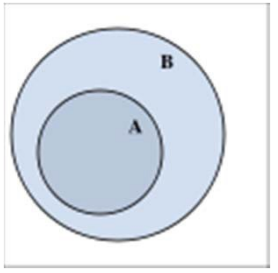
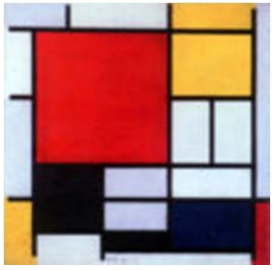
SCC120 Fundamentals of Computer Science
Unit 1: Abstractions and Sets



Jidong Yuan
yuanjd@bjtu.edu.cn



Relation to weeks 3-5 of SCC120 (data structures)

- in the previous material, you looked at the storage of data
- you looked at arrays, strings, objects, linked lists (chains)
- these linked structures are tools for organising collections of data
  - with different characteristics

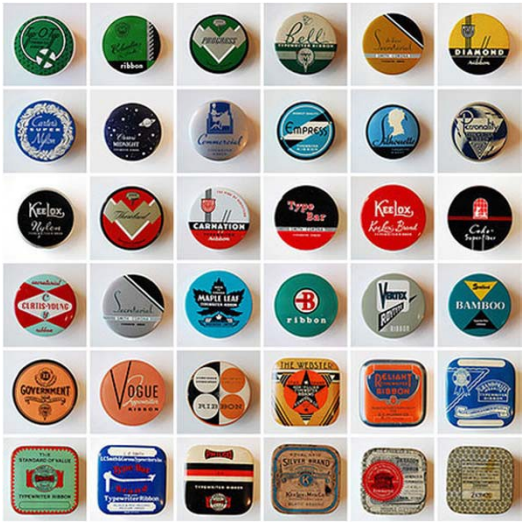
Learning Outcomes

At the end of this part of the course (weeks 5-10), you should be able to understand:

- general concept of abstract data types (ADT)
- characteristic operations and implementations of the stacks and queues data structure
- representations, features, and analysis of graphs and trees

Data Collections or Data Structures

- A data collection or data structure consists of a number of *elements* or *items*
- The elements could be numbers, characters, strings, or even themselves collections

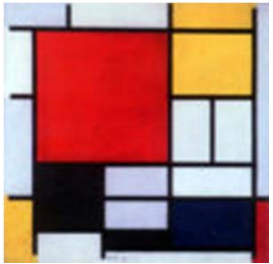


Abstraction

- Example: A queue is a type of data collection (similar to usual sense of a queue) can contain any type of data that we need; so we could have
  - a queue of integers
  - a queue of strings (for example, names of towns)
  - a queue of People objects
  - a queue of Button objects in a window
- but they all share the same queue properties
  - they are all “first in first out” collections

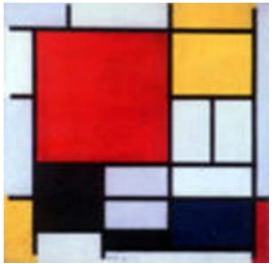
Abstraction

- so the idea of a queue is *abstract* or *general*
- we can think about a queue without worrying about what it contains
- this is called *data abstraction*



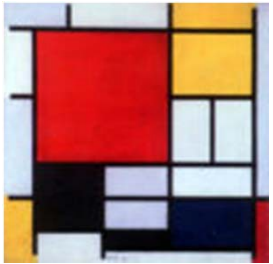
Abstraction

- abstraction is focussing on the essential logic of a data collection
- ignoring specific details of the elements it contains
  - that is, dealing with the details elsewhere, perhaps by creating a **class** to represent the elements



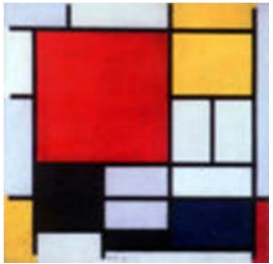
Abstract Data Types

- A particular class of data collection, which ignores the details of individual elements, is called an *Abstract Data Type* (ADT)
- so a queue is an Abstract Data Type
- We will concentrate on the *behaviour* of the ADT
  - for example, in a queue we can perform certain operations



Abstract Data Types

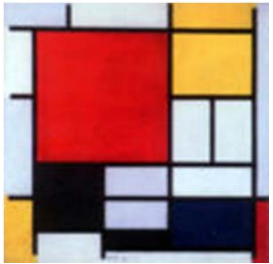
- We can organise things so that only the appropriate operations can be carried out on the ADT
- We can also try different internal organisations for the ADT
  - Examples: arrays, linked lists



Abstract Data Types

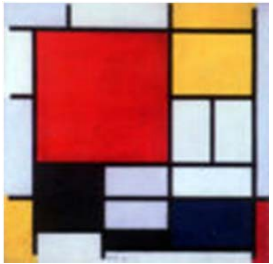
- “Being familiar with the main ADTs is a key to successful programming”

- We can use familiar techniques or existing (tested) tools for new problems



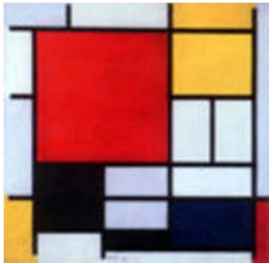
Static and Dynamic Collections

- a collection which is fixed in size is said to be *static*
- a collection whose size can change is said to be *dynamic*



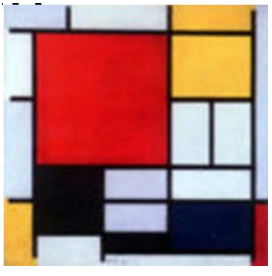
Dynamic ADTs

- There are two key operations
  - *add* or *insert* an element; the size is increased by one
  - *remove* or *delete* an element; the size is decreased by one



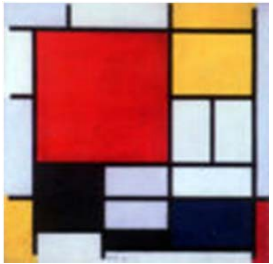
Other Operations

- There may be other operations to allow questions to be answered about the collection:
  - what is the current size of the collection; how many elements does it contain?
  - is the collection empty? (if so, we cannot remove an element)
  - is the collection full? (if so, we cannot add an element)
  - some data collections are essentially unlimited in size (we can always add another element)



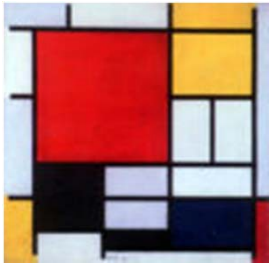
Other Operations

- Does the collection contain an element whose value is X?
  - or part of whose value is X, so we want to know the rest of its value
  - E.g., in a dictionary we look up a word and want the definition, or pronunciation, etc.

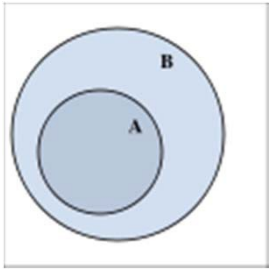


Some ADTs

- Set
- Stack
- Queue
- Graph
- Tree

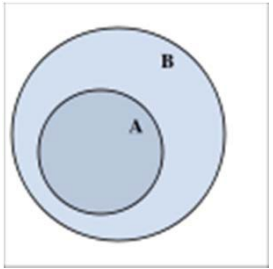


“Set” Abstract Data Type



The Set ADT

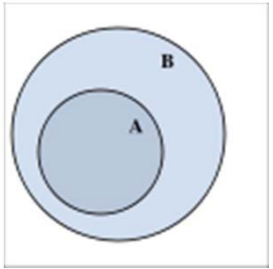
- Definition:
  - a collection in which no element is duplicated
  - unordered
- for example, a collection of numbers:
  - “3”, “5”, “10”
  - add “6”?
  - add “10” again?



Key Operations for a Set ADT

- *add* an element
  - fails if the element is already in t
- *delete* an element
  - fails if the element is not in the set
- *member* - does the set contain a specified element?
  - returns true or false
- *size* - how many elements are there in the set?
  - returns a non-negative number

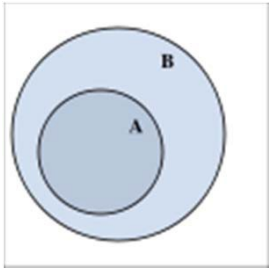
Think of these as maintenance operations in order to distinguish them from mathematical operations such as union, intersection etc.



Implementing a Set ADT

- We could use a linear array
- Or a linked list (a chain)

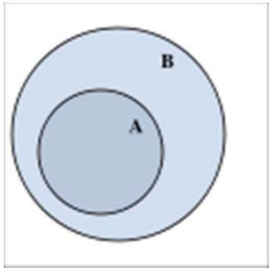
Remember: ADTs are not the same as the underlying data structures!



A Set ADT Using a Linear Array

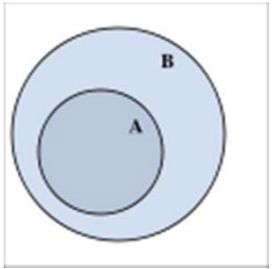
S

apple	big	cat	dog	null	null
0	1	2	3	4	5



The size Method

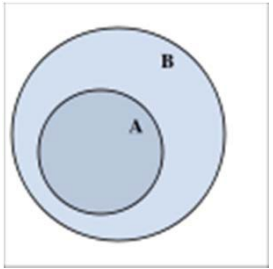
- What is the size of linear array?
- What is the size of Set ADT?
- How to get the size of Set ADT?



The size Method – Testing

- does this work for an empty array?
  - noElements = 0, i = 0
  - 0 < limit but S[0] == null; so leave loop
  - return 0

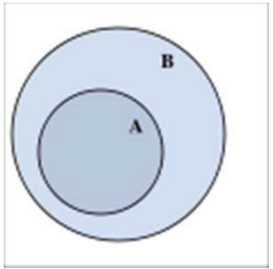
```
int noElements = 0 ;
int i = 0 ;
while ((i < limit ) && (S[i] != null))
    {
    noElements++ ;
    i++ ;
    }
return noElements ;
```



The size Method – Testing

- does this work for a part-full array?
  - say elements 0 to 3 occupied, 4 onwards null
  - go round the loop for i = 0, 1, 2, 3
  - then noElements = 4, i = 4
  - 4 < limit but S[4] == null; so leave loop
  - return 4

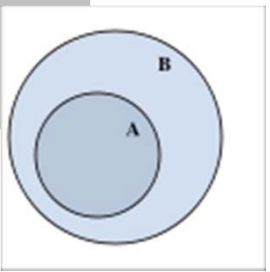
```
int noElements = 0 ;
int i = 0 ;
while ((i < limit ) && (S[i] != null))
    {
    noElements++ ;
    i++ ;
    }
return noElements ;
```



The size Method – Testing

- does this work for a full array?
  - go round the loop for i = 0, 1, 2, 3, 4, 5
  - then noElements = 6, i = 6
  - 6 == limit; so leave loop
  - return 6

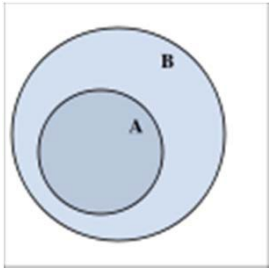
```
int noElements = 0 ;
int i = 0 ;
while ((i < limit ) && (S[i] != null))
    {
    noElements++ ;
    i++ ;
    }
return noElements ;
```



The size Method - Comments

- “&&” means if (i == limit), don’t check S[i]
- do we need separate variables i and noElements?

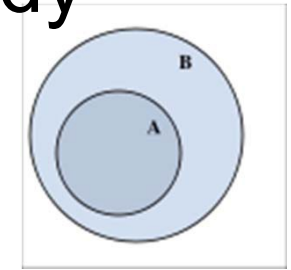
```
int noElements = 0 ;
int i = 0 ;
while ((i < limit ) && (S[i] != null))
    {
    noElements++ ;
    i++ ;
    }
return noElements ;
```



The add Method (DRAFT)

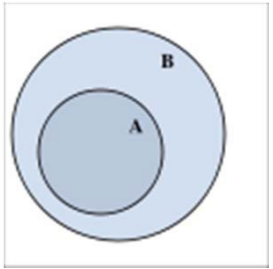
```
int i = 0 ;
while ((i < limit) && (S[i] != null))
    i++ ;
if (i == limit)
    PROBLEM - ARRAY FULL
else
    S[i] = X ;
```

- but we need to check for the element already being in the array



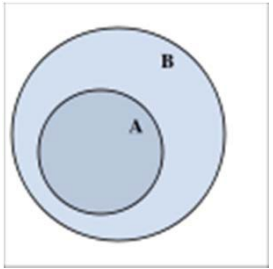
The add Method

```
int i = 0 ;
while ((i < limit) && (S[i] != null) && (S[i] != X))
        i++ ;
if (i == limit)
        PROBLEM - ARRAY FULL
else if (S[i] == X)
        PROBLEM - X ALREADY THERE
else
        S[i] = X ;
```



Handling Error Conditions

- What shall we do in the case of an error?
  - For example, the array is full when we want to add an element
    - output an error message (print on screen)
    - return an error signal (in function)



Adding ‘X = egg’

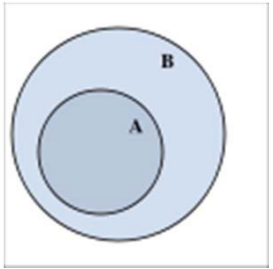
- i = 0
- go round the loop for i = 0, 1, 2, 3
- then i = 4
- (4 < limit) but (S[4] == null); so leave loop
- (4 < limit); so “if” fails
- (S[4] != egg); “if” fails
- “else” part : S[4] = egg

```
int i = 0 ;
while ((i < limit) && (S[i] != null)
          && (S[i] != X))
    i++ ;
if (i == limit)
    PROBLEM - ARRAY FULL
else if (S[i] == X)
    PROBLEM - X ALREADY THERE
else
    S[i] = X ;
```

Adding ‘X = egg’

S

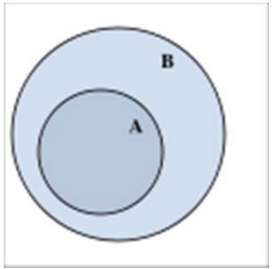
apple	big	cat	dog	egg	null
0	1	2	3	4	5



Adding ‘X = egg’

Test these yourself:

- does it work for an empty array?
- does it “work” for a full array?
- does it “work” if we try to insert “cat”?



The delete Method (first part is similar to “add”)

- int i = 0 ;

  while ((i < limit) && (S[i] != X) && (S[i] != null))

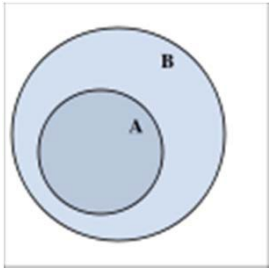
  i++ ;

  if (i == limit) || (S[i] == null))

  PROBLEM - NO SUCH ELEMENT

  else

  ...

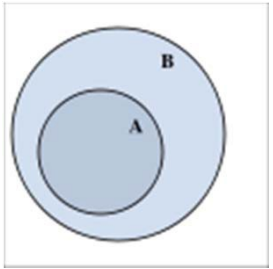


The delete Method (continued…)

```
{
        i++ ;
       while ((i < limit) && (S[i] != null))
        {
            S[i - 1] = S[i] ;
            i++ ;
        }
        S[i - 1] = null ;
    }
```

move remaining elements left by one place

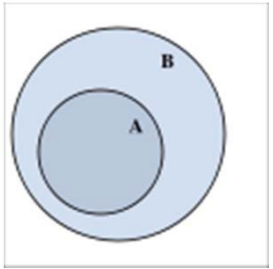
(an example is coming up)



The delete Method – Testing

S

apple	big	cat	dog	egg	null
0	1	2	3	4	5

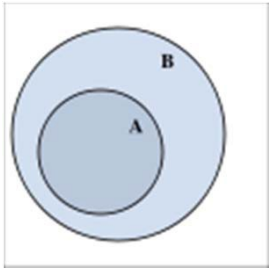


The delete Method – Testing

- suppose that we want to delete “cat” from the (updated) Set
- i = 0
- go round loop for i = 0, 1

```
int i = 0 ;
while ((i < limit) && (S[i] != X)
        && (S[i] != null))
    i++ ;
if (i == limit) || (S[i] == null))
    PROBLEM - NO SUCH…
else
```

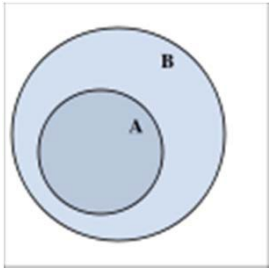
- then i = 2
- (2 < limit) but (S[2] == cat); so leave loop
- (2 != limit) and (S[2] != null); so “if” fails



The delete Method – Testing

- i = 3
- (3 < limit) and (S[3] != null), so S[2] = S[3] (i.e. dog), i = 4
- (4 < limit) and (S[4] != null), so S[3] = S[4] (i.e. egg), i = 5
- (5 < limit) but (S[5] == null); so leave loop
- S[4] = null

```
{
   i++ ;
   while ((i < limit) &&
            (S[i] != null))
   {
      S[i - 1] = S[i] ;
      i++ ;
   }
   S[i - 1] = null ;
}
```

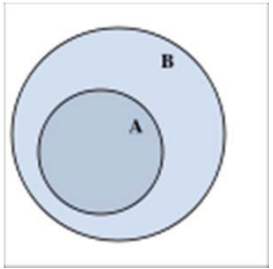


The delete Method – Testing

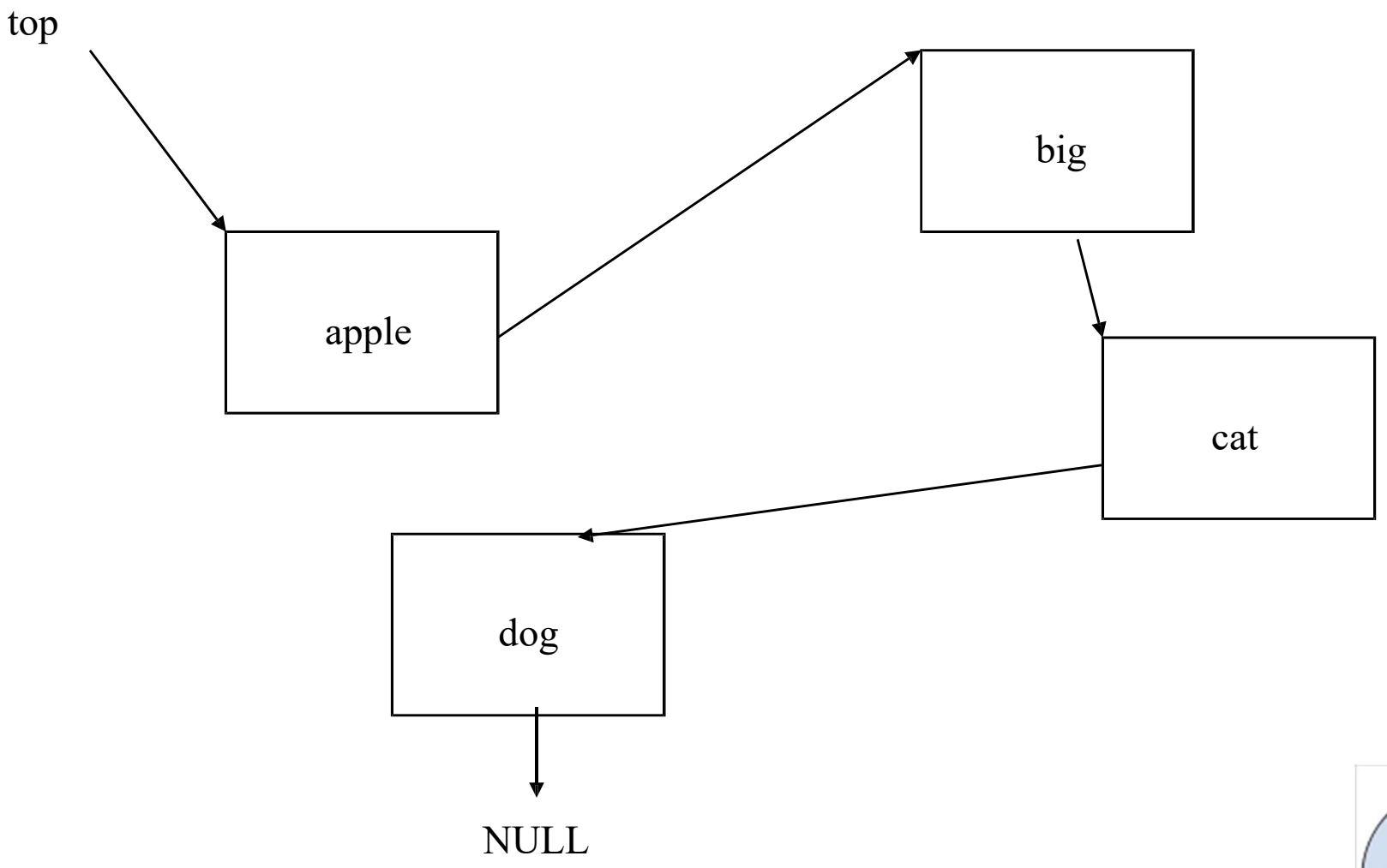
S

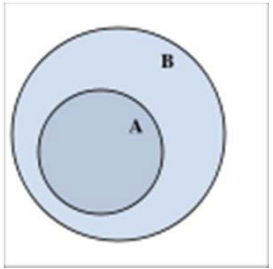
apple	big	dog	egg	null	null
0	1	2	3	4	5

moving elements left by one place is really inefficient – is there any way to improve this implementation?



A Set ADT Using a Linked List

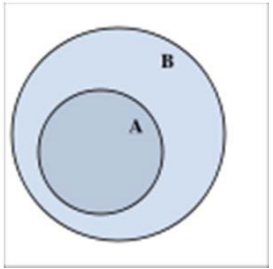




A Set ADT Using a Linked List

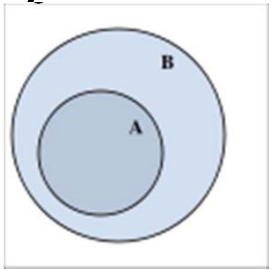
- We can:
  - check whether a linked list contains X
  - add the value X to linked list (to the front, or to the back) after checking that it is not already there
  - delete the element containing the value X

- So we can hold a Set in the form of a linked list



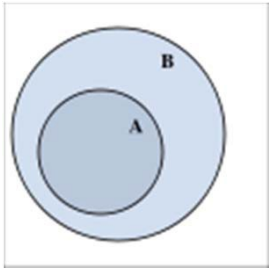
Efficiency of the Set Implementations

- Most operations require ...?
  - *size* to count the elements present
  - *add* to find where to insert the element
  - *delete* to find the element to be deleted
- These are all O(N) operations; that is, *linear* efficiency
  - the time increases in proportion to the amount of data
  - twice as much data means (about) twice as many times round the loop



Efficiency of the Set Implementations

- If we add to the front in the chain implementation, this would be a constant time O(1) operation
- But we need to check for duplication



Remember: ADTs are not the same as the underlying data structures!

Important Ideas

- Abstraction: an Abstract Data Type (ADT) is associated with its key operations (e.g. add, remove an element), and it does not need to know what kind of elements the ADT holds (e.g. numbers, strings)
- A “Set” is a type of ADT
- A “Set” can be implemented by an array or a linked list, and the implementation may affect the runtime of the operations

SCC120 ADT (weeks 5-10)

- Week 5 Abstractions; Set (key operations, implementations)
- Week 6
- Week 7
- Week 8
- Week 9
- Week 10