

---

CS 61B  
Spring 2019

---

ADTs & Sorting  
Discussion 12: April 15, 2019

# 1 Mechanical Sorting

1.1 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:

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

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

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

1 Stack
2   peek(); // returns front element of stack
3   pop(); // removes and returns front element of stack
4   push(element); // adds element to front of stack

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

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

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 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.

```

1  public class Queue<E> {
2      private Stack<E> stack = new Stack<E>();
3
4      public void offer(E element) {
5          stack.push(element);
6      }
7
8      public E poll() {
9          Stack<E> temp = new Stack<E>();
10         while (!stack.empty()) {
11             temp.push(stack.pop());
12         }
13         E toPop = temp.pop();
14         while (!temp.empty()) {
15             stack.push(temp.pop());
16         }
17         return toPop;
18     }
19 }

```

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

```

1  public class Queue {
2      private Stack<E> stack = new Stack<E>();
3
4      public void offer(E element) {
5          stack.push(element);
6      }
7
8      public E poll() {
9          return poll(stack.pop());
10     }
11
12     private E poll(E previous) {
13         if (stack.empty()) {
14             return previous;
15         }
16         E current = stack.pop();
17         E toReturn = poll(current);
18         offer(previous);
19         return toReturn;
20     }
21 }

```