CS 61B Spring 2018 — Abstract Data Types

Discussion 5: February 13, 2018

1 Assorted ADTs

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

```
List
  add(element); // adds element to the end of the list
  add(index, element); // adds element at the given index
  get(index); // returns element at the given index
  size(); // the number of elements in the list
```

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

```
Set
  add(element); // adds element to the collection
  contains(object); // checks if set contains object
  size(); // number of elements in the set
  remove(object); // removes specified object from set
```

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

```
Map
  put(key, value); // adds key-value pair to the map
  get(key); // returns value for the corresponding key
  containsKey(key); // checks if map contains the specified key
  keySet(); // returns set of all keys in map
```

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)` 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.

```
Deque
  addFirst(e); // adds e to front of deque
  removeFirst();  // removes and returns front element of deque
  getFirst(); // returns front element of deque
  addLast(e); // adds e to end of deque
  removeLast(); // removes and returns last element of deque
  getLast(); // returns last element of deque
```

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.

```
PriorityQueue
  add(e); // adds element e to the priority queue
  peek(); // looks at the highest priority element, but does not remove it from the PQ
  poll(); // pops the highest priority element from the PQ
```

2 Solving Problems with ADTs

2.1 For each problem, which of the ADTs given in the previous section might you use to solve each problem? Which ones will make for a better or more efficient implementation?

(a) Given a news article, find the frequency of each word used in the article.

Use a map. When you encounter a word for the first time, put the key into the map with a value of 1. For every subsequent time you encounter a word, get the value, and put the key back into the map with its value you just got, plus 1.

(b) Given an unsorted array of integers, return the array sorted from least to greatest.

Use a priority queue. For each integer in the unsorted array, enqueue the integer with a priority equal to its value. Calling dequeue will return the largest integer; therefore, we can insert these values from index length-1 to 0 into our array to sort from least to greatest.

(c) Implement the forward and back buttons for a web browser.

Use two stacks, one for each button. Each time you visit a new web page, add the previous page to the back button's stack. When you click the back button, add the current page to the forward button stack, and pop a page from the back button stack. When you click the forward button, add the current page to the back button stack, and pop a page from the forward button stack. Finally, when you visit a new page, clear the forward button stack.

2.2 Java supports many built-in ADTs and data structures that implement these ADTs. But if we want something more complicated, we'll have to build it ourselves.

If you wish to use sorting as part of your design, assume that it will take $\Theta(N \log N)$ time where the length of the sequence is N.

(a) Suppose we want an ADT called `BiDividerMap` that allows lookup in both directions: given a value, return its corresponding key, and vice versa. It should also support `numLessThan` which returns the number of mappings whose key is less than a given key. Assume that there are no duplicate values.

```
BiDividerMap
  put(k, V); // put a key, value pair
  getByKey(K); // get the value corresponding to a key
  getByValue(V); // get the key corresponding to a value
  numLessThan(K); // return number of keys in map less than K
```

Describe how you could implement this ADT by using existing Java ADTs as building blocks. Come up with an idea that is correct first before trying to make it more efficient.

Create two maps, `Map<K, V>` and `Map<V, K>`. Note that when a client calls `put`, the implementation must add the mapping into each of the two component maps.

When `numLessThan` is called, get the list of keys, sort it, and then iterate until you find a key that's equal to or greater than the given key. Binary search could yield slightly better real-world runtime, but the order of growth will still be dominated by the time spent sorting the list.

To improve runtime, keep a sorted list of keys. Whenever you put a new key-value pair, insert the new key into the list such that the list remains sorted. When `numLessThan` is called and the key is in the `BiDividerMap`, return the index of the key in the sorted list; otherwise, use binary search to find the key that's equal to or greater than the key and return the index of that key. Another implementation would be to use a priority queue to keep values sorted.

(b) Next, Suppose we would like to invent a new ADT called `MedianFinder` which is a collection of integers and supports finding the median of the collection.

```
MedianFinder
  add(x); // adds x to the collection of numbers
  median(); // returns the median from a collection of numbers
```

Describe how you could implement this ADT by using existing Java ADTs as building blocks. What's the most efficient implementation you can come up with?

Use a list. When you add, just insert to the back of the list. When computing the `median`, first sort the list. Then figure out the size of the list and get the middle item. For a faster solution, use an integer value for storing the current median and two priority queues, one for integers less than the median and one for integers greater than the median. When adding an integer, add it to the appropriate priority queue. Each time an integer is added, maintain balance between the priority queues by shifting integers over as needed.

2.3 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: Consider using two stacks.

```java
public class Queue<E> {
    private Stack<E> stack = new Stack<E>();

    public void push(E element) {
        stack.push(element);
    }

    public E pop() {
        Stack<E> temp = new Stack<E>();
        while (!stack.isEmpty()) {
            temp.push(stack.pop());
        }
        E toPop = temp.pop();
        while (!temp.isEmpty()) {
            stack.push(temp.pop());
        }
        return toPop;
    }
}
```

It can also be done using only one stack instance and recursion (which itself is a *call stack*).

```java
public class Queue {
    private Stack<E> stack = new Stack<E>();

    public void push(E element) {
        stack.push(element);
    }

    public E pop() {
        return pop(stack.pop());
    }

    private E pop(E previous) {
        if (stack.isEmpty()) {
            return previous;
        }
        E current = stack.pop();
        E toReturn = pop(current);
        push(previous);
        return toReturn;
    }
}
```