Session 7

Table of Contents

Linked Lists

Arrays have multiple drawbacks, not all of which are apparent at first sight.

1. Arrays have a fixed size. It is however not possible to always know the required size of the array at the very beginning.
2. To alleviate the above issue, arrays are often initialized with a larger size than required.
3. They are not suitable for fast insertions or deletions, since they involve shifting elements.

TL;DR:

1. Can fall short of space
2. Occupy too much space
3. Not good for fast random insertions/deletions

Linked lists solve these issues. A linked list simply consists of "nodes", where each node holds some data, and holds a pointer or reference to the next element in the list.

A typical simple Node class in Java may look like the following:

```
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
```

```
        this.next = null;
    }
}
```

Notice how one of the member variables in the class is a reference to another Node.

A fantastic visualization of various linked list operations can be found here.

Problems

1. https://www.hackerrank.com/challenges/print-the-elements-of-a-linked-list
2. https://www.hackerrank.com/challenges/insert-a-node-at-the-tail-of-a-linked-list
3. https://www.hackerrank.com/challenges/insert-a-node-at-the-head-of-a-linked-list
4. https://www.hackerrank.com/challenges/insert-a-node-at-a-specific-position-in-a-linked-list
5. https://www.hackerrank.com/challenges/delete-a-node-from-a-linked-list

Sorting

Bubble Sort

```
for i from 0 until n:
    for j from 0 until n-1:
        if a[j] > a[j + 1]:
            swap a[j] and a[j + 1]
```

Since you know that after k iterations, the last k integers are in their rightful positions, you can take advantage of this to speed up the algorithm a little bit:

```
for i from 0 until n:
    for j from 0 until n-i:
        if a[j] > a[j + 1]:
            swap a[j] and a[j + 1]
```

It is called *bubble sort* because the small values *bubble* their way to the top. It is also called *sink sort* because the large values *sink* to the bottom (end) of the array.

Colloquially, the individual elements in a bubble sort are also given nicknames. The smaller-valued ones are called turtles, since they slowly move to the left of the array. This is because in a single iteration, the smallest element will never move more than 1 step.
The larger valued ones, as you may have guessed, are called rabbits. This is because larger elements quickly make their way to the right of the array.

Selection Sort

```
for i from 0 until n:
    find the min in the range [i, n)
    swap it with the element at a[i]
```

Selection sort simply *selects* the smallest (or largest) element at every iteration, and puts it in its rightful place.

Selection sort also has the same property as bubble sort, in that in the first k iterations, the first k elements are in their final sorted order (and will never be moved again).

Insertion Sort

```
for i from 1 until n:
    tmp = a[i]

    j = i - 1

    while j >= 0 and tmp < a[j]:
```

```
        a[j + 1] = a[j]
        j = j - 1

    a[j + 1] = tmp
```

Insertion sort is typically useful when you already have a sorted array, and have to insert elements into it.

Merge Sort

```
def sort(arr):

    l = size(arr)

    if l == 1:
        return arr

    mid = (0 + l)/2

    leftCopy = arr[0, mid]
    rightCopy = arr[mid, l]

    # Recursive calls!
    leftSorted = sort(leftCopy)
    rightSorted = sort(rightCopy)

    return merge(leftSorted, rightSorted)

# Merges two sorted arrays
def merge(left, right):
    i = 0, j = 0
    ll = size(left)
    rl = size(right)

    sorted = list of size (ll + rl)

    k = 0

    while i < ll and j < rl:
        if left[i] < right[j]:
            sorted[k] = left[i]
            i++
        else:
            sorted[k] = right[j]
            j++

        k++

    while i < ll:
        sorted[k] = left[i]
        i++
        k++

    while j < rl:
        sorted[k] = right[j]
        j++
        k++

    return sorted
```

Merge sort is a divide-and-conquer sorting algorithm. At each iteration, it divides the array under consideration into two halves, and then recursively applied merge sort on each half. It then merges the two sorted arrays in linear time.

Quick Sort

```
# Note, this is an in place algorithm, it modifies the original array
def sort(a):
    p = partition(a)
    n = size(a)

    sort(a[0, p])
    sort(a[p, n-1])


def partition(a):
    n = size(a)
    pivot = a[n - 1]

    toSwap = 0

    for i from 0 until n - 1:
        if a[i] < pivot:
            swap a[toSwap] and a[i]
            toSwap = toSwap + 1

    swap a[n-1] and a[toSwap]
    return toSwap
```

At each iteration of a quick sort, the pivot element is put into its rightful place.

Again, some fantastic visualization of these algorithms can be found here and here.

A comparative visualization can be found here

Assignments

HackerRank

1. https://www.hackerrank.com/challenges/print-the-elements-of-a-linked-list-in-reverse
2. https://www.hackerrank.com/challenges/reverse-a-linked-list
3. https://www.hackerrank.com/challenges/compare-two-linked-lists
4. https://www.hackerrank.com/challenges/merge-two-sorted-linked-lists
5. https://www.hackerrank.com/challenges/get-the-value-of-the-node-at-a-specific-position-from-the-tail
6. https://www.hackerrank.com/challenges/delete-duplicate-value-nodes-from-a-sorted-linked-list
7. https://www.hackerrank.com/challenges/tutorial-intro
8. https://www.hackerrank.com/challenges/insertionsort1
9. https://www.hackerrank.com/challenges/insertionsort2
10. https://www.hackerrank.com/challenges/correctness-invariant
11. https://www.hackerrank.com/challenges/quicksort1
12. https://www.hackerrank.com/challenges/quicksort2
13. https://www.hackerrank.com/challenges/quicksort3
14. https://www.hackerrank.com/challenges/countingsort1
15. https://www.hackerrank.com/challenges/countingsort2
16. https://www.hackerrank.com/challenges/countingsort3
17. https://www.hackerrank.com/challenges/closest-numbers

Miscellaneous

1. In the directory sorting/problems, there are multiple sorting implementations. The main driver class is IntSorterTimer.java. Try running it, it will fail. Implement the various implementations, i.e. BubbleSorter.java, SelectionSorter.java, etc. Once you are confident your algorithms are complete, run the IntSorterTimer again. If your implementations are correct, you should ideally see something similar to this table:

Sort name/n	bubbleSort	selectionSort	insertionSort	mergeSort
1	0	0	0	0
10	0	0	0	0
100	1	0	0	1
1000	9	4	4	1
10000	279	91	24	5
100000	24091	3059	1378	28

While you may not see the exact same numbers as above, they should have the same "relative" performance differences, i.e. merge sort should be *insanely* faster than the others.

Tips: Instead of compiling each file separately, you can compile all of them with `javac *.java`

2. In the directory linkedlist/problems, there is an interface `ILinkedList.java`, and a class `Node.java`. Your job is to implement the interface, by making a class called `LinkedListImpl.java`, and filling out the definition of the Node class as well. Ensure that you handle every possible case for every possible operation, such as empty lists, single element lists, and so on. **Write your own driver class to test your implementation**

Tips:

1. Write the class one step (function) at a time, leaving all the others empty, and ensuring every function you write works with all edge cases. Do not fill out all the functions at your first go, because it will get difficult to debug if something is not working.
2. After you're done, look through the functions and check if you have copy pasted code from one function to another. Check if you have similar logic in more than two functions. Can you re-use other functions from the same class to reduce duplication?
3. When in doubt, use the rubber duck approach. When still in doubt, email me :)