CS 61B Spring 2018

Abstract Data Types

Exam Prep 5: February 12, 2018

1 Assorted ADTs

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

```
interface List<E> {
    boolean add(E element);
    void add(int index, E element);
    E get(int index);
    int size();
}
```

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

```
interface Set<E> {
    boolean add(E element);
    boolean contains(Object object);
    int size();
    boolean remove(Object object);
}
```

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

```
interface Map<K,V> {
    V put(K key, V value);
    V get(K key);
    boolean containsKey(Object key);
    Set<K> keySet();
}
```

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.

```
interface Deque<E> {
    void addFirst(E e);
    E removeFirst();
    E getFirst();
    void addLast(E e);
```

```
    E removeLast();
    E getLast();
}
```

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>`.

```
class PriorityQueue<E> {
    boolean add(E e);
    E peek();
    E poll();
}
```

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.

The fastest way to do this is with the help of a set (specifically, a HashSet, which has constant time add() and contains()). The key insight is that if a + b = x, then b = x - a. This means that we can look to see whether or not x - (current element) has been seen already. We can store every element that we look through in a set, and do a single pass through the array.
Code:

```
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 least 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.

Keep a count of all the words in the document using a HashMap <String, Integer>. After we go through all of the words, each word will be mapped to how many times it's appeared. What we can then do is put all the words into a MaxPriorityQueue<String>, using a custom comparator that compares words

based on the counts in the HashMap. We can then pop off the k most common words by just calling poll() on the MaxPriorityQueue k times.

Code:

```
1         public static void topFivePopularWords(String[] words, int k) {
2             Map<String, Integer> counts = new HashMap<>();
3             for (String word : words) {
4                 if (!counts.containsKey(word)) {
5                     counts.put(word, 1);
6                 } else {
7                     counts.put(word, counts.get(word) + 1);
8                 }
9             }
10            PriorityQueue<String> pq = new PriorityQueue<>(new Comparator<String>() {
11                @Override
12                public int compare(String a, String b) {
13                    return counts.get(b) - counts.get(a);
14                }
15            });
16            for (String word : counts.keySet()) {
17                pq.add(word);
18            }
19            for (int i = 0; i < k; i++) {
20                System.out.println(pq.poll());
21            }
22        }
23
```

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

Have two stacks; one to hold all the items, and one to be a buffer for when we add something to our queue. Lets call the first stack A and the second stack B. Whenever we add something to our Queue, we pop everything off of A and push onto B. We then push our new item onto A, and then pop everything back off of B and onto A again. Thus, our most recent elements are on the bottom of A, and our oldest elements are on the top of A, mimicking the behavior of a queue.
Code:

```
1     public Deque<Item> {
2         private Stack<Item> a;
3         private Stack<Item> b;
4
5         public Deque() {
6             a = new Stack<>();
7             b = new Stack<>();
8         }
9
10        public void push(Item t) {
11            while (!a.empty()) {
12                b.push(a.poll());
13            }
14            a.push(t);
15            while (!b.empty()) {
16                a.push(b.poll());
17            }
18        }
19
20        public Item poll() {
21            a.poll();
22        }
23    }
24
```

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.

The solution to this is very similar to that of the question above. Once again, we will have two stacks, A and B. A will hold all the items, and B will be our buffer again. This time, when we add something to the queue, we continue to pop items off from A and push it onto B until the next item that will be popped (we can access this via peek()) is greater than or equal to the item we're adding to it. At that point, we can push our item onto A, and then pop everything from B and push them back into A. Thus, we maintain the sorted-ness of our SortedStack.

Code:

```
1   public class SortedStack<Item extends Comparable<Item>> {
2       private Stack<Item> a;
3       private Stack<Item> b;
4
5       public SortedStack() {
6           a = new Stack<>();
7           b = new Stack<>();
8       }
9
10      public void push(Item t) {
11          while (!a.empty() && a.peek().compareTo(t) < 0) {
12              b.push(a.poll());
13          }
14          a.push(t);
15          while (!b.empty()) {
16              a.push(b.poll());
17          }
18      }
19
20      public Item poll() {
21          return a.poll();
22      }
23  }
24
```