Contents

Dictionary (Solutions)

Multiple Choices

1. Put a checkmark in the box corresponding to true statements.

   ☒ Any datatype can be used for keys, but they tend to be "simpler" than the datatype used for values.
   ☐ Two pairs (or `Cells`) with the same key can be stored in a dictionary, provided they have different values.
   ☒ A collision occurs when two keys (or, in general, two pieces of data) have the same hash.
   ☒ Open addressing and (separate) chaining are two methods to resolve collisions.
   ☐ Clustering is what makes dictionaries process data faster.

   Comments on the solution

   - Note that a collision actually occurs when the keys have the same hash *modulo the array size* in our case.
   - Clustering is actually negative: it means that data is often stored in the same place in the dictionary, making it more computationally costly to find the data.

Problem

1. Consider the implementation of "simple" dictionary `SDictionary` below:

```
using System;

public class SDictionary
{
  private class Cell
  {
    public string Value { get; set; }
    public string Key { get; set; }

    public Cell(
      string keyP,
```

```
      string valueP
    )
    {
      Key = keyP;
      Value = valueP;
    }

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

  private Cell[] table;

  public SDictionary(
    int size = 31
  )
  {
    table = new Cell[size];
  }

    public int GetIndex(string keyP, int countP)
    {
        return ((int)(keyP[0]) + countP) %
         ↪  table.Length;
    }
    public void Add(string keyP, string valueP)
    {
        int count = 0;
        int index = GetIndex(keyP, count);
        while (
          table[index] != null
        )
        {
            count++;
            index = GetIndex(keyP, count);
        }
        table[index] = new Cell(
          keyP,
          valueP
        );
    }
}
```

(Download this code)

Remember that, for example, `"Bob"[0]` is `'B'` and use the correspondence below between characters and their integer representation to help you (i.e., `(int)'B'` is 66):

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90

(a) Fill the `table` array below after the following have been performed:

```
SDictionary friends = new SDictionary(11);
friends.Add("Bob", null);
friends.Add("Pete", null);
friends.Add("Mary", null);
friends.Add("Lora", null);
```

0	1	2	3	4	5	6	7	8	9	10

Solution

0	1	2	3	4	5	6	7	8	9	10
"Bob"	"Mary"	`null`	"Pete"	`null`	`null`	`null`	`null`	`null`	`null`	"Lora"

The important point is to realize that

Expression	Value
`"Bob"[0]`	`'B'`
`(int)'B'`	66
`66 % 11`	0
`"Mary"[0]`	`'M'`
`(int)'M'`	77
`77 % 11`	0

So "Bob" and "Mary" are both stored at index 0. "Bob" is inserted first at index `0`, and since the `Add` method use linear open addressing, "Mary" is stored at the next available index, `1` in this case.

(b) What would happen if `friends.Add("Lora", null);` was executed again? Is it what is expected from a dictionary?

Solution

"Lora" would be inserted at index 2: 10, 0 and 1 being taken, `Add` goes to the next available index, 2. This is not expected, since a dictionary should reject entering *two* values with the same key.

(c) Write a `ToString` method for the `SDictionary` class, that returns a `string` containing all the keys and values stored in the dictionary.

Solution

```
public override string ToString()
{
    string returned =
      "";
    for (int i = 0; i < table.Length; i++)
    {
        returned += String.Format("\nIndex {0,
↪ -2} |", i);
        if (table[i] != null)
        {
            returned += String.Format(
              " {0, -10}| {1, -10} |",
              table[i].Key,
              table[i].Value
            );
        }
    }
    return returned;
}
```

(Download this code)

(d) What would happen if we were to try to insert 12 elements in our `friends` object?

Solution

When trying to insert the 12th element in the array of size 11, `Add` would loop forever, always circling through the array, looking for a cell containing `null`, while none are left.

(e) Consider the following `Delete` method:

```
public bool Delete(string keyP)
{
```

```
        int count = 0;
        int index = GetIndex(keyP, count);
        bool found = false;
        while (table[index] != null && !found)
        {
            if (table[index].Key.Equals(keyP))
            {
                found = true;
                table[index] = null;
            }
            count++;
            index = GetIndex(keyP, count);
        }
        return found;
    }
```

(Download this code)

Complete the series of instructions below such that `demo.Delete(error)` would return **`false`** even though the string `error` *is* the key of a value present in the `demo` dictionary object.

```
class Program{
    static void Main(){
    SDictionary demo = new SDictionary(    ); //
 ↪  Complete it.

    string error =                             //
     ↪  Fill me
    // To be completed.
    Console.WriteLine($"{error} was in demo:
 ↪  {demo.Delete(error)}.");
    }
}
```

Solution

The solution is to be in a position where the `error` value is "hidden after" a **`null`** value:

```
        SDictionary demo = new SDictionary(2);
        string error = "Alex";
        demo.Add("Alice", null);
        demo.Add(error, null);
        demo.Delete("Alice");
        Console.WriteLine($"{error} is in
 ↪  dictionary: {demo.Delete(error)}.");
```

(Download this code)