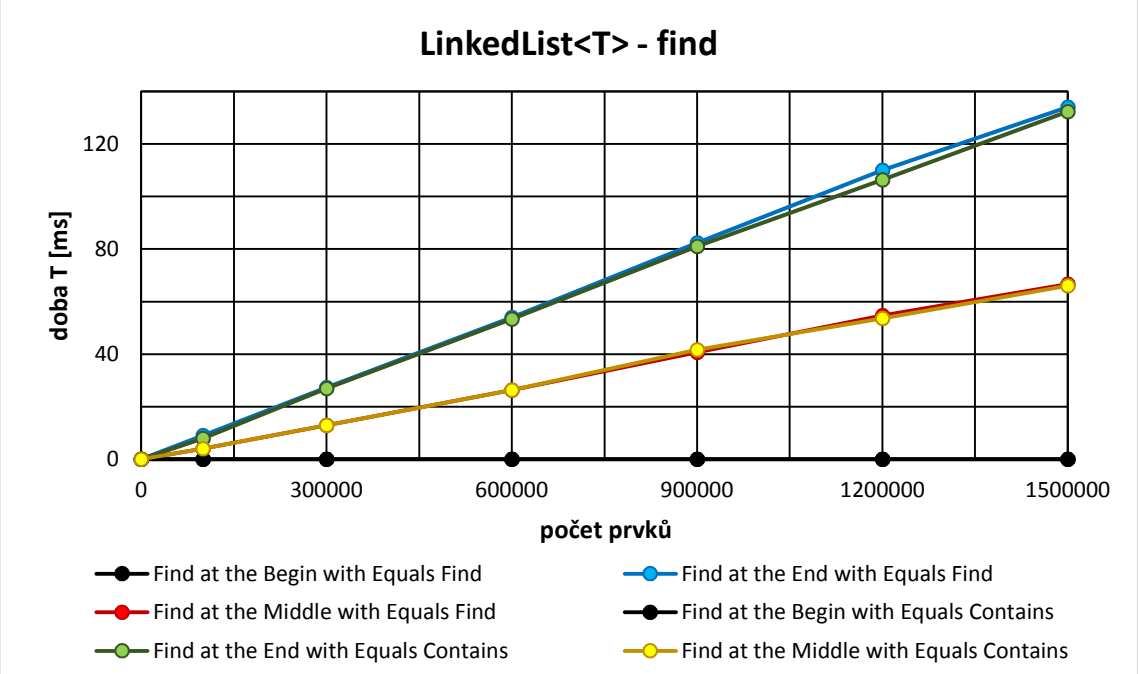
Operace vkládání do kolekce LinkedList<T> byla okamžitá. Z tohoto důvodu jsem jí vůbec nevynášel do grafu

Operace řazení (Sort)

Operace řazení u spojového seznamu není vůbec implementována.

Operace hledání (Find)

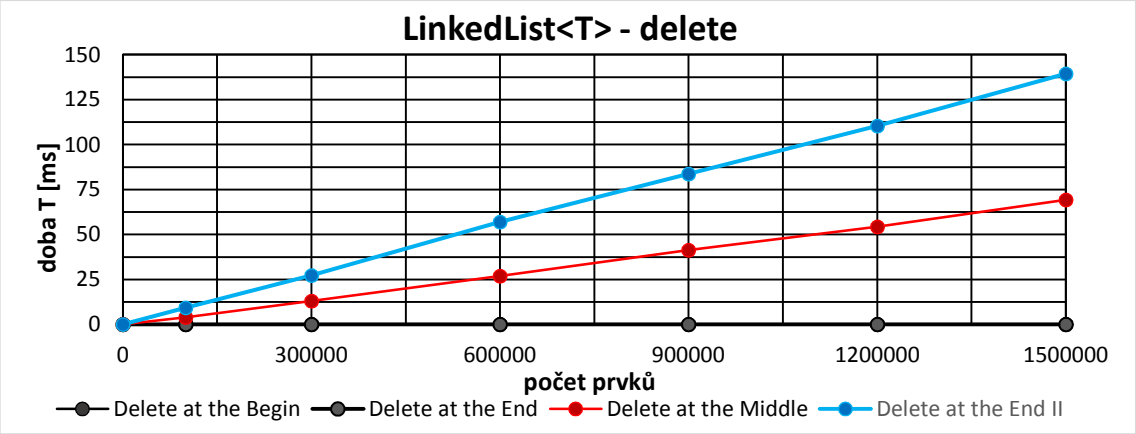
Zde evidentně probíhá sekvenční hledání, kdy se přechází z jednoho uzlu na následující a přitom se porovnává hodnota uzlu s hledanou hodnotou. Čím je hledaný prvek dál ve spojovém seznamu, tím je časová režie větší, podle lineární závislosti. Tento graf je dobré porovnat s grafem MyLinkedList<T> - find, zjistíme že oba grafy jsou si velmi podobné.



Graf 45: Kolekce LinkedList<T> - hledání

Operace mazání (Delete)

Tento graf je velmi podobný grafu MyLinkedList<T> - delete.



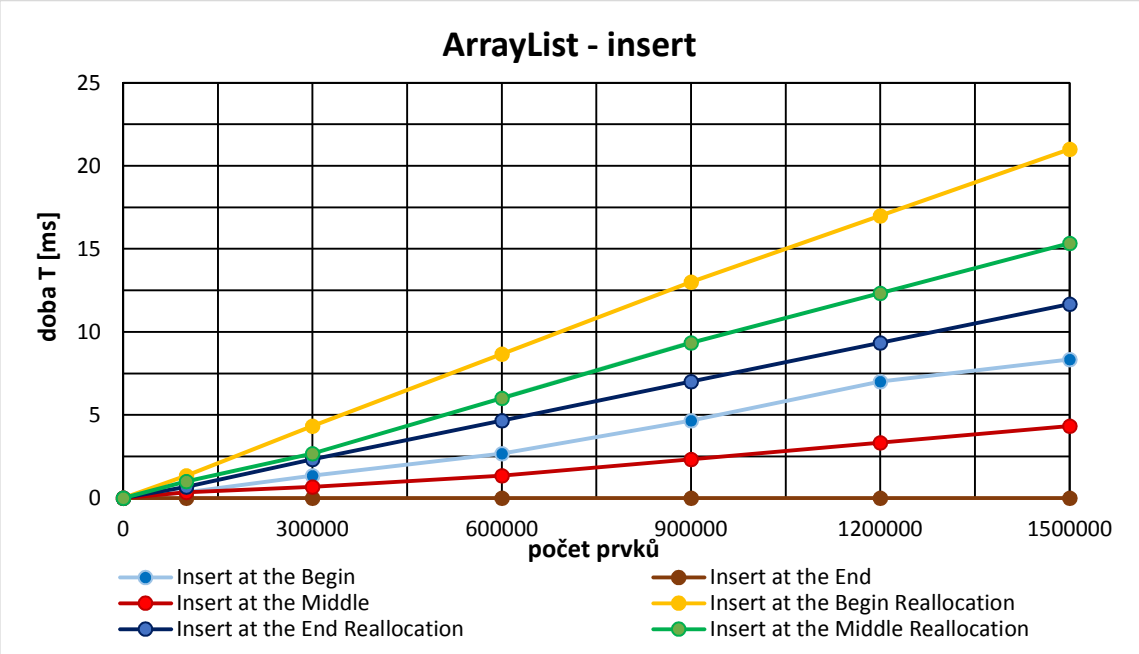
Graf 46: Kolekce LinkedList<T> - mazání

5.3 Beztypové kolekce

Kolekce ArrayList

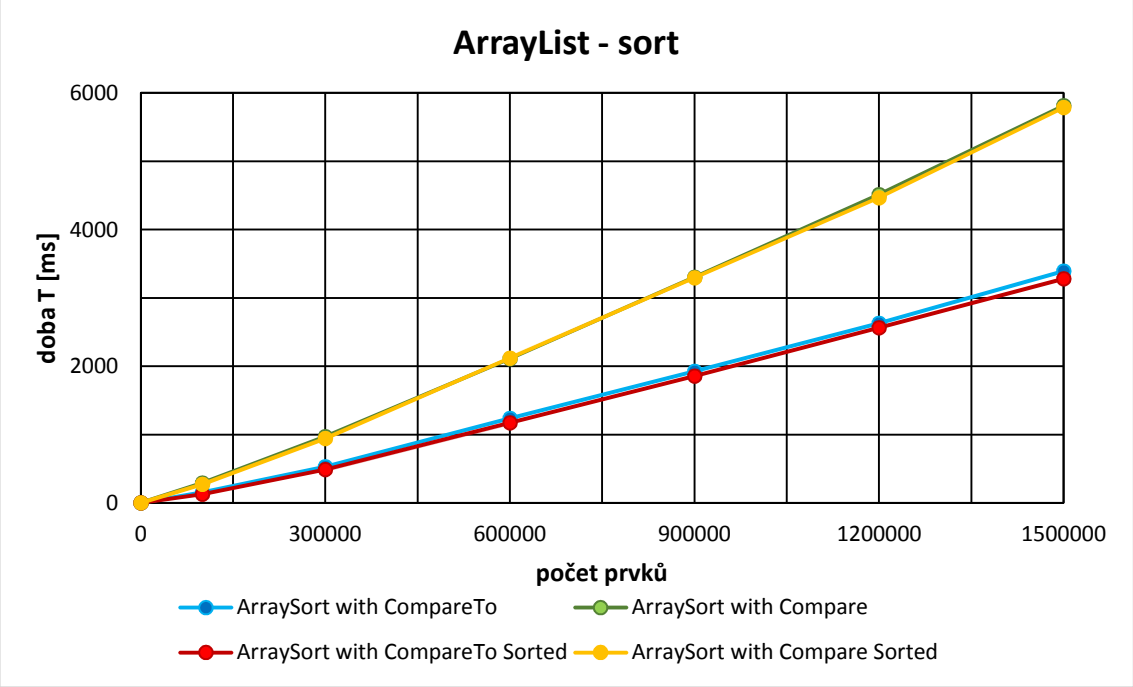
Teď budou následovat beztypové kolekce, pro které by byl komentář stejný jako pro předešlé typové kolekce, proto ho zde už nebudu psát, a jen uvedu grafy.

Operace vkládání (Insert)



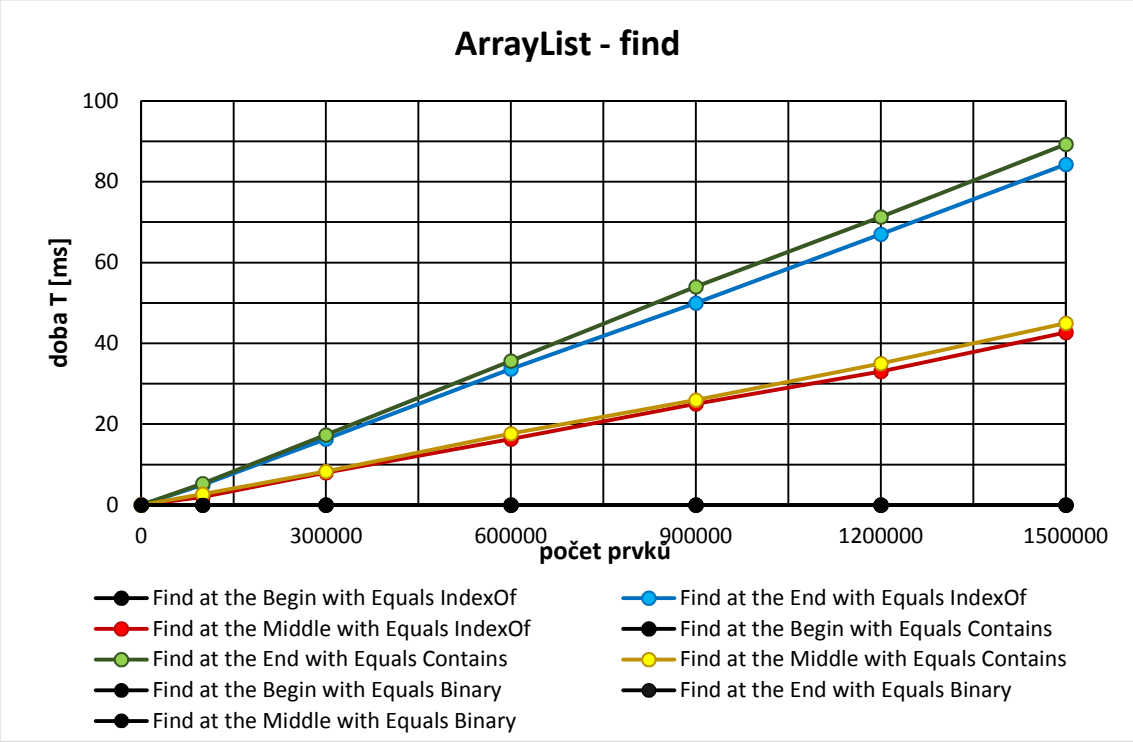
Graf 47: Kolekce ArrayList – vkládání

Operace řazení (Sort)



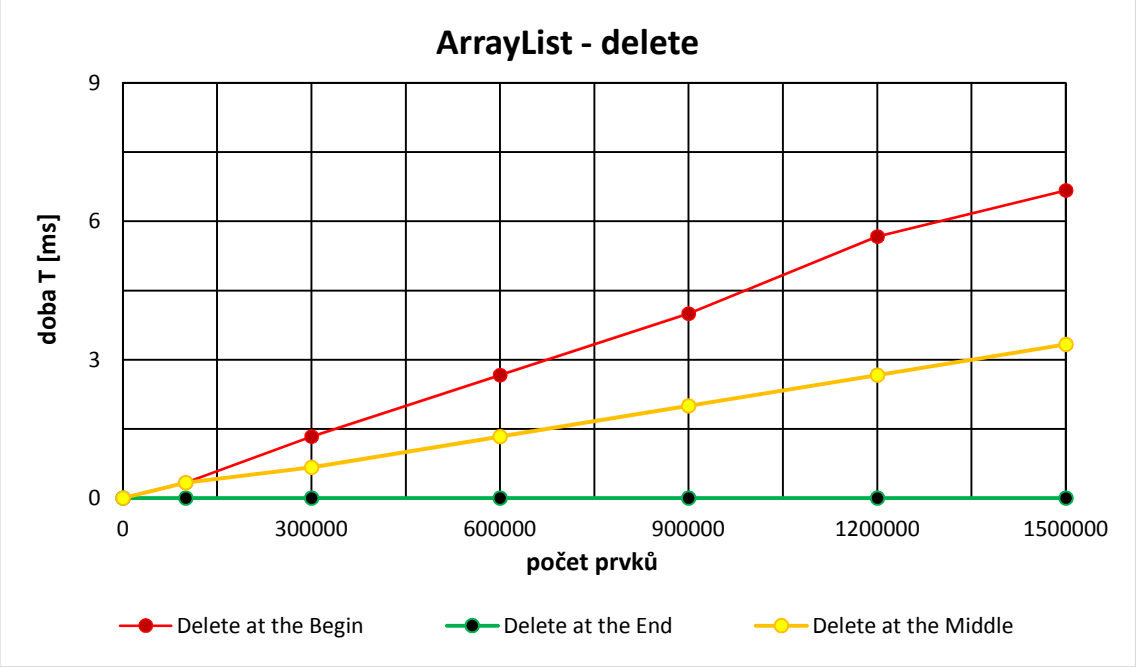
Graf 48: Kolekce ArrayList – řazení

Operace hledání (Find)



Graf 49: Kolekce ArrayList - hledání

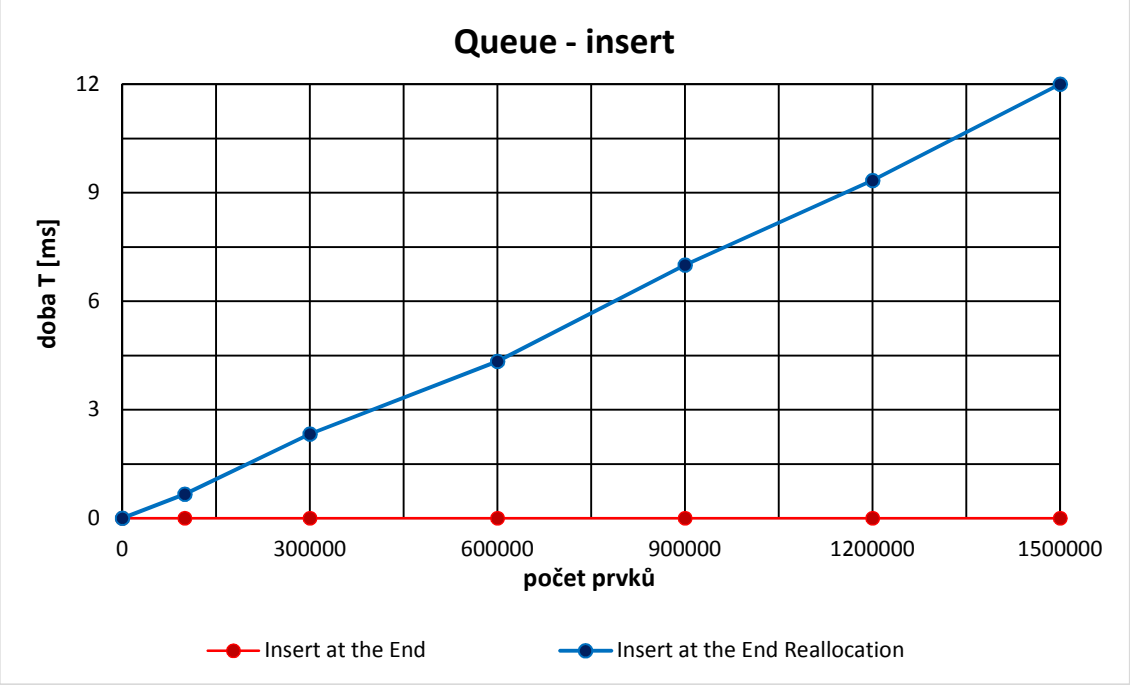
Operace mazání (Delete)



Graf 50: Kolekce ArrayList – mazání

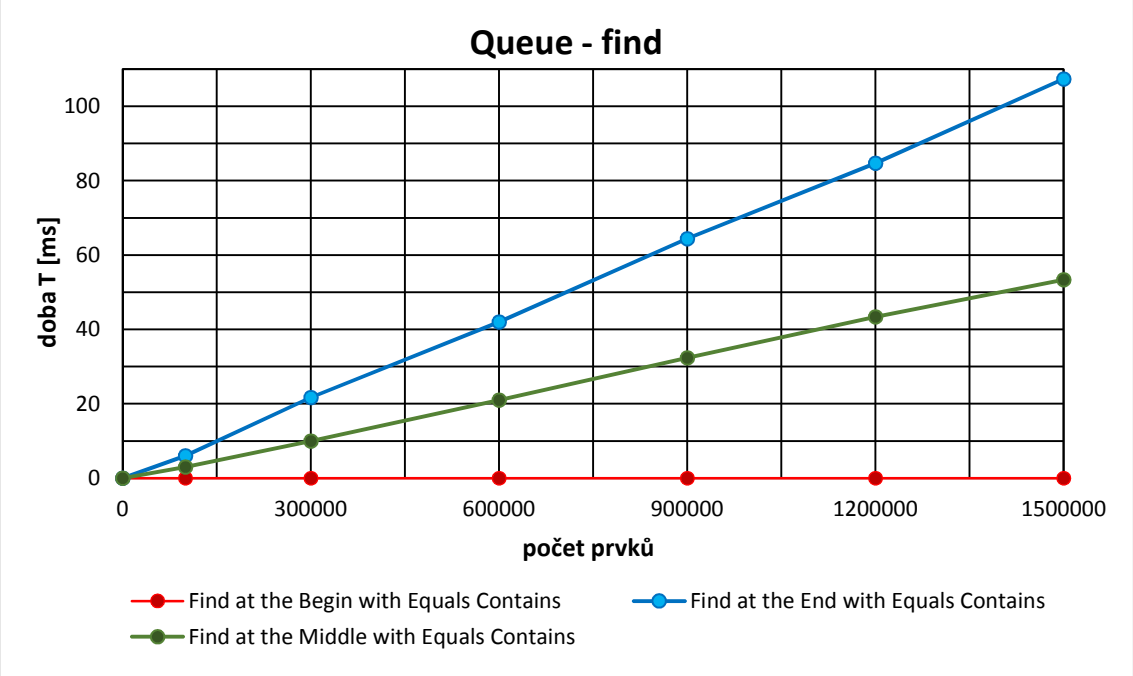
Kolekce Queue

Operace vkládání (Insert)



Graf 51: Kolekce Queue - vkládání

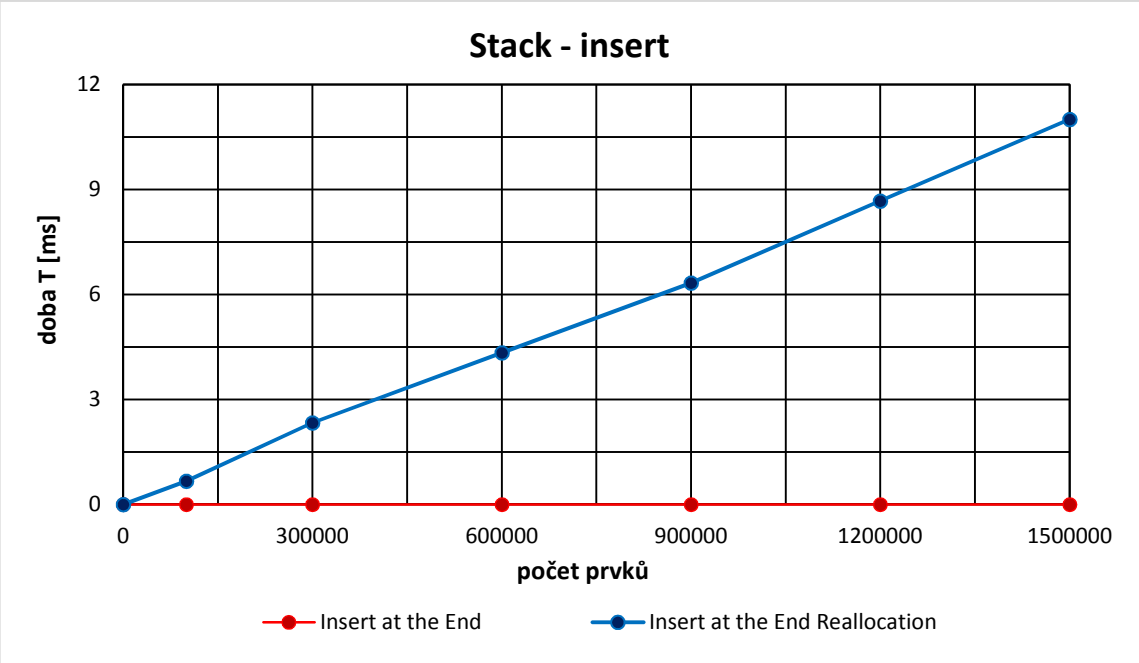
Operace hledání (Find)



Graf 52: Kolekce Queue – hledání

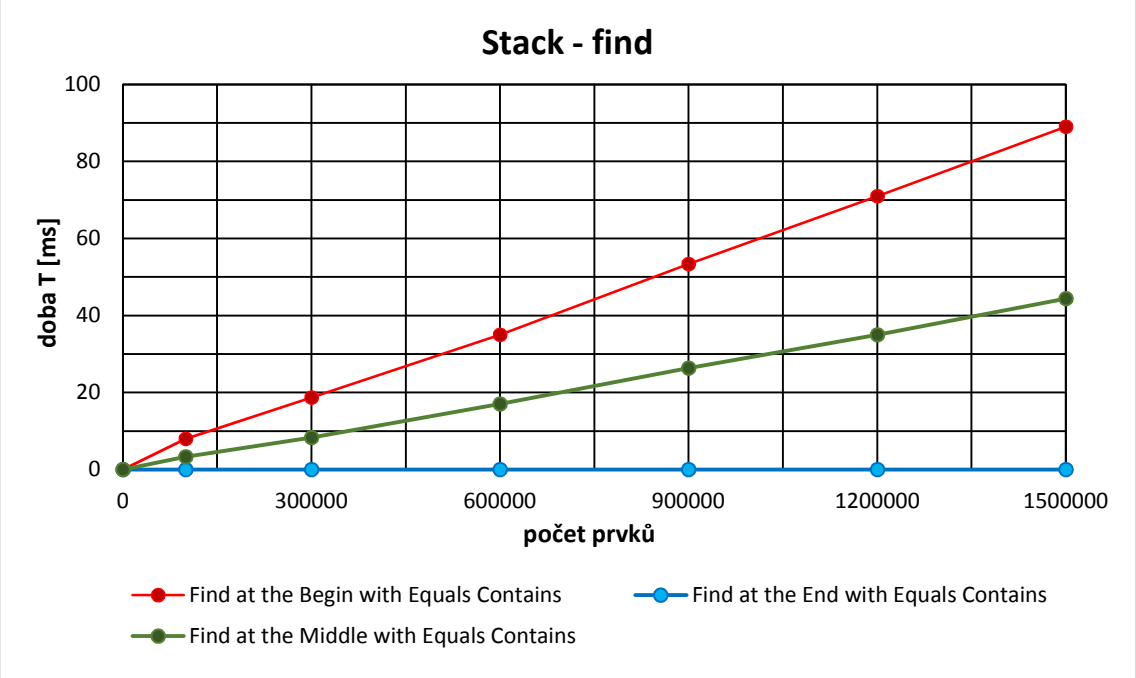
Kolekce Stack

Operace vkládání (Insert)



Graf 53: Kolekce Stack – vkládání

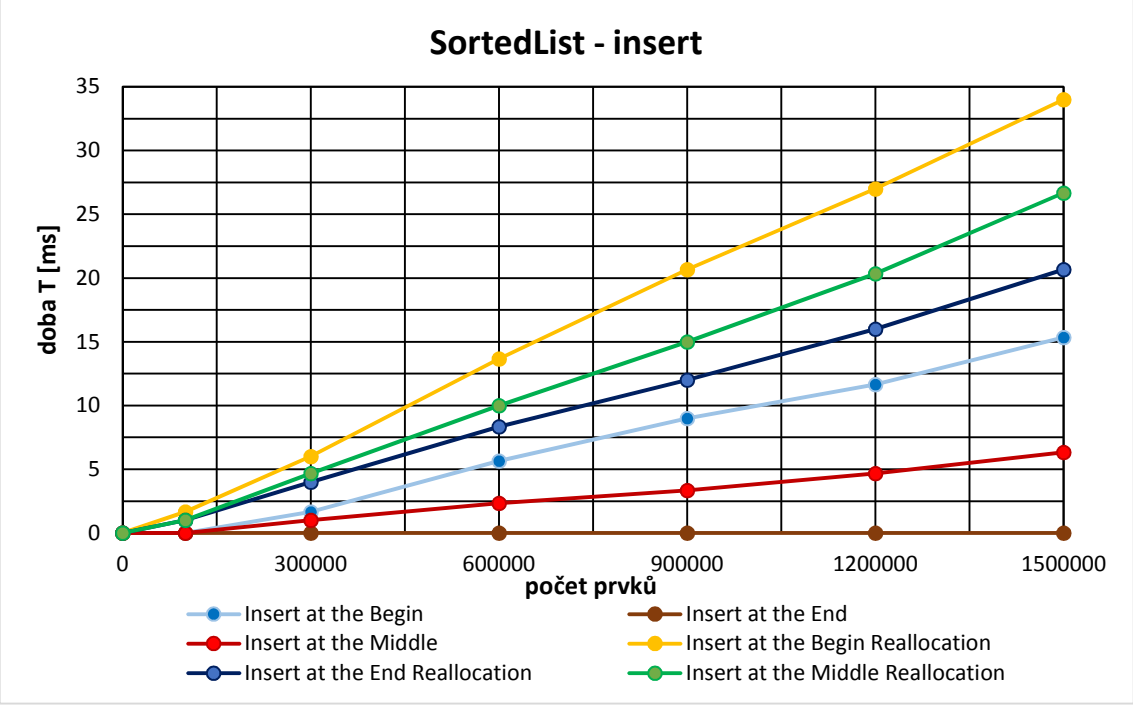
Operace hledání (Find)



Graf 54: Kolekce Stack – hledání

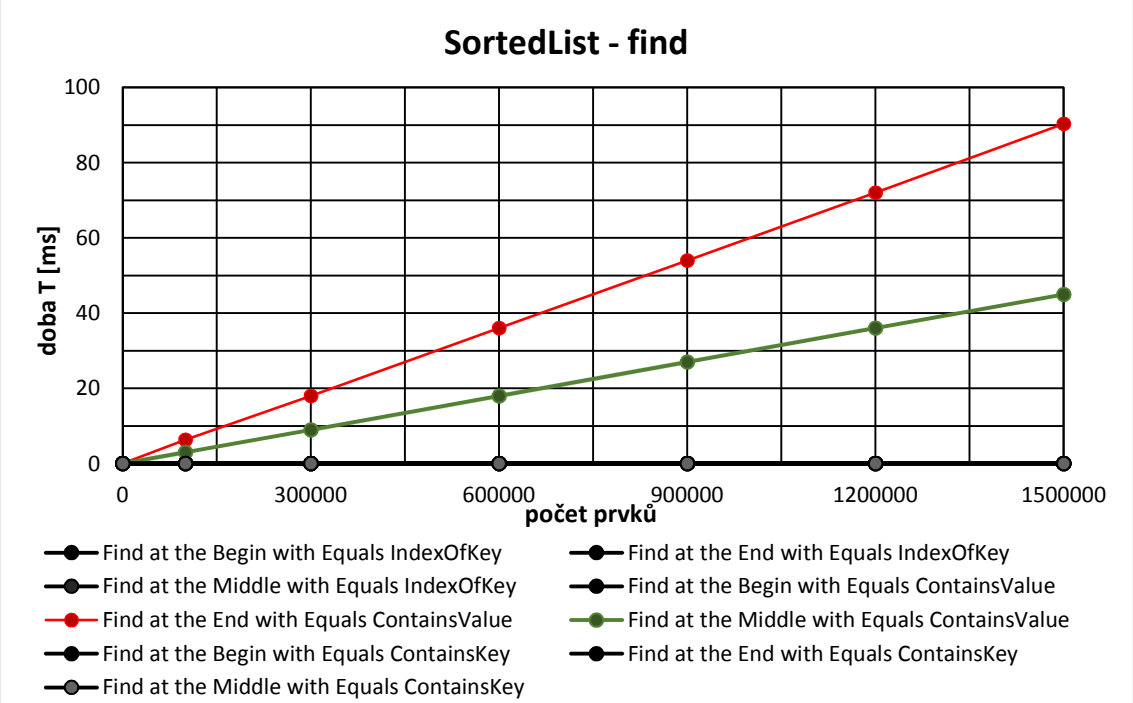
Kolekce SortedList

Operace vkládání (Insert)



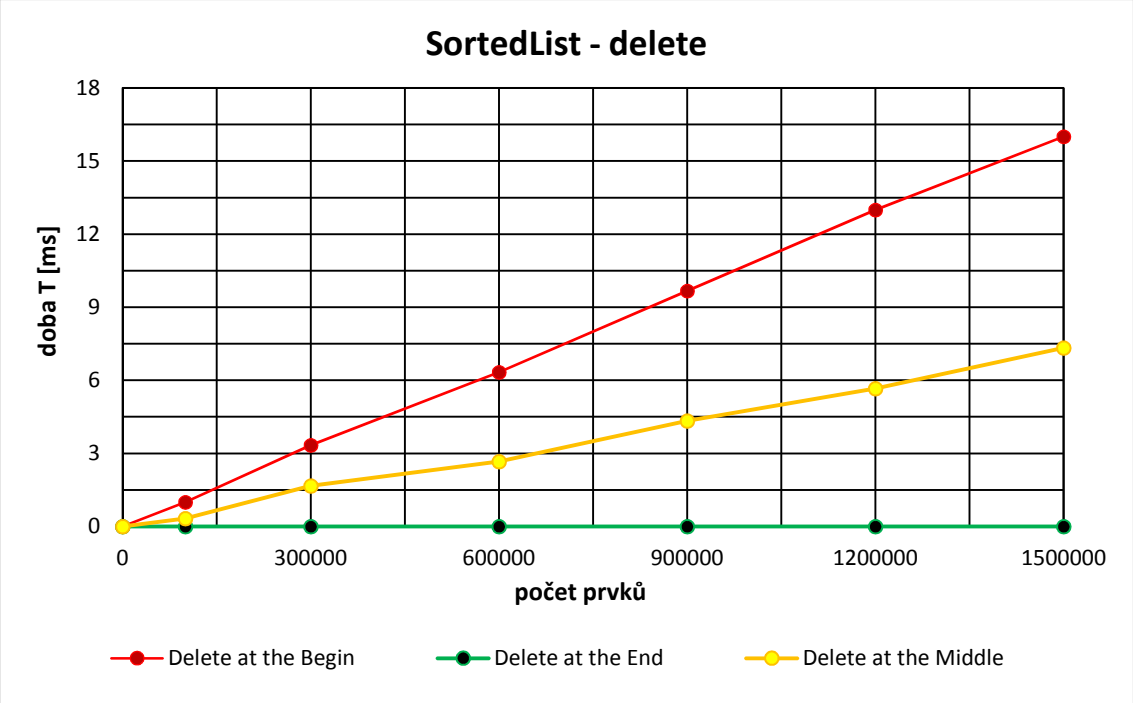
Graf 55: Kolekce SortedList - vkládání

Operace hledání (Find)



Graf 56: Kolekce SortedList - hledání

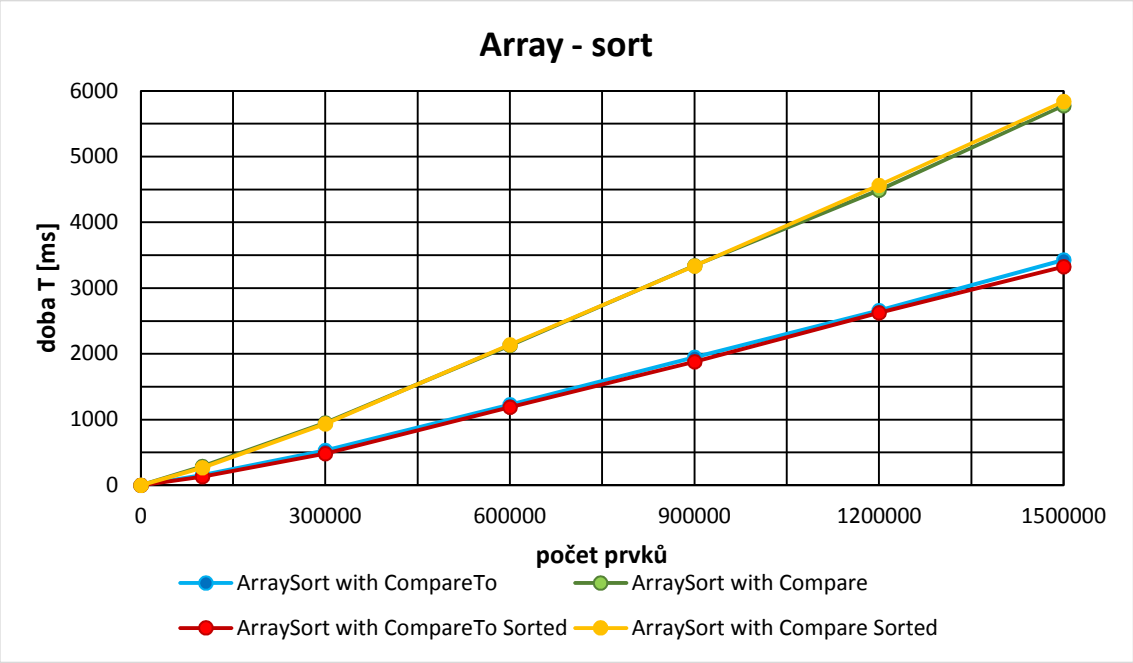
Operace mazání (Delete)



Graf 57: Kolekce SortedList - mazání

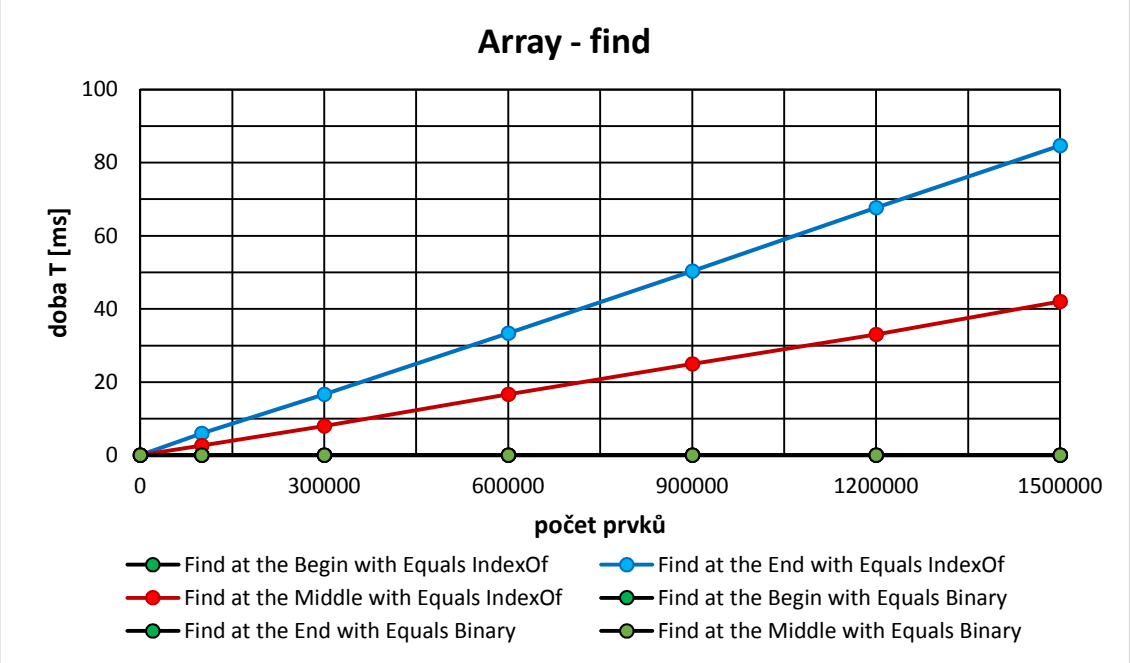
Třída Array

Operace řazení (Sort)



Graf 58: Kolekce Array - řazení

Operace hledání (Find)



Graf 59: Kolekce Array – hledání

5.4 Souhrnné vyhodnocení kolekcí

U všech kolekcí se všeobecně ukázalo, že vkládání prvku na konec kolekce je nejrychlejší. Při vkládání prvku do prostřed kolekce, časová náročnost začala lineárně narůstat. Nejvíce času při vkládání prvku do kolekce zabralo, vkládání prvku na začátek kolekce. Důvod je ten, že naše implementace *MyArrayList*, *MyList*, a zřejmě i kolekce .NET/C# jako například *ArrayList* a *List* používají ve své implementaci klasické pole. V tomto poli při vkládání prvku kdekoliv doprostřed, nebo na začátek, dochází k posunu všech zbylých prvků od indexu vkládaného prvku do konce pole. To je důvod proč časová náročnost lineárně roste. K dalšímu časovému nárůstu dochází, pokud při vkládání prvku do pole musí proběhnout navíc před tím realokace pole. Realokace pole probíhá tak, že se vytvoří nové pole o dvojnásobné velikosti a to původní se do něj nakopíruje.

Co se týká mazání prvku z kolekce, je postup přesně obrácený, než při vkládání prvku do kolekce. Na index mazaného prvku v poli se posunou všechny zbylé prvky do konce pole.

Když byla testována operace řazení u kolekcí .NET/C# jako například *ArrayList* a *List* ukázalo se, že časová náročnost roste lineárně v závislosti na počtu prvků. Dále se ukázalo, že nezáleží, jestli byly před tím prvky již seřazené nebo ne. Tuto vlastnost řazení vykazuje algoritmus *HeapSort*. Dále byly implementovány vlastní algoritmy řazení a diskutována jejich vhodnost pro řazení prvků.

Co se týká operace vyhledávání prvků v našich implementacích, jedná se o dvě základní metody. Sekvenční vyhledávání, kde se prochází postupně celé pole až k hledanému prvku, a binární vyhledávání, kde je využita metoda půlení intervalu. Na základě porovnání časové náročnosti této operace a operace vyhledávání u kolekcí .NET/C# jsme odhadli, že se jedná o týž algoritmus.

Nakonec jsme ještě dospěli k závěru, že některé kolekce v .NET/C# jako například *ArrayList* a *List*, používají pro některé své operace *Copy*, *BinarySearch*, *IndexOf* a *Sort* metody volané ze třídy *Array*. Když jsme se ve svých implementacích pokusili tyto metody nahradit vlastními metodami ze třídy *Algoritmy* (*AlgoritmyTYP*), časová náročnost těchto operací značně narostla.

6 Závěr a další kroky

V této práci byla úspěšně otestována rychlost kolekcí v .NET/C# a vynesena do grafů. Byly především testovány metody kolekcí vkládání, řazení, vyhledávání a mazání v závislosti na počtu prvků v kolekci.

Dále byly úspěšně implementovány a testovány vlastní implementace kolekcí. Byl potvrzen předpoklad, že některé metody vlastní implementace kolekce jsou více časově náročné k provedení daného úkolu, než metody standardní kolekce v .NET/C#. Z porovnání časů potřebných pro provedení dané operace, což bylo uděláno při porovnávání grafů, mezi standardní kolekcí v .NET/C# a naší vlastní implementací kolekcí, se došlo k následujícím závěrům. Třídy *List*, *ArrayList* a *SortedList* používají ve svých metodách některé metody z třídy *Array*. Používané metody z třídy *Array* jsou především *Copy*, *BinarySearch*, *Sort*, *IndexOf*. Pokud ve vlastních implementacích kolekcí byly použity též tyto metody z třídy *Array*, časová náročnost k provedení daného úkolu mezi naší kolekcí a kolekcí v .NET/C# byla velmi podobná. V momentě kdy i tyto metody z třídy *Array* byly nahrazovány vlastními, časová náročnost naší vlastní implementace kolekce začala narůstat.

Z porovnání časů potřebných k seřazení prvků v kolekci bylo dojito k závěru, že standardní kolekce *List* a *ArrayList* používají pro řazení prvků algoritmus *HeapSort*. V této práci mimo jiné byla provedena vlastní implementace známých algoritmů řazení a diskutována vhodnost jejich použití k řazení prvků v kolekci.

Též v této práci byly řešeny způsoby vyhledávání prvků v kolekci. Binární vyhledávání se ukázalo jako velmi efektivní oproti sekvenčnímu, kde se postupně prochází všechny prvky v kolekci. Dále bylo prokázáno, že spojový seznam *LinkedList*, který je implementovaný jako uzly spojené mezi sebou odkazy, je pro operaci sekvenčního prohledávání podle hodnot, mnohem více časově náročný, než například sekvenční prohledávání kolekcí *List* nebo *ArrayList*. Je to pochopitelné za předpokladu, že například *ArrayList* je definovaný jako pole typu *Object*.

Jako další krok by bylo vhodné se pokusit o vlastní implementaci zbylých kolekcí Queue, Stack, Dictionary, SortedDictionary, Hashtable. Dále by bylo vhodné program rozšířit o možnost uložit více prvků do kolekce a znovu testovat ty operace, které v našem případě proběhly okamžitě. Vzhledem k tomu, že kolekce disponují vlastními

enumerátory, které slouží pro procházení kolekcí, mohlo by se také testovat, s jakou rychlostí tyto kolekce prochází. Poslední věc, kterou bych doporučil je, že bych obohatil program o další algoritmy řazení.

Seznam použité literatury

[1] VIRIUS, Miroslav. *Jazyky C a C: Komplexní průvodce -2. /.* aktualizované vydání. Praha: Grada, 2011, 367 s. Knihovna programátora. ISBN 978-80-247-3917-5.

[2] VIRIUS, Miroslav. *C# 2010: hotová řešení*. 1. vyd. Brno: Computer Press, 2012, 424 s. K okamžitému použití (Computer Press). ISBN 978-80-251-3730-7.

[3] HEROUT, Pavel. *Učebnice jazyka C: hotová řešení*. 6. vyd. České Budějovice: Kopp, 2009, 271, viii s. K okamžitému použití (Computer Press). ISBN 978-80-7232-383-8.

[4] KERNIGHAN, Brian. *Programovací jazyk C: hotová řešení*. Vyd. 1. Brno: Computer Press, 2006, 286 s. K okamžitému použití (Computer Press). ISBN 80-251-0897-X.

[5] GUNNERSON, Eric. *Začínáme programovat v C #*. 1. vyd. Praha: Computer Press, 2001, 316 s. ISBN 80-722-6525-3.

[6] KEOGH, Jim. *Datové struktury bez předchozích znalostí*. Vyd. 1. Brno: Computer Press, 2006, 223 s. ISBN 80-251-0689-6.

[7] HUDEC, Bohuslav. Algoritmy v C++. Vyd. 1. Jihlava: Vysoká škola polytechnická Jihlava, 2011, 173 s. 0251664896

[8] Binární vyhledávání. *Algoritmy.net* [online]. [cit. 2015-02-05]. Dostupné z: http://www.algoritmy.net/article/21/Binarni-vyhledavani

[9] CORMEN, Thomas H. *Introduction to algorithms*. 3rd ed. Cambridge: MIT Press, 2009, xix, 1292 s. ISBN 978-0-262-03384-8.

[10] KNUTH, Donald Ervin. *The art of computer programming*. 3rd ed. Upper Saddle River: Addison-Wesley, 1998, xiii, 780 s. ISBN 02-018-9685-0.

[11] Vyhledávací algoritmy: Sekvenční vyhledávání. *Manualy.net* [online]. 2007 [cit. 2015-05-18]. Dostupné z: http://www.manualy.net/article.php?articleID=23

[12] Interpolační vyhledávání. *Itnetwork.cz* [online]. 2015 [cit. 2015-05-18]. Dostupné z: http://www.itnetwork.cz/algoritmus-interpolacni-vyhledavani-setridene-pole

[13] Selection Sort Vs Insertion Sort. *Cheetahonfire.blogspot.cz* [online]. 2009 [cit. 2015-05-18]. Dostupné z: http://cheetahonfire.blogspot.cz/2009/05/selection-sort-vs-insertion-sort.html

[14] Quicksort. *Algoritmy.net* [online]. 2013 [cit. 2015-05-18]. Dostupné z: http://www.algoritmy.net/article/10/Quicksort

[15] Bubble Sort. *Algolist.net* [online]. 2009 [cit. 2015-05-18]. Dostupné z: http://www.algolist.net/Algorithms/Sorting/Bubble_sort

[16] Binární halda. *Algoritmy.net* [online]. 2011 [cit. 2015-05-18]. Dostupné z: http://www.algoritmy.net/article/15/Binarni-halda

Seznam obrázků

Seznam grafů

Seznam tabulek

Seznam použitých zkratek

FIFO datová struktura fronta (first in, first out)

LIFO datová struktura zásobník (last in, first out)

Přílohy

1 Obsah přiloženého CD

Na přiloženém CD se v kořenovém adresáři nachází tato bakalářská práce ve formátu *bakalarska_prace.pdf* s jednoduchým návodem *navod.txt* pro obsluhu programu.