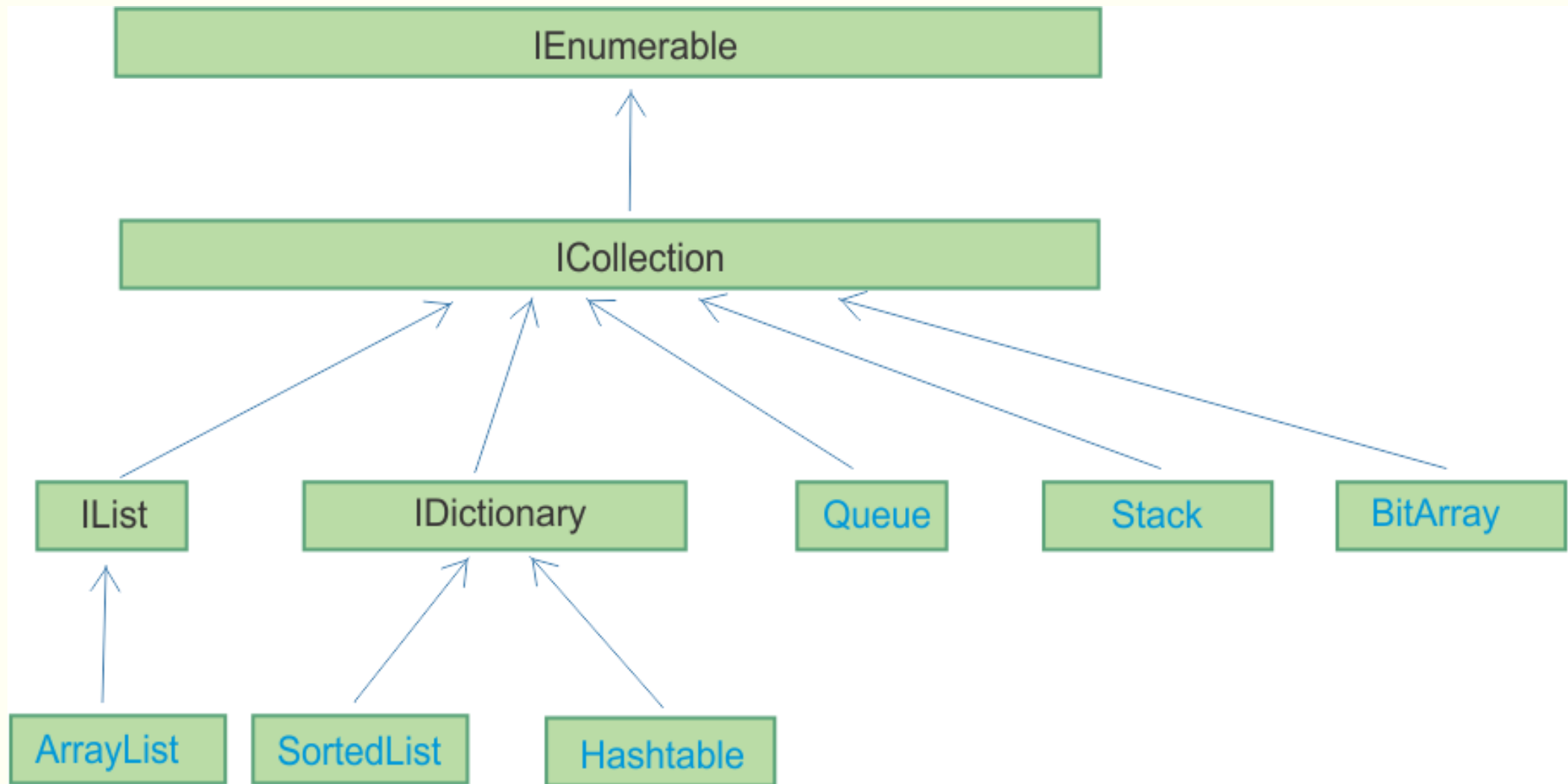




СТАНДАРТНІ СТРУКТУРИ ДАНИХ НА ПЛАТФОРМІ .NET

Питання 7.2.

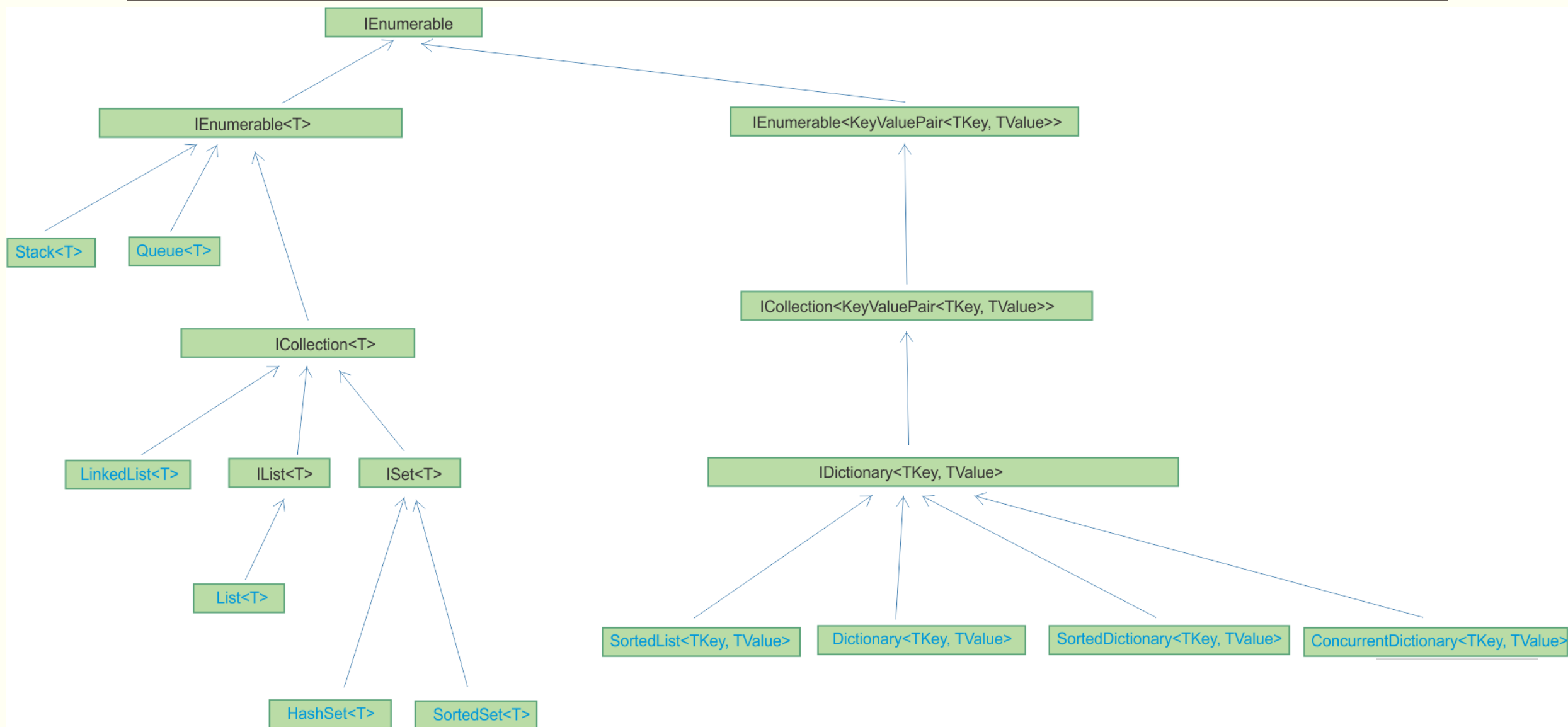
Неузагальнені колекції



Неузагальнені колекції

- **ВАЖЛИВО!** Уникайте використання колекцій з простору імен `System.Collections`!
 - Узагальнені та конкурентні версії колекцій рекомендуються для використання через їх вищу безпечність типів та інші покращення.
- Типи-замінники:
 - `BitArray` -> `List<bool>`, `IEnumerable<bool>`
 - `Queue` -> `Queue<T>`
 - `Stack` -> `Stack<T>`
 - `ArrayList` -> `List<T>`
 - `SortedList` -> `SortedSet<T>`
 - `Hashtable` -> `Dictionary<TKey, TValue>` (не 100%-ва заміна)

Узагальнені колекції



Бітові колекції. Тип BitArray

- Даний тип колекції може використовуватися для зберігання дуже великих серій бітів, якими можна маніпулювати як незалежно, так і колективно, як цілою групою.
 - Колекція BitArray реалізує властивості і методи інтерфейсу ICollection.
- Клас BitArray незвичайний тим, що не надає конструктор без параметрів.
 - Колекція BitArrays повинна бути повністю ініціалізована при створенні екземпляра з використанням одного з 6 конструкторів.
 - Кожен заповнює вміст колекції по-своєму. Найпростіший конструктор створює BitArray, що містить кілька логічних значень, які є хибними.
 - Необхідна кількість бітів передається як цілочисельний аргумент.
 - `BitArray flags = new BitArray (16);`
 - `BitArray flags = new BitArray (16, true);`
 - `bool[] bits = new bool[] { true, false, false, true };`
`BitArray flags = new BitArray(bits);`
 - `byte[] bytes = new byte[] { 1, 170 };`
`BitArray flags = new BitArray(bytes);`
 - `int[] values = new int[] { 1, 32767 };`
`BitArray flags = new BitArray(values);`

Бітові колекції. Тип BitArray

- Зчитування та запис окремих бітів:
 - ```
BitArray flags = new BitArray(16, false);
Console.WriteLine(flags[0]); // “False”
Console.WriteLine(flags.Get(1)); // "False”
flags[0] = true;
flags.Set(1, true);
```
- Запис кількох бітів:
  - ```
flags.SetAll (true);
```
- Оскільки клас `BitArray` реалізує `ICollection`, він включає властивість `Count`, яке повертає кількість бітів в колекції.
 - На додаток, `BitArray` надає властивість `Length`, яку можна використовувати як для запиту кількості елементів в колекції, так і для зміни довжини.
 - Якщо нова довжина менше існуючого числа елементів, колекція усікається, і всі записи з індексами, рівними або перевищуючими нову довжину, видаляються назавжди.
 - Якщо нова довжина більше поточної кількості елементів, то нові елементи додаються в `BitArray`.
 - Кожен додатковий біт спочатку `false`.

Бітові колекції. Тип BitArray

- Побітові операції:

- **Not** – інвертує біти в колекції:

- `BitArray flags = new BitArray(new bool[] { true, false, false, true });`
`flags.Not();`
`foreach (bool flag in flags) { Console.Write("{0}\t", flag); }`

- **And** – побітове І, потребує дві бітових колекції однакового розміру. Вміст колекції, яка використовується в якості параметра, не змінюється:

- `BitArray flags = new BitArray(new bool[] { true, false, false, true });`
`BitArray andFlags = new BitArray(new bool[] { false, true, false, true });`
`flags.And(andFlags);`
`foreach (bool flag in flags) { Console.Write("{0}\t", flag); }`

- **Or** – побітове АБО:

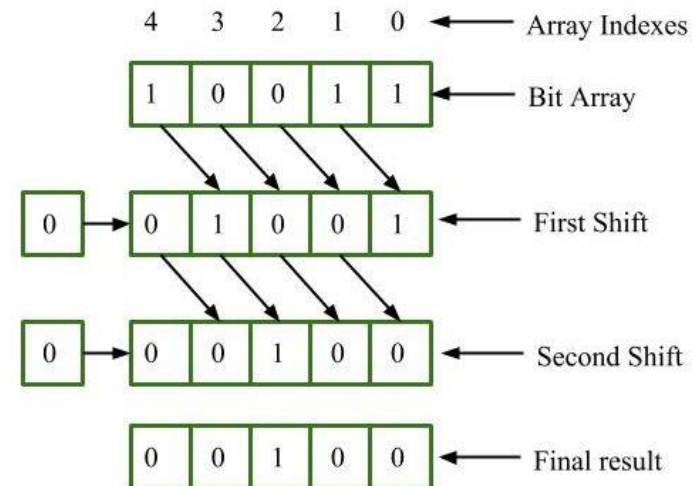
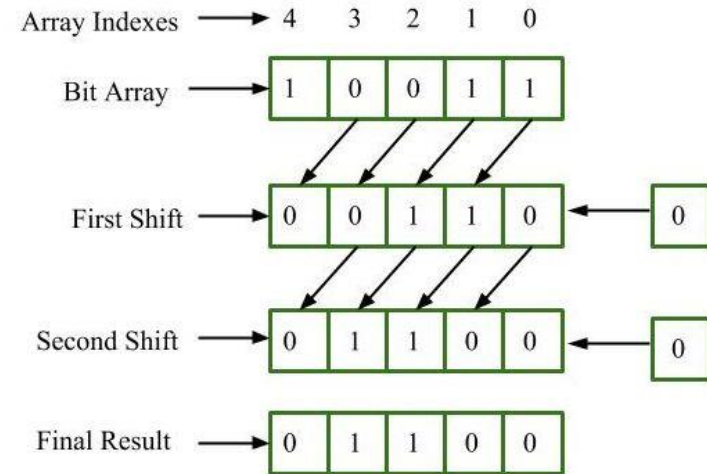
- `BitArray orFlags = new BitArray(new bool[] { false, true, false, true });`
`flags.Or(orFlags);`
`foreach (bool flag in flags) { Console.Write("{0}\t", flag); }`

- **Xor** – побітове виключне АБО:

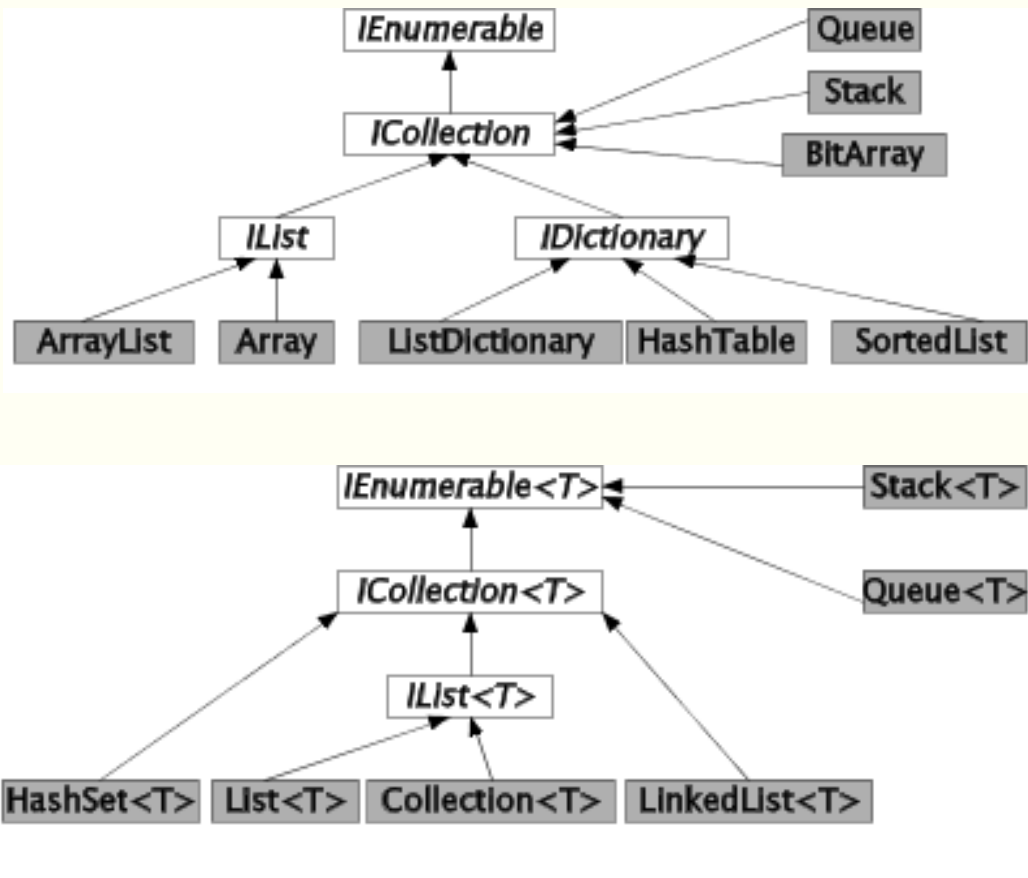
- `BitArray flags = new BitArray(new bool[] { true, false, false, true });`
`BitArray xorFlags = new BitArray(new bool[] { false, true, false, true });`
`flags.Xor(xorFlags);`
`foreach (bool flag in flags) { Console.Write("{0}\t", flag); }`

Бітові колекції. Тип BitArray

- Побітові операції:
 - LeftShift(Int32)** – побітовий зсув вліво:
 - `BitArr.LeftShift(2);`
 - RightShift(Int32)** – побітовий зсув вправо:
 - `BitArr.RightShift(2);`



Списки

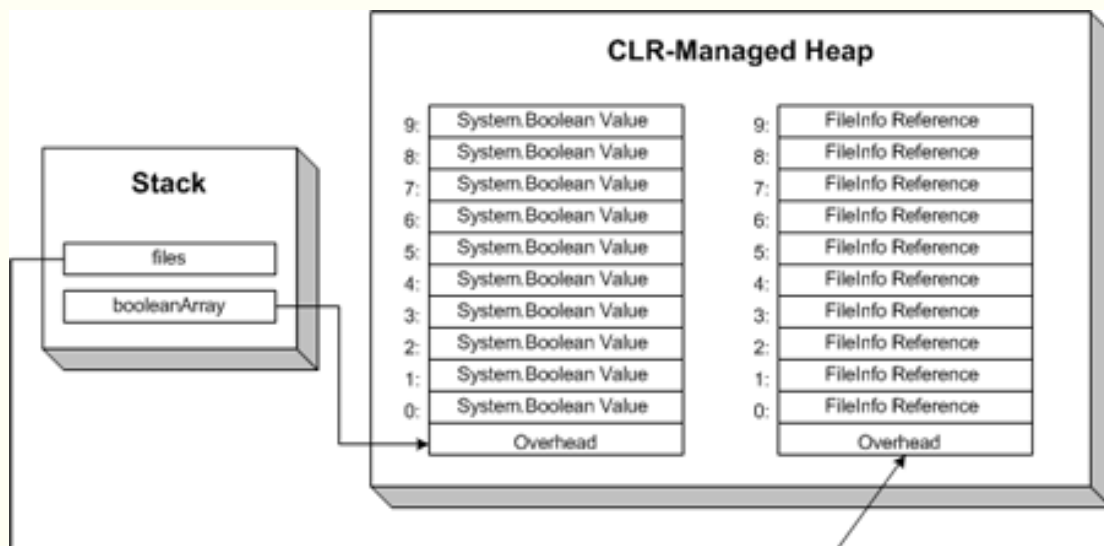


- **ArrayList** – клас для додавання, видалення елементів
- **SortedList** – клас, що зберігає набори пар "ключ-значення", відсортованих за ключем
- **List<T>** - клас для додавання, видалення елементів
- **LinkedList<T>** - двозв'язний список, у якому кожний елемент зберігає 2 посилання: на наступний та попередній елементи.
- **SortedList<TKey, TValue>** - клас, що зберігає набори пар "ключ-значення", відсортованих за ключем.

Масиви

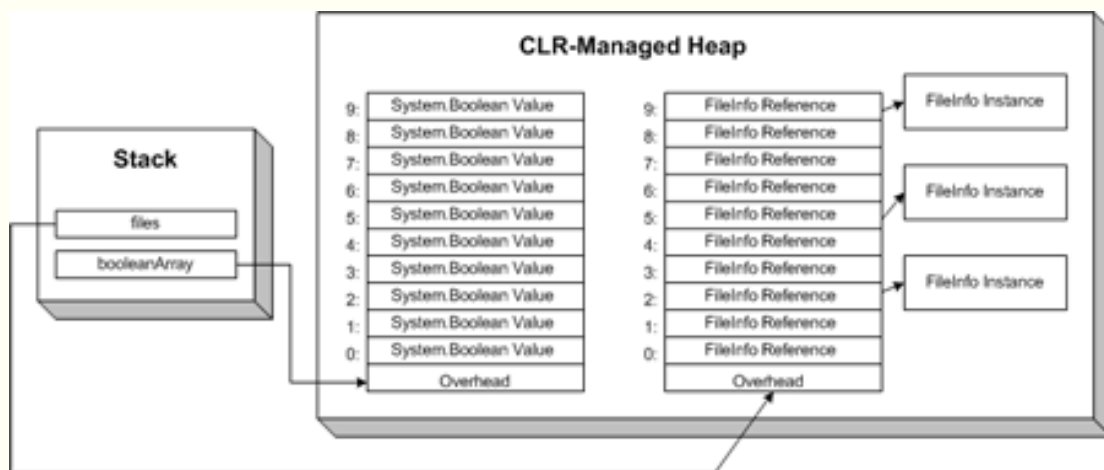
- Масив - одна з найпростіших і найбільш широко застосовуваних у комп'ютерних програмах структур даних.
- У будь-якій мові програмування масиви мають декілька спільних властивостей:
 - Вміст масиву зберігається в неперервній області пам'яті.
 - Всі елементи масиву мають однаковий тип; тому масиви називають однорідними (homogeneous) структурами даних.
 - Існує прямий доступ до елементів масиву.
 - В разі інших структур даних це необов'язково так.
 - Наприклад, у структурі даних скіп-список (SkipList) для доступу до конкретного елементу ви повинні попередньо здійснити пошук серед інших елементів скіп-списку, поки не знайдете необхідний елемент.
 - У випадку з масивами, якщо ви хочете отримати доступ до i-го елементу масиву, ви можете зробити це однією операцією: `arrayName[i]`.
- Типові операції для масивів:
 - Виділення елемента (Allocation)
 - Доступ до елемента (Accessing)
 - Зміна розмірів масиву (Redimensioning)

Масиви



■ Розглянемо приклад:

- `bool [] booleanArray;`
`FileInfo [] files;`
`booleanArray = new bool[10];`
`files = new FileInfo[10];`
- Тут `booleanArray` – масив елементів значимого типу `System.Boolean`, а `files` – це масив елементів посилального типу `System.IO.FileInfo`.
- На рисунках показано стан керованої кучі CLR після виконання цього коду.

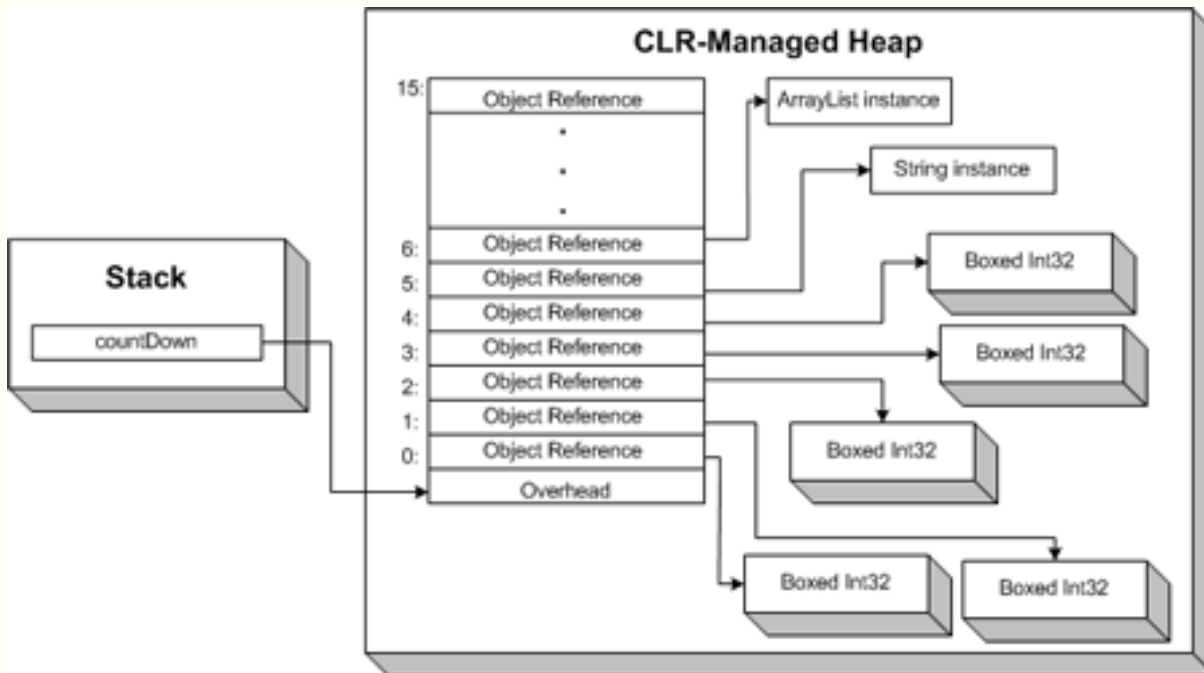


- Важливо пам'ятати, що 10 елементів масива `files` є посиланнями на екземпляри типу `FileInfo`.
- На нижньому рисунку показано вміст пам'яті, якщо ми присвоїмо деяким елементам масива `files` екземпляри типу `FileInfo`.

Масиви

- Масиви є чудовою структурою даних, якщо ми хочемо зберігати колекцію однорідних типів, до яких бажано мати прямий доступ.
 - Пошук в невідсортованому масиві займає лінійний від розміру масиву час.
 - Це прийнятно, якщо ми працюємо з маленькими масивами, або коли нам потрібно лише зрідка проводити пошук всередині масиву.
 - Якщо додаток зберігає великі масиви, в яких часто проводиться пошук, існують структури даних набагато краще пристосовані до таких ситуацій. Насправді, клас `Array` містить статичний метод `BinarySearch()`.
- `ArrayList` – неоднорідний масив змінного розміру.
 - звичайні масиви накладають певні обмеження на програміста, оскільки кожен масив може зберігати дані тільки одного типу (однорідність), а перед використанням масиву ви повинні виділити (`allocate`) певну кількість елементів.
 - Проте часто розробникам хочеться просту колекцію об'єктів потенційно різного типу, з якими можна було б просто працювати, не турбуючись про питання, пов'язані з виділенням елементів (`allocation`).
 - Базова бібліотека класів `.NET Framework` містить таку структуру даних - `System.Collections.ArrayList`.

Клас ArrayList



- ArrayList містить неперервну область посилань на екземпляри типу object.
 - Оскільки ArrayList зберігає масив об'єктів, ви повинні звести зчитане з ArrayList значення до того типу даних, який зберігався в даному елементі ArrayList-а.
 - Якщо ви спробуєте послатися на елемент ArrayList-а, номер якого перевищує розмір ArrayList-а, буде згенеровано виняток `System.ArgumentOutOfRangeException`.
- Хоча ArrayList забезпечує додаткову гнучкість в порівнянні зі звичайним масивом, досягається вона за рахунок швидкості роботи, особливо якщо ви зберігаєте в ArrayList-і значимі типи.
 - Внутрішній масив ArrayList-а є масивом посилань на екземпляри типу object.
 - Тому, навіть якщо ваш ArrayList не зберігає нічого, крім значимих типів, кожен елемент ArrayList-а є посиланням на упакований значимий тип (boxed value type).
 - Процеси упаковки і розпаковування, поряд з додатковими накладними витратами при зберіганні значимих типів в ArrayList-і, можуть негативно позначитися на продуктивності додатку при використанні великих ArrayList-ів з великою кількістю операцій читання / запису.
 - Як показує рисунок, використання пам'яті для посилальних типів виглядає однаково як у разі використання звичайних масивів, так і ArrayList-ів.

Клас ArrayList

- Автоматична зміна розміру ArrayList-а не повинна викликати погіршення продуктивності в порівнянні з масивом.
 - Якщо точний розмір невідомий, навіть у разі використання масиву, вам знадобитися збільшувати його розмір, якщо кількість елементів, що вставляються, перевищує розмір масиву.
- Класичне завдання computer science – визначення того, скільки місця необхідно виділяти при закінченні пам'яті в деякому буфері (області пам'яті).
 - (Рішення 1) додатково виділити один елемент з пам'яті буфера. Очевидно, що даний підхід є найбільш економним, однак він може бути занадто дорогим, оскільки за кожною вставкою нового елемента після заповнення масиву слід зміна його розміру.
 - (Рішення 2) Збільшення розміру масиву в 100 разів при закінченні наявного вільного місця для вставки нових елементів. Ясно, що такий підхід суттєво зменшує число змін розміру масиву, проте якщо небагато елементів будуть додані в масив, величезна кількість вільної пам'яті буде витрачена даремно.
 - (Компроміс) подвоєння поточного розміру масиву, коли закінчується вільне місце для нових елементів. Це і робить клас ArrayList, причому автоматично.
 - Асимптотичний час виконання операції ArrayList-а такий же, як і в звичайного масиву.

```

class Program
{
    private static ArrayList jobs = new ArrayList();
    private static int nextJobPos = 0;
    public static void AddJob(string jobName)
    {
        jobs.Add(jobName);
    }
    public static string GetNextJob()
    {
        if (nextJobPos > jobs.Count - 1)
            return "NO JOBS IN BUFFER";
        else
        {
            string jobName = (string)jobs[nextJobPos];
            nextJobPos++;
            return jobName;
        }
    }

    static void Main(string[] args)
    {
        AddJob("1");
        AddJob("2");
        Console.WriteLine(GetNextJob());
        AddJob("3");
        Console.WriteLine(GetNextJob());
        Console.WriteLine(GetNextJob());
        Console.WriteLine(GetNextJob());
        Console.WriteLine(GetNextJob());
        AddJob("4");
        AddJob("5");
        Console.WriteLine(GetNextJob());
    }
}

```

Недоліки ArrayList

- Нехай потрібно створити комп'ютерну службу, яка буде обробляти запити в порядку їх надходження.
 - Оскільки запити можуть з'являтися швидше, ніж служба їх оброблятиме, стане в нагоді буфер для запитів.
- Можливе рішення – застосувати ArrayList та цілочисельну змінну nextJobPos, у якій буде зберігатись індекс наступної задачі, що має відправлятися на обробку.
 - З приходом нового завдання будемо використовувати метод Add().
 - Такий підхід дуже простий, проте жахливо неефективний.

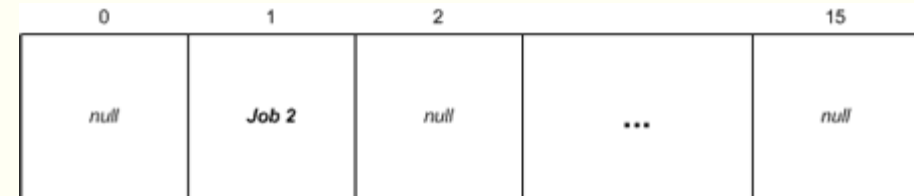
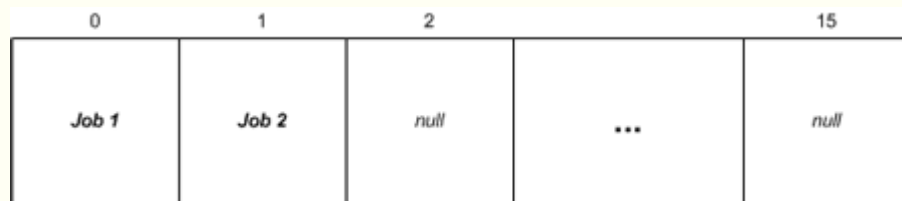
```

Консоль отладки Microsoft Visual Studio
1
2
3
NO JOBS IN BUFFER
NO JOBS IN BUFFER
4

```

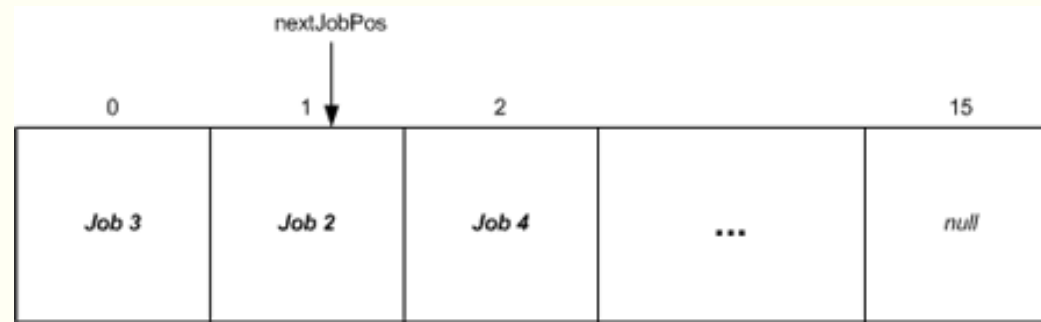
Недоліки ArrayList

- ArrayList буде необмежено зростати з кожним додаванням нового завдання в буфер, навіть у тому випадку, коли запити обробляються негайно після переміщення їх в буфер.
 - Наприклад, кожну секунду в буфер додається завдання і з буфера видаляється завдання: викликається метод `AddJob()`, який, в свою чергу, викликає метод `Add()` ArrayList-a.
 - При неперервному виклику методу `Add()`, розмір внутрішнього масиву ArrayList-a постійно подвоюється в міру необхідності. Після 5 хвилин (300 секунд) розмір внутрішнього масиву ArrayList-a досягне 512 елементів, незважаючи на те, що в буфері ніколи не знаходилося більше 1 завдання одночасно!
- Причина: пам'ять буфера, що використовувалась для старих запитів, не використовується повторно.
 - Розглянемо процес диспетчеризації задач.
 - Зауважте, що ArrayList на цей момент містить 16 елементів тому, що при створенні ArrayList-a за умовчанням створюється внутрішній масив з 16 елементів типу `object`.
 - Після цього викликається метод `GetNextJob ()`, який видаляє перше завдання з буфера.

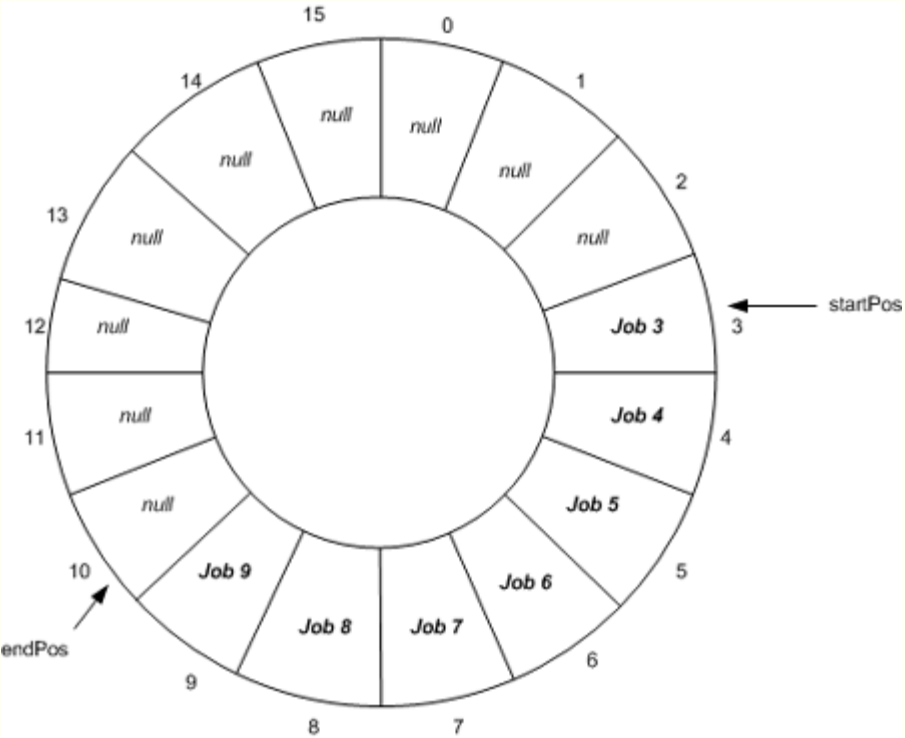


Недоліки ArrayList

- Коли виконується `AddJob("3")`, потрібно додати чергову задачу в буфер.
 - Перший елемент `ArrayList`-а (з індексом 0) можна використовувати повторно.
 - Однак, ми змушені відкинути такий підхід: якби ми помістили третє завдання по нульовому індексу, а четверте - в елемент з індексом 2, ми б отримали неприємну ситуацію:



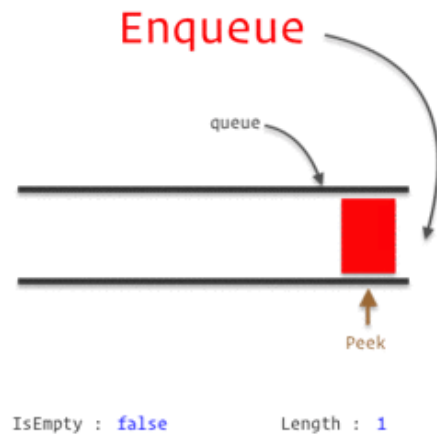
- Якщо викличемо `GetNextJob()`, з буфера буде видалена друга задача, значення `nextJobPos` буде збільшено на 1.
- Таким чином, при повторному виклику `GetNextJob()` з буфера буде видалена 4-те завдання, яке в результаті буде оброблене раніше за третє, що порушує порядок FCFS.
- Суть проблеми: `ArrayList` є списком задач у лінійному порядку. Це значить, що завжди потрібно додавати нові завдання після старих, щоб гарантувати коректний порядок обробки запитів.
- Кожного разу, коли ми натикаємось на кінець `ArrayList`-а, його розмір подвоюється навіть тоді, коли існують порожні елементи, які утворились через виклики `GetNextJob()`.



- Щоб виправити це, нам доведеться зробити ArrayList циклічним (circular).
 - Циклічний масив не має визначеного початку або кінця, точніше, кінець і початок масиву зберігаються в певних змінних.
 - При роботі з циклічним масивом, метод AddJob() поміщає нове завдання в елемент з індексом endPos, а потім "інкрементує" endPos.
 - Метод GetNextJob () забирає завдання з елемента з індексом startPos, привласнює елементу з індексом startPos значення null, а потім "інкрементує" startPos.
 - «Інкрементує» означає більш складну операцію, ніж просте додавання 1 до цілочисельного значення змінної.
 - Для випадку, коли endPos = 15, додавання 1 дасть значення 16.
 - При наступному виклику AddJob() буде намагання отримати доступ до елемента з номером 16, що призведе до винятку IndexOutOfRangeException.
 - Замість цього endPos має стати нулем:
 - ```
int increment(int variable) { return (variable + 1) % theArray.Length; }
```

# Черги

---



- Цей підхід чудово працює, якщо буфер ніколи не буде містити понад 16 елементів.
  - Інакше доведеться відповідним чином змінити розмір циклічного масиву, наприклад, подвоївши розмір масива.
- Вище описана функціональність надається стандартною структурою даних – Чергою.
  - Черга реалізується за допомогою класів `Queue` з простору імен `System.Collections` та `Queue<T>` з простору імен `System.Collections.Generic`.

# Черги

---

- Члени класу `Queue<T>`:
  - `Count` – властивість, яка повертає кількість елементів у черзі.
  - `Enqueue()` – метод, що додає елемент у кінець черги.
  - `Dequeue()` – метод, що зчитує та видаляє елемент з голови черги. Якщо на момент виклику метода елементів у черзі більше немає, генерується виняток `InvalidOperationException`.
  - `Peek()` – метод, що зчитує елемент з голови черги, проте не видаляє його.
  - `TrimExcess()` – метод, який змінює ємність черги. Метод `Dequeue()` видаляє елемент з черги, проте не змінює її ємності. `TrimExcess()` дозволяє усунути порожні елементи на початку черги.
- За кулісами, клас `Queue` використовує внутрішній циклічний масив елементів типу `object` та 2 змінні, що слугують маркерами його початку та кінця: `head` (голова) і `tail` (хвост).
  - За умовчанням початковий розмір черги – 32 елементи, хоч цю величину можна налаштувати при виклику конструктора черги.
  - Оскільки черга для зберігання даних використовує масив `object`-ів, в ній можна зберігати змінні будь-яких типів.

```

class Program
{
 static void Main(string[] args)
 {
 Queue<string> student = new Queue<string>();

 //Adding item in queue
 student.Enqueue("Mark");
 student.Enqueue("Jack");
 student.Enqueue("Sarah");
 student.Enqueue("Smith");
 student.Enqueue("Robbie");
 print(student);

 //Removing Item
 Console.WriteLine("\nRemoving {0} from List. \n
 New list is : ", student.Dequeue());
 print(student);

 //Copy Array Item to Queue
 string[] city = { "Newyork", "California", "Las Vegas" };
 Queue<string> citylist = new Queue<string>(city);
 Console.WriteLine("\nPrinting City List");
 print(citylist);

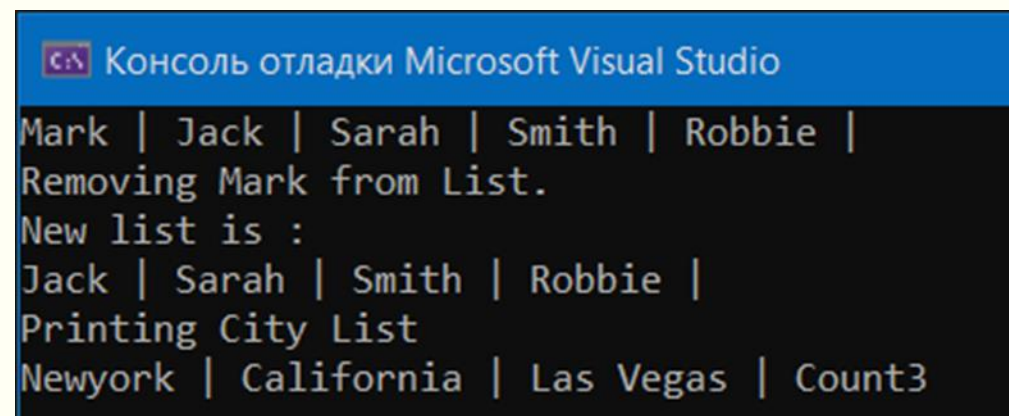
 //Count items in Queue
 Console.WriteLine("Count{0}", citylist.Count);
 }

 public static void print(Queue<string> q)
 {
 foreach (string s in q)
 {
 Console.Write(s.ToString() + " | ");
 }
 }
}
17.11.2020

```

## Черги

- При роботі з чергою, на відміну від ArrayList-а, ви не маєте доступу до її довільного елементу.
  - Наприклад, ви не можете подивитися третій елемент черзі, не видаливши з неї перші 2 елементи.
  - Проте клас Queue має метод Contains(), використовуючи який ви можете визначити, чи містить черга той чи інший конкретний елемент.
  - Якщо ви точно впевнені, що вам необхідний довільний доступ (random access), то використовуйте краще ArrayList.
  - Черга є ідеальним вибором в ситуаціях, коли необхідно виконувати обробку елементів даних точно в порядку їх надходження.

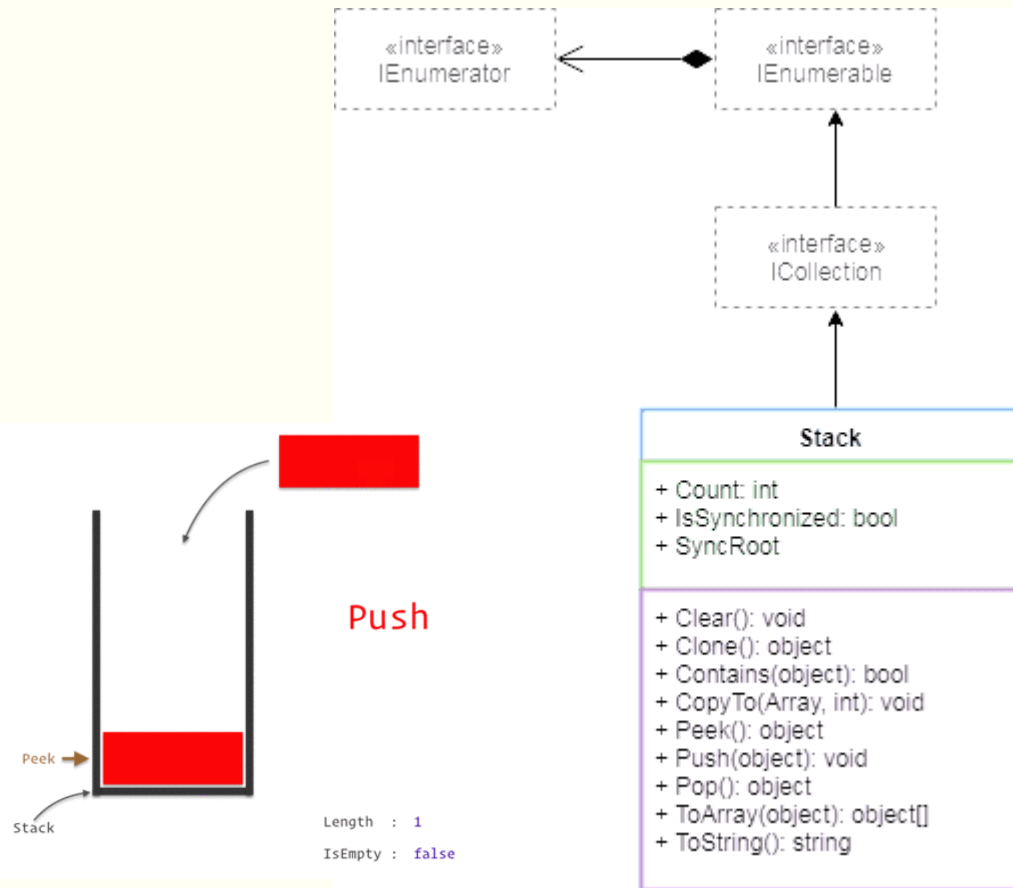


```

Консоль отладки Microsoft Visual Studio
Mark | Jack | Sarah | Smith | Robbie |
Removing Mark from List.
New list is :
Jack | Sarah | Smith | Robbie |
Printing City List
Newyork | California | Las Vegas | Count3

```

# Стек (неузагальнений)



class Program

```
{
 static void StackInfo(Stack st)
 {
 Console.WriteLine("Кількість елементів у стеку = " + st.Count);
 Console.WriteLine("Вершина стеку = " + st.Peek());

 foreach (Object obj in st) {
 Console.WriteLine("| " + obj + " |");
 Console.WriteLine("-----");
 }
 Console.WriteLine();
 }

 static void Main(string[] args)
 {
 Stack st = new Stack();

 st.Push(10);
 st.Push(20);
 st.Push(30);
 StackInfo(st);

 st.Pop();
 StackInfo(st);
 Console.WriteLine("Чи є у стеку трійка? " + st.Contains(3));
 }
}
```

Консоль отладки Microsoft Visual Studio

```
К?льк?сть елемент?в у стеку = 3
Вершина стеку = 30
30
20

10

К?льк?сть елемент?в у стеку = 2
Вершина стеку = 20
20
10

Чи є у стеку тр?йка? False
```

# Узагальнений стек

```
class Program
{
 static void Main(string[] args)
 {
 //Create Employee object
 Employee emp1 = new Employee() {
 ID = 101,
 Name = "Pranaya",
 Gender = "Male",
 Salary = 20000
 };
 Employee emp2 = new Employee() {
 ID = 102,
 Name = "Priyanka",
 Gender = "Female",
 Salary = 30000
 };
 Employee emp3 = new Employee() {
 ID = 103,
 Name = "Anurag",
 Gender = "Male",
 Salary = 40000
 };
 Employee emp4 = new Employee() {
 ID = 104,
 Name = "Sambit",
 Gender = "Female",
 Salary = 40000
 };

 Stack<Employee> stackEmployees = new Stack<Employee>();
 stackEmployees.Push(emp1);
 stackEmployees.Push(emp2);
 stackEmployees.Push(emp3);
 stackEmployees.Push(emp4);

 Console.WriteLine("Retrive Using Foreach Loop");
 foreach (Employee emp in stackEmployees) {
 Console.WriteLine(emp.ID + " - " + emp.Name + " - " + emp.Gender + " - " + emp.Salary);
 Console.WriteLine("Items left in the Stack = " + stackEmployees.Count);
 }
 Console.WriteLine("-----");
 Console.WriteLine("Retrive Using Pop Method");
 Employee e1 = stackEmployees.Pop();
 Console.WriteLine(e1.ID + " - " + e1.Name + " - " + e1.Gender + " - " + e1.Salary);
 Console.WriteLine("Items left in the Stack = " + stackEmployees.Count);
 Employee e2 = stackEmployees.Pop();
 Console.WriteLine(e2.ID + " - " + e2.Name + " - " + e2.Gender + " - " + e2.Salary);
 Console.WriteLine("Items left in the Stack = " + stackEmployees.Count);
 Employee e3 = stackEmployees.Pop();
 Console.WriteLine(e3.ID + " - " + e3.Name + " - " + e3.Gender + " - " + e3.Salary);
 Console.WriteLine("Items left in the Stack = " + stackEmployees.Count);
 Employee e4 = stackEmployees.Pop();
 Console.WriteLine(e4.ID + " - " + e4.Name + " - " + e4.Gender + " - " + e4.Salary);
 Console.WriteLine("Items left in the Stack = " + stackEmployees.Count);
 stackEmployees.Push(emp1);
 stackEmployees.Push(emp2);
 stackEmployees.Push(emp3);
 stackEmployees.Push(emp4);

 Console.WriteLine("Retrive Using Peek Method");
 Employee e105 = stackEmployees.Peek();
 Console.WriteLine(e105.ID + " - " + e105.Name + " - " + e105.Gender + " - " + e105.Salary);
 Console.WriteLine("Items left in the Stack = " + stackEmployees.Count);
 Employee e104 = stackEmployees.Peek();
 Console.WriteLine(e104.ID + " - " + e104.Name + " - " + e104.Gender + " - " + e104.Salary);
 Console.WriteLine("Items left in the Stack = " + stackEmployees.Count);

 Console.WriteLine("-----");
 if (stackEmployees.Contains(emp3)) { Console.WriteLine("Emp3 is in stack"); }
 else { Console.WriteLine("Emp3 is not in stack"); }
 Console.ReadKey();
 }
}
```

# Узагальнений стек

```
Retrive Using Foreach Loop
104 - Sambit - Female - 40000
Items left in the Stack = 4
103 - Anurag - Male - 40000
Items left in the Stack = 4
102 - Priyanka - Female - 30000
Items left in the Stack = 4
101 - Pranaya - Male - 20000
Items left in the Stack = 4

Retrive Using Pop Method
104 - Sambit - Female - 40000
Items left in the Stack = 3
103 - Anurag - Male - 40000
Items left in the Stack = 2
102 - Priyanka - Female - 30000
Items left in the Stack = 1
101 - Pranaya - Male - 20000
Items left in the Stack = 0
Retrive Using Peek Method
104 - Sambit - Female - 40000
Items left in the Stack = 4
104 - Sambit - Female - 40000
Items left in the Stack = 4

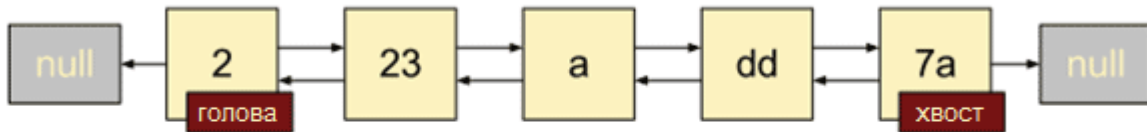
Emp3 is in stack
```

```
public class Employee
{
 public int ID { get; set; }
 public string Name { get; set; }
 public string Gender { get; set; }
 public int Salary { get; set; }
}
```

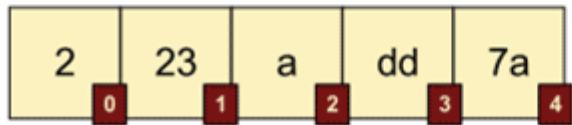


# Зв'язні списки

Связный список ( LinkedList)



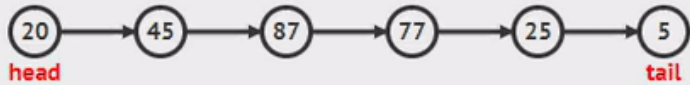
Массив ( Array и ArrayList)



|          | ArrayList | LinkedList     |
|----------|-----------|----------------|
| get()    | O(1)      | O(n)           |
| add()    | O(1)      | O(1) amortized |
| remove() | O(n)      | O(n)           |

- ArrayList всередині використовує динамічний масив, щоб зберегти елементи.
  - LinkedList базується на двозв'язному списку.
- Маніпуляції з ArrayList повільні: якщо видаляти елемент масиву, всі біти масиву після цього елементу зсуваються в пам'яті.
  - Операції з LinkedList швидші, оскільки двозв'язний список не вимагає зміщень у пам'яті.
- ArrayList доречніший для зберігання та доступу до даних.
  - LinkedList кращий для оперування даними.

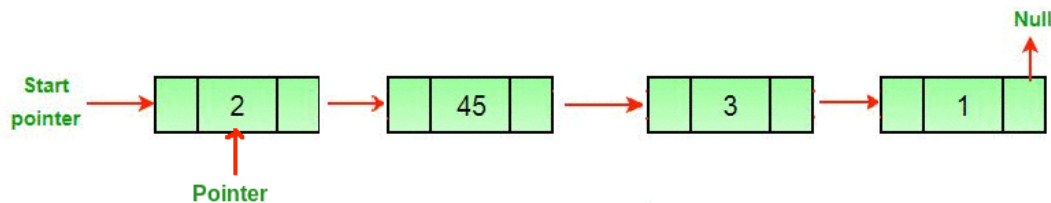
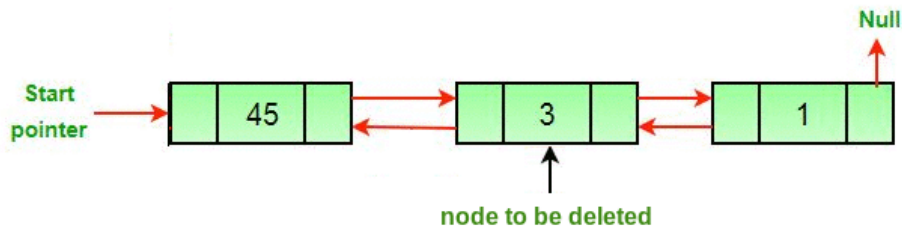
# Зв'язні списки



Зауваження щодо класу `LinkedList<T>`:

- Реалізує інтерфейси `ICollection<T>`, `IEnumerable<T>`, `IReadOnlyCollection<T>`, `ICollection`, `IEnumerable`, `IDeserializationCallback` та `ISerializable`.
- Також підтримує еnumератори.
- Можна видаляти вузли та повторно вставляти їх у той же чи інший список без виділення додаткових об'єктів з кучі.
- Кожний вузол (node) об'єкта `LinkedList<T>` має тип `LinkedListNode<T>`.
- Не підтримує `chaining`, `splitting`, `cycles` чи інші можливості, що можуть призвести до неузгодженого стану.
- Якщо зв'язний список порожній, властивості `First` та `Last` містять `null`.
- Ємність (capacity) зв'язного списку – це кількість елементів, які він здатний містити.

У зв'язному списку дозволяється зберігати дубльовані елементи, проте того ж типу.



# Зв'язні списки

---

- Клас `LinkedList<T>` має наступні властивості:
  - `Value`: саме значення вузла, представлене типом `T`
  - `Next`: ссылка на наступний елемент типу `LinkedListNode<T>` в списку. Якщо наступний елемент відсутній, то має значення `null`
  - `Previous`: ссылка на попередній елемент типу `LinkedListNode<T>` в списку. Якщо попередній елемент відсутній, то має значення `null`
- Використовуючи методи класу `LinkedList<T>`, можна звертатися до різних елементів, як до кінця, так і до початку списку:
  - `AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)`: вставляє вузол `newNode` в список після вузла `node`.
  - `AddAfter(LinkedListNode<T> node, T value)`: вставляє в список новий вузол з значенням `value` після вузла `node`.
  - `AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)`: вставляє в список вузол `newNode` перед вузлом `node`.
  - `AddBefore(LinkedListNode<T> node, T value)`: вставляє в список новий вузол з значенням `value` перед вузлом `node`.
  - `AddFirst(LinkedListNode<T> node)`: вставляє новий вузол в початок списку
  - `AddFirst(T value)`: вставляє новий вузол з значенням `value` в початок списку
  - `AddLast(LinkedListNode<T> node)`: вставляє новий вузол в кінець списку
  - `AddLast(T value)`: вставляє новий вузол з значенням `value` в кінець списку
  - `RemoveFirst()`: видаляє перший вузол з списку. Після цього новим першим вузлом стає вузол, наступний за видаленим
  - `RemoveLast()`: видаляє останній вузол з списку.

```
class Program
```

```
{
```

```
 static void Main(string[] args)
```

```
 {
```

```
 LinkedList<int> L = new LinkedList<int>();
```

```
 L.AddFirst(5);
```

```
 L.AddFirst(2);
```

```
 L.AddFirst(8);
```

```
 L.AddLast(4);
```

```
 L.AddLast(9);
```

```
 L.AddLast(1);
```

```
 L.AddBefore(L.Find(9), new LinkedListNode<int>(7));
```

```
 Console.WriteLine("Linked List elements are: ");
```

```
 foreach (int i in L)
```

```
 {
```

```
 Console.Write(i + " ");
```

```
 }
```

```
 L.RemoveFirst();
```

```
 L.RemoveLast();
```

```
 L.Find(2).Value = 12;
```

```
 Console.WriteLine("\nLinked List after deletion: ");
```

```
 foreach (int i in L)
```

```
 {
```

```
 Console.Write(i + " ");
```

```
 }
```

```
 Console.WriteLine("\nThe value 3 is present in Linked List: " + L.Contains(3));
```

```
 Console.WriteLine("The value 5 is present in Linked List: " + L.Contains(5));
```

```
 }
```

```
}
```

## Зв'язні списки



Консоль отладки Microsoft Visual Studio

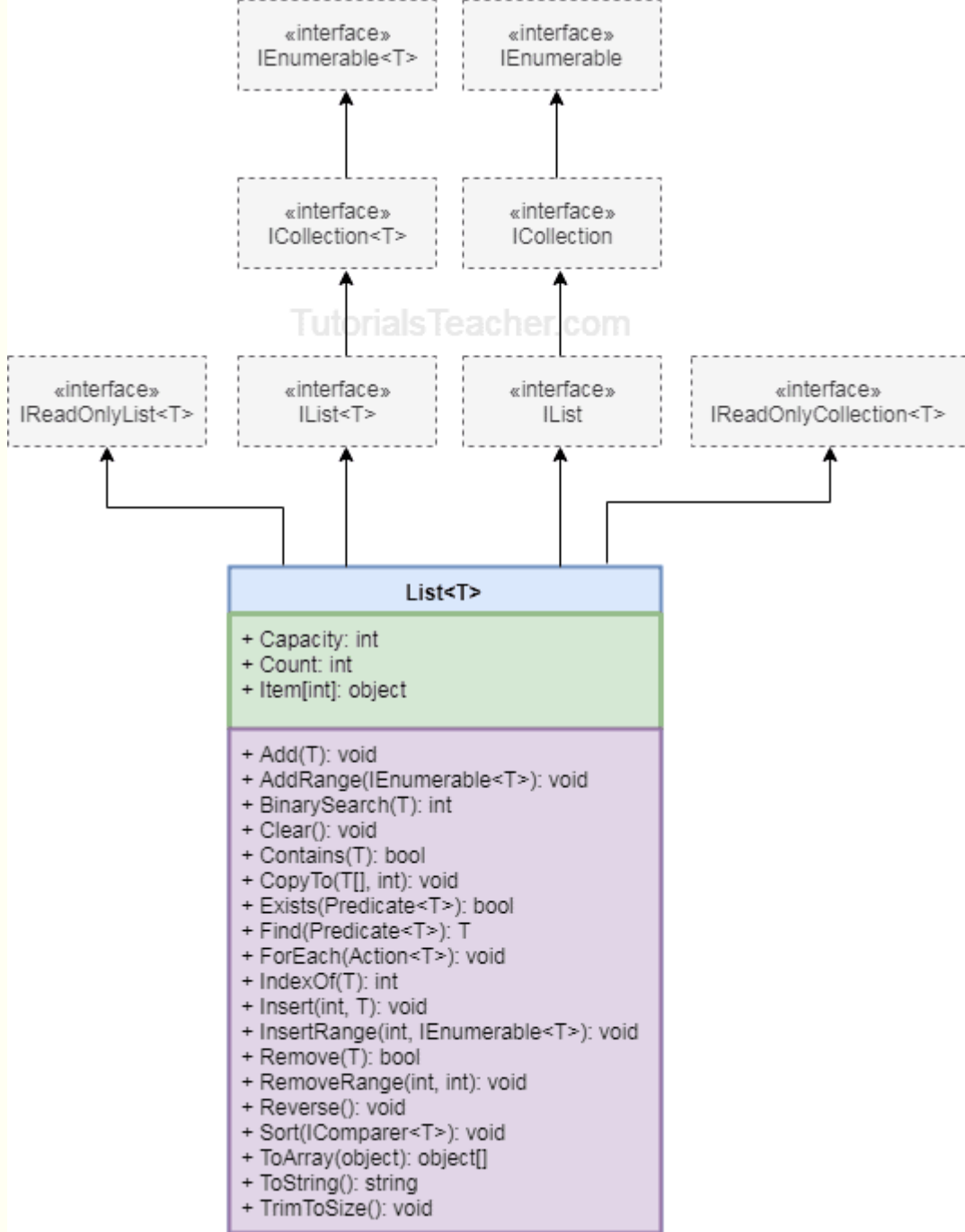
Linked List elements are: 8 2 5 4 7 9 1

Linked List after deletion: 12 5 4 7 9

The value 3 is present in Linked List: False

The value 5 is present in Linked List: True

# Клас List<T> - узагальнена версія ArrayList



- `List<T>` може містити елементи заданого типу.
  - Він забезпечує `compile-time type checking` та не виконує упаковання/розпакування, оскільки є узагальненням.
- Елементи можна вставляти за допомогою методів `Add()`, `AddRange()` чи синтаксису ініціалізаторів колекцій.
- Доступ до елементів здійснюється за індексом, наприклад, `myList[0]`. Індексуювання з 0.
- `List<T>` працює швидше та стабільніше, ніж `ArrayList`.

```

static void Main(string[] args)
{
 List<int> primeNumbers = new List<int>();
 primeNumbers.Add(2); // adding elements using add() method
 primeNumbers.Add(3);
 primeNumbers.Add(5);
 primeNumbers.Add(7);

 var cities = new List<string>();
 cities.Add("New York");
 cities.Add("London");
 cities.Add("Mumbai");
 cities.Add("Chicago");
 cities.Add(null); // nulls are allowed for reference type list

 //adding elements using collection-initializer syntax
 var bigCities = new List<string>()
 {
 "New York",
 "London",
 "Mumbai",
 "Chicago"
 };

 Console.WriteLine(primeNumbers[0]);
 Console.WriteLine(primeNumbers[1]);
 Console.WriteLine(primeNumbers[2]);
 Console.WriteLine(primeNumbers[3]);

 primeNumbers.Insert(1, 11); // inserts 11 at 1st index.
 primeNumbers.Remove(7);

 // using foreach LINQ method
 primeNumbers.ForEach(num => Console.Write(num + ", "));

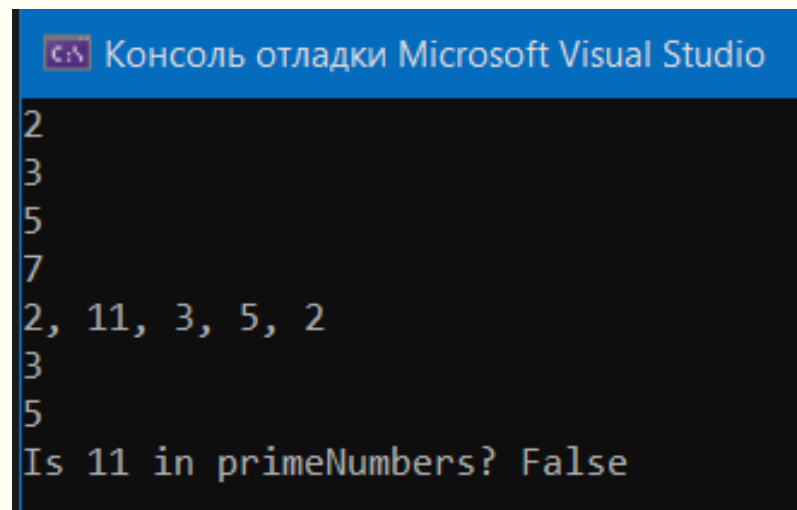
 primeNumbers.RemoveAt(1);

 // using for loop
 for (int i = 0; i < primeNumbers.Count; i++)
 Console.WriteLine(primeNumbers[i]);

 Console.WriteLine("Is 11 in primeNumbers? {0}", primeNumbers.Contains(11));
}

```

# Клас List<T> - узагальнена версія ArrayList



```

Консоль отладки Microsoft Visual Studio
2
3
5
7
2, 11, 3, 5, 2
3
5
Is 11 in primeNumbers? False

```

# Обмеження порядкового індексування елементів

|             | Name           | Phone    | Salary    | Dept. |
|-------------|----------------|----------|-----------|-------|
| 000-00-0000 |                |          |           |       |
| ...         | ...            |          |           |       |
| 455-11-0189 | Scott Mitchell | 333-4444 | \$134,500 | Sales |
| 455-11-0190 |                |          |           |       |
| 455-11-0191 | Jisun Lee      | 555-6666 | \$196,750 | Exec. |
| ...         | ...            |          |           |       |
| 999-99-9999 |                |          |           |       |

- Позиція потрібного елемента колекції часто нам невідома.
  - Нехай є БД працівників підприємства. Кожний працівник унікально ідентифікується за допомогою номера соціального страхування (в США – має вигляд DDD-DD-DDDD, де D - це цифра (0-9)).
  - Для невпорядкованого масиву пошук, наприклад, працівника з номером 111-22-3333 має здійснюватись серед усіх елементів масиву.
  - Краще попередньо відсортувати працівників за номером соціального страхування, зменшивши асимптотичний час пошуку до  $O(\log n)$ .
- В ідеалі, потрібно знаходити запис щодо конкретного робітника за час  $O(1)$ .
  - Один спосіб – створити величезний масив, у якому номер працівника відповідає номеру соціального страхування.
- Кожний запис про робітника містить деяку інформацію: ім'я (Name), номер телефону (Phone), зарплату (Salary), і т. д.
  - Кожний запис індексується за допомогою номеру соціального страхування, а асимптотичний час пошуку -  $O(1)$ .
  - Недолік підходу – жахлива розтрата пам'яті: тут існує мільярд різних номерів соціального страхування.
  - Для компанії з 1000 працівників масив фактично використовуватиме лише 0,0001% свого об'єму, що неприйнятно.

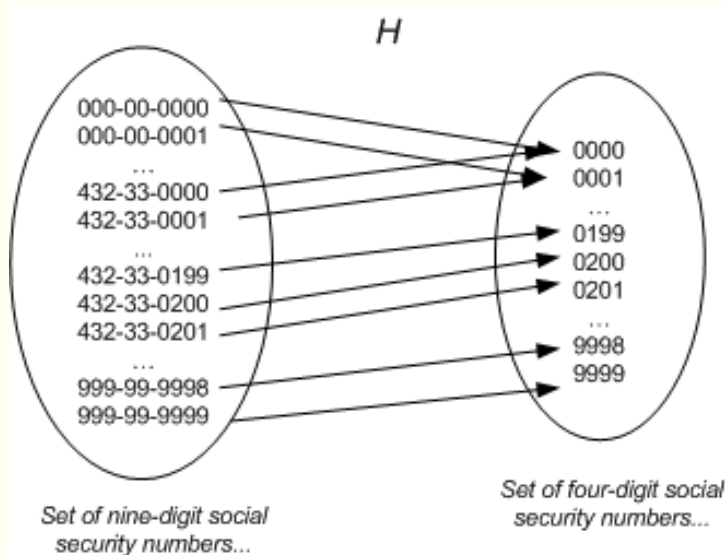
# Стиснення порядкового індексування за допомогою хеш-функції

|      | Name           | Phone    | Salary    | Dept. |
|------|----------------|----------|-----------|-------|
| 0000 | Dave Yates     | 111-2222 | \$75,000  | HR    |
| ...  | ...            |          |           |       |
| 0189 | Scott Mitchell | 333-4444 | \$134,500 | Sales |
| 0190 |                |          |           |       |
| 0191 | Jisun Lee      | 555-6666 | \$196,750 | Exec. |
| ...  | ...            |          |           |       |
| 9999 |                |          |           |       |

- Як варіант, можна використовувати для нумерації записів працівників не весь номер соціального страхування, а лише його останні 4 цифри.
  - Тоді отримаємо масив з діапазоном елементів від 0000 до 9999.
  - При такому підході матимемо постійний час пошуку та порівняно економну витрату пам'яті.
  - Вибір на користь останніх 4 цифр здійснено довільно, можна застосувати інші комбінації.
- Математичне перетворення 9-значного числа в 4-значне є прикладом **хешування** (від англ. Hash – перемелювати, перемішувати).
  - Масив, що використовує хешування для стиснення свого простору індексів, називається **хеш-таблицею** (*hash table*).



# Хеш-функція



- Хеш-функція – це функція, що здійснює процес хешування.
  - Для номеру соціального страхування хеш-функція,  $H$ , може бути описана так:
  - $H(x)$  = останні 4 цифри числа  $x$
  - В математичних термінах,  $H$  здійснює відображення (maps) множини 9-значних номерів соціального страхування на множину 4-значних чисел.
- Рисунок також демонструє явища, притаманні всім хеш-функціям – **колізії (collisions)**.
  - Колізія – це явище, коли хеш-функція відображає 2 різних елементи з більш широкої множини в один і той же елемент більш вузької множини.
  - Тут всі номери, що закінчуються на 0000, відобразяться хеш-функцією в 0000.
  - На рисунку, якщо спробуємо додати запис про працівника з номером 123-00-0191, виникнуть проблеми, оскільки в масиві вже існує працівник у комірці з номером 0191 (Jisun Lee).

|      | Name           | Phone    | Salary    | Dept. |
|------|----------------|----------|-----------|-------|
| 0000 | Dave Yates     | 111-2222 | \$75,000  | HR    |
| ...  | ...            |          |           |       |
| 0189 | Scott Mitchell | 333-4444 | \$134,500 | Sales |
| 0190 |                |          |           |       |
| 0191 | Jisun Lee      | 555-6666 | \$196,750 | Exec. |
| ...  | ...            |          |           |       |
| 9999 |                |          |           |       |

# Уникнення та вирішення колізій

---

- Якби колізія не виникла, ми б просто записали дані в елемент масиву, індекс якого було заздалегідь обчислено за допомогою хеш-функції.
  - Якщо маємо справу з колізією, потрібно виконати певні дії з її усунення.
  - Це сповільнює роботу, тому мета – зробити так, щоб колізії траплялись якомога рідше.
- Частота колізій напряду пов'язана з використаною хеш-функцією та розподілом даних, що подаються на вхід хеш-функції.
  - У прикладі хеш-функція практично ідеальна, якщо вважати, що номери соціального страхування присвоюються людям випадковим чином.
  - Проте якщо ці номери призначаються таким чином, що народжені в один рік та в сусідній місцевості люди отримують номери з однаковими 4ма цифрами, це може викликати велику кількість колізій (нерівномірний розподіл).
- Вибір відповідної хеш-функції називають **уникненням колізій**.
- При виникненні колізії існують різні підходи до її **вирішення**.
  - Задача вирішення колізії полягає в знаходженні іншого місця для об'єкта, що додається в зайняту комірку хеш-таблиці.

# Уникнення та вирішення колізій

|      | Name   | Phone | Salary | Dept. |
|------|--------|-------|--------|-------|
| 0000 |        |       |        |       |
| ...  | ...    |       |        |       |
| 1234 | Alice  | ...   | ...    | ...   |
| 1235 | Bob    | ...   | ...    | ...   |
| 1236 | Danny  | ...   | ...    | ...   |
| 1237 | Cal    | ...   | ...    | ...   |
| 1238 | Edward | ...   | ...    | ...   |
| ...  | ...    |       |        |       |
| 9999 |        |       |        |       |

- Один з найпростіших методів вирішення колізій – **лінійне зондування (*linear probing*)**:
  1. Коли новий елемент додається в хеш-таблицю, застосовуємо хеш-функцію, щоб визначити, в яке місце таблиці слід записати цей елемент.
  2. Перевіряємо, чи існує елемент у позиції, знайдений на кроці 1 за допомогою хеш-функції. Якщо позиція вільна, поміщаємо туди елемент, інакше переходимо до кроку 3.
  3. Нехай номер позиції, виданий хеш-функцією –  $i$ . Тоді перевіряємо чи зайнята позиція з номером  $i+1$ . Якщо  $i$  вона зайнята, перевіряємо позицію  $i+2$  і т. д., поки не знайдеться вільна комірка.
- Наприклад, маємо таблицю з 5ма працівниками:
  - Alice (333-33-1234), Bob (444-44-1234), Cal (555-55-1237), Danny (000-00-1235) та Edward (111-00-1235).
  - Вигляд хеш-таблиці показано на рисунку.

# Уникнення та вирішення колізій

|      | Name   | Phone | Salary | Dept. |
|------|--------|-------|--------|-------|
| 0000 |        |       |        |       |
| ...  | ...    |       |        |       |
| 1234 | Alice  | ...   | ...    | ...   |
| 1235 | Bob    | ...   | ...    | ...   |
| 1236 | Danny  | ...   | ...    | ...   |
| 1237 | Cal    | ...   | ...    | ...   |
| 1238 | Edward | ...   | ...    | ...   |
| ...  | ...    |       |        |       |
| 9999 |        |       |        |       |

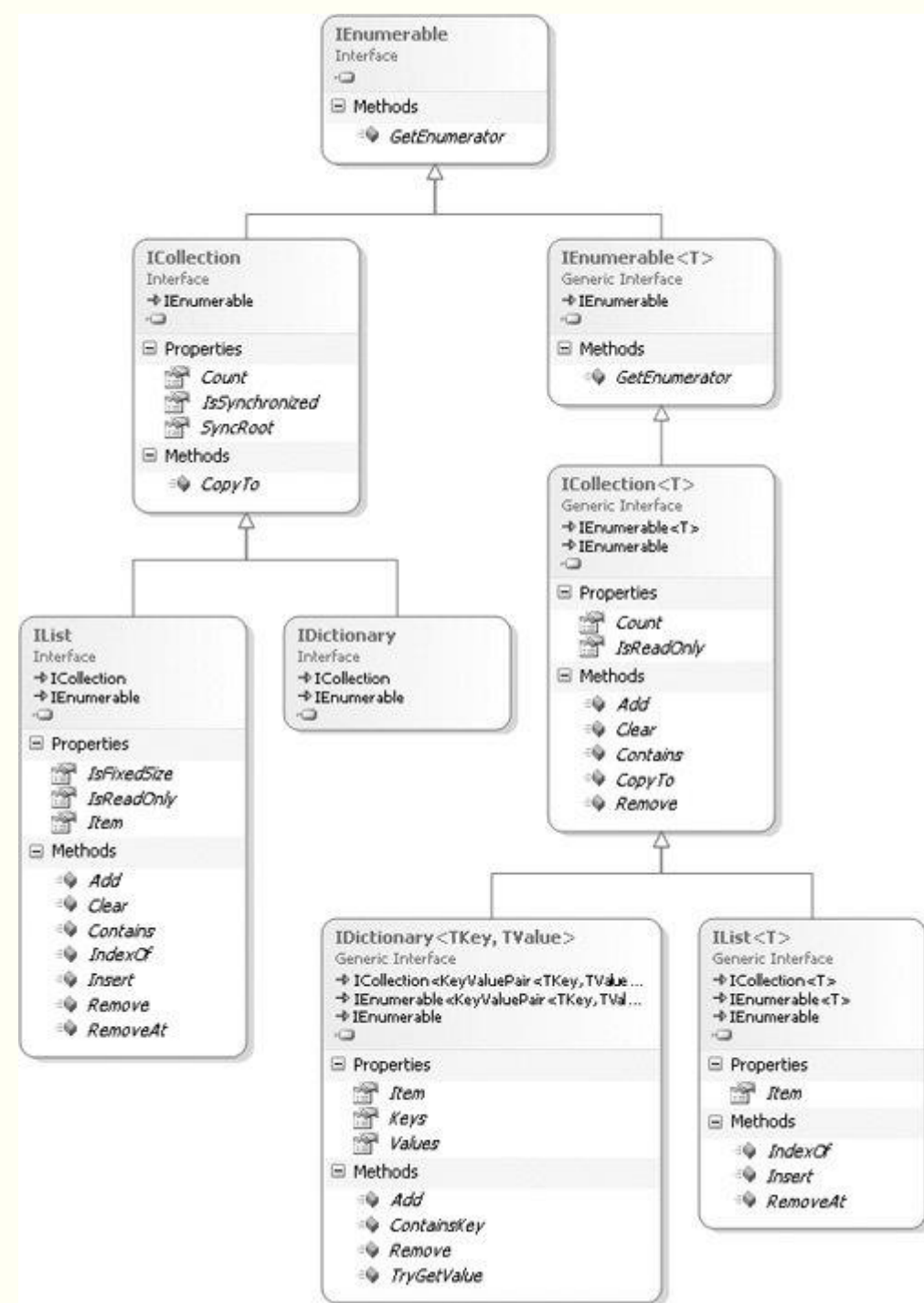
- Номер соціального страхування Аліси хешується в значення 1234, а запис поміщається в 1234-й елемент масиву.
  - Номер Боба теж хешується в 1234, проте комірка 1234 вже зайнята записом Аліси, тому Боб займає наступну вільну комірку - 1235.
  - Далі додаємо запис Кела, хеш-функція видає для його номеру соціального страхування значення 1237, запис Кела поміщається в комірку 1237.
  - Наступний – Денні, його номер відображається хеш-функцією в 1235, проте вона зайнята, і перевіряється комірка 1236. Оскільки вона вакантна, записуємо Денні в неї.
  - Наприкінці додаємо запис Едварда, номер соціального страхування якого також хешується в 1235. Відбувається послідовна перевірка комірок 1235, 1236, 1237, 1238, в останню заноситься запис Едварда.
- Через колізії виникають проблеми при пошуку в хеш-таблиці.
  - Наприклад, потрібно знайти інформацію про Едварда: беремо номер соціального страхування 111-00-1235, хешуємо його, отримуємо 1235 і починаємо пошук.
  - У комірці 1235 знаходимо Боба, а не Едварда.
  - Перевіряємо комірку 1236, проте там Денні.
  - Лінійний пошук буде продовжуватись доти, доки не знайдемо Едварда або не дійдемо до порожньої комрки.
  - Останнє означає, що такого працівника в хеш-таблиці не існує.

# Уникнення та вирішення колізій

|      | Name   | Phone | Salary | Dept. |
|------|--------|-------|--------|-------|
| 0000 |        |       |        |       |
| ...  |        | ...   |        |       |
| 1234 | Alice  | ...   | ...    | ...   |
| 1235 | Bob    | ...   | ...    | ...   |
| 1236 | Danny  | ...   | ...    | ...   |
| 1237 | Cal    | ...   | ...    | ...   |
| 1238 | Edward | ...   | ...    | ...   |
| ...  |        | ...   |        |       |
| 9999 |        |       |        |       |

- Незважаючи на простоту лінійного зондування, воно не є кращим способом вирішення колізій, оскільки веде до утворення кластерів (clustering).
  - Нехай перші 10 співробітників, яких додали в хеш-таблицю, всі мають номер соціального страхування, який закінчується на однакові 4 цифри, наприклад, 3344.
  - Тоді будуть зайняті 10 послідовних комірок масиву від 3344 до 3353.
  - При спробі пошуку даних будь-якого з цих співробітників відбуватиметься процес лінійної послідовності проб.
  - Більш того, додавання співробітників, для яких значення хеш-функції лежить в інтервалі від 3345 до 3353 призведе до подальшого розростання кластера.
  - Для швидкого пошуку, звичайно ж, краще мати рівномірний розподіл даних в хеш-таблиці, а не кластеризований в околиці деяких точок.
- Складнішою є квадратичне зондування (quadratic probing), яке починає перевіряти комірки на квадратичній відстані одна від одної.
  - Якщо комірка  $s$  зайнята, то спочатку перевіряється осередок  $(s+1)^2$ , потім  $(s-1)^2$ , потім  $(s+2)^2$ , потім  $(s-2)^2$ , після цього  $(s+3)^2$  і т. д.
  - Однак навіть квадратичний варіант може привести до утворення кластерів.

# Інтерфейси IDictionary / IDictionary<TKey, TValue>



```

public class Example {
 public static void Main() {
 IDictionary<string, string> openWith = new Dictionary<string, string>();
 openWith.Add("txt", "notepad.exe");
 openWith.Add("bmp", "paint.exe");
 openWith.Add("dib", "paint.exe");
 openWith.Add("rtf", "wordpad.exe");
 // The Add method throws an exception if the new key is already in the dict.
 try { openWith.Add("txt", "winword.exe");
 } catch (ArgumentException) {
 Console.WriteLine("An element with Key = \"txt\" already exists.");
 }
 // The Item property is another name for the indexer, so you
 // can omit its name when accessing elements.
 Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);
 // The indexer can be used to change the value associated with a key.
 openWith["rtf"] = "winword.exe";
 Console.WriteLine("For key = \"rtf\", value = {0}.",
 openWith["rtf"]);
 // If a key does not exist, setting the indexer for that key
 // adds a new key/value pair.
 openWith["doc"] = "winword.exe";
 // The indexer throws an exception if the requested key is
 // not in the dictionary.
 try { Console.WriteLine("For key = \"tif\", value = {0}.", openWith["tif"]);
 } catch (KeyNotFoundException) {
 Console.WriteLine("Key = \"tif\" is not found.");
 }
 }
}

```



```

// When a program often has to try keys that turn out not to be in the dictionary,
// TryGetValue can be a more efficient way to retrieve values.
string value = "";
if (openWith.TryGetValue("tif", out value)) {
 Console.WriteLine("For key = \"tif\", value = {0}.", value);
} else { Console.WriteLine("Key = \"tif\" is not found."); }
// ContainsKey can be used to test keys before inserting them.
if (!openWith.ContainsKey("ht")) {
 openWith.Add("ht", "hypertrm.exe");
 Console.WriteLine("Value added for key = \"ht\": {0}",
 openWith["ht"]);
}
// When you use foreach to enumerate dictionary elements,
// the elements are retrieved as KeyValuePair objects.
Console.WriteLine();
foreach(KeyValuePair<string, string> kvp in openWith) {
 Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
}
// To get the values alone, use the Values property.
ICollection<string> icoll = openWith.Values;
// The elements of the ValueCollection are strongly typed
// with the type that was specified for dictionary values.
Console.WriteLine();
foreach(string s in icoll) { Console.WriteLine("Value = {0}", s); }
// To get the keys alone, use the Keys property.
icoll = openWith.Keys;
// The elements of the ValueCollection are strongly typed
// with the type that was specified for dictionary values.
Console.WriteLine();
foreach(string s in icoll) { Console.WriteLine("Key = {0}", s); }
// Use the Remove method to remove a key/value pair.
Console.WriteLine("\nRemove(\"doc\")");
openWith.Remove("doc");
if (!openWith.ContainsKey("doc")) { Console.WriteLine("Key \"doc\" is not found."); }
}
}

```

## Продовження коду

```

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
Key = "tif" is not found.
Key = "tif" is not found.
Value added for key = "ht": hypertrm.exe

```

```

Key = txt, Value = notepad.exe
Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = rtf, Value = winword.exe
Key = doc, Value = winword.exe
Key = ht, Value = hypertrm.exe

```

```

Value = notepad.exe
Value = paint.exe
Value = paint.exe
Value = winword.exe
Value = winword.exe
Value = hypertrm.exe

```

```

Key = txt
Key = bmp
Key = dib
Key = rtf
Key = doc
Key = ht

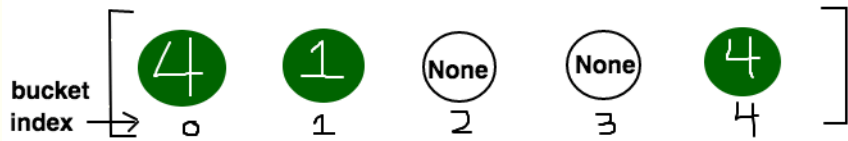
```

```

Remove("doc")
Key "doc" is not found.

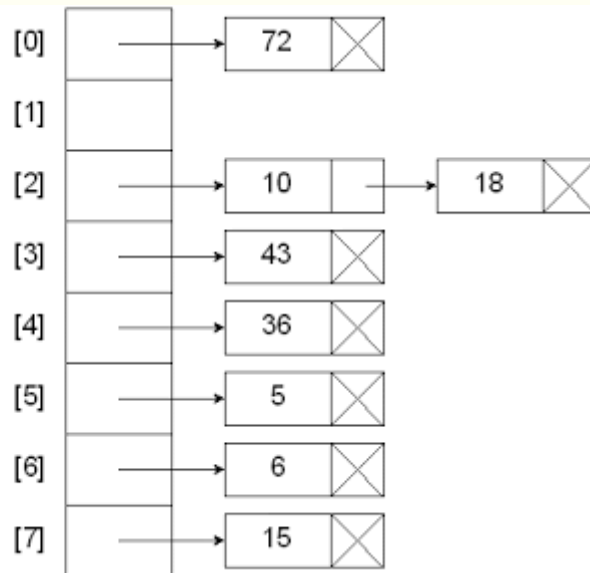
```

# Клас System.Collections.Hashtable (неузагальнена хеш-таблиця)



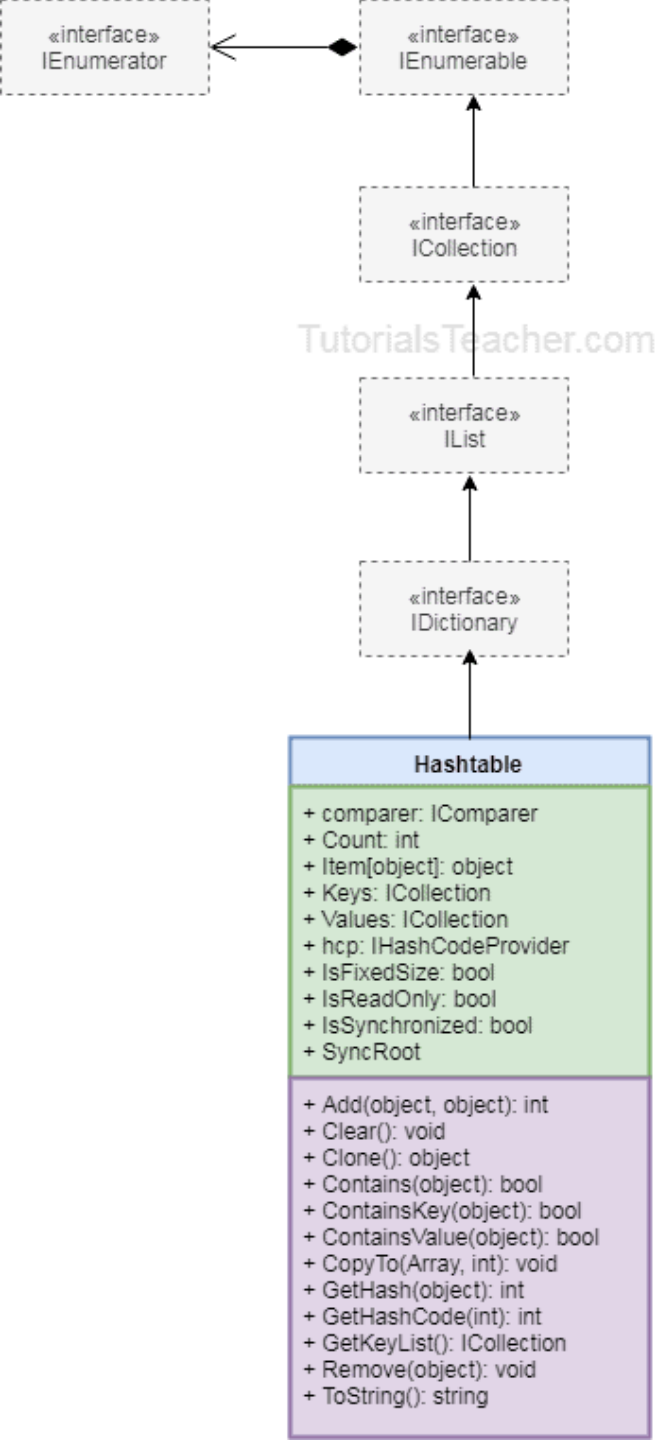
Hash key = key % table size

|   |   |    |   |   |
|---|---|----|---|---|
| 4 | = | 36 | % | 8 |
| 2 | = | 18 | % | 8 |
| 0 | = | 72 | % | 8 |
| 3 | = | 43 | % | 8 |
| 6 | = | 6  | % | 8 |
| 2 | = | 10 | % | 8 |
| 5 | = | 5  | % | 8 |
| 7 | = | 15 | % | 8 |



- При додаванні елемента в Hashtable ви повинні передати не тільки дані, але і унікальний ключ, за яким цей елемент може бути знайдений.
  - Як ключ так і дані можуть бути будь-якого типу.
  - У прикладі зі співробітниками ключем був номер соціального страхування.
  - Елементи додаються в Hashtable за допомогою методу Add().
- Кожен об'єкт, який використовується в якості елемента в Hashtable, повинен мати можливість створити свій хеш-код, використовуючи реалізацію методу GetHashCode.
  - Однак хеш-функцію також можна вказати для всіх елементів в Hashtable, використовуючи конструктор Hashtable, що приймає реалізацію IHashCodeProvider в якості одного зі своїх параметрів.





# Клас System.Collections.Hashtable (неузагальнена хеш-таблиця)

```

class Program
{
 static void Main(string[] args)
 {
 Hashtable numberNames = new Hashtable();
 numberNames.Add(1, "One"); //adding a key/value using the Add() method
 numberNames.Add(2, "Two");
 numberNames.Add(3, "Three");

 //The following throws run-time exception: key already added.
 //numberNames.Add(3, "Three");

 foreach (DictionaryEntry de in numberNames)
 Console.WriteLine("Key: {0}, Value: {1}", de.Key, de.Value);

 //creating a Hashtable using collection-initializer syntax
 var cities = new Hashtable(){
 {"UK", "London, Manchester, Birmingham"},
 {"USA", "Chicago, New York, Washington"},
 {"India", "Mumbai, New Delhi, Pune"}
 };

 Console.WriteLine();
 foreach (DictionaryEntry de in cities)
 Console.WriteLine("Key: {0}, Value: {1}", de.Key, de.Value);
 }
}

```

# Продовження коду

---

```
string citiesOfUK = (string)cities["UK"]; //cast to string
string citiesOfUSA = (string)cities["USA"]; //cast to string

Console.WriteLine();
Console.WriteLine(citiesOfUK);
Console.WriteLine(citiesOfUSA);

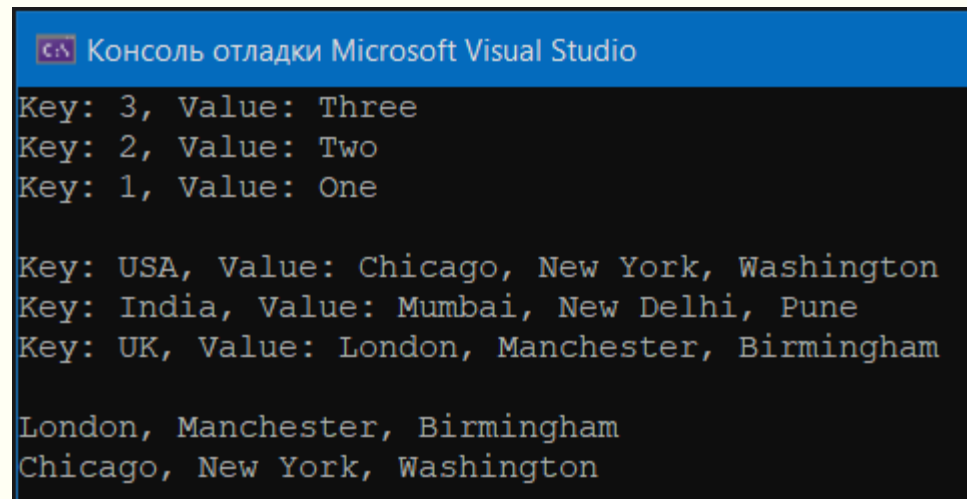
cities["UK"] = "Liverpool, Bristol"; // update value of UK key
cities["USA"] = "Los Angeles, Boston"; // update value of USA key

if (!cities.ContainsKey("France"))
{
 cities["France"] = "Paris";
}

cities.Remove("UK"); // removes UK
//cities.Remove("France"); //throws run-time exception: KeyNotFoundException

if (cities.ContainsKey("France"))
{ // check key before removing it
 cities.Remove("France");
}

cities.Clear(); //removes all elements
}
```



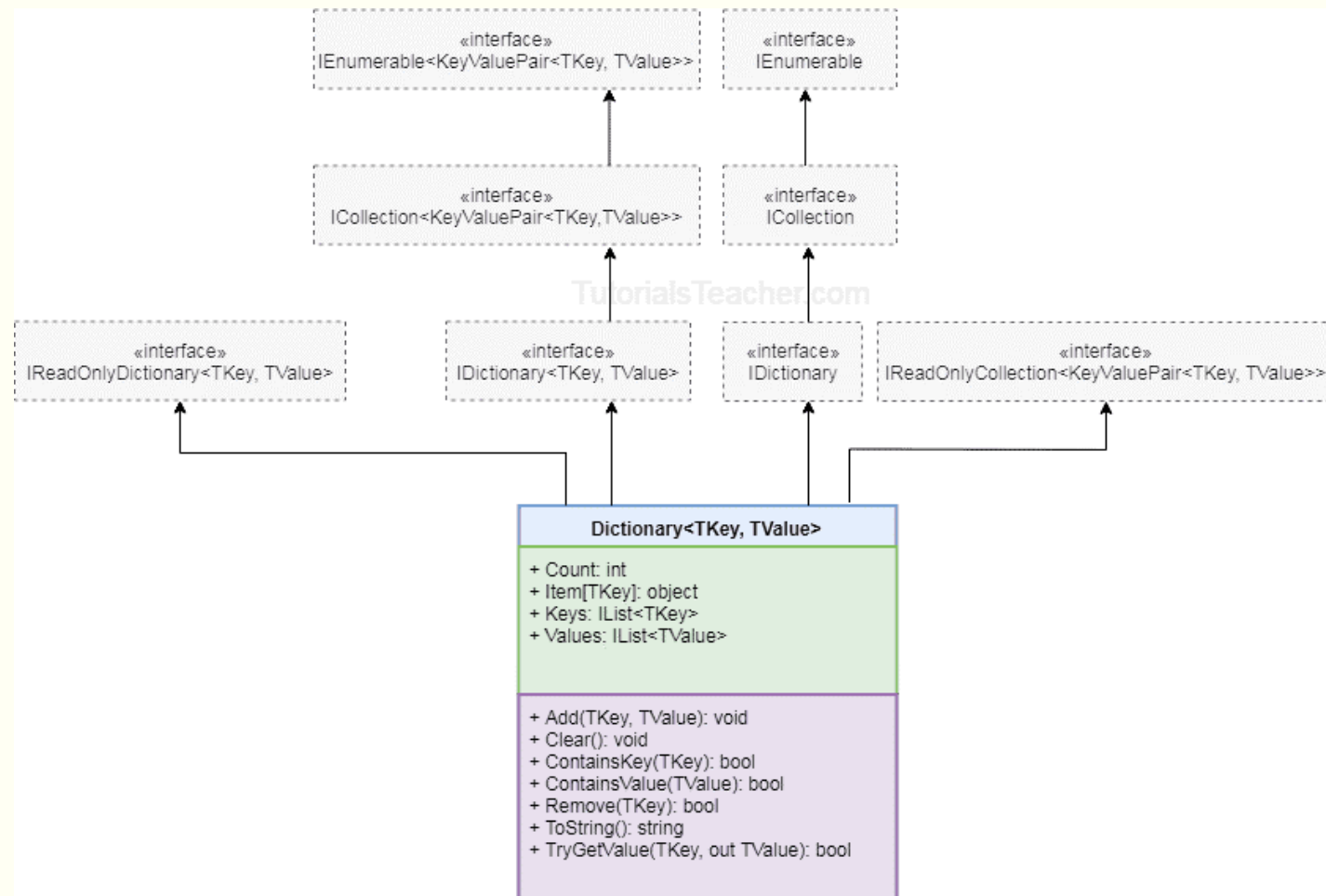
Консоль отладки Microsoft Visual Studio

Key: 3, Value: Three  
Key: 2, Value: Two  
Key: 1, Value: One

Key: USA, Value: Chicago, New York, Washington  
Key: India, Value: Mumbai, New Delhi, Pune  
Key: UK, Value: London, Manchester, Birmingham

London, Manchester, Birmingham  
Chicago, New York, Washington

# Узагальнена хеш-таблиця (словник System.Collections.Generic.Dictionary)



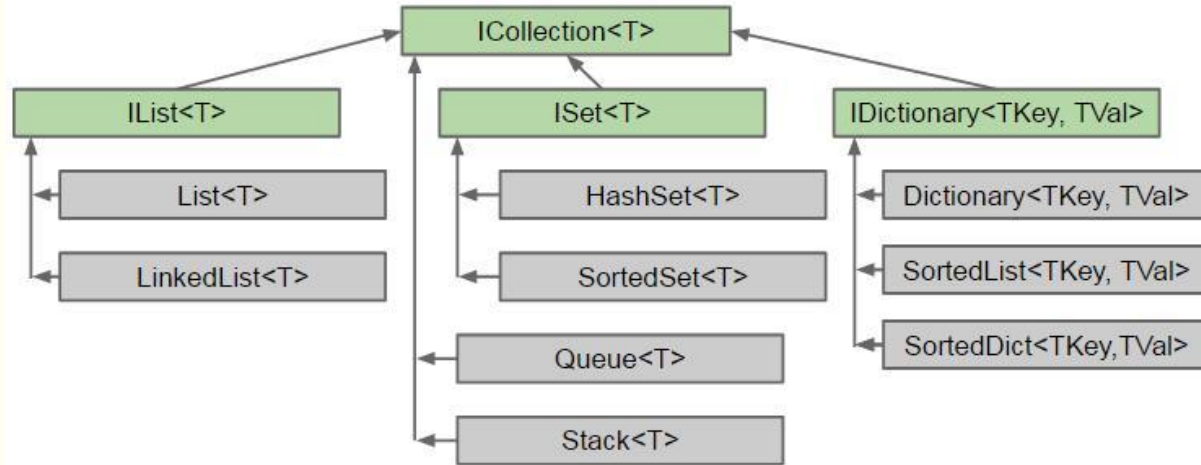
- `Dictionary<TKey, TValue>` зберігає пари «ключ-значення».
  - Знаходиться в просторі імен `System.Collection.Generic`.
  - Реалізує інтерфейс `IDictionary<TKey, TValue>`.
  - Ключі повинні бути унікальними та не можуть мати null-значення.
  - Значення можуть бути null або дублюватись.
  - Доступ до значень можна отримати, передаючи відповідний ключ в індексатор: `myDictionary[key]`
  - Елементи зберігаються як об'єкти типу `KeyValuePair<TKey, TValue>`.

# Відмінності між Hashtable та Dictionary

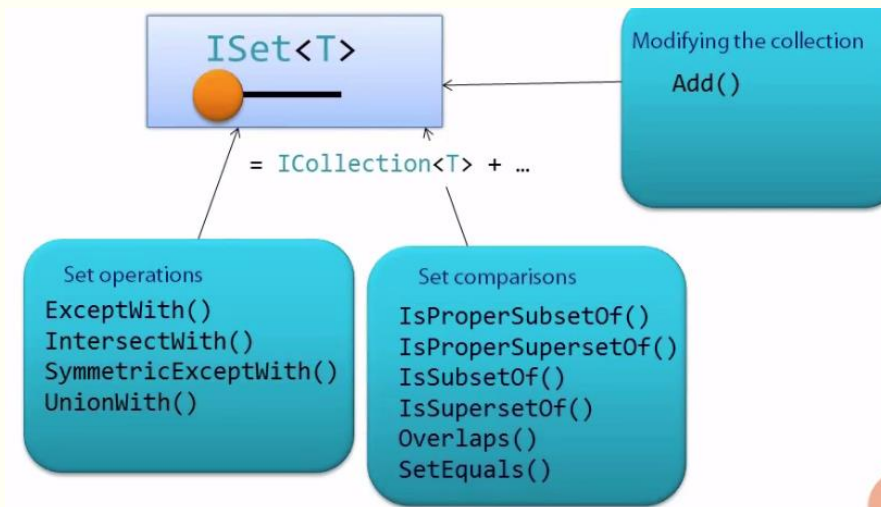
---

| Hashtable                                                               | Dictionary                                                               |
|-------------------------------------------------------------------------|--------------------------------------------------------------------------|
| Знаходиться в просторі імен System.Collections.                         | Знаходиться в просторі імен System.Collections.Generic.                  |
| Слабко типізована (loosely typed, неузагальнена, non-generic) колекція. | Узагальнена колекція, зберігає пари «ключ-значення» заданих типів даних. |
| Потокобезпечна колекція.                                                | Тільки відкриті статичні члени Dictionary є потокобезпечними.            |
| Повертає null при спробі знайти неіснуючий ключ.                        | Генерує виняток, якщо намагаємось знайти неіснуючий ключ.                |
| Отримання даних повільніше у зв'язку з упаковкою/розпаковкою.           | Отримання даних швидше, ніж у Hashtable.                                 |

# Множини. Інтерфейс ISet<T>



- Колекція, які містить тільки унікальні елементи, називається **множиною (set)**.
  - До складу .NET 4 входять 2 множини — `HashSet<T>` і `SortedSet<T>`.
  - Обидві реалізують інтерфейс `ISet<T>`.
- Властивості інтерфейсу `ISet<T>`:
  - Count – успадковано від ICollection<T>.
  - IsReadOnly – успадковано від ICollection<T>



## Методы интерфейсу ISet<T>

|                                                  |                                                                                                                                                                                    |
|--------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <u>Add(T)</u>                                    | Добавляет элемент в текущий набор и возвращает значение, указывающее, что элемент был добавлен успешно.                                                                            |
| <u>Clear()</u>                                   | Удаляет все элементы из коллекции <u>ICollection&lt;T&gt;</u> . (Унаследовано от <u>ICollection&lt;T&gt;</u> )                                                                     |
| <u>Contains(T)</u>                               | Определяет, содержит ли коллекция <u>ICollection&lt;T&gt;</u> указанное значение. (Унаследовано от <u>ICollection&lt;T&gt;</u> )                                                   |
| <u>CopyTo(T[], Int32)</u>                        | Копирует элементы коллекции <u>ICollection&lt;T&gt;</u> в массив <u>Array</u> , начиная с указанного индекса массива <u>Array</u> . (Унаследовано от <u>ICollection&lt;T&gt;</u> ) |
| <u>ExceptWith(IEnumerable&lt;T&gt;)</u>          | Удаляет все элементы указанной коллекции из текущего набора.                                                                                                                       |
| <u>GetEnumerator()</u>                           | Возвращает перечислитель, который осуществляет итерацию по коллекции. (Унаследовано от <u>IEnumerable</u> )                                                                        |
| <u>IntersectWith(IEnumerable&lt;T&gt;)</u>       | Изменяет текущий набор, чтобы он содержал только элементы, которые также имеются в заданной коллекции.                                                                             |
| <u>IsProperSubsetOf(IEnumerable&lt;T&gt;)</u>    | Определяет, является ли текущий набор должным (строгим) подмножеством заданной коллекции.                                                                                          |
| <u>IsProperSupersetOf(IEnumerable&lt;T&gt;)</u>  | Определяет, является ли текущий набор должным (строгим) подмножеством заданной коллекции.                                                                                          |
| <u>IsSubsetOf(IEnumerable&lt;T&gt;)</u>          | Определяет, является ли набор подмножеством заданной коллекции.                                                                                                                    |
| <u>IsSupersetOf(IEnumerable&lt;T&gt;)</u>        | Определяет, является ли текущий набор надмножеством заданной коллекции.                                                                                                            |
| <u>Overlaps(IEnumerable&lt;T&gt;)</u>            | Определяет, пересекаются ли текущий набор и указанная коллекция.                                                                                                                   |
| <u>Remove(T)</u>                                 | Удаляет первое вхождение указанного объекта из коллекции <u>ICollection&lt;T&gt;</u> . (Унаследовано от <u>ICollection&lt;T&gt;</u> )                                              |
| <u>SetEquals(IEnumerable&lt;T&gt;)</u>           | Определяет, содержат ли текущий набор и указанная коллекция одни и те же элементы.                                                                                                 |
| <u>SymmetricExceptWith(IEnumerable&lt;T&gt;)</u> | Изменяет текущий набор таким образом, чтобы он содержал только элементы, которые есть либо в нем, либо в указанной коллекции, но не одновременно там и там.                        |
| <u>UnionWith(IEnumerable&lt;T&gt;)</u>           | Изменяет текущий набор так, чтобы он содержал все элементы, которые имеются в текущем наборе, в указанной коллекции либо в них обоих.                                              |

# Клас HashSet<T>

---

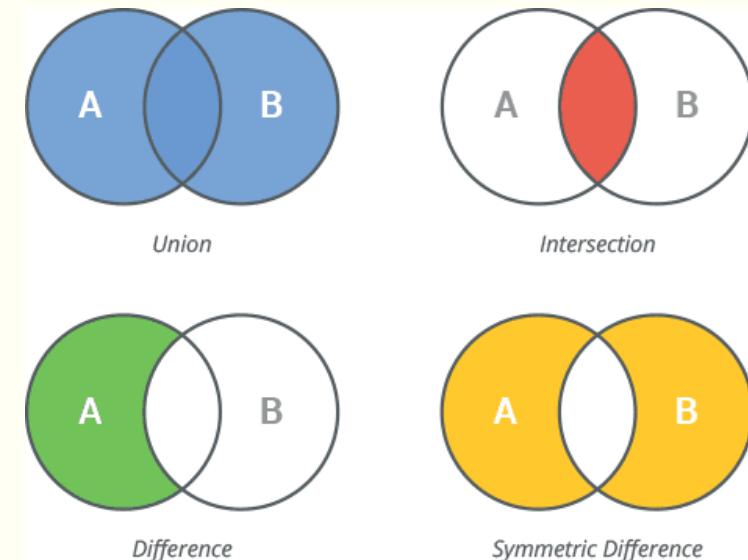
- Оптимізована колекція невідсортованих, унікальних елементів, яка забезпечує швидкий пошук (lookups) та високопродуктивні операції з множинами.
  - Представлений у .NET 3.5 та знаходиться в просторі імен System.Collection.Generic.
- HashSet не є відсортованою колекцією та не містить дублікати елементів.
  - Також не підтримує індекси, можна використовувати лише енумератори.
  - HashSet зазвичай використовується для високопродуктивних операцій над набором унікальних елементів.
- Клас HashSet<T> реалізує кілька інтерфейсів:
  - `public class HashSet<T> : System.Collections.Generic.ICollection<T>, System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.ICollection<T>, System.Collections.Generic.ISet<T>, System.Runtime.Serialization.IDeserializationCallback, System.Runtime.Serialization.ISerializable`
  - Оскільки HashSet містить тільки унікальні елементи, його внутрішня структура оптимізована для швидкого пошуку.
  - Зауважте, що доступне зберігання одного null-значення в хешсеті.



# Клас HashSet<T>

- Базується на моделі математичних множин та надає високопродуктивний набір операцій, подібних до доступу до ключів колекцій Dictionary<TKey,TValue> чи Hashtable.
  - Спрощено клас HashSet<T> можна розглядати як Dictionary<TKey,TValue> без значень (values).
  - Колекція HashSet<T> не є відсортованою та не допускає дублювання елементів.
  - Якщо порядок чи дублювання елементів важливіше за продуктивність додатку, розгляньте використання List<T> разом з методом Sort().
  - HashSet<T> постачає багато теоретико-множинних операцій, на зразок об'єднань (unions) та різниці (set subtraction):

| Операція HashSet           | Математичний еквівалент          |
|----------------------------|----------------------------------|
| <u>UnionWith</u>           | Об'єднання (Union, set addition) |
| <u>IntersectWith</u>       | Переріз (Intersection)           |
| <u>ExceptWith</u>          | Різниця (Set subtraction)        |
| <u>SymmetricExceptWith</u> | Симетрична різниця               |





```

class Program
{
 static void ShowColl(SortedSet<char> ss, string s)
 {
 Console.WriteLine(s);
 foreach (char ch in ss)
 Console.Write(ch + " ");
 Console.WriteLine("\n");
 }

 static void Main(string[] args)
 {
 // Создадим два множества
 SortedSet<char> ss = new SortedSet<char>();
 SortedSet<char> ss1 = new SortedSet<char>();

 ss.Add('A');
 ss.Add('B');
 ss.Add('C');
 ss.Add('Z');
 ShowColl(ss, "Первая коллекция: ");

 ss1.Add('X');
 ss1.Add('Y');
 ss1.Add('Z');
 ShowColl(ss1, "Вторая коллекция");
 ss.SymmetricExceptWith(ss1);
 ShowColl(ss, "Исключили разноименность (одинаковые элементы) двух множеств: ");
 ss.UnionWith(ss1);
 ShowColl(ss, "Объединение множеств: ");
 ss.ExceptWith(ss1);
 ShowColl(ss, "Вычитание множеств");
 Console.ReadLine();
 }
}

```

## Клас HashSet<T>

```

F:\csbc-github\oop-theory-repo\TimeStamp\HashSetOfTDemo\bin\Debug\netcoreapp
Первая коллекция:
A B C Z

Вторая коллекция
X Y Z

Исключили разноименность (одинаковые элементы) двух множеств:
A B C X Y

Объединение множеств:
A B C X Y Z

Вычитание множеств
A B C

```

# Відсортовані множини (клас SortedSet<T>)

---

- SortedSet – узагальнена відсортована колекція унікальних об'єктів з простору імен System.Collections.Generic.
  - Також постачає багато математичних операцій над множинами.
  - Це динамічна колекція (автоматично збільшується при додаванні нових елементів).
- Важливі риси:
  - SortedSet не включає хешування, тобто виконує лінійний пошук, значно повільніший, ніж у HashSet.
  - Всередині SortedSet реалізований як бінарне дерево з вузлом Root (коренем) та вузлами Left і Right для кожного вузла.
  - Кожний вузол необхідно алокувати з керованої кучі.
  - Кожна операція вставки розташовує новий елемент у відсортованому порядку. Тому складність  $O(\log n)$ .
  - SortedSet<T> повинен виконувати бінарний пошук, щоб знайти коректне місце для нового елементу.

|                             | List          | HashSet       | SortedSet   |
|-----------------------------|---------------|---------------|-------------|
| Iteration                   | $O(n)$        | $O(n)$        | $O(n)$      |
| Search                      | $O(n)$        | $O(1)$        | $O(n)$      |
| Add                         | $O(n)$        | $O(1)$        | $O(\log n)$ |
| Remove                      | $O(n)$        | $O(1)$        | $O(\log n)$ |
| Enumerating in Sorted Order | $O(n \log n)$ | $O(n \log n)$ | $O(n)$      |
| Allow Duplicates            | YES           | NO            | NO          |

```

class Program
{
 static void Main(string[] args)
 {
 // Create sorted set.
 SortedSet<string> set = new SortedSet<string>();
 bool a = set.Add("sam");
 bool b = set.Add("rally");
 bool c = set.Add("landra");
 bool d = set.Add("steve");
 bool e = set.Add("mark");
 bool f = set.Add("mark");

 // Remove all elements where first letter is "s".
 set.RemoveWhere(element => element.StartsWith("s"));

 // Display.
 foreach (string val in set) { Console.WriteLine(val); }

 Console.WriteLine($"Insertion: {a} {b} {c} {d} {e} {f}");

 List<string> list = new List<string>();
 list.Add("ark");
 list.Add("mark");
 list.Add("marka");

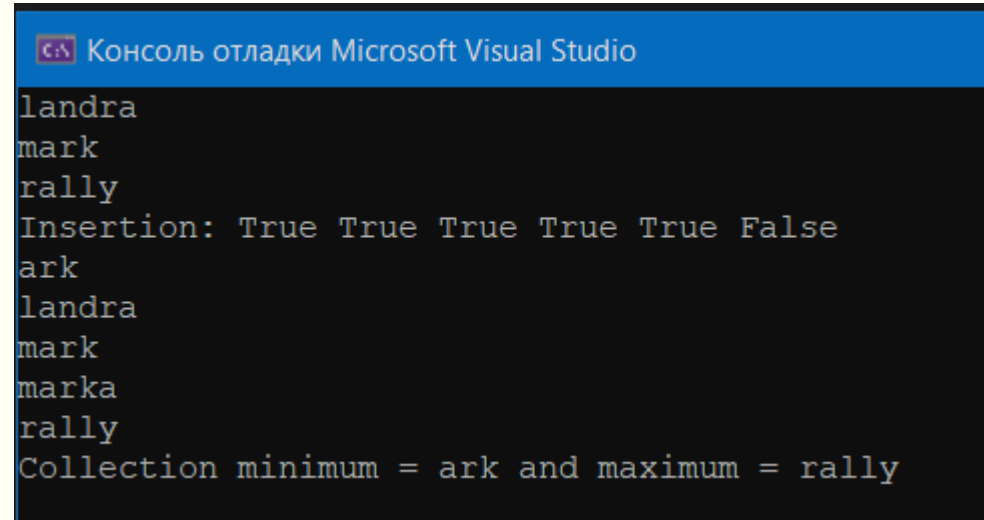
 // Union the two collections.
 set.UnionWith(list);

 // Enumerate.
 foreach (string val in set) { Console.WriteLine(val); }

 Console.WriteLine($"Collection minimum = {set.Min} and maximum = {set.Max}");
 }
}

```

## Відсортовані множини (клас SortedSet<T>)



```

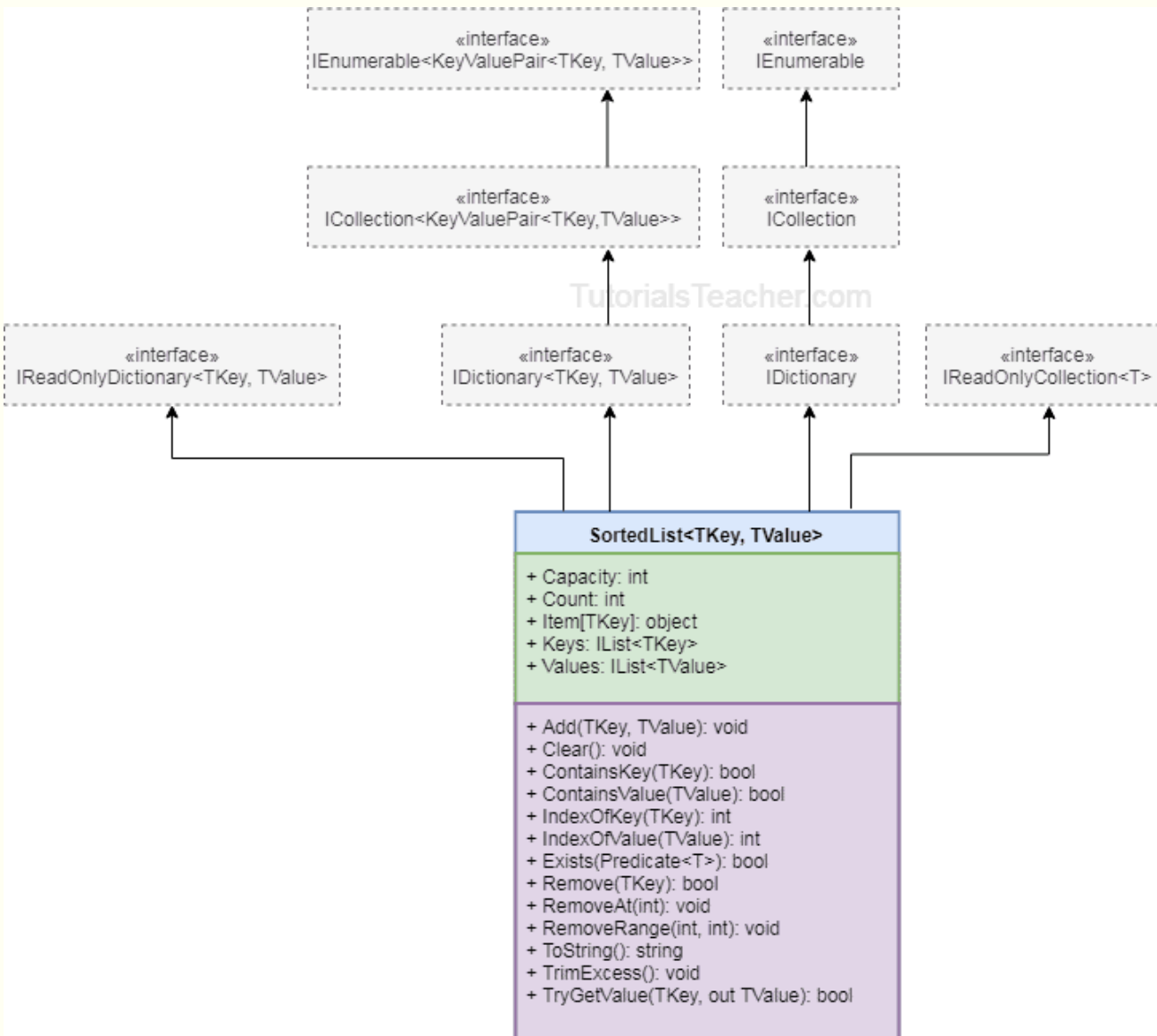
Консоль отладки Microsoft Visual Studio

landra
mark
rally
Insertion: True True True True True False
ark
landra
mark
marka
rally
Collection minimum = ark and maximum = rally

```

### ■ Повне демо

# Класи SortedList / SortedList<TKey, TValue>



- Основні риси класу SortedList<TKey, TValue>:
  - Масив пар «ключ-значення», відсортованих за ключем.
  - Сортує елементи при їх вставці: ключі примітивного типу та ключі-об'єкти сортуються на основі інтерфейсу IComparer<T>.
  - Ключ повинен бути унікальним та не бути null.
  - Значення може бути null або повторюватись.
  - Доступ до значення можливий шляхом передачі відповідного ключа в індексатор: sortedList[key]
  - Містить елементи типу KeyValuePair<TKey, TValue>
  - Використовує менше пам'яті, ніж SortedDictionary<TKey, TValue>.
  - Швидший за SortedDictionary<TKey, TValue> в отриманні даних після сортування, проте програє у швидкості вставки та видалення пар «ключ-значення».

# Використання SortedList<TKey, TValue>

```
class Program
{
 static void Main(string[] args)
 {
 SortedList<int, string> numberNames = new SortedList<int, string>()
 {
 {3, "Three"},
 {1, "One"},
 {2, "Two"}
 };

 Console.WriteLine(numberNames[1]); //output: One
 Console.WriteLine(numberNames[2]); //output: Two
 Console.WriteLine(numberNames[3]); //output: Three
 //Console.WriteLine(numberNames[10]); //run-time KeyNotFoundException

 numberNames[2] = "TWO"; //updates value
 numberNames[4] = "Four"; //adds a new key-value if a key does not exists

 Console.WriteLine();
 for (int i = 0; i < numberNames.Count; i++)
 Console.WriteLine("key: {0}, value: {1}", numberNames.Keys[i], numberNames.Values[i]);

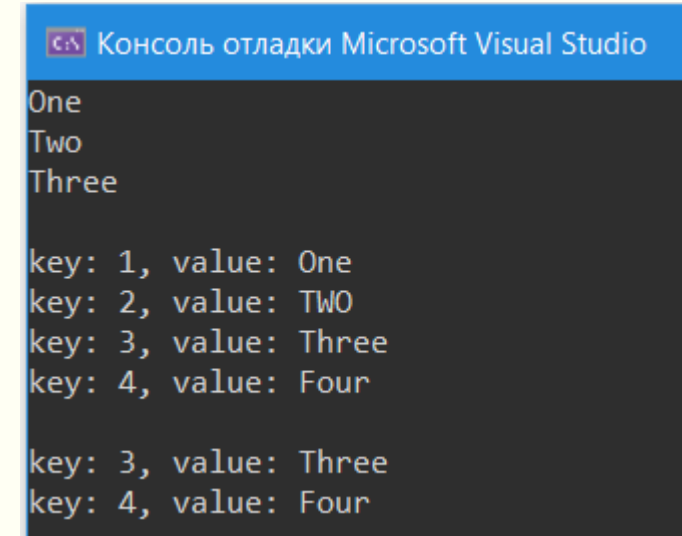
 numberNames.Remove(1); //removes key 1 pair
 numberNames.Remove(10); //removes key 1 pair, no error if not exists

 numberNames.RemoveAt(0); //removes key-value pair from index 0
 //numberNames.RemoveAt(10); //run-time exception: ArgumentOutOfRangeException

 Console.WriteLine();
 foreach (var kvp in numberNames)
 Console.WriteLine("key: {0}, value: {1}", kvp.Key, kvp.Value);
 }
}
```

17.11.2020

@Марченко С.В., ЧДБК, 2020



```
Консоль отладки Microsoft Visual Studio

One
Two
Three

key: 1, value: One
key: 2, value: TWO
key: 3, value: Three
key: 4, value: Four

key: 3, value: Three
key: 4, value: Four
```

## Клас SortedDictionary<TKey, TValue>

---

- Реалізує інтерфейси IDictionary, IDictionary<TKey, TValue>, ICollection, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable и IEnumerable<KeyValuePair<TKey, TValue>>.
- Надає конструктори:
  - public SortedDictionary()
  - public SortedDictionary(IDictionary<TKey, TValue> dictionary)
  - public SortedDictionary(IComparer<TKey> comparer)
  - public SortedDictionary(IDictionary<TKey, TValue> dictionary, IComparer<TKey> comparer)
- Перший конструктор створює порожній словник, другий – словник з указаною кількістю елементів dictionary.
  - Третій конструктор допускає вказівку способу порівняння за допомогою об'єкта comparer, що буде використаний для сортування.
  - Четвертий конструктор ініціалізує словник із вказівкою способу порівняння.

# Клас SortedDictionary<TKey, TValue>

---

- Найчастіше використовуються методи:
  - Add(): Додає в словар пару "ключ-значення", визначену параметрами key і value. Якщо ключ key вже знаходиться в словарі, то його значення не змінюється, і генерується виключення ArgumentException
  - ContainsKey(): Повертає логічне значення true, якщо викликаючий словар містить об'єкт key як ключ; в протилежному випадку — логічне значення false
  - ContainsValue(): Повертає логічне значення true, якщо викликаючий словар містить значення value, в протилежному випадку — логічне значення false
  - Remove(): Видаляє ключ key з словаря. При успішному результаті операції повертається логічне значення true, а якщо ключ key відсутній в словарі — логічне значення false
- Слід мати на увазі, що ключі і значення, що містяться в колекції, доступні окремими списками за допомогою властивостей Keys і Values.
  - В колекціях типу SortedDictionary<TKey, TValue>.KeyCollection і SortedDictionary<TKey, TValue>.ValueCollection реалізуються як загальні, так і спеціалізовані форми інтерфейсів ICollection і IEnumerable.
- І нарешті, в класі SortedDictionary<TKey, TValue> реалізується наведений нижче індикатор, визначений в інтерфейсі IDictionary<TKey, TValue>:
  - public TValue this[TKey key] { get; set; }
  - Цей індикатор служить для отримання і встановлення значення елемента колекції, а також для додавання в колекцію нового елемента. Але в даному випадку в якості індексу служить ключ елемента, а не сам індекс.

# Клас SortedDictionary<TKey, TValue>

```
class UserInfo {
 public static SortedDictionary<string, string> MyDic(int i)
 {
 SortedDictionary<string, string> dic = new SortedDictionary<string, string>();
 string s, s1;
 for (int j = 0; j < i; j++) {
 Console.WriteLine("\nВведіть ключ: ");
 s1 = Console.ReadLine();
 Console.WriteLine("Введіть ім'я працівника");
 Console.WriteLine("Name{0} --> ", j);
 s = Console.ReadLine();
 dic.Add(s1, s);
 }
 return dic;
 }
}

class Program {
 static void Main() {
 Console.WriteLine("Скільки працівників додати? ");
 try {
 int i = int.Parse(Console.ReadLine());
 SortedDictionary<string, string> dic = UserInfo.MyDic(i);

 ICollection<string> keys = dic.Keys;

 Console.WriteLine("\nВідсортована база даних містить: ");
 foreach (string j in keys)
 Console.WriteLine("ID -> {0} Name -> {1}", j, dic[j]);
 }
 catch (FormatException) {
 Console.WriteLine("Некоректне введення!");
 }
 Console.ReadLine();
 }
}
```

F:\csbc-github\oop-theory-repo\TimeSta

Скільки працівників додати? 4

Введіть ключ: D

Введіть ім'я працівника

Name0 --> Dmytro

Введіть ключ: O

Введіть ім'я працівника

Name1 --> Olena

Введіть ключ: A

Введіть ім'я працівника

Name2 --> Alex

Введіть ключ: J

Введіть ім'я працівника

Name3 --> John

Відсортована база даних містить:

ID -> A Name -> Alex

ID -> D Name -> Dmytro

ID -> J Name -> John

ID -> O Name -> Olena



## Загальне порівняння

| Operation            | Dictionary<K,V>  | SortedDictionary<K,V> | SortedList<K,V>       |
|----------------------|------------------|-----------------------|-----------------------|
| this[key]            | $O(1)$           | $O(\log n)$           | $O(\log n)$ or $O(n)$ |
| Add(key, value)      | $O(1)$ or $O(n)$ | $O(\log n)$           | $O(n)$                |
| Remove(key)          | $O(1)$           | $O(\log n)$           | $O(n)$                |
| ContainsKey(key)     | $O(1)$           | $O(\log n)$           | $O(\log n)$           |
| ContainsValue(value) | $O(n)$           | $O(n)$                | $O(n)$                |



---

---

# ДЯКУЮ ЗА УВАГУ!

**Наступне запитання: доступні тільки для читання та спостережувані колекції**

---

---