Lecture 14: Basic Sorting

Review Exercises

1. How do you declare a balanced search tree based map in Java with key String and value Integer (i.e., create a variable of that type)? How do you declare a max heap in Java of Integers?

2. Give an algorithm for getting the $k$th largest element of a list of numbers.

3. Start with an empty AVL tree and add the following values: 1, 2, 3, 4, 5, 6, 7. What does the final AVL tree look like?

4. (Hard Interview Question) You are watching the price of a financial instrument over time. You have its price after each second over several days (given in an array $p$ of length $n$). At each second $s$ you want to know what its highest price was over the previous $k$ seconds. Output an array of size $n - k + 1$ of the various maxes. More precisely, the first entry should be the max of the prices with indices $0, \ldots, k - 1$ in $p$. The second entry will be the max of the prices with indices $1, \ldots, k$ in $p$, and so forth.

   public static double[][] maxes(double[] prices, int k)

   (a) (★★) Give a $\Theta(n \log k)$ implementation.
   (b) (★★★) Give a $\Theta(n)$ implementation.

Review Solutions

1. Tree-based map:

   TreeMap<K,V> map = new TreeMap<>();

   Max heap:

   PriorityQueue<Integer> pq =
       new PriorityQueue<>((Collections.reverseOrder()));

2. One algorithm maintains a min heap of size $k$. Each time we add the next element to the heap we removeMin to return the heap to size $k$. At the end the minimum element is the $k$th largest. This gives a runtime of $\Theta(n \log k)$ and uses $\Theta(k)$ memory.

   A faster algorithm is to make a max heap out of all of the numbers, and removeMax $k$ times. Runtime is $\Theta(n + k \log n)$ (since making the heap is $\Theta(n)$) but the space usage is $\Theta(n)$. We will see a different way to do this related to quicksort later.
4. (a) We keep a sliding window of width $k$. We also keep a max heap that stores all of the elements of the window. When we slide the window we must add the next price and remove the last price. To remove a specific item from the heap we require a special heap that gives us handles into it (like node references). This is sometimes called an augmented heap. When you add an element to the heap you are given a Handle object. The heap maintains an array of Handle objects, one for each heap element (that is, the data in the heap array has type Handle). Each Handle object has a value field, and an index field that says where the corresponding value lies in the heap array. Every time we swap in sift-up or sift-down we will update the corresponding handle indices. We then implement a remove method that takes a Handle and allows for $\Theta(\log k)$ removal, where $k$ is the heap size. If we want to support a safer interface, we can invalidate handles when the corresponding object is removed by setting the index to $-1$.

(b) This implementation is similar, but harder than our implementation of a stack that let’s us retrieve the max in $\Theta(1)$ time. We will maintain 2 structures, a queue of size $k$ that holds the sliding window, and a deque of maxes. When an element is added to the window, we check if it is larger than the front of the deque of maxes. While it is strictly larger than the front of the deque of maxes we remove the front of the deque. Then we add the new element to the front of the deque. When we remove an element from the sliding window, we check the back of the deque, and remove it from the back of the deque if it matches. Using this system two properties are maintained:

i. The deque of maxes is always in descending order from back to front.

ii. The back of the deque of maxes always has the maximum value in the sliding window.

The first property above gives this data structure/algorithm its name, the monotonic queue. Note that a single add can cause up to $k$ removals from the deque. That said, when processing the entire array we will remove at most $n$ values from the deque. Due to this amortized analysis we see the worst-case runtime is $\Theta(n)$. 
Other Trees

We will not be covering Tries or B-Trees in the course, but they are worth looking into if you are curious.

Basic Sorting: Selection, Insertion and Bubble Sort

Our first sort we will look at is selection sort. It works by finding the maximum element of an array, swapping it into the last position, and then repeats that process on the first \( n - 1 \) elements of the array. That is, it then finds the second largest element, and swaps it into the next to last position, and so forth.

We have already discussed the idea of insertion sort earlier in the class. It uses the concept of inserting a new element into an already sorted list. Given any list of numbers, if we focus on the sublist consisting only of its first element we trivially obtain a sorted list. Insertion sort then inserts the second element into this sorted list so that the first two elements form a sorted list. Then it inserts the third, and so forth.

Our final basic sorting algorithm is bubble sort. We give a naive implementation below:

```java
public static void swap(int[] arr, int a, int b) {
    int tmp = arr[a];
    arr[a] = arr[b];
    arr[b] = tmp;
}

public static void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length; ++i) {
        for (int j = 1; j < arr.length; ++j) {
            if (arr[j] < arr[j - 1]) swap(arr,j-1,j);
        }
    }
}
```

Basic Sorting Exercises

1. Consider the array \{4,3,2,1\}.
   
   (a) Show step-by-step what selection sort does on it.

   (b) Show step-by-step what insertion sort does on it.

   (c) Show step-by-step what bubble sort does on it.

2. (a) Implement `findMax` which returns the index of the largest element in an array between indices \( a \) and \( b \), inclusive.

   ```java
   public static int findMax(int[] arr, int a, int b)
   ```

   (b) Implement selection sort. What is the runtime?
public static void selectionSort(int[] arr)

(c) Implement insertion sort. What is the runtime?

public static void insertionSort(int[] arr)

3. Is there a way to greatly reduce the worst-case number of comparisons in insertion sort?

4. (⋆) Prove that bubble sort actually sorts.

5. Give a slightly faster implementation of bubble sort that shortens the loops above.

Basic Sorting Solutions

1. The steps are below.

   (a) Selection sort

   
   
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

   (b) Insertion sort

   
   
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
(c) Bubble sort

2. The implementations of all parts are given below.

Sorts.java

```java
import java.util.Arrays;

public class Sorts {
    public static void swap(int[] arr, int a, int b) {
        int tmp = arr[a];
        arr[a] = arr[b];
        arr[b] = tmp;
    }

    // Returns index of max value between indices //a and b, inclusive
    public static int findMax(int[] arr, int a, int b) {
        int maxI = a;
        for (int i = a + 1; i <= b; ++i)
            if (arr[i] > arr[maxI]) maxI = i;
        return maxI;
    }

    public static void selectionSort(int[] arr) {
        for (int i = arr.length - 1; i >= 1; --i)
            swap(arr, i, findMax(arr, 0, i));
    }

    // Inserts value at index i into the sorted array // in indices 0,...,i-1
    public static void insert(int[] arr, int i) {
    }
}
```
```java
public static void insertionSort(int[] arr)
{
    for (int i = 1; i < arr.length; ++i)
        insert(arr, i);
}

public static void bubbleSort(int[] arr)
{
    for (int i = 0; i < arr.length - 1; ++i)
        for (int j = 1; j < arr.length - i; ++j)
            if (arr[j - 1] > arr[j])
                swap(arr, j - 1, j);
}

public static void main(String[] args)
{
    int[] arr1 = {2, 4, 1, 5, 9, 7, 3, 10, 6, 8};
    int[] arr2 = Arrays.copyOf(arr1, arr1.length);
    int[] arr3 = Arrays.copyOf(arr1, arr1.length);
    selectionSort(arr1);
    insertionSort(arr2);
    bubbleSort(arr3);
    System.out.println(Arrays.toString(arr1));
    System.out.println(Arrays.toString(arr2));
    System.out.println(Arrays.toString(arr3));
}
```

Selection sort and insertion sort both have a $\Theta(n^2)$ worst-case runtime. In the best-case insertion sort can be $\Theta(n)$ but selection sort is always $\Theta(n^2)$.

3. Binary search for the insertion point.

4. After the $k$th iteration of the outer loop, the last $k$ elements of the array are in their correct positions.

5. Implemented above with the other sorts.