**Dictionary**

**Plan:**

* Dictionary nədir? Nə zaman istifadə olunur?
* Dictionary yaratmaq və element əlavə etmək;
* Dictionary-dən məlumat çəkmək və yoxlamaq;
* Elementləri silmək;
* Dictionary üzrə iterasiya;
* Qabaqcıl xüsusiyyətləri və optimizasiyası
* Dictionary – də Serialization/Deserialization (JSON, XML, File)
* Dictionary ilə bağlı müsahibə sualları

**Dictionary nədir?**

**Dictionary<TKey, TValue> C# - da açar – dəyər cütlərini saxlamaq üçün istifadə olunan kolleksiya tipidir. Bu, verilənlərə indekslə deyil, unikal açarla müraciət etməyə imkan verir.**

**Yaddaşda Heap və Stack bölmələrdə müxtəlif formalarda saxlanılır. Dictionary reference type olduğu üçün object kimi heap-da saxlanılır. Lakin içindəki TKey və TValue tiplərindən asılı olaraq məlumatlar həm stack, həm də heap-da saxlanıla bilər. Məsələn,**

**Dictionary<int, string> dict = new Dictionary<int, string>();**

**dict.Add(1, “Leyla”); - Burada, dict dəyişəni stack-da saxlanılır, Dictionary<int, string> əmri isə heap-da yeni object yaradır və stack-dakı dict ona göstərici olur.**

**Int (TKey) value type olduğu üçün stack-da saxlanılır.**

**String (TValue) reference type olduğu üçün isə heap-da saxlanılır, stack-dakı dict onun göstəricisi olur.**

**Nə zaman istifadə olunur?**

* Məlumatları tez-tez əldə etmək lazım olduqda (search və lookup əməliyyatları üçün);
* Unikal açarlar ilə əlaqəli dəyərləri saxlamaq lazım olduqda;
* Konfiqurasiya və parametrləri saxlamaq lazım olduqda;
* API və JSON formatında gələn dataları saxlamaq və işlətmək lazım olduqda.

**Dictionary vs List**

|  |  |  |
| --- | --- | --- |
| **Xüsusiyyət** | **Dictionary** | **List** |
| Müraciət metodu | Açar ilə (key) | İndeks ilə |
| Məlumat əldə etmə sürəti | O(1) çox sürətlidir | O(n) element sayl artdıqca yavaşlayır |
| Dublikatlara icazə | Açarlar unikal olduğu üçün təkrarlana bilməz. | Dublikat elementlər ola bilər |
| Tətbiq sahəsi | Key-value əsaslı saxlama və lookup əməliyyatları | Sırf indekslə işləmə |

**Dictionary yaratmaq.**

C# - da Dictionary**<TKey, TValue> istifadə edərək açar – dəyər cütləri saxlaya bilərik. TKey açar tipini, TValue isə dəyər tipini göstərir. Dictionary yaratmağın bir neçə üsulu var:**

1. Boş Dictionary yaratmaq:  
   **Dictionary<int, string> students = new Dictionary<int, string>();  
   Burada int açar, string isə dəyər tipidir.**
2. **Başlanğıc dəyərlərlə Dictionary yaratmaq:  
   var students = new Dictionary<int, string>()  
   {  
    {1, “Leyla”},  
    {2, “Həsən”},  
    {3, “Aysel”}  
   };  
   Burada var istifadə edərək Dictionary<int, string> avtomatik təyin edilir.**
3. **Dictionary üçün KeyValuePair istifadə etmək:  
   var students = new Dictionary<int, string>()  
   {  
    new KeyValuePair(1, “Leyla”),  
    new KeyValuePair(2, “Həsən”)  
   };  
   Bu üsul da eyni işi görür, lakin KeyValuePair<TKey, TValue> obyektindən istifadə edir. Daha səliqəli kod yazmağa imkan verir, açar və dəyəri aydın şəkildə oxumağa kömək edir. Əgər həm açar, həm də dəyər qaytarılmalıdırsa, KeyValuePair yaxşı seçimdir. Ondan List və ya başqa kolleksiyalarda da açar – dəyər tipli data saxlamaq üçün istifadə etmək olar.  
     
   Dictionary vs KeyValuePair**

|  |  |
| --- | --- |
| Bütün açar – dəyər cütlərini saxlayır. | Bir açar – dəyər cütü saxlayır. |
| Add, Remove, ContainsKey kimi metodları var. | Yalnız key və value xassələri var. |
| Açarlar unikal olmalıdır. | Tək bir cütü saxladığı üçün belə məhdudiyyəti yoxdur. |
| Performans üçün optimallaşdırılmışdır | Sadəcə məlumat struktur kimi istifadə edilir. |

Dictionary-ə elemet əlavə etmək üçün:

1. Add metodu:  
   **students.Add(4, “Orxan”);  
   students.Add(5, “Nİgar”);**Add metodu eyni açarla təkrar element əlavə etməyə imkan vermir və əks halda, ArgumentException qaytarır.
2. [ ] operatoru ilə əlavə etmək, yaxud dəyəri yeniləmək:  
   **students[6] = “Elvin” –** Yeni dəyər əlavə edir;  
   **students[3] = “Lalə”** – mövcud açarın dəyərini yeniləyir

Dictionary – də elementlərin sayını bilmək üçün Count istifadə olunur:  
**students.Count;**

**Dictionary – dən məlumat almaq və ya yoxlamaq.**

1. Açara görə dəyəri götürmək ([ ] ilə):  
   Ən sadə üsul açarın dəyərini indeks operatoru ([ ]) vasitəsilə əldə etməkdir.  
   **Console.WriteLine(students[1]); //”Leyla”**Əgər açar mövcud deyilsə, KeyNotFoundException xətası qaytaracaq.
2. Açarın mövcudluğunu yoxlamaq:  
   Bunun üçün ContainsKey istifadə olunur.  
   **if(students.ContainsKey(2))  
   {** **Console.WriteLine(“Key exists!”);  
   }  
   else  
   {  
    Console.WriteLine(“Key doesn’t exist!”);  
   }**Xətanın qarşısının alınması üçün indeks operatoru əvəzinə birbaşa ContainsKey daha effektivdir.
3. Dəyərin mövcudluğunu yoxlamaq:  
   Bunun üçün də ContainsValue işlədilir.  
   **if(students.ContainsValue(“Elvin”);  
   {  
    Console.WriteLine(“Value exists!”);  
   }**ContainsValue ContainsKey-ə nisbətən yavaşdır. Çünki, bütün dəyərləri yoxlamalıdır.
4. Təhlükəsiz dəyər almaq:  
   Əgər açar varsa, dəyəri qaytarsın, yoxdursa, xətaya səbəb olmasın deyə TryGetValue var.  
   **if(students.TryGetValue(2, out string name)  
   {  
    Console.WriteLine(“Found!”);  
   }  
   else  
   {  
    Console.WriteLine(“Not Found!”);  
   }**

**Elementləri silmək.**

1. Remove metodu:  
   Bir acara uygun dəyəri və açarı silir. Geriyə bool dəyər qaytarır.  
   **bool isRemoved = students.Remove(2);**Əgər mövcuddursa, silir və true qaytarır. Açar yoxdursa, false qaytarır.
2. Bütün Dictionary-ni təmizləmək:  
   Bunun üçün Clear() metodur istifadə olunur.  
   **students.Clear();**Sürətli və effektivdir.
3. Şərtə əsasən elementləri silmək:  
   **var keysToRemove = students.Where( x => x.Value == “Orxan”).Select(x => x.Key).ToList()$  
   foreach(var key in keysToRemove)  
   {  
    students.Remove(key);  
   };**ToList etməsək, Where birbaşa Dictionary – dən iterasiya edərkən, CollectionModifiedException qaytaracaq.

**Dictionary üzrə iterasiya.**

1. foreach ilə iterasiya (ən optimal):  
   **foreach(KeyValuePair<int, string> student in students)**Sadə və optimallaşdırılmış üsuldur.
2. foreach ilə key və dəyərlər üzərindən keçmək:  
   Təkcə açarları/dəyərləri oxumaq lazımdırsa, Keys və ya Values kolleksiyalarından istifadə edə bilərik.  
   **foreach(var key in students.Keys)  
   foreach(var value in students.Value)**Bu göstərilənər (var student in students) - ə nisbətən, yaddaşda daha az yer tutur. Bütün obyektləri götürməyə ehtiyac qalmır.
3. for dövrü ilə:  
   for dövrü Dictionary ilə birbaşa işləyə bilmədiyi üçün ElementAt() istifadə olunur.  
   **for(int i = 0; i < students.Count; i++)  
   {  
    var item = students.ElementAt(i);  
   };**Bu üsul heç də məsləhətli deyil. Çünki ElementAt() O(n) vaxt tələb edir, for dövrü O(n2) olur.
4. LINQ ilə iterasiya:  
   Dictionary – də LINQ sorğularını istifadə etməklə xüsusi şərtlərə görə iterasiya mümkündür.  
   **var filtered = students.Where(s => s.Key > 1).ToList();  
   foreach(var item in filtered)**şərtə görə filter etsə də, böyük Dictionary – lərdə çox Where istifadəsi performansa təsir edə bilər.

**Qabaqcıl xüsusiyyətləri və Optimizasiyası.**

Dictionary performans baxımından çox ekkeftiv bir kolleksiyadır. Onun arxasında hashing və hash table strukturu dayanır.

1. Əməliyyatların Big – O analizi

|  |  |  |  |
| --- | --- | --- | --- |
| **Əməliyyat** | **Orta hal** | **Pish al** | **İzah** |
| Add(key, value) | O(1) | O(n) | Çox vaxt 1 olur, lakin hash conflict olarsa (collison) O(n) ola bilər. |
| Remove(key) | O(1) | O(n) | Əsasən, 1 olur. Collison səbəbilə n ola bilər. |
| TryGetValue(key, out type value) | O(1) | O(n) | Ən optimaldır. |
| Keys və ya Values | O(n) | O(n) | Hamısı oxunur. |
| Clear() | O(n) | O(n) | Bütün elementləri nəzərə alır. |
| foreach iterasiya | O(n) | O(n) | Hər element bir dəfə oxunur. |

1. Collison problem:  
   Dictionary açarları hash funksiyası ilə saxlayır. Əgər iki açarın hash dəyəri eyni olacaqsa, çaxnaşma baş verir. Hash çaxnaşmaları çox olarsa, dictionary sürətini itirə bilər və bəzi əməliyyatlar O(n) - ə qədər yavaşlaya bilər. Məsələn,  
   **students.Add(7, “Natiq”); - Hash(7) => 1007  
   students.Add(8, “Oqtay”); - Hash(8) => 1008  
   students.Add(18, “Amaliya”); - Hash(18) => 1008**Bu zaman 8 və 18 çaxnaşır.
2. Performansı yaxşılaşdırmaq:  
   3.1. Mümkün olduğu qədər unikal açar istifadəsi.   
   Əgər int/Guid olarsa, max fərqli olsun. Dəyər kimi string olsa, böyük verilənlərdə performans aşağı düşə bilər.  
     
   3.2. Dictionary – ni tıqribi ölçü ilə yaratmaq.   
   Əgər çoxlu element əlavə olunacaqsa, Capacity təyin etmək rehash qarşısını alır.   
   **Dictionary<int, string> dict = new Dictionary<int, string>(10000);**Əgər Capacity əvvəlcədən təyin olarsa, daxilindəki array tez – tez böyüməyə məcbur qalmayacaq, nəticədə performans stabil qalacaq.  
     
   3.3. TryeGetValue istifadəsi.   
   Daha sürətlidir, birbaşa açarl yoxlayır və götürür.  
     
   3.4. Search istifadəsi (for yox).   
   İndeks əsaslı olmamasına görə foreach daha effektivdir.
3. Thread – Safe alternativləri.  
   C# - da Dictionary<TKey, TValue> default olaraq thread – safe deyil. Yəni bir neçə thread eyni anda eyni Dictionary üzərində əməliyyat apararsa, məlumat pozula və xəta baş verə bilər. Bunun qarşısını almaq üçün bir neçə alternativlər istifadə oluna bilər.  
     
   4.1. ConcurrentDictionary<TKey, TValue>.  
   Ən yaxşı seçimdir, multiuse üçün uyğundur. Eyni anda bir neçə thread rahatlıqla yazma, oxuma və silmə apara bilər. Performansı optimallaşdırılıb, Dictionary – dən daha yaxşı işləyir.  
     
   4.2. lock istifadəsi.  
   **Dictionary<int, string> dict = new Dictionary<int, string>();  
   object \_lock = new Object();  
   Parallel.For(0, 1000, i => {  
    lock( \_lock)  
    {  
    dict[i] = I;  
    }  
   });  
   Console.WriteLine($”Element count: {dict.Count})** //Həmişə 1000 olacaq.  
   Advantages: Yalnız bir thread – ın eyni anda dictionary üzərində işləməsinə icazə verir.   
   Disadvantages: Performansı azaldır, parallellism düşür.  
     
   4.3. ReaderWriterLockSlim istifadə etmək.  
   Oxuma əməliyyatları üçün daha optimaldır.  
   **...dict...**  
   **ReaderWriterLockSlim rwLock = new ReadWriterLockSlim();  
   Parallel.For(0, 1000, i =>   
    {  
    rwLock.EnterWriteLock();  
    try  
    {  
    dict[i] = i;  
    }  
    finally  
    {  
    rwLock.ExitWriteLock();  
    }  
   });**Burada element sayı həmişə 1000 olur.  
   Advantages: Oxuma və yazma əməliyyatlarını fərqli tənzimləyir. Eyni anda multhread oxuuya, amma singlethread yaza bilir.

**Dictionary – də Serialization/Deserialization.**Dictionary – dəki məlumatları fayl, JSON, XML formatına və ya şəbəkə üzərindən göndərmək üçün serialization etmək lazımdır.

1. Serialization/Deserialization. JSON.  
   Dictionary – ni JSON formatına çevirmək üçün System.Text.Json kitabxanası lazımdır. Bayaqkı misallarda işlətdiyimiz students dictionary – sini JSON formatına çevirək.  
   **string json = Json.Serializer.Serialize(students);**Result: **“{“1” : “Leyla”, “2” : “Həsən”, “3” : “Lalə”, “4” : “Orxan”, “5” : “Nigar”, “6” : “Elvin”, “7” : “Natiq”, “8” : “Oqtay”, “18” : “Amaliya”}”**API – lərdə geniş istifadə olunur.  
   Deserialization üçün aşağıdakı alqoritmi tətbiq edə bilərik:  
   **string json = {\“1\”:\”Alma\”, \”2\” : \”Armud\”};  
   Dictionary<int, string> fruits = Json.Serializer.Deserialize<Dictionary<int, string>>(json);**
2. Serialization/Deserialization. XML:  
   **[XLMRoot(“Dictionary”)]  
   public class SerializableDictionary<TKey, TValue>   
   {  
    [XMLArray(“Items”)]  
    [XMLArrayItem(“Item”)]  
    public List<KeyValuePair<TKey, Tvalue>** **Items { get; set } = new();  
    public SerializableDictoinary()  
    {  
    }  
    public SerializableDictoinary(Dictionary<TKey, Tvalue> dictionary)  
    {  
    Items = new List<KeyValuePair<TKey, Tvalue>>(dictionary);  
    }  
    public Dictionary<TKey, Tvalue> ToDictionary()  
    {  
    return new Dictionary<TKey, TValue>(Items);  
    }  
   }  
     
   Dictionary<int, string> fruits = new Dictionary<int, string>  
   {  
    {1, “Alma”},  
    {2, “Armud”},  
    {3, “Heyva”}  
   };**//XML converting  
   **XMLSerializer serializer = new XMLSerializer(typeof(SerializableDictionary<int, string>));  
   using (StringWriter writer = new StringWriter())  
   {  
    Serializer.Serialize(writer, new SerializableDictionary<int, string>(fruits));  
   }**//Result  
   **<Dictionary>  
    <Items>  
    <Item>  
    <Key> 1 </Key>  
    <Value> Alma </Value>  
    </Item>  
    <Item>  
    <Key> 2 </Key>  
    <Value> Armud </Value>  
    </Item>  
    <Item>  
    <Key> 3 </Key>  
    <Value> Heyva </Value>  
    </Item>  
    </Items>  
   </Dictionary>**Deserialization yazmaq üçün fərz edək ki, string xmlData yuxarıdakı nəticəyə bərabərdir.  
   **var serializer = new XMLSerializer(typeof(SerializableDictionary<int, string>));  
   using (StringReader reader = new StringReader(xmlData))  
   {  
    SerializableDictionary<int, string> deserializedFruits = (SerializableDictionary<int, string>) serializer.Deserialize(reader);  
   }  
   Dictionary<int, string> fruits = deserializedFruits.ToDictionary();**
3. Serialization/Deserialization. Binary:  
   Əgər məlumatı faylda və ya şəbəkədən ötürmək lazımdırsa, BinaryFormatter istifadə olunur. System.Runtime.Serialization.Formatters.Binary;  
   **[Serializable]  
   public class SerializableDictionary<TKey, TValue> Dictionary<TKey, TValue>{}  
   var fruits = new Dictionary<int, string>()  
   {  
    {1, “Alma”},  
    {2, “Armud”}  
   };**//Convert to Binay file  
   **using (FileStream fs = new FileStream(“dictionary.dat”, FileMode.Create))  
   {  
    BinaryFormatter formatter = new BinaryFormatter();  
    formatter.Serialize(fs, fruits);  
   }**Deserialize  
   **SerializableDictionary<int, string> loadedFruits;  
   using(FileStream fs = new FileStream(“dictionary.dat”, FileMode.Open))  
   {  
    BinaryFormatter formatter = new BinaryFormatter();  
    loadedFruits = (SerializableDictionary<int, string>) formatter.Deserialize(fs);  
   }**.NET 5+ versiyalarında safety səbəbilə BinaryFormatter məsləhət görülmür.