Chapter 5

Sets

A set contains a number of values, in no particular order. The values within the set are distinct from one another.

Generally set implementations tend to check that a value is not in the set before adding it, avoiding the issue of repeated values from ever occurring.

This section does not cover set theory in depth; rather it demonstrates briefly the ways in which the values of sets can be defined, and common operations that may be performed upon them.

The notation $A = \{4, 7, 9, 12, 0\}$ defines a set A whose values are listed within the curly braces.

Given the set A defined previously we can say that 4 is a member of A denoted by $4 \in A$, and that 99 is not a member of A denoted by $99 \notin A$.

Often defining a set by manually stating its members is tiresome, and more importantly the set may contain a large number of values. A more concise way of defining a set and its members is by providing a series of properties that the values of the set must satisfy. For example, from the definition $A = \{x|x > 0, x \ \% \ 2 = 0\}$ the set A contains only positive integers that are even. x is an alias to the current value we are inspecting and to the right hand side of | are the properties that x must satisfy to be in the set A. In this example, x must be > 0, and the remainder of the arithmetic expression $x/2$ must be 0. You will be able to note from the previous definition of the set A that the set can contain an infinite number of values, and that the values of the set A will be all even integers that are a member of the natural numbers set $\mathbb{N}$, where $\mathbb{N} = \{1, 2, 3, ...\}$.

Finally in this brief introduction to sets we will cover set intersection and union, both of which are very common operations (amongst many others) performed on sets. The union set can be defined as follows $A \cup B = \{x \mid x \in A \ or \ x \in B\}$, and intersection $A \cap B = \{x \mid x \in A \ and \ x \in B\}$. Figure 5.1 demonstrates set intersection and union graphically.

Given the set definitions $A = \{1, 2, 3\}$, and $B = \{6, 2, 9\}$ the union of the two sets is $A \cup B = \{1, 2, 3, 6, 9\}$, and the intersection of the two sets is $A \cap B = \{2\}$.

Both set union and intersection are sometimes provided within the framework associated with mainstream languages. This is the case in .NET 3.5[1] where such algorithms exist as extension methods defined in the type *System.Linq.Enumerable*[2], as a result DSA does not provide implementations of

[1] `http://www.microsoft.com/NET/`

[2] `http://msdn.microsoft.com/en-us/library/system.linq.enumerable_members.aspx`

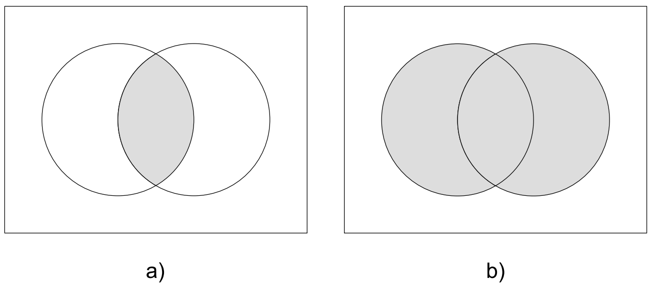

Figure 5.1: a) $A \cap B$; b) $A \cup B$

these algorithms. Most of the algorithms defined in *System.Linq.Enumerable* deal mainly with sequences rather than sets exclusively.

Set union can be implemented as a simple traversal of both sets adding each item of the two sets to a new union set.

```
1) algorithm Union($set1$, $set2$)
2)     Pre: $set1$, and $set2 \neq \emptyset$
3)          $union$ is a set
3)     Post: A union of $set1$, and $set2$ has been created
4)     foreach $item$ in $set1$
5)         $union$.Add($item$)
6)     end foreach
7)     foreach $item$ in $set2$
8)         $union$.Add($item$)
9)     end foreach
10)    return $union$
11) end Union
```

The run time of our *Union* algorithm is $O(m + n)$ where m is the number of items in the first set and n is the number of items in the second set. This runtime applies only to sets that exhibit $O(1)$ insertions.

Set intersection is also trivial to implement. The only major thing worth pointing out about our algorithm is that we traverse the set containing the fewest items. We can do this because if we have exhausted all the items in the smaller of the two sets then there are no more items that are members of both sets, thus we have no more items to add to the intersection set.

```
1) algorithm Intersection($set1$, $set2$)
2)     Pre: $set1$, and $set2 \neq \emptyset$
3)          $intersection$, and $smallerSet$ are sets
3)     Post: An intersection of $set1$, and $set2$ has been created
4)     if $set1$.Count < $set2$.Count
5)        $smallerSet \leftarrow set1$
6)     else
7)        $smallerSet \leftarrow set2$
8)     end if
9)     foreach $item$ in $smallerSet$
10)       if $set1$.Contains($item$) and $set2$.Contains($item$)
11)          $intersection$.Add($item$)
12)       end if
13)    end foreach
14)    return $intersection$
15) end Intersection
```

The run time of our *Intersection* algorithm is $O(n)$ where n is the number of items in the smaller of the two sets. Just like our *Union* algorithm a linear runtime can only be attained when operating on a set with $O(1)$ insertion.

5.1 Unordered

Sets in the general sense do not enforce the explicit ordering of their members. For example the members of $B = \{6, 2, 9\}$ conform to no ordering scheme because it is not required.

Most libraries provide implementations of unordered sets and so DSA does not; we simply mention it here to disambiguate between an unordered set and ordered set.

We will only look at insertion for an unordered set and cover briefly why a hash table is an efficient data structure to use for its implementation.

5.1.1 Insertion

An unordered set can be efficiently implemented using a hash table as its backing data structure. As mentioned previously we only add an item to a set if that item is not already in the set, so the backing data structure we use must have a quick look up and insertion run time complexity.

A hash map generally provides the following:

1. $O(1)$ for insertion
2. approaching $O(1)$ for look up

The above depends on how good the hashing algorithm of the hash table is, but most hash tables employ incredibly efficient general purpose hashing algorithms and so the run time complexities for the hash table in your library of choice should be very similar in terms of efficiency.

5.2 Ordered

An ordered set is similar to an unordered set in the sense that its members are distinct, but an ordered set enforces some predefined comparison on each of its members to produce a set whose members are ordered appropriately.

In DSA 0.5 and earlier we used a binary search tree (defined in §3) as the internal backing data structure for our ordered set. From versions 0.6 onwards we replaced the binary search tree with an AVL tree primarily because AVL is balanced.

The ordered set has its order realised by performing an inorder traversal upon its backing tree data structure which yields the correct ordered sequence of set members.

Because an ordered set in DSA is simply a wrapper for an AVL tree that additionally ensures that the tree contains unique items you should read §7 to learn more about the run time complexities associated with its operations.

5.3 Summary

Sets provide a way of having a collection of unique objects, either ordered or unordered.

When implementing a set (either ordered or unordered) it is key to select the correct backing data structure. As we discussed in §5.1.1 because we check first if the item is already contained within the set before adding it we need this check to be as quick as possible. For unordered sets we can rely on the use of a hash table and use the key of an item to determine whether or not it is already contained within the set. Using a hash table this check results in a near constant run time complexity. Ordered sets cost a little more for this check, however the logarithmic growth that we incur by using a binary search tree as its backing data structure is acceptable.

Another key property of sets implemented using the approach we describe is that both have favourably fast look-up times. Just like the check before insertion, for a hash table this run time complexity should be near constant. Ordered sets as described in 3 perform a binary chop at each stage when searching for the existence of an item yielding a logarithmic run time.

We can use sets to facilitate many algorithms that would otherwise be a little less clear in their implementation. For example in §11.4 we use an unordered set to assist in the construction of an algorithm that determines the number of repeated words within a string.