CS 61B
Fall 2020

ADTs & Sorting
Discussion 12: November 9, 2020

1 Mechanical Sorting

Show the steps taken by each sort on the following unordered list:

```
0, 4, 2, 7, 6, 1, 3, 5
```

(a) Insertion sort

```
0 | 4 2 7 6 1 3 5
0 4 | 2 7 6 1 3 5
0 2 4 | 7 6 1 3 5
0 2 4 7 | 6 1 3 5
0 2 4 6 7 | 1 3 5
0 1 2 4 6 7 | 3 5
0 1 2 3 4 6 7 | 5
0 1 2 3 4 5 6 7 |
```

(b) Selection sort

```
0 | 4 2 7 6 1 3 5
0 1 | 2 7 6 4 3 5
0 1 2 | 7 6 4 3 5
0 1 2 3 | 6 4 7 5
0 1 2 3 4 | 6 7 5
0 1 2 3 4 5 | 7 6
0 1 2 3 4 5 6 | 7
0 1 2 3 4 5 6 7 |
```

(c) Merge sort

[0 4 2 7 6 1 3 5]
[0 4 2 7] [6 1 3 5]
[0 4] [2 7] [6 1] [3 5]
[0] [4] [2] [7] [6] [1] [3] [5]
[0 4] [2 7] [1 6] [3 5]
[0 2 4 7] [1 3 5 6]
[0 1 2 3 4 5 6 7]

(d) Use heapsort to sort the following array (hint: draw out the heap). Draw out the array at each step:

```
0, 6, 2, 7, 4
```

```
7 6 2 0 4 (turns the array into a valid heap)
6 4 2 0 7 ('delete' 7, then sink 4)
4 0 2 6 7 ('delete' 6, then sink 0)
```

```
2 0 4 6 7 ('delete' 4, then sink 2)
0 2 4 6 7 ('delete' 2)
0 2 4 6 7 ('delete' 0)
```

2 Abstract Data Types

Recall the following ADTs when answering this question:

```
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(); // returns the number of elements in the list
```

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

```
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
```

```
Stack
  peek();  // returns front element of stack
  pop(); // removes and returns front element of stack
  push(element); // adds element to front of stack
```

```
Queue
  peek();  // returns front element of queue without removing it
  poll(); // removes and returns front element of queue
  offer(element); // adds element to front of queue
```

```
PriorityQueue
  add(element); // adds element to the PQ
  peek(); // returns front element of PQ without removing it
  poll(); // removes and returns the highest priority element in the PQ
```

(a) 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?

1. 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.

2. 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.

3. 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.

(b) Define a `Queue` class that implements the `offer` and `poll` methods of a queue ADT using only a `Stack` class which implements the stack ADT.

Hint: Consider using two stacks.

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

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

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

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

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

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

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

```java

    private E poll(E previous) {
        if (stack.empty()) {
            return previous;
        }
        E current = stack.pop();
        E toReturn = poll(current);
        offer(previous);
        return toReturn;
    }
}
```