Contents

Dictionaries

Introduction

Abstract Data Type

A *dictionary*, also called a *hash*, an *associative array*, a *map*, or a *hashmap*, is a key-value store: it stores values (that can be of any type) and indexes them using a key (which is in general of a simple type, such as `int`).

Described abstractly, a dictionary) is

- a finite collection of elements,
- in *no* particular order,
- that may contain the same element multiple times.

The fact that it may contain the same element multiple times makes it different from a set, the fact that it is ordered makes it different from a multiset.

Generally, it has operations to…

- … create an empty dictionary,
- … test for emptiness,
- … insert or update a value,
- … remove a key-value pair,
- … test for existence of a key.

Possible Implementation

Using Open Addressing

A description is given at https://en.wikibooks.org/wiki/Data_Structures/Hash_Tables#Open_addressing.

```
using System;
using System.Collections.Generic;

public class CDictionary<TKey, TValue>
```

```
  where TKey : IComparable<TKey>
{
  // Two enumerated types that we will be using
  // for our implementation of Dictionary.
  public enum StatusType
  {
    Empty,
    Active,
    Deleted,
  };

  public enum CollisionRes
  {
    Linear,
    Quad,
    Double,
  };

  private class Cell
  {
    public StatusType Status { get; set; }
    public TValue Value { get; set; }
    public TKey Key { get; set; }

    public Cell(
      TKey aKey = default(TKey),
      TValue aValue = default(TValue),
      StatusType aStatus = StatusType.Empty
    )
    {
      Key = aKey;
      Value = aValue;
      Status = aStatus;
    }

    public override string ToString()
    {
      return Key + ":" + Value;
    }
  }

  // The hash table is an array of Cells,
  // and a collision strategy.
  private Cell[] table;
  private readonly CollisionRes Strategy;
```

```
public override string ToString()
{
  string returned = "";
  int i = 0;
  foreach (Cell pos in table)
  {
    returned += $"Position {i}: {pos}\n";
    i++;
  }
  return returned;
}

public CDictionary(
  int size = 31,
  CollisionRes aCollisionStrategy = CollisionRes.Double
)
{
  table = new Cell[PrimeHelper.NextPrime(size)];
  Strategy = aCollisionStrategy;
}

public void Clear()
{
  for (int i = 0; i < table.Length; i++)
    if (table[i] != null)
      table[i].Status = StatusType.Deleted; // Reuse
↪  cells by setting them to Empty
}

public void Add(TKey aKey, TValue aValue)
{
  /*
   * First, we find an empty cell (e.g. cell is null,
↪  status empty or deleted)
   * - We computer a possible index:
   *      - We first use GetHashCode() to generate a
↪  hash code,
   *      - then shift it using collisionR.
   * - We check if the cell at this index is available,
   * - If it is not, we try with the next one,
   * - If all cells are occupied, we throw an error.
   */
  int count = 0;
  int index = GetIndex(aKey, count);
  // Collision resolution
  while (
```

```
      table[index] != null
      && !table[index].Status.Equals(StatusType.Deleted)
    )
    {
      count++;
      if (count == table.Length) // If table is full,
       ↪  throw an exception.
        throw new ApplicationException("Table is full.");
      index = GetIndex(aKey, count);
    }

    if (table[index] == null) // table slot is empty (e.g.
     ↪  never been used)
      table[index] = new Cell(
        aKey,
        aValue,
        StatusType.Active
      );
    else if (
      table[index].Key.Equals(aKey) == true
      && table[index].Status == StatusType.Active
    ) // duplicate key found
      throw new ArgumentException(
        "Dictionary Error: Don't add duplicate keys: "
          + aKey.ToString()
      );
    else if (table[index].Status == StatusType.Deleted)
     ↪  // previously used item, reuse the cell
    {
      table[index].Value = aValue;
      table[index].Key = aKey;
      table[index].Status = StatusType.Active;
    }
    else
      throw new ApplicationException(
        "Something went wrong in Add method."
      );
  }

  /// <summary>
  /// Returns the data associated with the key
  /// </summary>
  /// <param name="aKey"></param>
  /// <returns>data item</returns>
  public TValue Find(TKey aKey)
  {
```

```
  // search until found or empty
  int count = 0;
  int index = GetIndex(aKey, count);
  while (
    table[index] != null
    && table[index].Status != StatusType.Deleted
    && !table[index].Key.Equals(aKey)
  )
  {
    count++;
    if (count == table.Length) // in case table is full,
     ↪  kicks out of inf loop
      throw new ApplicationException("Table is full");
    index = GetIndex(aKey, count);
  }

  if (table[index] == null)
    throw new KeyNotFoundException(
      "The key " + aKey.ToString() + " was not found"
    );
  else if (
    table[index].Status == StatusType.Active
    && table[index].Key.Equals(aKey) == true
  )
    return table[index].Value;
  else
    throw new ApplicationException(
      "Something went wrong in Find method."
    );
}

// The following is used to compute the
// integer hash code of the key, and shift it if needed
// using countP.
private int GetIndex(TKey aKey, int countP)
{
  // countP captures the number of times we had to solve
  // a collision.
  return (
      Math.Abs(aKey.GetHashCode())
      + collisionR(countP, aKey)
    ) % table.Length;
}

// This is the how collision are handled.
// It depends on the strategy picked.
```

```
// This overall strategy is called open addressing.
//
 ↪  https://en.wikibooks.org/wiki/Data_Structures/Hash_Tables#Open_addressing
private int collisionR(int i, TKey aKey)
{
  if (i == 0)
    return 0;
  else
  {
    if (Strategy == CollisionRes.Linear)
      return i++;
    else if (Strategy == CollisionRes.Quad)
      return i * i;
    else if (Strategy == CollisionRes.Double)
      // This is double hashing.
      return i * (31 - (aKey.GetHashCode() % 31)); // i
       ↪  * hash2(aKey) where hash2 is 31 - (key % 31)
       ↪  and will always be > 0
    else
      throw new ApplicationException(
        "Unknown collision startegy."
      );
  }
}

public void Remove(TKey aKey)
{
  //int index = Search(aKey, IsDeletedOrFound);
  int count = 0;
  int index = GetIndex(aKey, count);
  while (
    table[index] != null
    && (
      table[index].Status == StatusType.Deleted
      || !table[index].Key.Equals(aKey)
    )
  )
  {
    count++;
    if (count == table.Length) // in case table is full,
     ↪  kicks out of inf loop
      throw new ApplicationException("Table is full");
    index = GetIndex(aKey, count);
  }
  // Search will keep looking until found or empty cell.
  if (table[index] == null)
```

```
      throw new KeyNotFoundException(
        "Cannot delete missing key: " + aKey.ToString()
      );
    else if (
      table[index].Status == StatusType.Active
      && table[index].Key.Equals(aKey)
    )
      table[index].Status = StatusType.Deleted; // Found
↪   it! Mark the cell as deleted.
    else
      throw new ApplicationException(
        "Something went wrong in the Remove method."
      );
  }

  // The following allows the use of [].
  public TValue this[TKey aKey]
  {
    get { return Find(aKey); }
    set
    {
      // find empty cell (e.g. cell is null, status empty
       ↪  or deleted)
      int count = 0;
      int index = GetIndex(aKey, count);
      while (
        table[index] != null
       && !table[index].Status.Equals(StatusType.Deleted)
      )
      {
        count++;
        if (count == table.Length) // in case table is
         ↪  full, kicks out of inf loop
          throw new ApplicationException("Table is full");
        index = GetIndex(aKey, count);
      }
      // table slot is empty
      if (table[index] == null)
        table[index] = new Cell(
          aKey,
          value,
          StatusType.Active
        );
      // duplicate key found
      else if (
        table[index].Key.Equals(aKey) == true
```

```
        && table[index].Status == StatusType.Active
      )
        table[index].Value = value;
      // previously used item, reuse it
      else if (table[index].Status == StatusType.Deleted)
      {
        table[index].Value = value;
        table[index].Key = aKey;
        table[index].Status = StatusType.Active;
      }
      else
        throw new ApplicationException(
          "Something went wrong in [] set."
        );
    }
  }
}
```

(Download this code)

```
/*
 * Why prime numbers are needed is explained for example
 * at
 * https://cs.stackexchange.com/questions/11029
 */

public static class PrimeHelper
{
  public static bool IsPrime(int n)
  {
    // "A prime number is a natural number greater than 1
    ↪  that is not a product of two smaller natural
    ↪  numbers."
    // https://en.wikipedia.org/wiki/Prime_number
    if (n < 2)
      return false;
    if (n == 2 || n == 3)
      return true;
    if (n % 2 == 0)
      return false;
    for (int i = 3; i * i <= n; i += 2)
      if (n % i == 0)
        return false;
    return true;
  }

  public static int NextPrime(int n)
```

```
  {
    if (n < 2)
    {
      n = 2;
    }
    else
    {
      // Since 2 is the only even prime,
      // we make the n even if it is divisible
      // by 2.
      if (n % 2 == 0)
        n++;

      while (!IsPrime(n))
      {
        n += 2;
      }
    }
    return n;
  }
}
```

(Download this code)

```
using System;
using System.Collections.Generic;

class Program
{
  /*
   * Demonstrating how to use
   * enum type, cf.
   * https://learn.microsoft.com/en-
 ↪  us/dotnet/csharp/language-reference/builtin-
 ↪  types/enum
   */
  public enum Level
  {
    Low,
    Medium,
    High,
  }

  static void Main(string[] args)
  {
    // Demonstrating enum type.
```

```
Level lvl1 = Level.Medium; // To access the value, we prefix with Level.
Level lvl2 = (Level)0; // We can cast an int into a Level.
Console.WriteLine(lvl1 + " " + lvl2);
// Will display "Medium Low"

// Demonstrating PrimeHelper class:
for (int i = 0; i < 10; i++)
{
  Console.WriteLine(
    "The smallest prime greater than or equal to "
      + i
      + " is "
      + PrimeHelper.NextPrime(i)
      + "."
  );
}

// Demonstrating GetHashCode:
Console.WriteLine(
  "The hash code of an empty array of 12 int is: "
    + (new int[12]).GetHashCode()
    + "."
);
Console.WriteLine(
  "The hash code of an empty array of 14 string is: "
    + (new string[14]).GetHashCode()
    + "."
);
Console.WriteLine(
  "The hash code of \"test string\" is: "
    + "test string".GetHashCode()
    + "."
);
Console.WriteLine(
  "The hash code of 12 is: " + 12.GetHashCode() + "."
);

// Example of using the CDictionary class
CDictionary<string, int> ht = new CDictionary<
  string,
  int
>(13, CDictionary<string, int>.CollisionRes.Linear);
// Key of type string, value of type int.
ht.Add("one", 1);
```

```
    ht.Add("twenty", 20);
    ht.Add("fourteen", 14);
    ht.Add("two", 2);
    ht.Add("seventeen", 17);
    ht["fifteen"] = 15;
    Console.Write(ht);
    Console.WriteLine(ht["two"]);
    ht["two"] = 10;
    Console.WriteLine(ht["two"]);

    int x = ht.Find("two");
    Console.WriteLine($"Found x = {x}");
    try
    {
      int y = ht.Find("zzz");
      Console.WriteLine($"Found x = {y}");
    }
    catch (Exception)
    {
      Console.WriteLine($"Didn't find zzz");
    }

    ht.Remove("two");
    try
    {
      int y = ht.Find("two");
      Console.WriteLine($"Should not find two = {y}");
    }
    catch (Exception)
    {
      Console.WriteLine(
        $"Didn't find two since it was removed"
      );
    }
    try
    {
      ht.Remove("two");
      int y = ht.Find("two");
      Console.WriteLine($"Should not find two = {y}");
    }
    catch (Exception)
    {
      Console.WriteLine(
        $"Shoud throw when trying to remove two since it
↪   was removed"
      );
```

```
    }
  }
}
```

(Download this code)