An Extensive Examination of Data Structures Using C# 2.0

Visual Studio 2005

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: This sixth installment of the article series examines how to implement a common mathematical construct, the set. A set is an unordered collection of unique items that can be enumerated and compared to other sets in a variety of ways. In this article we'll look at data structures for implementing general sets as well as disjoint sets. (20 printed pages)

Download the DataStructures20.msi sample file.

> **Editor's note** This six-part article series originally appeared on MSDN Online starting in November 2003. In January 2005 it was updated to take advantage of the new data structures and features available with the .NET Framework version 2.0, and C# 2.0. The original articles are still available at http://msdn.microsoft.com/vcsharp/default.aspx?pull=/library/en-us/dv_vstechart/html/datastructures_guide.asp.

> **Note** This article assumes the reader is familiar with C#.

Contents

Introduction

One of the most basic mathematical constructs is a *set*, which is an unordered collection of unique objects. The objects contained within a set are referred to as the set's *elements*. Formally, a set is denoted as a capital, italic letter, with its elements appearing within curly braces (`{ . . . }`). Examples of this notation can be seen below:

```
S = { 1, 3, 5, 7, 9 }
T = { Scott, Jisun, Sam }
U = { -4, 3.14159, Todd, x }
```

In mathematics, typically sets are comprised strictly of numbers, such as set *S* above, which contains the odd positive integers less than 10. But notice that the elements of a set can be anything—numbers, people, strings, letters, variables, and so on. Set *T*, for example, contains peoples' names; set *U* contains a mix of numbers, names, and variables.

In this article we'll start with a basic introduction of sets, including common notation and the operations that can be performed on sets. Following that, we'll examine how to efficiently implement a set data structure with a defined universe. The article concludes with an examination of disjoint sets, and the best data structures to use.

The Fundamentals of Sets

Recall that a set is simple a collection of elements. The "element of" operator, denoted *x* Î *S*, implies that *x* is an element in the set *S*. For example, if set *S* contains the odd positive integers less than 10, then 1Î *S*. When reading such notation, you'd say, "1 is an element of *S*." In addition to 1 being an element of *S*, we have 3 Î *S*, 5 Î *S*, 7 Î *S*, and 9 Î *S*. The "not and element of" operator, denoted *x* Ï *S*, means that *x* is not an element of set *S*.

The number of unique elements in a set is the set's *cardinality*. The set {1, 2, 3} has cardinality 3, just as does the set {1, 1, 1, 1, 1, 1, 1, 2, 3} (because it only has three unique elements). A set may have no elements in it at all. Such a set is called the *empty set*, and is denoted as {} or Æ, and has a cardinality of 0.

When first learning about sets, many developers assume they are tantamount to collections, like a List. However, there are some subtle differences. A List is an *ordered* collection of elements. Each element in a List has an associated ordinal index, which implies order. Too, there can be duplicate elements in a List.

A set, on the other hand, is *unordered* and contains *unique* items. Because sets are unordered, the elements of a set may be listed in any order. That is, the sets {1, 2, 3} and {3, 1, 2} are considered equivalent. Also, any duplicates in a set are considered redundant. The set {1, 1, 1, 2, 3} and the set {1, 2, 3} are equivalent. Two sets are equivalent if they have the same elements. (Equivalence is denoted with the = sign; if S and T are equivalent they are written as *S* = *T*.)

> **Note** In mathematics, an ordered collection of elements that allows duplicates is referred to as a list. Two lists, L_1 and L_2 are considered equal if and only if for i ranging from 1 to the number of elements in the list the ith element in L_1 equals the ith element in L_2.

Typically the elements that can appear in a set are restricted to some *universe*. The universe is the set of all possible values that can appear in a set. For example, we might only be interested in working with sets whose universe are integers. By restricting the universe to integers, we can't have a set that has a non-integer element, like 8.125, or Sam. (The universe is denoted as the set *U*.)

Relational Operators of Sets

There are a bevy of relational operators that are commonly used with numbers. Some of the more often used ones, especially in programming languages, include <, <=, =, !=, >, and >=. A relational operator determines if the operand on the left hand side is related to the operand on the right hand side based on criteria defined by the relational operator. Relational operators return a "true" or "false" value, indicating whether or not the relationship holds between the operands. For example, *x* < *y* returns true if *x* is less than *y*, and false otherwise. (Of course the meaning of "less than" depends on the data type of *x* and *y*.)

Relational operators like <, <=, =, !=, >, and >= are typically used with numbers. Sets, as we've seen, use the = relational operator to indicate that two sets are equivalent (and can likewise use != to denote that two sets are not equivalent), but relational operators <, <=, >, and >= are not defined for sets. After all, how is one to determine if the set {1, 2, 3} is less than the set {Scott, 3.14159}?

Instead of notions of < and <=, sets use the relational operators *subset* and *proper subset*, denoted Í and Ì, respectively. (Some older texts will use Ì for subset and Í for proper subset.) *S* is a subset of *T*—denoted *S* Í *T*—if every element in *S* is in *T*. That is, *S* is a subset of *T* if it is *contained* within *T*. If *S* = {1, 2, 3}, and *T* = {0, 1, 2, 3, 4, 5}, then *S* Í *T* because every element in *S* – 1, 2 and 3—is an element in *T*. *S* is a proper subset of *T*—denoted *S* Ì *T*—if *S* Í *T* and *S* [1] *T*. That is, if *S* = {1, 2, 3} and *T* = {1, 2, 3}, then *S* Í *T* because every element in *S* is an element in *T*, but *S* Ë *T* because *S* = *T*. (Notice that there is a similarity between the relational operators < and <= for numbers and the relational operators Ì and Í for sets.)

Using the new subset operator, we can more formally define set equality. Given sets *S* and *T*, *S* = *T* if and only if *S* Í *T* and *T* Í *S*. In English, *S* and *T* are equivalent if and only if every element in *S* is in *T*, and every element in *T* is in *S*.

> **Note** Because Í is analogous to <=, it would make sense that there exists a set relational operator analogous to >=. This relational operator is called superset, and is denoted Ê; a proper superset is denoted É. Like with <= and >=, S Ê T if and only if T Í S.

Set Operations

As with the relational operators, many operations defined for numbers don't translate well to sets. Common operations on numbers include addition, multiplication, subtraction, exponentiation, and so on. For sets, there are four basic operations:

1. **Union:** the union of two sets, denoted *S* È *T*, is akin to addition for numbers. The union operator returns a set that contains all of the elements in *S* and all of the elements in *T*. For example, {1, 2, 3} È {2, 4, 6} equals {1, 2, 3, 2, 4, 6}. (The duplicate 2 can be removed to provide a more concise answer, yielding {1, 2, 3, 4, 6}.) Formally, *S* È *T* = {*x* : *x* Î *S* or *x* Î *T*}. In English, this translates to *S* union *T* results in the set that contains an element *x* if *x* is in *S* or in *T*.
2. **Intersection:** the intersection of two sets, denoted *S* Ç *T*, is the set of elements that *S* and *T* have in common. For example, {1, 2, 3} Ç {2, 4, 6} equals {2}, because that's the only element both {1, 2, 3} and {2, 4, 6} share in common. Formally, *S* Ç *T* = {*x* : *x* Î *S* and *x* Î *T*}. In English, this translates to *S* intersect *T* results in the set that contains an element *x* if *x* is both in *S* **and** in *T*.
3. **Difference:** the difference of two sets, denoted *S* – *T*, are all of the elements in *S* that are **not** in *T*. For example, {1, 2, 3} – {2, 4, 6} equals {1, 3}, because 1 and 3 are the elements in *S* that are not in *T*. Formally, *S* – *T* = {*x* : *x* Î *S* and *x* Ï *T*}. In English, *S* set difference *T* results in the set that contains an element *x* if *x* is in *S* and **not** in *T*.
4. **Complement:** Earlier we discussed how typically sets are limited to a known universe of possible values, such as the integers. The complement of a set, denoted *S*', is *U* – *S*. (Recall that *U* is the universe set.) If our universe is the integers 1 through 10, and *S* = {1, 4, 9, 10}, then *S*' = {2, 3, 5, 6, 7, 8}. Complementing a set is akin to negating a number. Just like negating a number twice will give you the original number back. That is, --*x* = *x* – complementing a set twice will give you the original set back – *S*'' = *S*.

When examining new operations, it is always important to get a solid grasp on the nature of the operations. Some questions to ask yourself when learning about any operation, be it one defined for numbers or one defined for sets, are:

- **Is the operation commutative?** An operator *op* is commutative if *x op y* is equivalent to *y op x*. In the realm of numbers, addition is an example of a commutative operator, while division is not commutative.
- **Is the operation associative?** That is, does the order of operations matter. If an operator *op* is associative, then *x op* (*y op z*) is equivalent to (*x op y*) *op z*. Again, in the realm of numbers addition is associative, but division is not.

For sets, the union and intersection operations are both commutative and associative. *S* È *T* is equivalent to *T* È *S*, and *S* È (*T* È *V*) is equivalent to (*S* È *T*) È *V*. Set difference, however, is neither commutative nor associative. (To see that set difference is not commutative, consider that {1, 2, 3} – {3, 4, 5} = {1, 2}, but {3, 4, 5} – {1, 2, 3} = {4, 5}.)

Finite Sets and Infinite Sets

All of the set examples we've looked at thus far have dealt with finite sets. A finite set is a set that has a finite number of elements. While it may seem counterintuitive at first, a set can contain an infinite number of elements. The set of positive integers, for example, is an infinite set because there is no bounds to the number of elements in the set.

In mathematics, there are a couple infinite sets that are used so often that they are given a special symbol to represent them. These include:

N = {0, 1, 2, ...}

Z = {..., -2, -1, 0, 1, 2, ...}

Q = {a/b: a Î Z, b Î Z, and b ¹ 0}

R = *set of real numbers*

N is the set of *natural numbers*, or positive integers greater than or equal to 0. z is the set of *integers*. N is the set of *rational numbers*, which are numbers that can be expressed as a fraction of two integers. Finally, R is the set of *real numbers*, which are all rational numbers, plus irrational numbers as well (numbers that cannot be expressed as a fraction of two integers, such as *pi*, and the square root of 2).

Infinite sets, of course, can't be written down in their entirety, as you'd never finish jotting down the elements, but instead are expressed more tersely using mathematical notation like so:

```
S = {x : xÎ N and x > 100}
```

Here *S* would be the set of all natural numbers greater than 100.

In this article we will be looking at data structures for representing finite sets. While infinite sets definitely have their place in mathematics, rarely will we need to work with infinite sets in a computer program. Too, there are unique challenges with representing and operating upon infinite sets, because an infinite set's contents cannot be completely stored in a data structure or enumerated.

> **Note** Computing the cardinality of finite sets is simple. Just count up the number of elements in the set. But how does one compute the cardinality of an infinite set? This discussion is far beyond the scope of this article, but realize that there's different types of cardinality for infinite sets. For instance, the set of positive integers has the same cardinality as the set of **all** integers, but the set of real numbers has a larger cardinality than the set of all integers.

Sets in Programming Languages

C++, C#, Visual Basic .NET, and Java don't provide inherent language features for working with sets. If you want to use sets, you need to create your own set class with the appropriate methods, properties, and logic. (We'll do precisely this in the next section.) There have been programming languages in the past, though, that have offered sets as a fundamental building block in the language. Pascal, for example, provides a set construct that can be used to create sets with an explicitly defined universe. To work with sets, Pascal provides the `in` operator to determine if

an element is in a particular set. The operators +, *, and − are used for union, intersection, and set difference, respectively. The following Pascal code illustrates the syntax used to work with sets:

```
/* declares a variable named possibleNumbers, a set whose universe is the
  set of integers between 1 and 100... */
var
  possibleNumbers = set of 1..100;

...

/* Assigns the set {1, 45, 23, 87, 14} to possibleNumbers */
possibleNumbers := [1, 45, 23, 87, 14];

/* Sets possibleNumbers to the union of possibleNumbers and {3, 98} */
possibleNumbers := possibleNumbers + [3, 98];

/* Checks to see if 4 is an element of possible numbers... */
if 4 in possibleNumbers then write("4 is in the set!");
```

Other previous languages have allowed for more powerful set semantics and syntax. A language called SETL (an acronym for SET Language) was created in the 70s and offered sets as a first-class citizen. Unlike Pascal, when using sets in SETL you are not restricted to specifying the set's universe.

Implementing an Efficient Set Data Structure

In this section we'll look at creating a class that provides the functionality and features of a set. When creating such a data structure, one of the first things we need to decide is how to store the elements of the set. This decision can greatly affect the asymptotic efficiency of the operations performed on the set data structure. (Keep in mind that the operations we'll need to perform on the set data structure include: union, intersection, set difference, subset, and element of.)

To illustrate how storing the set's elements can affect the run time, imagine that we created a set class that used an underlying ArrayList to hold the elements of the set. If we had two sets, S_1 and S_2 that we wanted to union (where S_1 had m elements and S_2 had n elements), we'd have to perform the following steps:

1. Create a new set class, T, that holds the union of S_1 and S_2.
2. Iterate through the elements of S_1, adding it to T.
3. Iterate through the elements of S_2. If the element does not already exist in T, then add it to T.

How many steps would performing the union take? Step (2) would require m steps through S_1's m elements. Step (3) would take n steps, and for each element in S_2, we'd have to determine if the element was in T. To determine if an element is in an unsorted List the entire List must be enumerated linerarly. So, for each of the n elements in S_2 we might have to search through the m elements in T. This would lead to a quadratic running time for union of $O(m * n)$.

The reason a union with a List takes quadratic time is because determining if an element exists within a set takes linear time. That is, to determine if an element exists in a set, the set's List must be exhaustively searched. If we could reduce the running time for the "element of" operation to a constant, we could improve the union's running time to a linear $O(m + n)$. Recall from Part 2 of this article series that Hashtables and Dictionaries provides constant running

time to determine if an item resides within the data structure. Hence, a Hashtable or Dictionary would be a better choice for storing the set's elements than a List.

If we require that the set's universe be known, we can implement an even more efficient set data structure using a bit array. Assume that the universe consists of elements e_1, e_2, ..., e_k. Then we can denote a set with a *k*-element bit array; if the *i*th bit is 1, then the element e_i is in the set; if, on the other hand, the *i*th bit is 0, then the element e_i is not in the set. Representing sets as a bit array not only provides tremendous space savings, but also enables efficient set operations, as these set-based operations can be performed using simple bit-wise instructions. For example, determining if element e_i exists in a set takes constant time because only the *i*th bit in the bit array needs to be checked. The union of two sets is simply the bit-wise OR of the sets' bit arrays; the intersection of two sets is the bit-wise AND of the sets' bit arrays. Set difference and subset can be reduced down to bit-wise operations as well.

> **Note** A bit array is a compact array composed of 1s and 0s, typically implemented as an integer array. Because an integer in the .NET Framework has 32 bits, a bit array can store 32 bit values in one element of an integer array (rather than requiring 32 array elements).

Bit-wise operations are ones that are performed on the individual bits of an integer. There are both binary bit-wise operators and unary bit-wise operators. The bit-wise AND and bit-wise OR operators are binary, taking in two bits each, and returning a single bit. Bit-wise AND returns 1 only if both inputs are 1, otherwise it returns 0. Bit-wise OR returns 0 only if both inputs are 0, otherwise it returns 1.

For a more in-depth look at bit-wise operations in C# be sure to read Bit-Wise Operators in C#.

Let's look at how to implement a set class that uses C#'s bit-wise operations.

Creating the PascalSet Class

Understand that to implement a set class that uses the efficient bit-wise operators the set's universe must be known. This is akin to the way Pascal uses sets, so in honor of the Pascal programming language I have decided to name this set class the `PascalSet` class. `PascalSet` restricts the universe to a range of integers or characters (just like the Pascal programming language). This range can be specified in the `PascalSet`'s constructor.

```
public class PascalSet : ICloneable, ICollection, IEnumerable
{
   // Private member variables
   private int lowerBound, upperBound;
   private BitArray data;

   public PascalSet(int lowerBound, int upperBound)
   {
      // make sure lowerbound is less than or equal to upperbound
      if (lowerBound > upperBound)
         throw new ArgumentException("The set's lower bound cannot be
            greater than its upper bound.");

      this.lowerBound = lowerBound;
      this.upperBound = upperBound;

      // Create the BitArray
      data = new BitArray(upperBound - lowerBound + 1);
```

```
    }

    ...
}
```

So, to create a `PascalSet` whose universe is the set of integers between -100 and 250, the following syntax could be used:

```
PascalSet mySet = new PascalSet(-100, 250);
```

Implementing the Set Operations

`PascalSet` implements the standard set operations—union, intersection, and set difference—as well as the standard relational operators—subset, proper subset, superset, and proper superset. The set operations union, intersection, and set difference, all return a new `PascalSet` instance, which contains the result of unioning, intersecting, or set differencing. The following code for the `Union(`*`PascalSet`*`)` method illustrates this behavior:

```
public virtual PascalSet Union(PascalSet s)
{
    if (!AreSimilar(s))
        throw new ArgumentException("Attempting to union two dissimilar
           sets.  Union can only occur between two sets with the same universe.");

    // do a bit-wise OR to union together this.data and s.data
    PascalSet result = (PascalSet)Clone();
    result.data.Or(s.data);

    return result;
}

public static PascalSet operator +(PascalSet s, PascalSet t)
{
   return s.Union(t);
}
```

The `AreSimilar(`*`PascalSet`*`)` method determines if the `PascalSet` passed has the same lower and upper bounds as the `PascalSet` instance. Therefore, union (and intersection and set difference) can only be applied to two sets with the same universe. You could make a modification to the code here to have the returned `PascalSet`'s universe be the union of the two universe sets, thereby allowing sets with non-disjoint universes to be unioned. If the two `PascalSet`s have the same universe, then a new `PascalSet`—`result`—is created, which is an exact duplicate of the `PascalSet` instance whose `Union()` method was called. This cloned `PascalSet`'s contents are modified using the bit-wise OR method, passing in the set whose contents are to be unioned. Notice that the `PascalSet` class also overloads the `+` operator for union (just like the Pascal programming language).

Similarly, the **PascalSet** class provides methods for intersection and set difference, along with the overloaded operators `*` and `-`, respectively.

Enumerating the PascalSet's Members

Because sets are an *unordered* collection of elements, it would not make sense to have `PascalSet` implement `IList`, as collections that implement `IList` imply that the list has some ordinal order. Because `PascalSet` is a

collection of elements, though, it makes sense to have it implement `ICollection` and `IEnumerable`. Because `PascalSet` implements `IEnumerable`, it needs to provide a `GetEnumerator()` method that returns an `IEnumerator` instance allowing a developer to iterate through the set's elements. This method simply iterates through the set's underlying `BitArray`, returning the corresponding value for each bit with a value of 1.

```
public IEnumerator GetEnumerator()
{
    int totalElements = Count;
    int itemsReturned = 0;
    for (int i = 0; i < this.data.Length; i++)
    {
        if (itemsReturned >= totalElements)
            break;
        else if (this.data.Get(i))
            yield return i + this.lowerBound;
    }
}
```

To enumerate the PascalSet, you can simply use a `foreach` statement. The following code snippet demonstrates creating two PascalSet instances and then enumerating the elements of their union:

```
PascalSet a = new PascalSet(0, 255, new int[] {1, 2, 4, 8});
PascalSet b = new PascalSet(0, 255, new int[] { 3, 4, 5, 6 });

foreach(int i in a + b)
   MessageBox.Show(i);
```

This code would display seven messageboxes, one after the other, displaying values 1, 2, 3, 4, 5, 6, and 8.

The complete code for the `PascalSet` class is included as a download with this article. Along with the class, there is an interactive Windows Forms testing application, `SetTester`, from which you can create a `PascalSet` instance and perform various set operations, viewing the resulting set.

Maintaining a Collection of Disjoint Sets

Next time you do a search at Google notice that with each result there's a link titled "Similar Pages." If you click this link, Google displays a list of URLs that are related to the item whose "Similar Pages" link you clicked. While I don't know how Google particularly determines how pages are related, one approach would be the following:

- Let x be the Web page we are interested in finding related pages for.
- Let S_1 be the set of Web pages that x links to.
- Let S_2 be the set of Web pages that the Web pages in S_1 link to.
- Let S_3 be the set of Web pages that the Web pages in S_2 link to.
- ...
- Let S_k be the set of Web pages that the Web pages in S_{k-1} link to.

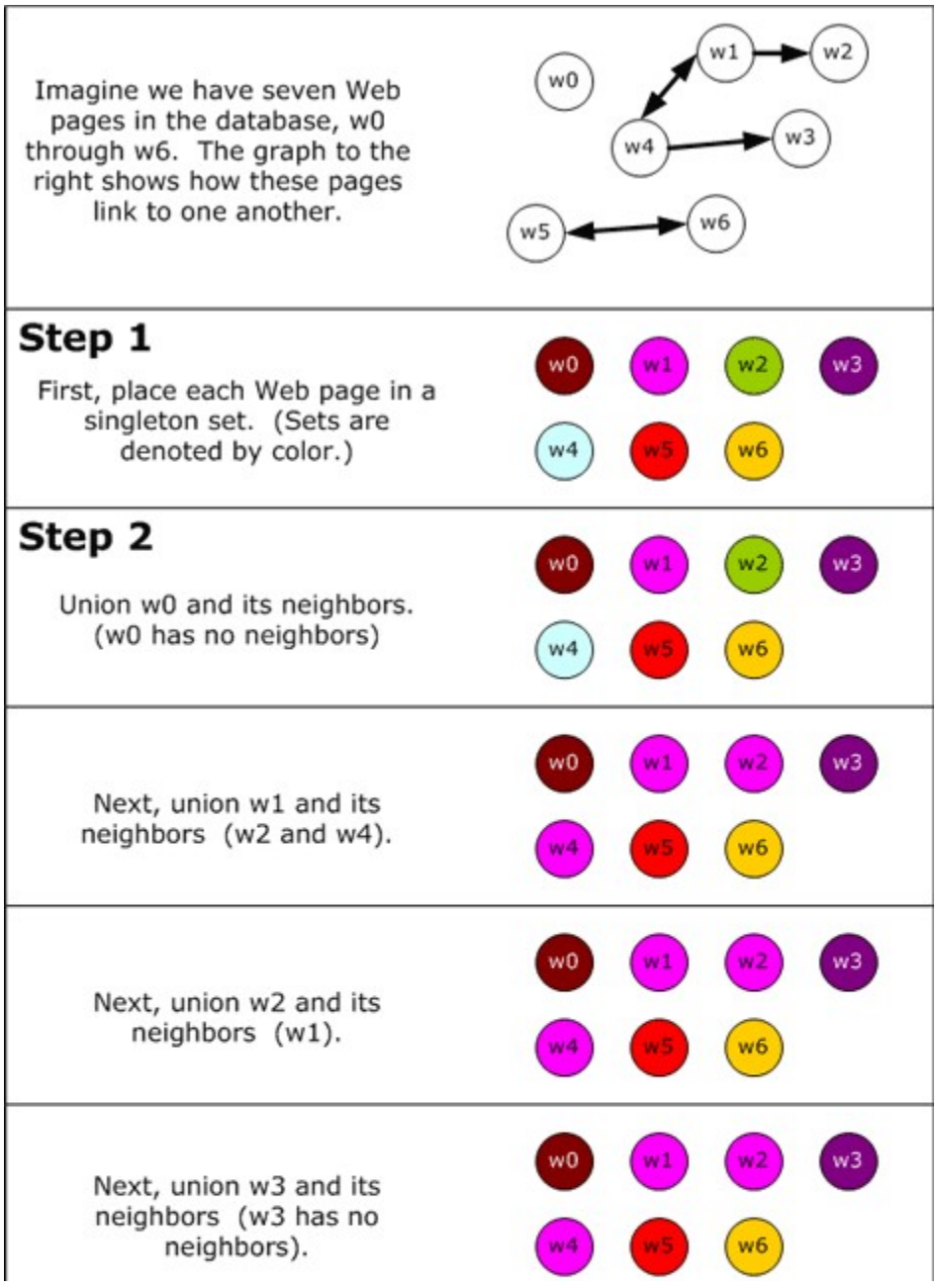
All of the Web pages in S_1, S_1, up to S_k are the related pages for x. Rather than compute the related Web pages on demand, we might opt to create the set of related pages for *all* Web pages once, and to store this relation in a

database or some other permanent store. Then, when a user clicks on the "Similar Pages" link for a search term, we simply query the display to get the links related to this page.

Google has some sort of database with all of the Web pages it knows about. Each of these Web pages has a set of links. We can compute the set of related Web pages using the following algorithm:

1. For each Web page in the database create a set, placing the single Web page in the set. After this step completes, if we have *n* Web pages in the database, we'll have *n* one-element sets.
2. For a Web page *x* in the database, find all of those Web pages it directly links to. Call these linked-to pages *S*. For each element *p* in *S*, union the set containing *p* with *x*'s set.
3. Repeat step 2 for all Web pages in the database.

After step 3 completes, the Web pages in the database will be partitioned out into related groups. To view a graphical representation of this algorithm in action, consult Figure 1.



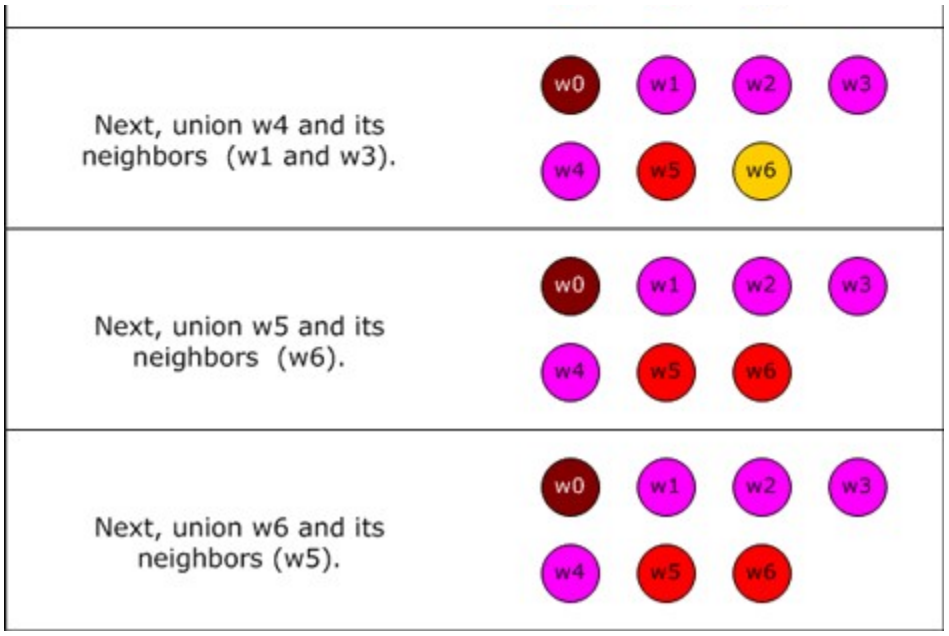


Figure 1. A graphical representation of an algorithm for grouping linked web pages.

Examining Figure 1, notice that in the end, there are three related partitions:

- w0
- w1, w2, w3, and w4
- w5 and w6

So, when a user clicks the "Similar Pages" link for w2, they would see links to w1, w3, and w4; clicking the "Similar Pages" link for w6 would show only a link to w5.

Notice that with this particular problem only one set operation is being performed—union. Furthermore, all of the Web pages fall into *disjoint sets*. Given an arbitrary number of sets, these sets are said to be *disjoint* if they share no elements in common. {1,2,3} and {4,5,6} are disjoint, for example, while {1,2,3} and {2,4,6} are not, because they share the common element 2. In all stages shown in Figure 1, each of the sets containing Web pages are disjoint. That is, it's never the case that one Web page exists in more than one set at a time.

When working with disjoint sets in this manner, we often need to know what particular disjoint set a given element belongs to. To identify each set we arbitrarily pick a *representative*. A representative is an element from the disjoint set that uniquely identifies that entire disjoint set. With the notion of a representative, I can determine of two given elements are in the same set by checking to see if they have the same representative.

A disjoint set data structure needs to provide two methods:

- **GetRepresentative(element):** this method accepts an element as an input parameter and returns the element's representative element.
- **Union(element, element):** this method takes in two elements. If the elements are from the same disjoint set, then `Union()` does nothing. If, however, the two elements are from different disjoint sets, then `Union()` combines the two disjoint sets into one set.

The challenge that faces us now is how to efficiently maintain a number of disjoint sets, where these disjoint sets are often merged from two sets into one. There are two basic data structures that can be used to tackle this problem—one uses a series of linked lists, the other collection of trees.

Maintaining Disjoint Sets with Linked Lists

In Part 4 of this article series we took a moment to look at a Quick Primer on Linked Lists. Recall that linked lists are a set of nodes that typically have a single reference to their next neighbor. Figure 2 shows a linked list with four elements.

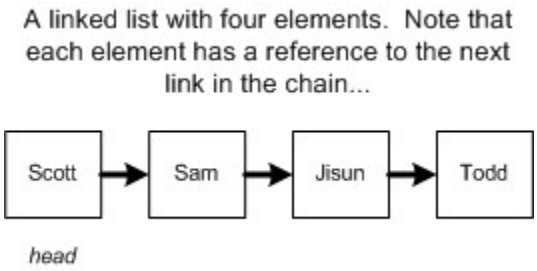


Figure 2. A linked list with four elements

For the disjoint set data structure, a set is represented using a modified linked list. Rather than just having a reference to its neighbor, each node in the disjoint set linked list has a reference to the set's representative. As Figure 3 illustrates, *all* nodes in the linked list point to the *same* node as their representative, which is, by convention, the head of the linked list. (Figure 3 shows the linked list representation of the disjoint sets from the final stage of the algorithm dissected in Figure 1. Notice that for each disjoint set there exists a linked list, and that the nodes of the linked list contain the elements of that particular disjoint set.)

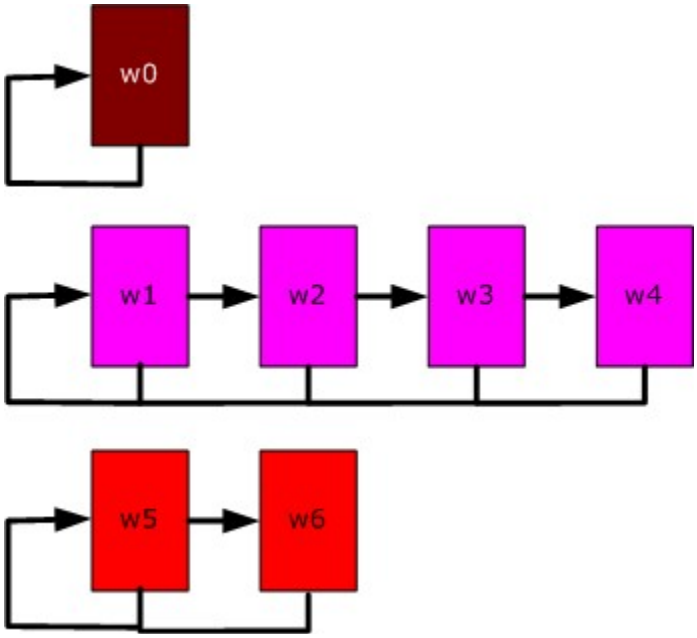


Figure 3. A linked list representation of the disjoint sets from the final stage of the algorithm dissected in Figure 1.

Because each element in a set has a direct reference back to the set's representative, the `GetRepresentative(element)` method takes constant time. (To understand why, consider that regardless of how many elements a set

has, it will always take one operation to find a given element's representative, because it involves just checking the element's representative reference.)

Using the linked list approach, combining two disjoint sets into one involves adding one linked list to the end of another, and updating the representative reference in each of the appended nodes. The process of joining two disjoint sets is depicted in Figure 4.

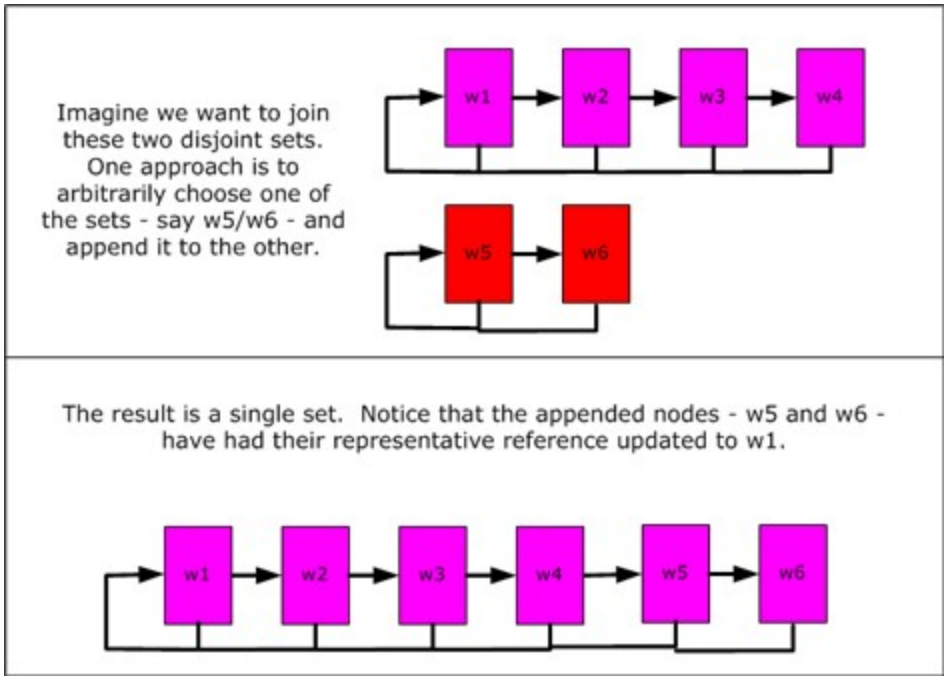


Figure 4. The process of joining two disjoint sets

When unioning together two disjoint sets, the correctness of the algorithm is not affected by which of the two sets is appended to the other. However, the running time can be. Imagine that our union algorithm randomly chose one of the two linked lists to be appended to the other. By a stroke of bad luck, imagine that we always chose the longer of the two linked lists to append. This can negatively impact the running time of the union operation since we have to enumerate all of the nodes in the appended linked list to update their representative reference. That is, imagine we make n disjoint sets, S_1 to S_n. Each set would have one element. We could then do n - 1 unions, joining all n sets into one big set with n elements. Imagine the first union joined S_1 and S_2, having S_1 be the representative for this two element unioned set. Since S_2 only has one element, only one representative reference would need to be updated. Now, imagine S_1—which has two elements—is unioned with S_3, and S_3 is made the representative. This time two representative references—S_1's and S_2's—will need to be updated. Similarly, when joining S_3 with S_4, if S_4 is made the representative of the new set, three representative references will need to be updated (S_1, S_2, and S_3). In the (n-1)th union, n-2 representative references will need to be updated.

Summing up the number of operations that must be done for each step, we find that the entire sequence of steps—n make set operations and n-1 unions—takes quadratic time—$O(n^2)$.

This worst-case running time can transpire because it is possible that union will choose the longer set to append to the shorter set. Appending the longer set requires that more nodes' representative references need to be updated. A better approach is to keep track of the size of each set, and then, when joining two sets, to append the smaller of the two linked lists. The running time when using this improved approach is reduced to $O(n \log_2 n)$. A thorough time

analysis is a bit beyond the scope of this article, and is omitted for brevity. Refer to the readings in the References section for a formal proof of the time analysis.

To appreciate the improvement of $O(n \log_2 n)$ from $O(n^2)$, observe Figure 5, which shows the growth rate of n^2 in blue, and the grown rate of $n \log_2 n$ in pink. For small values of n, these two are comparable, but as n exceeds 32, the $n \log_2 n$ grows much slower than n^2. For example, performing 64 unions would require over 4,000 operations using the naive linked list implementation, while it would take only 384 operations for the optimized linked list implementation. These differences become even more profound as n gets larger.

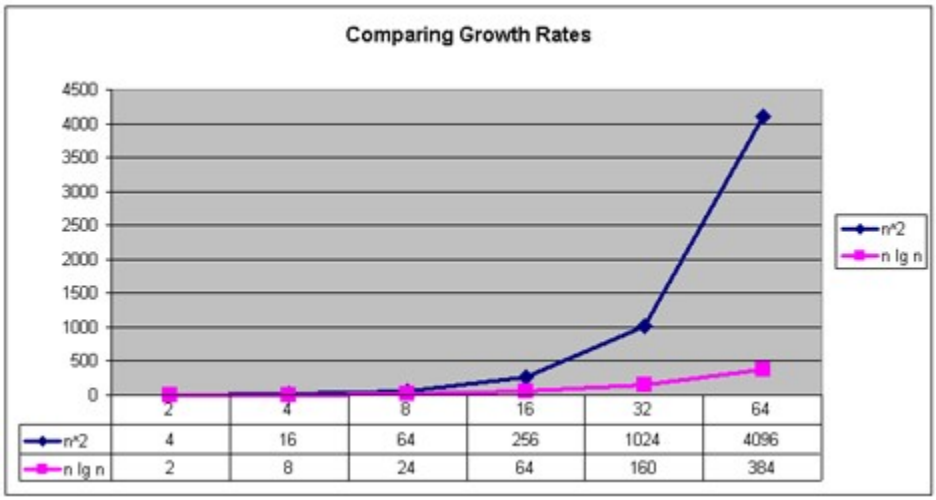


Figure 5. Growth rates of n^2 and $n \log_2$

Maintaining Disjoint Sets with a Forest

Disjoint sets can also be maintained using a *forest*. A forest is a set of trees (get it?). Recall that with the linked list implementation, the set's representative was the head of the list. With the forest implementation, each set is implemented as a tree, and the set's representative is the root of the tree. (If you are unfamiliar with what trees are, consider reading Part 3 of this article series, where we discussed trees, binary trees, and binary search trees.)

With the linked list approach, given an element, finding its set's representative was fast because each node had a direct reference to its representative. However, with the linked list approach unioning took longer because it involved appending one linked list to another, which required that the appended nodes' representative references be updated. The forest approach aims at making unions fast, at the expense finding a set's representative given an element in the set.

The forest approach implements each disjoint set as a tree, with the root as the representative. To union together two sets, one tree is appended as a child of the other. Figure 6 illustrates this concept graphically.

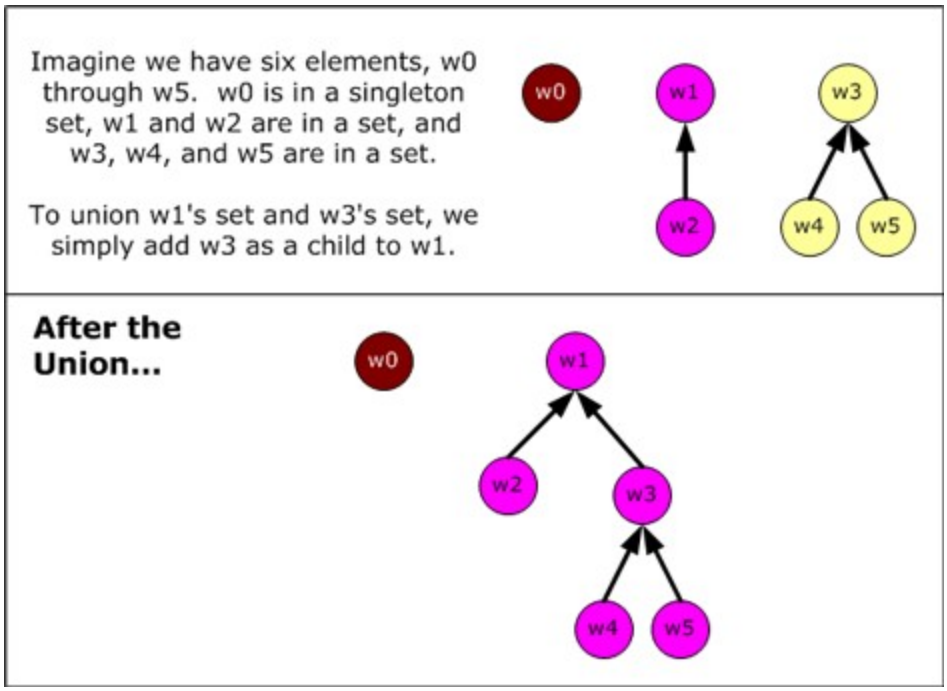


Figure 6. The union of two sets

To union two sets together requires constant time, as only one node needs to have its representative reference updated. (In Figure 6, to union together the w1 and w3 sets, all we had to do was have w3 update its reference to w1—nodes w4 and w5 didn't need any modification.)

Compared to the linked list implementation, the forest approach has improved the time required for unioning two disjoint sets, but has worsened the time for finding the representative for a set. The only way we can determine a set's representative, given an element, is to walk up the set's tree until we find the root. Imagine that we wanted to find the representative for w5 (after sets w1 and w3 had been unioned). We'd walk up the tree until we reached the root—first to w3, and then to w1. Hence, finding the set's representative takes time relative to the depth of the tree, and not constant time as it does with the linked list representation.

The forest approach offers two optimizations that, when both employed, yield a linear running time for performing *n* disjoint set operations, meaning that each single operation has an average constant running time. These two optimizations are called union by rank and path compression. What we are trying to avoid with these two optimizations is having a sequence of unions generate a tall, skinny tree. As discussed in Part 3 of this article series, the ratio of a tree's height to breadth typically impacts its running time. Ideally, a tree is fanned out as much as possible, rather than being tall and narrow.

The Union by Rank Optimization

Union by rank is akin to the linked list's optimization of appending the shorter list to the longer one. Specifically, union by rank maintains a rank for each sets' root, which provides an upperbound on the height of the tree. When unioning two sets, the set with the smaller rank is appended as a child of the root with the larger rank. Union by rank helps ensure that our trees will be broad. However, even with union by rank we might still end up with tall, albeit wide, trees. Figure 7 shows a picture of a tree that might be formed by a series of unions that adhere only to the union by rank optimization. The problem is that leaf nodes on the right hand side still must perform a number of operations to find their set's representative.

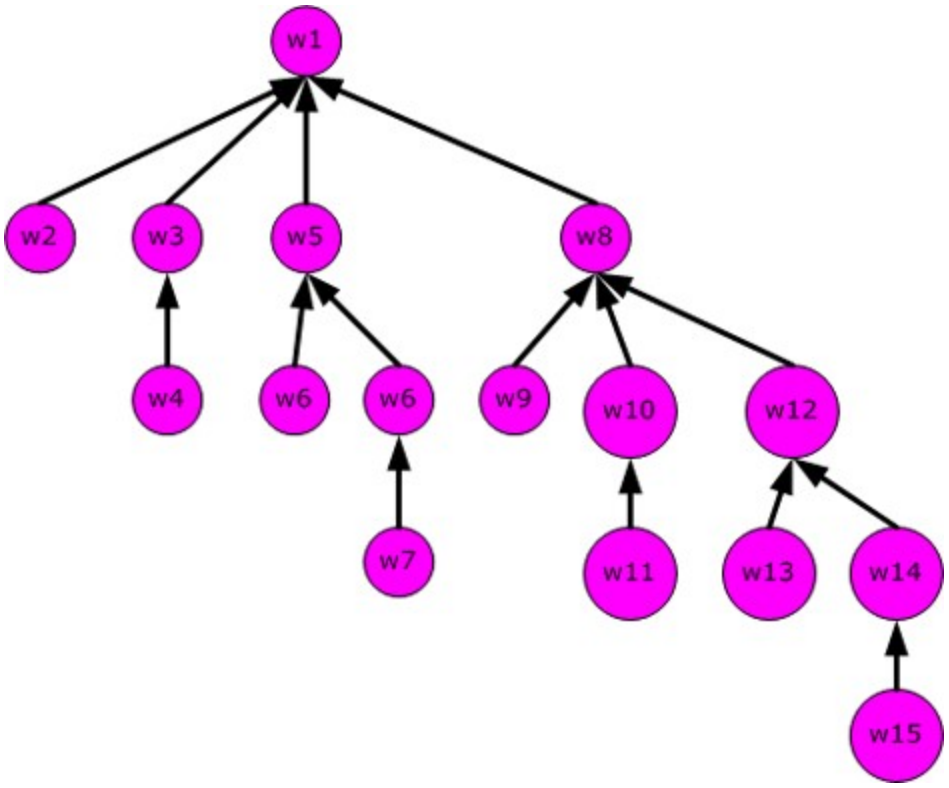


Figure 7. A tree that might be formed by a series of unions that adhere only to the union by rank optimization

> **Note** The forest approach, when implementing just the union by rank optimization, has the same running time as the optimized link list implementation.

The Path Compression Optimization

Because a tall tree makes finding a set's representative expensive, ideally we'd like our trees to be broad and flat. The path compression optimization works to flatten out a tree. As we discussed earlier, whenever an element is queries for its set's representative, the algorithm walk up the tree to the root. The way the path compression optimization works is in this algorithm, the nodes that are visited in the walk up to the root have their parent reference updated to the root.

To understand how this flattening works, consider the tree in Figure 7. Now, imagine that we need to find the set representative for w13. The algorithm will start at w13, walk up to w12, then to w8, and finally to w1, returning w1 as the representative. Using path compression, this algorithm will also have the side effect of updating w13 and w12's parents to the root—w1. Figure 8 shows a screenshot of the tree after this path compression has occurred.

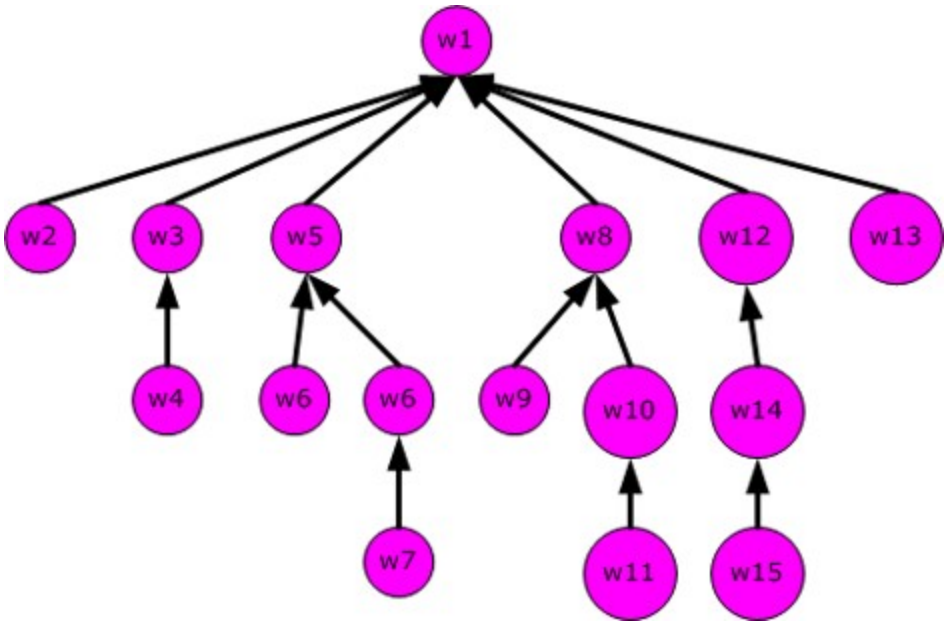


Figure 8. A tree after path compression

Path compression pays a slight overhead the first time when finding a representative, but benefits future representative lookups. That is, after this path compression has occurred, finding the set representative for w13 takes one step, because w13 is a child of the root. In Figure 7, prior to path compression, finding the representative for w13 would have taken three steps. The idea here is that you pay for the improvement once, and then benefit from the improvement each time the check is performed in the future.

When employing both the union by rank and path compression algorithms, the time is takes to perform *n* operations on disjoint sets is linear. That is, the forest approach, utilizing both optimizations, has a running time of *O*(*n*). You'll have to take my word on this, as the formal proof for the time complexity is quite lengthy and involved, and could easily fill several printed pages. If you are interested, though, in reading this multi-page time analysis, refer to the "Introduction to Algorithms" text listed in the references.

References

- Alur, Rajeev. "Disjoint Sets." Available online at: http://www.cis.upenn.edu/~cse220/h29.pdf.
- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.
- Devroye, Luc. "Disjoint Set Structures." Available online at: http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic24/.

Scott Mitchell, author of six books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies because January 1998. Scott works as an independent consultant, trainer, and writer, and holds a Masters degree in Computer Science from the University of California – San Diego. He can be reached at mitchell@4guysfromrolla.com, or via his blog at http://ScottOnWriting.NET.