Custom Collections

There are a great many collections well known in the computer science literature, which haven't made the cut for being included in the .NET Framework. Some of them are fairly common and your applications may benefit from using them instead of the built-in ones. Furthermore, most of them offer sufficiently simple algorithms that can be implemented in a reasonable time. Although it is not our intent to explore the large variety of collections, below are two examples that differ greatly from the existing .NET collections and offer insight into situations where custom collections may be useful.

Disjoint-Set (Union-Find)

The *disjoint-set* data structure (often called *union-find*) is a collection in which partitions of elements into disjoint subsets are stored. It differs from all .NET collections because you do not store elements in it. Instead, there is a domain of elements in which each element forms a single set, and consecutive operations on the data structure join sets together to form larger sets. The data structure is designed to perform two operations efficiently:

- *Union*: Join two subsets together to form a single subset.
- *Find*: Determine to which subset a particular element belongs. (Most commonly used to determine whether two elements belong to the same subset.)

Typically, sets are manipulated as representative elements, with a single representative for each set. The union and find operations, then, receive and return representatives instead of entire sets.

A naïve implementation of union-find involves using a collection to represent each set, and merging the collections together when necessary. For example, when using a linked list to store each set, the merge takes linear time and the find operation may be implemented in constant time if each element has a pointer to the set representative.

The Galler-Fischer implementation has *much* better runtime complexity. The sets are stored in a forest (set of trees); in each tree, every node contains a pointer to its parent node, and the root of the tree is the set representative. To make sure that the resulting trees are balanced, when trees are merged, the smaller tree is always attached to the root of the larger tree (this requires tracking the tree's depth). Additionally, when the find operation executes, it compresses the path from the desired element to its representative. Below is a sketch implementation:

```
public class Set<T> {
  public Set Parent;
  public int Rank;
  public T Data;
  public Set(T data) {
   Parent = this;
   Data = data;
  }
  public static Set Find(Set x) {
   if (x.Parent != x) {
   x.Parent = Find(x.Parent);
   }
   return x.Parent;
  }
  public static void Union(Set x, Set y) {
   Set xRep = Find(x);
   Set yRep = Find(y);
   if (xRep == yRep) return; //It's the same set
   if (xRep.Rank < yRep.Rank) xRep.Parent = yRep;
   else if (xRep.Rank > yRep.Rank) yRep.Parent = xRep;
   else {
   yRep.Parent = xRep;
   ++xRep.Rank; //Merged two trees of equal rank, so rank increases
   }
  }
}
```

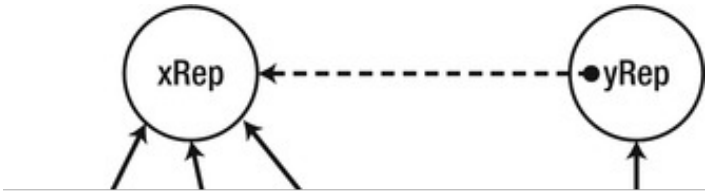

Figure 5-6. Merge of two sets x and y where y's set is smaller. The dashed arrow is the result of the merge

Accurate runtime analysis of this data structure is quite complex; a simple upper bound is that the amortized time per operation in a forest with n elements is $O(\log^* n)$, where $\log^* n$ (the iterated logarithm of n) is the number of times the logarithm function must be applied to get a result that is smaller than 1, i.e. the minimal number of times "log" must appear in the inequality $\log \log \log \ldots \log n \leq 1$. For practical values of n, e.g., $n \leq 10^{50}$, this is no more than 5, which is "effectively constant."

Skip List

A *skip list* is a data structure that stores a sorted linked list of elements and allows lookup in $O(\log n)$ time, comparable to a binary search in an array or lookup in a balanced binary tree. Clearly, the major problem with performing a binary search in a linked list is that linked lists do not allow random access by index. Skip lists address this limitation by using a hierarchy of increasingly sparse linked lists: the first linked list links together all the nodes; the second linked list links together nodes 0, 2, 4, ...; the third linked list links together nodes 0, 4, 8, ...; the fourth linked list links together nodes 0, 8, 16, ...; and so forth.

To perform a lookup for an element in the skip list, the procedure iterates the sparsest list first. When an element is encountered that is greater than or equal to the desired element, the procedure returns to the previous element and drops to the next list in the hierarchy. This repeats until the element is found. By using $O(\log n)$ lists in the hierarchy, $O(\log n)$ lookup time can be guaranteed.

Unfortunately, maintaining the skip list elements is not at all trivial. If the entire linked list hierarchy must be reallocated when elements are added or removed, it would offer no advantages over a trivial data structure such as `SortedList<T>`, which simply maintains a sorted array. A common approach is to randomize the hierarchy of lists (see Figure 5-7), which results in expected logarithmic time for insertion, deletion, and lookup of elements. The precise details of how to maintain a skip list can be found in William Pugh's paper, "Skip lists: a probabilistic alternative to balanced trees" (ACM, 1990).

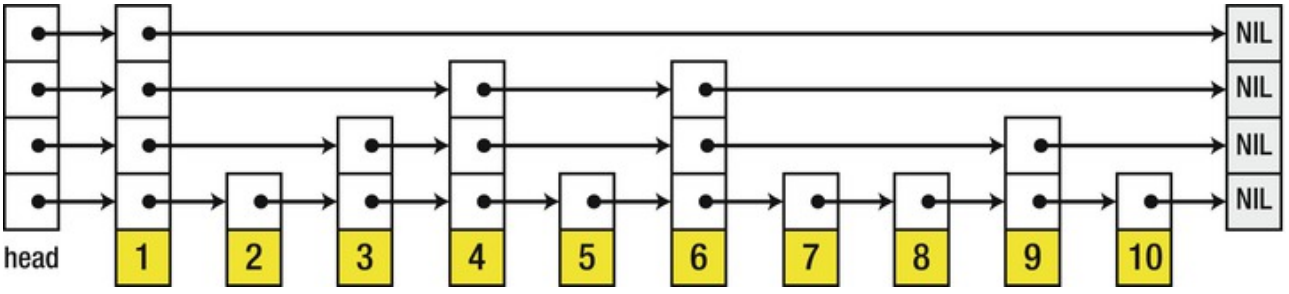

Figure 5-7. Skip list structure with four randomized lists in the hierarchy (Image from Wikipedia: http://upload.wikimedia.org/wikipedia/commons/8/86/Skip_list.svg, released into the public domain.)

One-Shot Collections

It may also be the case that you have a unique situation which warrants the use of a completely custom collection. We call these *one-shot collections*, because they might be an undeniably new invention tailored to your specific domain. As time goes by you may find that some of the one-shot collections you implemented are in fact quite reusable; in this subsection we will take a look at one example.

Consider the following application. You are running a candy exchange system, which keeps candy traders up to date with prices for various candy types. Your main data table is stored in memory and contains a row for each type of candy, listing its current price. Table 5-4 is an example of the data at a certain instant in time:

***Table 5-4.** Example of data table for the candy exchange system*

Type of Candy	Price ($)
Twix	0.93
Mars	0.88
Snickers	1.02
Kisses	0.66

There are two types of clients in your system:

- Candy traders are connected to your system through a TCP socket and periodically ask you for up-to-date information on a certain type of candy. A typical request from a trader is “What’s the price of Twix?” and your response is “$0.93”. There are tens of thousands of these requests per second.
- Candy suppliers are connected to your system through a UDP socket and periodically send you candy price updates. There are two subtypes of requests:
    - “Update the price of Mars to $0.91”. No response is necessary. There are thousands of these requests per second.
    - “Add a new candy, Snowflakes, with a starting price of $0.49”. No response is necessary. There are no more than a few dozens of these requests per day.

It is also known that 99.9 % of operations read or update the price of candy types that existed at the beginning of the trade; only 0.1 % of operations access candy types that were added by the add operation.

Armed with this information, you set out to design a data structure — a collection — to store the data table in memory. This data structure must be thread-safe, because hundreds of threads may be competing for access to it at a given time. You need not concern yourself with copying the data to persistent storage; we are inspecting only its in-memory performance characteristics.

The shape of the data and the types of requests our system serves suggest strongly that we should use a hash table to store the candy prices. Synchronizing access to a hash table is a task best left to `ConcurrentDictionary <K,V>`. Reads from a concurrent dictionary can be satisfied without any synchronization, whereas update and add operations require synchronization at a fairly fine-grained level. Although this may be an adequate solution, we set ourselves a higher bar: we would like reads and updates to proceed without any synchronization in the 99.9 % of operations on pre-existing candy types.

A possible solution to this problem is a *safe-unsafe cache*. This collection is a set of two hash tables, the *safe table* and the *unsafe table*. The safe table is prepopulated with the candy types available at the beginning of the trade; the unsafe table starts out empty. Operations on the safe table are satisfied without any locks because it is not mutated; new candy types are added to the unsafe table. Below is a possible implementation using `Dictionary <K,V>` and `ConcurrentDictionary <K,V>`:

```
//Assumes that writes of TValue can be satisfied atomically, i.e. it must be a reference
//type or a sufficiently small value type (4 bytes on 32-bit systems).
public class SafeUnsafeCache<TKey, TValue> {
  private Dictionary<TKey, TValue> safeTable;
  private ConcurrentDictionary<TKey, TValue> unsafeTable;
public SafeUnsafeCache(IDictionary<TKey, TValue> initialData) {
   safeTable = new Dictionary<TKey, TValue>(initialData);
   unsafeTable = new ConcurrentDictionary<TKey, TValue>();
  }
  public bool Get(TKey key, out TValue value) {
   return safeTable.TryGetValue(key, out value) || unsafeTable.TryGetValue(key, out value);
  }
  public void AddOrUpdate(TKey key, TValue value) {
   if (safeTable.ContainsKey(key)) {
   safeTable[key] = value;
   } else {
   unsafeTable.AddOrUpdate(key, value, (k, v) => value);
   }
  }
}
```

A further refinement would be to periodically stop all trade operations and merge the unsafe table into the safe table. This would further improve the expected synchronization required for operations on the candy data.

IMPLEMENTING IENUMERABLE <T>AND OTHER INTERFACES

Almost any collection will eventually implement `IEnumerable <T>` and possibly other collection-related interfaces. The advantages of complying with these interfaces are numerous, and as of .NET 3.5 have to do with LINQ as much as anything. After all, any class that implements `IEnumerable <T>` is *automatically* furnished with the variety of extension methods offered by `System.Linq` and can participate in C# 3.0 LINQ expressions on equal terms with the built-in collections.

Unfortunately, naïvely implementing `IEnumerable<T>` on your collection sentences your callers to pay the price of interface method invocation when they enumerate it. Consider the following code snippet, which enumerates a `List<int>`:

```
List<int> list = ...;
IEnumerator<int> enumerator = list.GetEnumerator();
long product = 1;
while (enumerator.MoveNext()) {
    product *= enumerator.Current;

}
```

There are two interface method invocations per iteration here, which is an unreasonable overhead for traversing a list and finding the product of its elements. As you may recall from Chapter 3, inlining interface method invocations is not trivial, and if the JIT fails to

inline them successfully, the price is steep.

There are several approaches that can help avoid the cost of interface method invocation. Interface methods, when invoked directly on a value type variable, can be dispatched directly. Therefore, if the `enumerator` variable in the above example had been a value type (and not `IEnumerator<T>`), the interface dispatch cost would have been prevented. This can only work if the collection implementation could return a value type directly from its `GetEnumerator` method, and the caller would use that value type instead of the interface.

To achieve this, `List<T>` has an explicit interface implementation of `IEnumerable<T>.GetEnumerator`, which returns `IEnumerator<T>`, and another public method called `GetEnumerator`, which returns `List<T>.Enumerator`— an inner value type:

```
public class List<T> : IEnumerable<T>, ... {
  public Enumerator GetEnumerator() {
    return new Enumerator(this);
  }
  IEnumerator<T> IEnumerable<T>.GetEnumerator() {
    return new Enumerator(this);
  }
  ...
  public struct Enumerator { ... }
}
```

This enables the following calling code, which gets rid of the interface method invocations entirely:

```
List<int> list = ...;
List.Enumerator<int> enumerator = list.GetEnumerator();
long product = 1;
while (enumerator.MoveNext()) {
    product *= enumerator.Current;
}
```

An alternative would be to make the enumerator a reference type, but repeat the same explicit interface implementation trick on its `MoveNext` method and `Current` property. This would also allow callers using the class directly to avoid interface invocation costs.