

1 Assorted ADTs

A **list** is an ordered collection, or *sequence*.

```
1 interface List<E> {  
2     boolean add(E element);  
3     void add(int index, E element);  
4     E get(int index);  
5     int size();  
6 }
```

A **set** is a (usually unordered) collection of unique elements.

```
1 interface Set<E> {  
2     boolean add(E element);  
3     boolean contains(Object object);  
4     int size();  
5     boolean remove(Object object);  
6 }
```

A **map** is a collection of key-value mappings, like a dictionary in Python. Like a set, the keys in a map are unique.

```
1 interface Map<K,V> {  
2     V put(K key, V value);  
3     V get(K key);  
4     boolean containsKey(Object key);  
5     Set<K> keySet();  
6 }
```

Stacks and queues are two similar types of linear collections with special behavior.

A **stack** is a last-in, first-out ADT: elements are always added or removed from one end of the data structure. A **queue** is a first-in, first-out ADT. Both data types support three basic operations: `push(E e)` which adds an element, `peek()` which returns the next element, and `poll()` which returns and removes the next element.

Java defines an interface that combines both stacks and queues in the Deque. A **deque** (double ended queue, pronounced “deck”) is a linear collection that supports element insertion and removal at both ends.

```
1 interface Deque<E> {  
2     void addFirst(E e);  
3     E removeFirst();  
4     E getFirst();  
5     void addLast(E e);
```

```

6     E removeLast();
7     E getLast();
8 }

```

Generally-speaking, a **priority queue** is like a regular queue except each element has a priority associated with it which determines in what order elements are removed from the queue. In Java, `PriorityQueue` is a class, a heap data structure implementing the priority queue ADT. The priority is determined by either natural order (`E implements Comparable<E>`) or a supplied `Comparator<E>`.

```

1 class PriorityQueue<E> {
2     boolean add(E e);
3     E peek();
4     E poll();
5 }

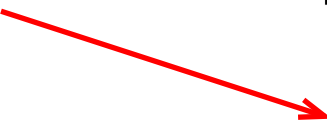
```

2 Use them!

- a. Given an array of integers A and an integer k , return true if any two numbers in the array sum up to k , and return false otherwise. How would you do this? Give the main idea and what ADT you would use.

I will use set. Firstly I will convert the array to a set (using the `Set()` method), then do a traversal from the 0th element of the original array, and checks whether $(k - \text{array}[i])$ is contained in the set. If so, return true; otherwise, continue the loop; end of loop, return false.

It is the classic two sum question. Instead of converting array to set, we can just build another set to store values. But I think the two approaches are almost the same.



```

public boolean twoSum(int[] A, int k) {
    Set<Integer> prevSeen = new HashSet<>();
    for (int i : A) {
        if (prevSeen.contains(k - i)) {
            return true;
        }
        prevSeen.add(i);
    }
    return false;
}

```

- b. Find the k most common words in a document. Assume that you can represent this as an array of Strings, where each word is an element in the array. You might find using multiple data structures useful.

I will use a map to store {words: frequency} pairs. Then I construct a priority queue with all the keys, associated with their frequency as the priority. For k most common words, I just deque the pq k times.

3 Mutant ADTs

- a. Define a Queue class that implements the push and poll methods of a queue ADT using only a Stack class which implements the stack ADT.

Hint: Try using two stacks

see discussion 5

- b. Suppose we wanted a data structure SortedStack that takes in integers, and maintains them in sorted order. SortedStack supports two operations: push(int i) and pop(). Pushing puts an int onto our SortedStack, and popping returns the next smallest item in the SortedStack. For example, if we inserted 10, 4, 8, 2, 14, and 3 into a SortedStack, and then popped everything off, we would get 2, 3, 4, 8, 10, 14.

```
import java.util.Stack;
import java.lang.Integer;

public class SortedStack<Item extends Comparable<Item>>{ // copied this line from solution

    7 usages
    private Stack<Item> stack1;
    4 usages
    private Stack<Item> stack2;

    1 usage
    public SortedStack() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    6 usages
    public void push(Item item) {
        // Step1: move all the small elements in stack1 to stack2
        while (!stack1.isEmpty() && stack1.peek().compareTo(item) < 0) {
            stack2.push(stack1.pop());
        }
        // Step2: push item to stack1
        stack1.push(item);
        // Step3: move all the elements in stack2 to stack1
        while (!stack2.isEmpty()) {
            stack1.push(stack2.pop());
        }
    }

    1 usage
    public Item poll() { return stack1.pop(); }
```

the smallest is at
the top at the stack