Tutorial 3

Set ADT and Data Structure using Arrays

Lecturer: Dr Andrew Hines *TA: Esri Ni*

3.1 Set ADT

3.1.1 Definition

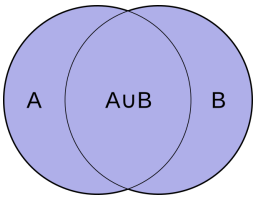
A set is an *unordered* collection of *distinct* objects.

- There are no duplicate elements in a set.
- There is no notion of a key or an order – elements are not organised in the set.

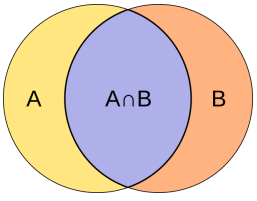
Unlike a sequence, we are concerned with the contents but not with the index of items, i.e. $S = \{1, 2, 3\}$ is equivalent to $S = \{2, 1, 3\}$. However we do not want duplicates, so $S = \{1, 2, 2\}$ would not be valid.
Sets are used a lot in data science/data analysis.

3.1.2 Set Operations

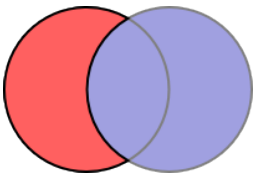
A reminder of set operations from your Venn diagrams math classes.
Union: Combine set A with the union of A and B, that is, $A \cup B$ is the combined (but without duplicate) elements of A and B.



Intersection: Intersection of A and B, that is, $A \cap B$ is the elements common to both sets.



Subtraction/Difference: Difference of A and B, that is, $A - B$, i.e. remove the elements common to both from A.



3.1.3 Example: Puzzle

A jigsaw puzzle is a set of puzzle pieces. Each piece is unique – there is only one of each piece in terms of the pattern/image and shape.
Let us imagine a jigsaw with a landscape photograph – green and brown fields and mountains below and blue and white cloudy sky above.
We can categorise our pieces by shape into groups edges and middle (non-edges). We can also categorise them into groups by colour – ground (green/brown) or sky (blue/white).
Let's assign these groups to sets.

$$E = \{Edges\}$$

$$M = \{Middle\}$$

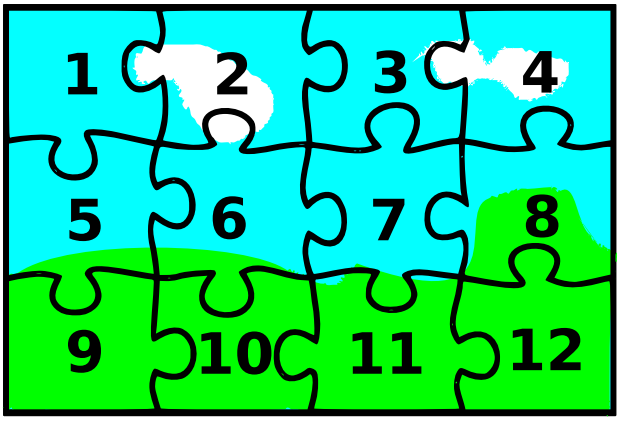
$$S = \{Sky\}$$

$$G = \{Ground\}$$

3.1.4 Exercise: Puzzle

For the jigsaw below put the puzzle piece indexes into the appropriate sets. If a piece is a mixture of colour take the greatest colour (e.g. piece 8 is in sets G and E).
Compute $E \cup S$, $E \cap S$ and $S - E$.

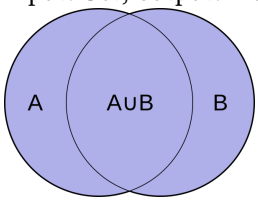
$E = \{1, 2, 3, 4, 5, 8, 9, 10, 11, 12\}$
$M = \{6, 7\}$
$S = \{1, 2, 3, 4, 5, 6, 7\}$
$G = \{8, 9, 10, 11, 12\}$
$E \cup S = \{1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 6, 7\}$
$E \cap S = \{8, 9, 10, 11, 12\}$
$S - E = \{6, 7\}$



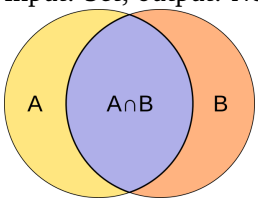
3.1.5 The Set Abstract Data Type

The ADT is defined with methods to support the key operations and a number of other support methods. You can think of it as a class definition. So the operations are from the perspective of one of the sets. This means, for example, that if you have a set A and you call the difference method, `A.difference(B)`, you will expect it to perform the operations on set A so take set A and replace the contents of A with $A - B$.
The set supports 3 fundamental methods:

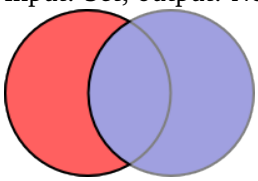
- union(s): Replace A with the union of A and B, that is, assign $A \cup B$ to A.
  Input: Set; output: None



- intersection(s): Replace A with the intersection of A and B, that is, assign $A \cap B$ to A.
  Input: Set; output: None



- subtraction(s)/difference(s): Replace A with the difference of A and B, that is, assign $A - B$ to A.
  Input: Set; output: None



These support methods should also be defined:

- size(): Return the number of objects in the set.
  Input: none; Output: integer
- is_empty(): Return a boolean value that indicates whether the set is empty.
  Input: none; Output: boolean
- contains(e): Return a boolean, true if the set contains the element.
  Input: element; Output: boolean
- add(e): adds the element e to the set, if not present already.
  Input: element; Output: none
- remove(e): remove the element e from the set, if present.
  Input: element; Output: none

3.1.6 Exercise: An Array-Based Set

In this exercise, you are asked to write the pseudo-code algorithms of the operations on the set ADT: `size` (example: solution provided), `add`, `remove`, `is_empty`, `union`, `intersection`, `difference`, assuming the set ADT is implemented using a dynamic (unbounded) array.

Algorithm 1 size

Input: A an array representing a set, n the size of the set
Output: the number of elements in the set (0 if empty)
  return n

Algorithm 2 add

Input: A an array representing a set, n the size of the set, e an element
Output: none - e is added to the set if not already present
?

Algorithm 3 remove

Input: A an array representing a set, n the size of the set, e an element
Output: none - e is removed to the set
?

Algorithm 4 is_empty

Input: A an array representing a set, n the size of the set
Output: $true$ if the set is empty
?

Algorithm 5 union

Input: A and B two arrays representing sets, n_A and n_B the size of the sets
Output: A gets the union of the two sets (elements contained in any set without duplicates)
?

Algorithm 6 intersection

Input: A and B two arrays representing sets, n_A and n_B the size of the sets
Output: A gets the intersection of the two sets (elements contained in both sets)
?

Algorithm 7 difference

Input: A and B two arrays representing sets, n_A and n_B the size of the sets
Output: A gets the difference between A and B (elements that are in A but not in B)
?

Solutions

The original, simple, naive (and not well written) solution is this one:

Algorithm 8 add

Input: A an array representing a set, n the size of the set, e an element
Output: none - e is added to the set if not already present

```
found ← false
for item ∈ A do
    if item = e then
        found ← true
    end if
end for
if found then
    add e to A
end if
```

It is naive because it does not optimise the comparisons (the for loop keeps going through the array even after the element has been found). It is not well written because it uses the function add which is not defined (well actually it is the one we're trying to define) and the $\in$ sign which is not clear/defined. An optimisation is this one (note that the output is different, a boolean, this time)

Algorithm 9 add (better)

Input: A an array representing a set, n the size of the set, e an element
Output: boolean - true if e is added to the set and false if e was already present

```
for i = 0 to n − 1 do
    if A[i] = e then
        return false
    end if
end for
increase the size of A (+1)
A[n] ← e
n ← n + 1
return true
```

Another option, if you don't want to change the output, would be to use a while loop instead of a for loop, with an extra condition. If the element has been found in the array ($found = true$) then the loop terminates otherwise you can safely add the element at the end of the array.

Algorithm 10 add (better with while)

Input: A an array representing a set, n the size of the set, e an element
Output: none - e is added to the set if not already present

```
found ← false
i ← 0
while i < n and found = false do
    if A[i] = e then
        found ← true
    end if
    i ← i + 1
end while
if found = false then
    increase the size of A (+1)
    A[n] ← e
    n ← n + 1
end if
```

Note how Algorithm 11 (remove) is made of only 1 single for loop (while a more simple algorithm would probably have 2: one to figure out whether the element is in the array and one to shift all the elements after the element up one position). Here the algorithm goes through all the elements: checks whether $A[i] = e$ and when this has been found ($found = true$) then all the elements are shifted: $A[i-1] \leftarrow A[i]$.

Algorithm 11 remove

Input: A an array representing a set, n the size of the set, e an element
Output: none - e is removed to the set

```
found ← false
for i = 0 to n − 1 do
    if found = true then
        A[i − 1] ← A[i]
    else
        if A[i] = e then
            found ← true
        end if
    end if
end for
if found = true then
    shrink A (-1)
    n ← n − 1
end if
```

Algorithm 12 size

Input: A an array representing a set, n the size of the set
Output: the number of elements in the set (0 if empty)
  return n

Algorithm 13 is_empty

Input: A an array representing a set, n the size of the set
Output: $true$ if the set is empty
  return $n = 0$

Note how the inner loop in the union algorithm (Algorithm 14) is a while with two conditions (one which checks whether the element has been found). The outer loop is a for as we want to got through ALL the elements of the set/array B.

Algorithm 14 union

Input: A and B two arrays representing sets, n_A and n_B the size of the sets
Output: A gets the union of the two sets (elements contained in any set without duplicates)

```
for i = 0 to n_B − 1 do
  found ← false
  j ← 0
  while j < n_A and found = false do
    if B[i] = A[j] then
      found ← true
    end if
    j ← j + 1
  end while
  if found = false then
    increase the size of A (+1)
    A[n_A] ← B[i]
    n_A ← n_A + 1
  end if
end for
```

Algorithm 15 (intersection) works very much like the union but the difference is that we might have to remove several elements in the set/array – hence the variable $shifts$ which keeps how many elements have been found

Algorithm 15 intersection

Input: A and B two arrays representing sets, n_A and n_B the size of the sets
Output: A gets the intersection of the two sets (elements contained in both sets)

```
shifts ← 0
for i = 0 to n_A − 1 do
    found ← false
    j ← 0
    while j < n_B and found = false do
        if A[i] = B[j] then
            found ← true
        end if
        j ← j + 1
    end while
    if found = false then
        shifts ← shifts + 1
    else
        A[i − shifts] ← A[i]
    end if
end for
n_A ← n_A − shifts
shrink A (-shifts)
```

Algorithm 16 is very similar to the previous one (intersection): only one condition differs: only if an element of A is **not** found is B we keep it.

Algorithm 16 difference

Input: A and B two arrays representing sets, n_A and n_B the size of the sets
Output: A gets the difference between A and B (elements that are in A but not in B)

```
shifts ← 0
for i = 0 to n_A − 1 do
    found ← false
    while j < n_B and found = false do
        if A[i] = B[j] then
            found ← true
        end if
        j ← j + 1
    end while
    if found = true then
        shifts ← shifts + 1
    else
        A[i − shifts] ← A[i]
    end if
end for
n_A ← n_A − shifts
shrink A (-shifts)
```