17V2

Tahle otázka je úplně špatně :)

**SEŘAZENÉ**

SortedDictionary

SortedSet

SortedList (cringe)

**NESEŘAZENÉ**

Zbytek

### Uložení dat

Větší programy, nejspíš budou obsahovat více dat. Pamatujte, počítač by si měl vše pamatovat, ne vy. Představme si, že si chceme uložit něco kolem milionu čísel. Některá tam mohou být vícekrát, některá tam nemusí být ani jednou. V čem je uložit?

Uložit do pole bude jednoduché, ale bude vznikat hodně jiných problémů. Budeme třeba chtít najít maximum, ale co když z pole bude právě maximum odstraněné. Jednak nám v poli budou vznikat díry a jednak pro každé vyhledání maxima budeme muset pole prohledat znovu. Je lepší použít jinou strukturu.

### Kterou strukturu kdy použít

Strukturu vybereme tedy podle toho, jaké operace bychom chtěli nejčastěji provádět nad danými daty. Jaké operace chceme? Například najít prvek, najít maximum/minimum, najít předchůdce prvku, vložit prvek, odstranit, seřadit prvky atd...

# **Seřazené datové struktury**

### Pole

V datové struktuře pole můžete uložit více proměnných **stejného typu**. Deklarujete pole zadáním typu jeho prvků. Chcete-li, aby pole ukládalo prvky libovolného typu, můžete jako typ zadat object. Pole je v podstatě hromada krabic poskládaných vedle sebe a k tomu očíslovaných přirozenými čísly. Je dobrým zvykemčíslovat od nuly**.**

*Create:*

type[] nazevPole;

### Jednorozměrné pole

Jednorozměrné pole si z matematického hlediska je tedy možné představit jako konečnou množinu prvků.

Samotná definice se skládá ze dvou částí. V první definujeme proměnnou typu pole. V dalším kroku je třeba tuto proměnnou inicializovat, tj. vytvořit instanci pole. To se provádí klíčovým slovem new. Tyto dva kroky se většinou spojí do jednořádkové deklarace, která vypadá následovně.

datový\_typ[] pole = new datový\_typ[počet\_prvků];

Pokud chceme získávat, nebo upravovat hodnoty určitého prvku v poli, ukazujeme na pole pomocí indexů. Prvky v poli se indexují od nuly, takže za předpokladu, že je v poli *n* prvků, tak poslední bude mít index (n – 1) a první samozřejmě 0. K prvku se dostaneme pomocí názvu pole a do hranatých závorek uvedeme index prvku.

int[] pole = new int[12];

pole[0] = 963;

**Délka pole** - Pole má svoje vlastnosti a metody. Důležitou vlastností pole je jeho délka - Length. U vícerozměrného pole potom počet sloupců a řádku získáváme pomocí metody GetLength(), přičemž jestli se jedná o řádek či sloupec stanovujeme argumentem funkce. Pokud jako argument uvedeme nulu, vrací počet řádků a pokud jedničky, počet sloupců.

### List

List je oproti poli dynamická datová struktura, můžeme ho tedy měnit za běhu, přidávat, ubírat, ptát se na jeho velikost atd.

Navíc lze List převést do pole toArray(). Jde jen o to, že list zkopíruje všechny své prvky do nového pole se stejnou velikostí jako má list.

List je chytřejší, dynamické pole, tzn. interně dělá to, co jsme si popsali výše u pole a co jsme většinou líní implementovat ručně. Všechny časové složitosti, týkající se vyhledávání, vložení, smazání atd. jsou stejné jako u pole. Výhoda je však ta, že místo složité syntaxe, kterou jsme museli dříve popisovat chování funkce, můžeme jednoduše volat jednotlivé metody přímo na listu: List.Add(), List.Delete(), List.Sort(), List.Contains()... List je vnitřně implementován pomocí pole, při přidání nového záznamu se tedy při vyčerpání kapacity vytvoří interně pole nové a prvky se do něj zkopírují.

List je tzv. **generická kolekce**. Při deklaraci Listu musíme specifikovat datový typ objektů, které v něm budou uloženy.

Vlastnost:

**Count** - Funguje jako Length na poli, vrací počet prvků v kolekci.

Metody:

**Add(položka)** - Metodu Add() jsme si již vyzkoušeli, jako parametr bere položku, kterou vloží na konec listu.

**AddRange(kolekce)** - Přidá do listu více položek, např. z pole.

**Clear()** - Vymaže všechny položky v listu.

**Contains(položka)** - Vrací true/false podle toho, zda List obsahuje předanou položku.

**IndexOf(položka)** - Vrátí index prvního výskytu položky (jako u pole). Vrací -1 při neúspěchu.

**Insert(index, položka)** - Vloží položku na daný index (pozici) v Listu.

**Remove(položka)** - Vymaže první nalezenou položku.

**RemoveAt(index)** - Vymaže položku na daném indexu.

**Sort()** - Sort() již také známe, setřídí položky v listu. Metoda opět nic nevrací.

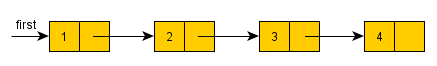
### Spojový seznam(LinkedList)

Spojový seznam (Lineární seznam, Linked list) je kontejner určený k ukládání dat předem neznámé délky. Základní stavební jednotkou spojového seznamu je uzel, který vždy obsahuje ukládanou hodnotu a ukazatel na následující prvek.

Varianty spojového seznamu(LinkedList)

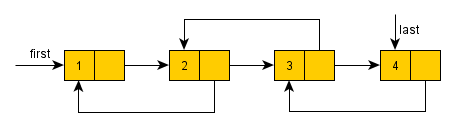
Jednosměrně zřetězený

Jednosměrně zřetězený seznam je základní variantou této datové struktury, ve které jednotlivé uzly obsahují kromě dat pouze ukazatel na další uzel (poslední uzel ukazuje na *null*), čímž umožňují pouze traverzování jedním směrem. Samotná struktura seznamu pak obsahuje pouze ukazatel na první prvek a data jsou přidávána vždy na začátek seznamu. Dílčím vylepšením je přidání ukazatele také na poslední prvek a umisťování nových prvků na konec seznamu.



#### Obousměrně zřetězený spojový seznam

V obousměrně zřetězeném spojovém seznamu prvky obsahují nejen ukazatel na další prvek, ale také ukazatel na předchozí prvek. Tímto se sice poněkud zkomplikuje implementace struktury, ale toto je vyváženo zvýšenou flexibilitou, protože lze traverzovat v obou směrech.



Kruhový spojový seznam  
Vyhledávání ve spojovém seznamu můžeme zjednodušit použitím zarážky – zarážka je uzel zvláštního typu umístěný na konec seznamu (v případě použití hlídky bude hlídka zarážkou). Do zarážky vždy umístíme data, která vyhledáváme, čímž si zajistíme, že je také najdeme. Tímto trikem eliminujeme nutnost neustálé kontroly toho, jestli již nejsme na konci seznamu.

#### Iterátor

Pro efektivní využití libovolného typu spojového seznamu je zapotřebí vytvořit tzv. *iterátor*, což je objekt, který při průchodu seznamem udržuje odkaz na aktuální pozici a umožní z ní pokračovat dále nebo se vrátit zpět (při použití obousměrně zřetězeného seznamu). Bez této struktury by byl průchod seznamem neúnosně drahou operací, protože by se při každém přístupu na další položku musel tento prvek vždy nejprve vyhledat (tj. projít celý seznam od začátku až na danou pozici).

### SortedSet

SortedSet je kolekce objektů v seřazeném pořadí. Jedná se o obecný typ kolekce a je definován v System.Collections.Generic namespace.

Jedná se o dynamickou kolekci, což znamená, že velikost SortedSet se automaticky zvyšuje, když jsou přidány nové prvky.

Třída SortedSet implementuje rozhraní ICollection, IEnumerable, IReadOnlyCollection, ISet, ICollection, IEnumerable, IDeserializationCallback a ISerializable interface.

-Kapacita SortedSet je počet prvků, které může obsahovat.

-V SortedSet musí být prvky jedinečné.

-V SortedSet je pořadí prvku vzestupné.

-Obvykle se používá, když chceme použít třídu SortedSet, pokud musíte ukládat jedinečné prvky a udržovat vzestupné pořadí.

-V SortedSet může uživatel ukládat pouze stejný typ prvků.

### Dictionary

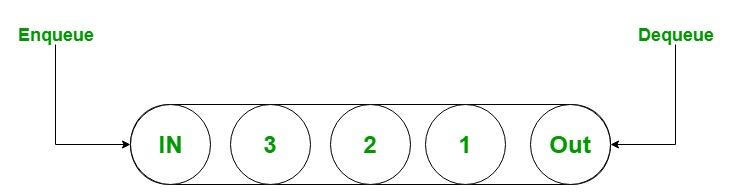
V C # je Slovník obecná kolekce, která se obvykle používá k ukládání párů klíč / hodnota. Práce se slovníkem je docela podobná nongeneric hashtable. Výhodou slovníku je, že jde o generický typ. Slovník je definován v System.Collection.Generic namespace. Jeho dynamický charakter znamená, že velikost slovníku roste podle potřeby.

Třída Slovník implementuje

* IDictionary <TKey, TValue> Interface
* IReadOnlyCollection <KeyValuePair <TKey, TValue >> Rozhraní
* IReadOnlyDictionary <TKey, TValue> Interface
* IDictionary Interface
* Ve slovníku nemůže být klíč nulový, ale hodnota může být.
* Ve slovníku musí být klíč jedinečný. Duplicitní klíče nejsou povoleny, pokud se pokusíte použít duplicitní klíč, kompilátor vyvolá výjimku.
* Ve slovníku můžete ukládat pouze stejné typy prvků.
* Kapacita slovníku je počet prvků, které může slovník obsahovat.

### Queue (fronta)

Queue se používá k reprezentaci kolekce FIFO (first-in, first out, FIFO). Používá se, když potřebujete first-in, first-out přístup k položkám. Je to non-generic typ kolekce, který je definován v oboru názvů System.Collections. Slouží k vytvoření dynamické kolekce, která roste podle potřeby vašeho programu. V Queue můžete ukládat prvky stejného typu a různých typů. Obecně je queue užitečná, když k těmto informacím přistupujete stejným způsobem, jakým byly uloženy ve sbírce, a ukládání dat je dočasným úložištěm.



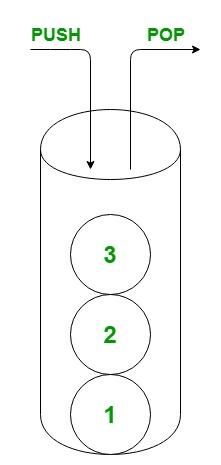
* Třída Queue implementuje rozhraní IEnumerable, ICollection a ICloneable.
* Když přidáte položku do seznamu, nazývá se enqueue.
* Když odstraníte položku, nazývá se to dequeue.
* Queue přijímá null jako platnou hodnotu pro typy odkazů.
* Jak jsou prvky přidávány do fronty, kapacita se automaticky zvyšuje podle potřeby přerozdělením interního pole.
* V Queue můžete ukládat duplicitní prvky.
* Queue fronty je počet prvků, které může queue obsahovat.

*4 konstruktory*

* **Queue ():** Tento konstruktor se používá k vytvoření instance třídy Queue, která je prázdná a má výchozí počáteční kapacitu a používá výchozí růstový faktor.
* **Queue (ICollection):** Tento konstruktor se používá k vytvoření instance třídy Fronta, která obsahuje prvky zkopírované ze zadané kolekce, má stejnou počáteční kapacitu jako počet zkopírovaných prvků a používá výchozí růstový faktor.
* **Queue (Int32):** Tento konstruktor se používá k vytvoření instance třídy Queue, která je prázdná a má stanovenou počáteční kapacitu, a používá výchozí růstový faktor.
* **Queue (Int32, Single):** Tento konstruktor se používá k vytvoření instance třídy Queue, která je prázdná a má stanovenou počáteční kapacitu, a používá zadaný růstový faktor.

### Stack

Stack(Zásobník) se používá k reprezentaci last-in, first out kolekce objektů. Používá se, když potřebujete přístup k položkám last-in, first out. Jedná se o generický a negenerický typ kolekce. Generický stack je definován v System.Collections.Generic namespace, zatímco ne-generický stack je definován v namespace System.Collections. Stack se používá k vytvoření dynamické kolekce, která roste podle potřeby vašeho programu. Do stack můžete ukládat prvky stejného typu nebo různých typů.



* Třída Stack implementuje rozhraní IEnumerable, ICollection a ICloneable.
* Když přidáte položku do seznamu, nazývá se tlačení prvku.
* Když ji odeberete, nazývá se praskání prvku.
* Kapacita zásobníku je počet prvků, které může zásobník obsahovat. Jak jsou prvky přidávány do zásobníku, kapacita se automaticky zvyšuje podle potřeby prostřednictvím realokace.
* V zásobníku můžete ukládat duplicitní prvky.
* Zásobník přijímá null jako platnou hodnotu pro referenční typy.

*tři konstruktery*

* **Stack ():** Tento konstruktor se používá k vytvoření instance třídy Stack, která je prázdná a má výchozí počáteční kapacitu.
* **Stack (ICollection):** Tento konstruktor se používá k vytvoření instance třídy Stack, která obsahuje prvky zkopírované ze zadané kolekce a má stejnou počáteční kapacitu jako počet zkopírovaných prvků.
* **Stack (Int32):** Tento konstruktor se používá k vytvoření instance třídy Stack, která je prázdná a má určenou počáteční kapacitu nebo výchozí počáteční kapacitu, podle toho, co je větší.

# **Neseřazené datové struktury**

### HashSet

V C # je HashSet neuspořádanou sbírkou jedinečných prvků.

* Obvykle se používá, když chceme zabránit tomu, aby do kolekce byly umístěny duplicitní prvky.   
  Výkon HashSetu je ve srovnání s listem mnohem lepší.
* Třída HashSet implementuje rozhraní ICollection, IEnumerable, IReadOnlyCollection, ISet, IEnumerable, IDeserializationCallback a ISerializable interface.
* V HashSet není pořadí prvku definováno. Prvky HashSetu nelze třídit.
* V HashSet musí být prvky jedinečné.
* V HashSetu nejsou duplicitní prvky povoleny.
* Poskytuje mnoho operací s matematickými množinami, jako je průnik, sjednocení a rozdíl.
* Kapacita zařízení HashSet je počet prvků, které může obsahovat.
* HashSet je dynamická kolekce, což znamená, že se velikost HashSet automaticky přidá po přidání nových prvků.
* V HashSet můžete ukládat pouze stejný typ prvků.

### ConcurrentBag

ConcurrentBag je jednou z tříd kolekce bezpečných pro vlákno

ConcurrentBag umožňuje ukládat objekty neuspořádaným způsobem. Na rozdíl od třídy ConcurrentDictionary umožňuje ukládat duplicitní objekty.

ConcurrentBag umožňuje ukládání více podprocesů. Je optimalizován pro scénáře, kde stejné vlákno působí jako výrobce i spotřebitel. To znamená, že stejné vlákno přidává a načítá data.

Například existují dvě vlákna Thread1 a Thread2. Vlákno1 přidalo čtyři objekty 1,2,3,4. Thread2 přidal tři objekty 5,6,7. Poté, co obě vlákna přidaly data, Thread1 začne načítat data. Když Thread1 přidal 1,2,3,4 objektů, získají tyto položky preference nad 5,6,7. Poté, co Thread1 načte všechny čtyři položky, Thread1 přejde k načtení Thread2 vložených dat 5,6,7.

### ConcurrentDictionary

ConcurrentDictionary je třída kolekce bezpečná pro vlákna k ukládání párů klíč / hodnota. ConcurrentDictionary lze použít s více vlákny současně.

Bez třídy ConcurrentDictionary, pokud musíme použít třídu Dictionary s více podprocesy, musíme použít zámky k zajištění bezpečnosti podprocesů, který je vždy náchylná k chybám. ConcurrentDictionary vám poskytuje snadnou možnost. Interně spravuje zamykání a poskytuje snadné rozhraní pro přidávání / aktualizaci položek.

### SortedDictionary

V C # se třída SortedDictionary <TKey, TValue> používá k reprezentaci kolekce párů klíč / hodnota. Tento pár je ve tříděné podobě a třídění se provádí na klíči. Ve třídě SortedDictionary jsou klíče neměnné, vždy jedinečné a nemohou být null. Pokud je typ hodnoty referenčního typu, můžete použít hodnotu null. Třída SortedDictionary poskytuje nejrychlejší operace vkládání a odebírání netříděných dat. Pár klíč / hodnota třídy SortedDictionary je získán pomocí struktury KeyValuePair.