**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.ToString();  
    }  
   });  
   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 ReaderWriterLockSlim();  
   Parallel.For(0, 1000, i =>   
    {  
    rwLock.EnterWriteLock();  
    try  
    {  
    dict[i] = i.ToString();  
    }  
    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.

**Müsahibə sualları**

1. Dictionary<TKey, TValue> nədir, necə işləyir?
2. Dictionary – də ContainsKey və ContainsValue fərqləri.
3. Dictionary vs List.
4. Dictionary – də elementə necə əlçatanlıq olur?
5. Dictionary – də key üçün hansı tipər istifadə olunur?
6. Dictionary yaddaşda necə saxlanılır?
7. Dictionary – nin key, value – ləri harada saxlanılır?
8. Reference və value tipləri Dictionary – də necə fərqli işləyirlər?
9. KeyValuePair nədir və nə üçün istifadə olunur?
10. DictionaryEntry vs KeyValuePair.
11. foreach ilə Dictionary – də iterasiya edərkən geriyə nə döndərir?
12. Dictionary performansı necə təmin edir və hashtable ilə necə müqayisə olunur?
13. Dictionary – də GetHashCode() və Equals().
14. Rehashing nə zaman baş verir?
15. Capacity vs Count.
16. Serialization/Deserialization JSON.
17. Serialization/Deserialization XML.
18. Serialization/Deserialization Binary.
19. Dictionary nə üçün thread safe deyil?
20. Multithreading alternativləri.
21. ConcurrentDictionary nədir?
22. Lock ilə necə safe edə bilərik?
23. ReaderWriterLockSlim nə zaman istifadə olunur?
24. Dictionary<TKey, TValue> vs SortedDictionary<TKey, TValue>.
25. Dictionary – də custom comporer necə təyin olunur?
26. Dictionary – də IEqualityComparer<TKey> istifadəsi.
27. Dictionary – də Lazy Initialization necə baş verir?
28. Dictionary – də Default Value necə təyin olunur?
29. Dictionary açar-dəyər əlaqəli data saxlamaq üçün bir kolleksiyadır. TKey Açarı, TValue isə dəyərin tipini göstərir. Əsasən, search və lookup əməliyyatları üçün istifadə olunur. Açarları unikal dəyər almalıdır.
30. ContainsKey dictionarydə müəyyən açarın, ContainsValue isə dəyərin mövcudluğunu yoxlayır və uyğun olaraq bool dəyər (true/false) qaytarır. Hər ikisi exception olmasının qarşısını alır. ContainsValue() performans baxımından zəifdir, çünki O(n) zaman kompleksliyi ilə işləyir. ContainsKey() isə O(1) və ya O(log n) olur (GetHashCode()-un keyfiyyətindən asılıdır).
31. List indeksləmə əsasında işləyir, dictionary isə açar. Listdə təkrarlanma olması mümkündür, dictionarydə isə açarların unikallıq məhdudiyyəti duplikatların qarşısını alır. Dictionary daha böyük və geniş anlayışdır. Dictionary əsasən sürətli lookup üçündür (O(1) və ya O(log n)), amma List O(n) zaman kompleksliyi ilə axtarış edir.
32. Dictionary-də elementləri bir neçə üsulla yoxlamaq və gətirmək olar.  
    A. [] ilə. Bu zaman açar bildirilir və onun uyğun dəyəri geri qaytarılır. Lakin, açar mövcud deyilsə, geriyə exception qaytarır. Məsələn;   
    var employees = new Dictionary<int, string>()  
    {  
     {1, "Kamal"},   
    {2, "Humay"},   
    {3, Nuray"}   
    };   
    employees[2] //Humay dəyərinə bərabərdir.   
    B. TryGetValue istifadə edə bilərik. Bu verilən açarın mövcud olduğu halda uyğun dəyəri qaytarır, olmadıqda isə geriyə exception atmır. Bayaqkı dictionary əsasında, baxsaq: employees.TryGetValue(3, out string name) - Bu zaman dəyər Nuray qaytarılacaq.  
    C. Keys və Values kolleksiyaları istifadə oluna bilər. Məsələn,   
    var keys = employees.Keys.ToList(); Burada açarlar listə mənimsədilir (1,2,3)   
    var values = employees.Values.ToList(); Burada isə dəyərlər listə mənimsədilir. ("Kamal", "Humay", "Nuray")   
    D. LINQ sorğuları ilə data yoxlayıb gətirə bilərik, bu zaman müəyyən şərtlər də bildirmək olar. Məsələn,   
    employees.Where(x => x.Key > 1).Select(x => x.Value).ToList(); Burada açar dəyəri 1-dən böyük olan dəyərlər gələcək (Humay və Nuray). ToList() çağırmaq lazımdırsa, bu deferred execution (təxirə salınmış icra) ilə əlaqəlidir. Əgər Where() birbaşa istifadə edilsə və sonra FirstOrDefault() çağırılsa, boş nəticəyə null reference səhvi verə bilər.
33. Hər bir tipi yazmaq olar həm key həm də value kimi. Amma Key üçün GetHashCode() və Equals() metodları düzgün işləməlidir.
34. Dictionary-nin yaddaşda saxlanması, onun tərkibindən çox asılıdır. Məsələn, yuxarıdakı nümunədə employees dəyişəni Stackda saxlanılır və heap-də isə Dictionary<int, string> tipində obyekt yaradılır və bu employees dəyişəni onun üçün göstərici olur, referans edir. Bu nümunədə açar tipi primitiv (int) olduğundan birbaşa stackda, dəyər tipi(string) isə reference hərəkət edir. Yenə də name dəyişəni stackda yaranır və dictionary üçün göstərici olur. Bu nümunədən başqa, açar və dəyər tiplərinə uyğun yadda saxlanılır. Hər iksi primitiv, hər iksi referans tiplər ola bilər. 1ci halda iksi də stackda, 2ci halda isə qeyd etdiyim heap qaydasinda saxlanılacaq. Açar (Key) tipi primitiv (məsələn, int) olsa belə, Dictionary-də internal struktura görə heap-də saxlanıla bilər. Çünki Dictionary, açarların GetHashCode() və Equals() metodlarını çağırır. Bu isə hash table-ın yaddaşda necə yerləşəcəyini müəyyən edir. Dəyər (Value) referans tipdirsə (string, class və s.), heap-də saxlanılır.
35. Tipindən asılı olaraq müəyyən olunur. Əgər TKey və TValue hər ikisi value type (struct) olsa, onda bütün elementlər heap-də saxlanmaz, çünki Dictionary onları boks etmədən işlədə bilər. Əgər TKey və TValue hər ikisi reference type olsa, onda yalnız referanslar stack-də, obyektlərin özü isə heap-də olar.
36. Heap-də tipinə uyğun obyekt yaradılır və stackdakı dictionary dəyişəninə reference edir. Dictionary daxilində böyük bir array saxlayır. Həmin array elementləri də heap-də yerləşir. Hər bir Key-Value cütü heap-də saxlanılır, çünki Dictionary hashing üçün onları bir-birində saxlamalıdır.
37. KeyValuePair özlüyündə yalnız bir cüt açar-dəyər tipi saxlayam kolleksiyadır. Həm List kimi indekslə işləyən kolleksiyalarda, həm də dictionary tərkibində işləyə bilir. Dictionary-dın məlumat çəkərkən, yaradarkən və s istifadə edə bilir, daha oxunaqlı və clean code təmin edir. Xüsusi unikallıq məhdudiyyəti və optimallaşdırılması yoxdur, çünki yalnız 1 cüt açar-dəyər saxlayır.
38. DictionaryEntry – System.Collections namespace-i altında olan və Hashtable üçün istifadə edilən bir açar-dəyər strukturudur. Yəni, Dictionary<TKey, TValue> üçün yox, daha köhnə Hashtable üçün işlədilir. KeyValuePair isə generikdir və daha geniş istifadə olunur.  
    Dictionary və keyvaluepair arasında fərq:   
    A. Dictionary çoxlu sayda açar-dəyər cütlükləri saxlayır, keyvaluepair isə ancaq 1 cüt.   
    B. Dictionary-də açar unikallıq məhdudiyyəti var, digərində yoxdur.   
    C. Dictionary-də optimizasiya üçün müxtəlif metodlar var, o birində yoxdur.   
    D. Dictionary daha geniş anlayışdır, Keyvaluepair isə özü müxtəlif kolleksiyalarda member kimi istifadə olunur.
39. foreach dövrü ilə iterasiya zamanı value dəyərlər qaytarılır. Əlbəttə sırf dictionary.Keys çoxluğuna da müraciət edə bilərik.
40. Dictionary singlethread ilə işləyir, açar dəyərləri hashlayir. Amma hashtable nəzərən daha çevik və optimaldır. Iakin, collison baş verə bilir; belə olduqda ən pis halda performans O(n) düşə bilir. Hash table isə artıq istifadəyə yararlı sayılmır, necə deyərlər old sayılır.
41. Equals açarın unikal olub-olmadığını yoxlamaq üçün istifadə olunur. GetHashCode isə həmin açarın hash kodunu qaytarır ki, Dictionary onun yerini düzgün tapa bilsin.
42. Çaxnaşma baş verər və performans düşər. Çünki açar dəyər üçün unikallıq məhdudiyyəti vardır.
43. Capacity əvvəlcədən təyin olunmazsa, Dictionary özünü avtomatik genişləndirir. Bu genişlənmə zamanı yeni bir array yaradılır və köhnə elementlər ora köçürülür. Bu proses əlavə performans xərci yaradır. Əgər Dictionary-də çoxlu element olacaqsa, əvvəlcədən Capacity təyin etmək daha optimaldır. Capacity həmişə müəyyən bir həcm olur, məsələn 4, 8, 16 və s. (2-nin qüvvəti şəklində artır). Amma Count isə faktiki daxil edilən elementlərin sayını göstərir. Capacity > Count olur, çünki həmişə müəyyən boşluq saxlanılır ki, performans yaxşı olsun
44. SON serializasiya üçün əsas metodlar JsonConvert.SerializeObject() və JsonConvert.DeserializeObject<T>()-dir (Newtonsoft.Json üçün). .NET-in özündə isə System.Text.Json namespace-i ilə işləmək olur: JsonSerializer.Serialize() və JsonSerializer.Deserialize<T>().
45. SerializableDictionary yaratmaq olur. Amma əslində Dictionary birbaşa JSON-a serialize olur, amma XML üçün bu mümkün deyil, çünki Dictionary IXmlSerializable implement etmir. Ona görə XML-də bunu əllə yazmaq lazımdır.
46. BinaryFormatter istifadə edərək dictionary – ni file – a və ya şəbəkə üzərindən ötürməyə uyğun formata çevirə bilərik. Lakin, .Net 5 + versiyalarda BinaryFormatter məsləhətli üsul hesab edilmir.
47. Çünki, dictionary yalniz single thread ilə işləyə bilir. Eyni zamanda multithread yazıb oxuma əməliyyatları aparmaq istəsə, xəta baş verəcək. Bunu üçün bir neçə alternativlər mövcuddur. Dictionary-də eyni anda bir neçə thread Add, Remove, və ya Get kimi əməliyyatlar aparsa, racecondition və ya InvalidOperationException baş verə bilər. Ona görə thread-safe deyil.
48. ConcurrentDictionary, ReaderWriterLockSlim, lock, ImmutableDictionary.
49. ConcurrentDictionary<TKey, TValue> thread safety üçün ən optimal alternativdir. Multithreading ilə rahat işləyir.  AddOrUpdate, TryAdd, TryRemove, GetOrAdd kimi metodlar sayəsində paralel əməliyyatlar üçün çox uyğundur. Performansı da yüksəkdir.
50. private static readonly object \_lock = new object();  
    private Dictionary<int, string> \_myDict = new();  
    public void SafeAdd(int key, string value)  
    {  
     lock(\_lock)  
     {  
     myDict[key] = value;  
     }  
    }  
    Bu üsulda bütün oxuma və yazma əməliyyatları lock daxilində edilir. Bu isə single-threaded access təmin edir, amma performans baxımından zəif ola bilər, çünki bütün thread-lər növbəyə dayanır.
51. Əgər çoxlu sayda oxuma əməliyyatları, lakin az sayda yazma əməliyyatları varsa, ReaderWriterLockSlim daha effektivdir. Bu struktur eyni anda çoxlu sayda oxumağa icazə verir, amma yazmaq üçün tək thread girişi təmin edir.
52. Qısa müqayisə:

* Dictionary: Məlumatlar random order-da saxlanılır, çox sürətlidir (O(1) vaxtda axtarış).
* SortedDictionary: Məlumatlar sorted order-da saxlanılır (məsələn, int keys üçün sıralanır), axtarış O(log n) vaxt aparır, çünki Balanced Tree strukturu istifadə edir.

1. Dictionary constructor-u vasitəsilə:  
   var myDict = new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase);  
   Yəni IEqualityComparer<TKey> interfeysini implement edən obyekt verilir.
2. Yuxarıdakının davamı kimi:  
   IEqualityComparer<TKey> bizə custom key müqayisəsi imkanı verir. Məsələn, case-insensitive, trimmed string, və ya hətta complex object müqayisəsi üçün istifadə olunur.
3. Əgər açar yoxdursa, onu on-the-fly əlavə etmək üçün GetOrAdd metodu yoxdur (ancaq ConcurrentDictionary-də var). Amma belə yaza bilərik:  
   if (!dict.TryGetValue(key, out var value))  
   {  
    value = new MyType();  
    dict[key] = value;  
   }  
   Yəni, ilk dəfə istifadə zamanı dəyəri yaratmaq və dictionary-ə əlavə etmək.
4. Dictionary default value saxlamır. Amma TryGetValue istifadə olunarsa və tapılmazsa, default(TValue) qaytarır:  
   dict.TryGetValue(key, out var value); // value null ola bilər reference type isə  
   Əgər istədiyin default dəyər null deyilsə, sən onu özün handle etməlisən:  
   var value = dict.ContainsKey(key) ? dict[key] : "default";