**Datové struktury**

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 tedy 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;

pole[2 \* 2] = 21;

// Toto ale vyhodí chybu, protože jsme překročili celkový počet prvků

pole[12] = 3124;

Pokud již při definici víme, jaké prvky bude pole obsahovat, můžeme využít zápis definice i s určením prvků.

int[] pole = new int[5] { 20, 9, 195, 1, -569 };

// nebo ještě kratší zápis

int[] pole = { 20, 9, 195, 1, -569 };

Vícerozměrná pole

Vícerozměrné pole je takové pole, na jehož každém prvku je definované další pole. U vícerozměrných polí dále rozlišujeme, jestli je pole pravoúhlé, tj. má pevně definovaný počet sloupců a pole, u něhož počet sloupců neznáme a jednotlivé prvky skutečně definujeme jako nová pole.

int[,] matice = new int[2, 2];

matice[0, 0] = 1;

matice[1, 1] = 1;

Pole s neurčenými rozměry dimenzí se definují tak, že zdvojíme hranaté závorky při definici a jednotlivým prvkům musíme předávat opět pole stejného typu. Nyní máme příklad, kdy chceme uchovávat rodiny a k těmto rodinám přiřazovat jména jejich členů.

string[][] rodina = new string[2][];

rodina[0] = new string[3] {"Marek", "Petr", "Eva"};

rodina[1] = new string[2] {"Tomáš", "Zuzana" };

Console.Write("Člen rodiny: " + rodina[0][1]);

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.

*Operace:*

*Vytvoření pole*

**int** [] array\_of\_integers = **new** **int** [100];

*Výpis pole pomocí for cyklu*

**for** (i = 0; i < 100; i++ ) {

System.out.println("Na pozici [" + i + "] je " + pole[i]);

}

*Vlož/ uprav prvek*

Pokud budeme vkládat prvek na konec takového pole, je potřeba jej projít celé. V prvním případě je složitost O(n), neboť se nám může stát, že musíme projít celé pole.

Pokud víme kolik prvků máme momentálně obsazených, vložíme prvek na konec pole v čase O(1), tedy konstantním.

A pokud chceme upravit prvek na konkrétním indexu, zvládneme to přímo v O(1):

array[i] = nejaky\_prvek;

Vyhledat prvek složitost této operace O(n), protože musíme projít celé pole

**int** searched\_item = nejaky\_prvek;

**for** (**int** i = 0; i < n; i++) {

**if** (array[i] == searched\_item ) {

System.out.println("Prvek " + searched\_item + " nalezen na pozici ["+ i + "].");

**break**;

}

}

*Smaž prvek* pokud bychom měli pole kladných čísel, mohli bychom prvek smazat tak, že bychom jeho hodnotu nastavili na 0. To by trvalo v O(1). Pokud bychom chtěli pole i zmenšit, tato operace by trvala v O(n), protože se prvky za smazaným prvkem musí posunout o jednu pozici doleva a zaplnit tak "díru", která v poli vznikla:

**for**( **int** i = index\_smazaneho\_prvku; i < n - 1; i++) {

array[i] = array[i+1];

}

*Seřadit pole* Na seřazení prvků v poli můžeme použít algoritmy. Selection sort, haldové třídění atd.

### Rozdíl mezi listem a polem

Pole má jednu velkou výhodu i nevýhodu, je statické. Často však nevíme, jak bude pole dlouhé, nebo z něj chceme za běhu programu odebírat prvky nebo je do něj přidávat. Existuje na to takový malý trik.

Mějme pole délky n. Pokud do něj chceme vložit další prvek, tzn. na index n + 1, založíme nové pole o velikosti 2n a zkopírujeme prvky původního pole do tohoto nového. Takže přidání prvku většinu času stojí O(1) a jednou za čas O(n), když dojde kapacita pole, což nám dohromady dá O(1) (říká se tomu amortizovaná časová složitost) .Potom, až toto pole zase naplníme, zdvojnásobíme kapacitu atd. Takto dostaneme dynamické pole, které se může měnit. Není to však příliš elegantní pro běžné použití, proto v mnoha jazycích můžeme použít dynamické pole – List, který tuto práci dělá za nás.

**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.

// Creating a List of integers

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

// displaying the number of elements of List<T>

Console.WriteLine(firstlist.Count);

// adding elements in firstlist

        firstlist.Add(1);

        firstlist.Add(2);

        firstlist.Add(3);

        firstlist.Add(4);

// Printing the Capacity of firstlist

        Console.WriteLine("Capacity Is: " + firstlist.Capacity);

// Printing the Count of firstlist

Console.WriteLine("Count Is: " + firstlist.Count);

OUTPUT

Capacity Is: 4

Count Is: 4

// Checking whether 4 is present in List or not

Console.Write(firstlist.Contains(4));

**Output:**

True

// 3 will remove from the List and 4 will come at index 3

firstlist.RemoveAt(3);

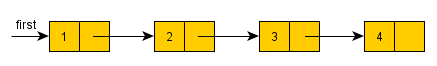
**Spojový seznam(LinkedList)**

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)

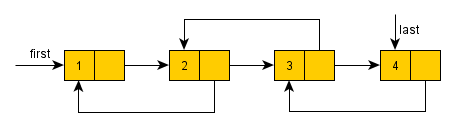
Jednosměrně zřetězený

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 zvláštího 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 s 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).

Složitost základních operací

Operace *vkládání* na začátek seznamu proběhne s [asymptotickou složitostí](https://www.algoritmy.net/article/102/Asymptoticka-slozitost) O(1), protože se pouze zalokuje nový prvek, který se vloží na počátek seznamu. Operace *mazání* a *čtení* prvku na indexu *i* proběhnou v čase O(i), protože spojový seznam neumožňuje náhodný přístup - k prvku je zapotřebí doiterovat.

LinkedList neobsahuje rovnou naše prvky jako tomu bylo u Listu, ale jsou v něm uloženy položky typu LinkedListNode. Jsou to uzly, které na sebe navzájem ukazují (odkazují se, chcete-li) a disponují vlastností Value. Právě v té je teprve uložen náš prvek, který uzel obaluje. Dodává tak našemu prvku ony vazby na prvky okolní. Ukažme si metody, které má LinkedList oproti klasickému Listu navíc:

* **AddAfter()** - Přidá nový prvek za daný prvek.
* **AddBefore()** - Přidá nový prvek před daný prvek.
* **AddFirst()** - Přidá nový prvek na začátek seznamu.
* **AddLast()** - Přidá nový prvek na konec seznamu.
* **First** - Vlastnost vracející první prvek.
* **Last** - Vlastnost vracející poslední prvek.
* **RemoveFirst()** - Odstraní první prvek.
* **RemoveLast()** - Odstraní poslední prvek.

LinkedList<**int**> seznam = **new** LinkedList<**int**>();

LinkedListNode<**int**> hlava = seznam.AddFirst(5);

seznam.AddAfter(hlava, 10);

Console.WriteLine(seznam.First.Value);

Console.WriteLine(seznam.First.Next.Value);

**Output:**

**5, 10**

// inicializace a naplnění spojového seznamu

LinkedList<**int**> seznam = **new** LinkedList<**int**>();

seznam.AddLast(1);

seznam.AddLast(2);

LinkedListNode<**int**> prostredni = seznam.AddLast(3);

seznam.AddLast(4);

seznam.AddLast(5);

// přidávání a mazání v prostředku seznamu

seznam.AddAfter(prostredni, 32);

seznam.AddAfter(prostredni, 31);

seznam.Remove(prostredni);

// výpis seznamu

**foreach** (**int** i **in** seznam)

Console.Write("{0}, ", i);

Console.ReadKey();

**Output: 1, 2, 31, 32, 4, 5**

Práce se spojovým seznamem je díky uzlům o něco komplikovanější než s Listem a většinou používáme spíše List. Po "spojáku" sáhneme hlavně v případě, kdy chceme často vkládat a mazat prvky doprostřed seznamu, což by u Listu bylo velmi pomalé.

// Creating a LinkedList of Strings

        LinkedList<String> myList = new LinkedList<String>();

        // Adding nodes in LinkedList

        myList.AddLast("Geeks");

        myList.AddLast("for");

        myList.AddLast("Data Structures");

        myList.AddLast("Noida");

        // To check if LinkedList is empty or not

        if (myList.Count > 0)

            Console.WriteLine("LinkedList is not empty");

        else

            Console.WriteLine("LinkedList is empty");

    }

 // Creating a LinkedList of Strings

        LinkedList<String> myList = new LinkedList<String>();

        // Adding nodes in LinkedList

        myList.AddLast("GeeksforGeeks");

        myList.AddLast("GFG");

        myList.AddLast("Data Structures");

        myList.AddLast("Noida");

        // ------- Count Property -------

        // To get the first node of the LinkedList

        if (myList.Count > 0)

            Console.WriteLine(myList.First.Value);

        else

            Console.WriteLine("LinkedList is empty");

        // ------- Last Property -------

        // To get the last node of the LinkedList

        if (myList.Count > 0)

            Console.WriteLine(myList.Last.Value);

        else

            Console.WriteLine("LinkedList is empty");

    }

 // Creating a LinkedList of Strings

        LinkedList<String> myList = new LinkedList<String>();

        // Adding nodes in LinkedList

        myList.AddLast("A");

        myList.AddLast("B");

        myList.AddLast("C");

        myList.AddLast("D");

        myList.AddLast("E");

        // To check if a value is in LinkedList

        Console.WriteLine(myList.Contains("B"));

**Output:**

**TRUE**

// Creating a LinkedList of Integers

        LinkedList<int> myList = new LinkedList<int>();

        // Adding nodes in LinkedList

        myList.AddLast(2);

        myList.AddLast(4);

        myList.AddLast(6);

        myList.AddLast(8);

        // To get the count of nodes in LinkedList

        // before removing all the nodes

        Console.WriteLine("Total nodes in myList are : " + myList.Count);

        // Displaying the nodes in LinkedList

        foreach(int i in myList)

        {

            Console.WriteLine(i);

        }

        // Removing the first node from the LinkedList

        myList.Remove(myList.First);

        // To get the count of nodes in LinkedList

        // after removing all the nodes

        Console.WriteLine("Total nodes in myList are : " + myList.Count);

        // Displaying the nodes in LinkedList

        foreach(int i in myList)

        {

            Console.WriteLine(i);

        }

**Output:**

Total nodes in myList are : 4

2

4

6

8

Total nodes in myList are : 3

4

6

8

# **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.

*Create a SortedSet*

*SortedSet<type\_of\_sortedset> sortedset\_name = new SortedSet<type\_of\_sortedset>();*

 // Creating SortedSet

        // Using SortedSet class

        SortedSet<int> my\_Set1 = new SortedSet<int>();

        // Add the elements in SortedSet

        // Using Add method

        my\_Set1.Add(101);

        my\_Set1.Add(1001);

        my\_Set1.Add(10001);

        my\_Set1.Add(100001);

        Console.WriteLine("Elements of my\_Set1:");

        // Accessing elements of SortedSet

        // Using foreach loop

        foreach(var val in my\_Set1)

        {

            Console.WriteLine(val);

        }

Elements of my\_Set1:

101

1001

10001

100001

* [**Remove(T)**](https://www.geeksforgeeks.org/c-remove-a-specified-item-from-sortedset/)**:**Tato metoda se používá k odebrání určité položky z SortedSet.
* **RemoveWhere (Predicate):** Tato metoda se používá k odstranění všech prvků, které odpovídají podmínkám definovaným zadaným predikátem z SortedSet.
* **Clear ():** Tato metoda se používá k odstranění všech prvků ze sady

 // Creating SortedSet

        // Using SortedSet class

        SortedSet<int> my\_Set = new SortedSet<int>();

        // Add the elements in SortedSet

        // Using Add method

        my\_Set.Add(101);

        my\_Set.Add(1001);

        my\_Set.Add(10001);

        my\_Set.Add(100001);

        // After using Remove method

        Console.WriteLine("Total number of elements "+

               "present in my\_Set:{0}", my\_Set.Count);

        // Remove element from SortedSet

        // Using Remove method

        my\_Set.Remove(1001);

        // Before using Remove method

        Console.WriteLine("Total number of elements "+

               "present in my\_Set:{0}", my\_Set.Count);   
Total number of elements present in my\_Set:4

Total number of elements present in my\_Set:3

V SortedSet můžete zkontrolovat, zda daný prvek je nebo není přítomen pomocí metody Contains

  // Creating SortedSet

        // Using SortedSet class

        SortedSet<int> my\_Set = new SortedSet<int>();

        // Add the elements in SortedSet

        // Using Add method

        my\_Set.Add(101);

        my\_Set.Add(1001);

        my\_Set.Add(10001);

        my\_Set.Add(100001);

        // Check the given element present

        // in the SortedSet or not

        // Using Contains method

        if (my\_Set.Contains(101) == true)

        {

            Console.WriteLine("Element is available..!");

        }

        else

        {

            Console.WriteLine("Element is not available..!");

        }

    }

**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.

*Create a Dictionary*

Dictionary dictionary\_name = new Dictionary();

 // Creating a dictionary

        // using Dictionary<TKey,TValue> class

        Dictionary<int, string> My\_dict1 =

                       new Dictionary<int, string>();

          // Adding key/value pairs

          // in the Dictionary

          // Using Add() method

          My\_dict1.Add(1123, "Welcome");

          My\_dict1.Add(1124, "to");

          My\_dict1.Add(1125, "GeeksforGeeks");

          foreach(KeyValuePair<int, string> ele1 in My\_dict1)

          {

              Console.WriteLine("{0} and {1}",

                        ele1.Key, ele1.Value);

          }

          Console.WriteLine();

          // Creating another dictionary

          // using Dictionary<TKey,TValue> class

      // adding key/value pairs without

          // using Add method

      Dictionary<string, string> My\_dict2 =

              new Dictionary<string, string>(){

                                  {"a.1", "Dog"},

                                  {"a.2", "Cat"},

                                {"a.3", "Pig"} };

          foreach(KeyValuePair<string, string> ele2 in My\_dict2)

          {

              Console.WriteLine("{0} and {1}", ele2.Key, ele2.Value);

          }

**Output:**

1123 and Welcome

1124 and to

1125 and GeeksforGeeks

a.1 and Dog

a.2 and Cat

a.3 and Pig

*Remove elements*

My\_dict.Remove(1123);

My\_dict.Clear();

**ContainsKey:** Tato metoda se používá ke kontrole, zda slovník <TKey, TValue> obsahuje zadaný klíč.

**ContainsValue:** Tato metoda se používá ke kontrole, zda slovník <TKey, TValue> obsahuje konkrétní hodnotu.

if (My\_dict.ContainsKey(1122)==true)

          {

              Console.WriteLine("Key is found...!!");

          }

          else

          {

               Console.WriteLine("Key is not found...!!");

          }

          // Using ContainsValue() method to check

          // the specified value is present or not

          if (My\_dict.ContainsValue("GeeksforGeeks")==true)

          {

              Console.WriteLine("Value is found...!!");

          }

          else

          {

               Console.WriteLine("Value is not found...!!");

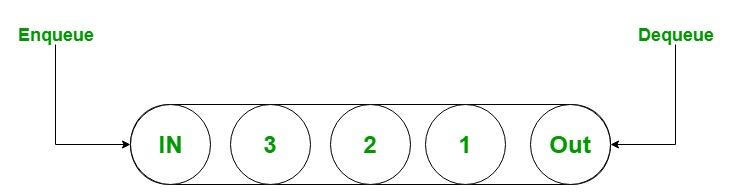
**Output:**

Key is not found...!!

Value is found...!!

# **Queue**

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.

  // Create a queue

        // Using Queue class

        Queue my\_queue = new Queue();

        // Adding elements in Queue

        // Using Enqueue() method

        my\_queue.Enqueue("GFG");

        my\_queue.Enqueue(1);

        my\_queue.Enqueue(100);

        my\_queue.Enqueue(null);

        my\_queue.Enqueue(2.4);

        my\_queue.Enqueue("Geeks123");

        // Accessing the elements

        // of my\_queue Queue

        // Using foreach loop

        foreach(var ele in my\_queue)

        {

            Console.WriteLine(ele);

        }

**Output:**

GFG

1

100

2.4

Geeks123

*Delete elements*

// Create a queue

        // Using Queue class

        Queue my\_queue = new Queue();

        // Adding elements in Queue

        // Using Enqueue() method

        my\_queue.Enqueue("GFG");

        my\_queue.Enqueue(1);

        my\_queue.Enqueue(100);

        my\_queue.Enqueue(2.4);

        my\_queue.Enqueue("Geeks123");

        Console.WriteLine("Total elements present in my\_queue: {0}",

                                                    my\_queue.Count);

        my\_queue.Dequeue();

        // After Dequeue method

        Console.WriteLine("Total elements present in my\_queue: {0}",

                                                    my\_queue.Count);

        // Remove all the elements from the queue

        my\_queue.Clear();

        // After Clear method

        Console.WriteLine("Total elements present in my\_queue: {0}",

                                                    my\_queue.Count);

    }

**Output:**

Total elements present in my\_queue: 5

Total elements present in my\_queue: 4

Total elements present in my\_queue: 0

**Peek:** Tato metoda vrací objekt na začátku fronty bez jeho odstranění.

**Dequeue:** Tato metoda vrací objekt na začátku fronty s úpravou, což znamená, že tato metoda odstraní nejvyšší prvek fronty.

*Příklad:*

// Create a queue

        // Using Queue class

        Queue my\_queue = new Queue();

        // Adding elements in Queue

        // Using Enqueue() method

        my\_queue.Enqueue("GFG");

        my\_queue.Enqueue("Geeks");

        my\_queue.Enqueue("GeeksforGeeks");

        my\_queue.Enqueue("geeks");

        my\_queue.Enqueue("Geeks123");

        Console.WriteLine("Total elements present in my\_queue: {0}",

                                                    my\_queue.Count);

        // Obtain the topmost element of my\_queue

        // Using Dequeue method

        Console.WriteLine("Topmost element of my\_queue"

                     + " is: {0}", my\_queue.Dequeue());

        Console.WriteLine("Total elements present in my\_queue: {0}",

                                                    my\_queue.Count);

        // Obtain the topmost element of my\_queue

        // Using Peek method

        Console.WriteLine("Topmost element of my\_queue is: {0}",

                                               my\_queue.Peek());

        Console.WriteLine("Total elements present in my\_queue: {0}",

                                                    my\_queue.Count);

    }

**Output:**

Total elements present in my\_queue: 5

Topmost element of my\_queue is: GFG

Total elements present in my\_queue: 4

Topmost element of my\_queue is: Geeks

Total elements present in my\_queue: 4

*Contains:*

my\_queue.Enqueue("GFG");

       my\_queue.Enqueue("Geeks");

        my\_queue.Enqueue("GeeksforGeeks");

        my\_queue.Enqueue("geeks");

        my\_queue.Enqueue("Geeks123");

        // Checking if the element is

        // present in the Queue or not

        if (my\_queue.Contains("GeeksforGeeks") == true) {

            Console.WriteLine("Element available...!!");

        }

        else {

            Console.WriteLine("Element not available...!!");

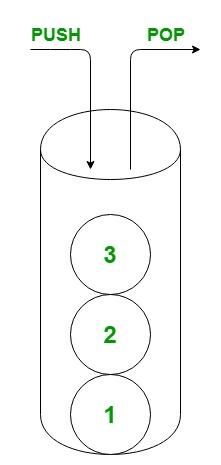
        }

**Output:**

Element available...!!

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.

// Create a stack

        // Using Stack class

        Stack my\_stack = new Stack();

        // Adding elements in the Stack

        // Using Push method

        my\_stack.Push("Geeks");

        my\_stack.Push("geeksforgeeks");

        my\_stack.Push('G');

        my\_stack.Push(null);

        my\_stack.Push(1234);

        my\_stack.Push(490.98);

        // Accessing the elements

        // of my\_stack Stack

        // Using foreach loop

        foreach(var elem in my\_stack)

        {

            Console.WriteLine(elem);

        }

**Output:**

490.98

1234

G

geeksforgeeks

Geeks

*Remove elements*

**Clear:** Tato metoda se používá k odstranění všech objektů ze zásobníku.

**Pop:** Tato metoda odstraní počáteční prvek zásobníku.

 // Using Push method

        my\_stack.Push("Geeks");

        my\_stack.Push("geeksforgeeks");

        my\_stack.Push("geeks23");

        my\_stack.Push("GeeksforGeeks");

        Console.WriteLine("Total elements present in"+

                    " my\_stack: {0}", my\_stack.Count);

        my\_stack.Pop();

        // After Pop method

        Console.WriteLine("Total elements present in "+

                      "my\_stack: {0}", my\_stack.Count);

        // Remove all the elements

        // from the stack

        my\_stack.Clear();

        // After Pop method

        Console.WriteLine("Total elements present in "+

                      "my\_stack: {0}", my\_stack.Count);

    }

**Output:**

Total elements present in my\_stack: 4

Total elements present in my\_stack: 3

Total elements present in my\_stack: 0

**Topmost element**

        // Adding elements in the Stack

        // Using Push method

        my\_stack.Push("Geeks");

        my\_stack.Push("geeksforgeeks");

        my\_stack.Push("geeks23");

        my\_stack.Push("GeeksforGeeks");

        Console.WriteLine("Total elements present in"+

                     " my\_stack: {0}",my\_stack.Count);

        // Obtain the topmost element

        // of my\_stack Using Pop method

        Console.WriteLine("Topmost element of my\_stack"

                          + " is: {0}",my\_stack.Pop());

        Console.WriteLine("Total elements present in"+

                    " my\_stack: {0}", my\_stack.Count);

        // Obtain the topmost element

        // of my\_stack Using Peek method

        Console.WriteLine("Topmost element of my\_stack "+

                              "is: {0}",my\_stack.Peek());

        Console.WriteLine("Total elements present "+

                 "in my\_stack: {0}",my\_stack.Count);

**Output:**

Total elements present in my\_stack: 4

Topmost element of my\_stack is: GeeksforGeeks

Total elements present in my\_stack: 3

Topmost element of my\_stack is: geeks23

Total elements present in my\_stack: 3

// Adding elements in the Stack

        // Using Push method

        my\_stack.Push("Geeks");

        my\_stack.Push("geeksforgeeks");

        my\_stack.Push("geeks23");

        my\_stack.Push("GeeksforGeeks");

        // Checking if the element is

        // present in the Stack or not

        if (my\_stack.Contains("GeeksforGeeks") == true)

        {

            Console.WriteLine("Element is found...!!");

        }

        else

        {

            Console.WriteLine("Element is not found...!!");

        }

**Output:**

Element is found...!!

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.

using System.Collections.Generic;

*Create a HashSet*

HashSet<Type\_of\_hashset> Hashset\_name = new HashSet<Type\_of\_hashset>();

  // Creating HashSet

        // Using HashSet class

        HashSet<string> myhash1 = new HashSet<string>();

        // Add the elements in HashSet

        // Using Add method

        myhash1.Add("C");

        myhash1.Add("C++");

        myhash1.Add("C#");

        myhash1.Add("Java");

        myhash1.Add("Ruby");

        Console.WriteLine("Elements of myhash1:");

        // Accessing elements of HashSet

        // Using foreach loop

        foreach(var val in myhash1)

        {

            Console.WriteLine(val);

        }

Elements of myhash1:

C

C++

C#

Java

Ruby

*ODSTRANIT elementy*

Remove elements methods:

* [**Remove(T)**](https://www.geeksforgeeks.org/c-remove-the-specified-element-from-a-hashset/)**:**Tato metoda se používá k odebrání určeného prvku z objektu HashSet.
* [**RemoveWhere(Predicate)**](https://www.geeksforgeeks.org/c-remove-elements-from-a-hashset-with-conditions-defined-by-the-predicate/)**:** Tato metoda se používá k odstranění všech prvků, které odpovídají podmínkám definovaným predikátem z kolekce HashSet.
* [**Clear**](https://www.geeksforgeeks.org/c-remove-all-elements-from-a-hashset/)**:** Tato metoda se používá k odebrání všech prvků z objektu HashSet.

**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.

*creating ConcurrentBag instance*.

ConcurrentBag<int> bag = new ConcurrentBag<int>();

Můžeme také inicializovat ConcurrentBag s existující kolekcí. Můžeme předat libovolnou kolekci v parametru a zkopíruje všechny objekty kolekce do svého vlastního seznamu. Níže je uveden příklad.

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

ints.Add(1);

ints.Add(2);

ints.Add(3);

ints.Add(4);

ConcurrentBag<int> bag = new ConcurrentBag<int>(ints);

int count = bag.Count; //returns 4

*Přidat nové položky*

Chcete-li přidat nové položky, existuje metoda Add. Níže je uveden příklad.

ConcurrentBag<int> bag = new ConcurrentBag<int>();

bag.Add(1);

bag.Add(2);

*Přidat více položek*

ConcurrentBag neposkytuje žádnou metodu AddRange, pro každou položku musíme ručně volat metodu Add.

int[] arr = { 1, 2, 3 };

ConcurrentBag<int> bag = new ConcurrentBag<int>();

foreach(var item in arr)

{

    bag.Add(item);

}

*Načíst položky*

Dvě metody:

**TryPeek**

**TryTake**

Metoda **TryPeek** vrací jednu položku z batohu, ale neodstraní ji z batohu. Vrací true, pokud úspěšně načte položku, jinak vrací false.

ConcurrentBag<int> bag = new ConcurrentBag<int>();

bag.Add(1);

bag.Add(2);

int item;

bool isSucess = bag.TryPeek(out item); //isSuccess=True, item = 2

isSuccess = bag.TryPeek(out item); //isSuccess=True, item = 2

isSuccess = bag.TryPeek(out item); //isSuccess=True, item = 2

Metoda **TryTake** vrátí jednu položku z batohu a odstraní ji z batohu. Vrací hodnotu true, pokud úspěšně načte položku jinak false.

ConcurrentBag<int> bag = new ConcurrentBag<int>();

bag.Add(1);

bag.Add(2);

int item;

bool isSuccess = bag.TryTake(out item); //isSuccess=True, item = 2

isSuccess = bag.TryTake(out item); //isSuccess=True, item = 1

# **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, která je vždy náchylná k chybám. ConcurrentDictionary vám poskytuje snadnou možnost. Interně spravuje zamykání a poskytuje snadné rozhraní pro přidávání / aktualizaci položek.

## Add items

ConcurrentDictionary<string, string> dict = new ConcurrentDictionary<string, string>();

    bool firstItem = dict.TryAdd("1", "First");  //returns true

    bool secondItem = dict.TryAdd("2", "Second");  //returns  true

    bool thirdItem = dict.TryAdd("1", "Third"); //returns false;

## Retrieve Single Item using Key

Musíme poskytnout klíč v metodě TryGetValue. Parametr vrací hodnotu klíče. TryGetValue vrací true, pokud existuje klíč, nebo vrací false, pokud klíč neexistuje ve slovníku.

ConcurrentDictionary<string, string> dict = new ConcurrentDictionary<string, string>();

    bool firstItem = dict.TryAdd("1", "First");

    bool secondItem = dict.TryAdd("2", "Second");

    string itemValue1;

    string itemValue2;

bool isItemExists1 = dict.TryGetValue("1", out itemValue1);  //returns true

    Console.WriteLine(itemValue1); //Print "First"

bool isItemExists2 = dict.TryGetValue("3", out itemValue2);  //returns false

    Console.WriteLine(itemValue2); //Print blank

## Retrieve all items

Protože ConcurrentDictionary implementuje rozhraní IEnumerable, můžeme použít loop foreach k výčtu všech položek ve slovníku. Nevrací všechny položky v pořadí, jak jsme je přidali do slovníku.

ConcurrentDictionary<string, string> dictionary = new ConcurrentDictionary<string, string>();

    dictionary.TryAdd("1", "First");

    dictionary.TryAdd("2", "Second");

    dictionary.TryAdd("3", "Third");

    dictionary.TryAdd("4", "Fourth");

    foreach(var item in dictionary)

    {

       Console.WriteLine(item.Key + "-" + item.Value);

    }

    //Returns

    // 4- Fourth

    // 3 - Third

    // 2 - Second

    // 1 – First

## Update item

ConcurrentDictionary poskytuje metody TryUpdate k aktualizaci libovolného klíče. TryUpdate potrebuje tři parametry. Tři parametry jsou:

1. Key - key to modify
2. New Value - new value of the key
3. Old Value - old value of the key

string newValue;

    bool returnTrue = dictionary.TryUpdate("1", "New Value", "First"); //Returns true

    dictionary.TryGetValue("1", out newValue);

    Console.WriteLine(newValue); //Print "New Value"

    bool returnsFalse = dictionary.TryUpdate("2", "New Value 2", "No Value"); //Returns false

    dictionary.TryGetValue("2", out newValue);  //Returns "Second" Old value

    Console.WriteLine(newValue);    //Print "Second"

## Remove item

string removedItem;

bool result = dictionary.TryRemove("2", out removedItem); //Returns true

Console.WriteLine(removedItem); //Print "Second"

## Remove all items

dictionary.Clear();

    foreach (var item in dictionary) //enumerate 0 times

    {

        Console.WriteLine(item.Key);

    }

## Count all Items

 int totalItems = dictionary.Count(); // Returns 4

## Check Key exists

    dictionary.TryAdd("1", "First");

    dictionary.TryAdd("2", "Second");

    dictionary.TryAdd("3", "Third");

    dictionary.TryAdd("4", "Fourth");

    bool firstItemFound = dictionary.ContainsKey("1"); //Returns true

    bool fifthItemFound = dictionary.ContainsKey("5"); //Returns false

# **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.

 // Create a new SortedDictionary

        // of strings, with int keys.

        SortedDictionary<string, string> myDr =

          new SortedDictionary<string, string>();

        // Adding key/value pairs in myDr

        myDr.Add("One", "C");

        myDr.Add("Two", "C++");

        myDr.Add("Three", "C#");

        // Display the key/value pairs

        foreach(KeyValuePair<string, string> pair in myDr)

        {

            Console.WriteLine("Key: {0} and Value: {1}",

                                  pair.Key, pair.Value);

        }

**Output:**

Key: One and Value: C

Key: Three and Value: C#

Key: Two and Value: C++

 SortedDictionary<int, string> myDr =

        new SortedDictionary<int, string>();

        // Adding key/value pairs in myDr

        myDr.Add(1, "Dog");

        myDr.Add(2, "Cat");

        myDr.Add(3, "Birds");

        myDr.Add(4, "Rabbits");

        myDr.Add(5, "Fish");

        myDr.Add(6, "Hamster");

        myDr.Add(7, "Turtle");

        // Display the total number of

        // key/value pairs present in myDr

        Console.WriteLine("Total number of pairs "+

              "present in myDr : {0}", myDr.Count);

    }

**Output:**

Total number of pairs present in myDr : 7