A4 - Linked List (5+1 points)

All the required python files are in 'Ch7' directory. Find `PositionalList` class in `positional_list.py`.

A4-1 (1 point)

Describe in detail how to swap two nodes *x* and *y* (and not just their contents) in a singly linked list ***L*** given references only to *x* and *y* (You may use `L.head` for the reference to the head node of the ***L***). Repeat this exercise for the case when L is a doubly linked list. Which algorithm takes more time?

A4-2 (1 point)

Implement `__reversed__` method for the `PositionalList` class. This method is similar to the given `__iter__`, but it iterates the elements in *reversed* order.

A4-3 (1 point)

Modify `add_last` and `add_before` methods, only using methods in the set `{is_empty, first, last, before, after, add_after, add_first}`.

A4-4 (1 point)

Update the `PositionalList` to support an additional method `max()`, that returns the maximum element from a `PositionalList`. For example. for instance ***L*** containing comparable elements, you execute the function by calling `L.max()`.

A4-5 (1 point)

Update the `PositionalList` to support an additional method `find(e)`, which returns the position of the (first occurance of) element ***e*** in the list (or **None** if not found).

A4-Bonus (1 point)

Update the previous `find(e)` using recursion. Your method should not contain any loops. How much space does your method use in addition to the space used for ***L***?

A5 - Trees (7 points)

A5-1 (1 point)

Answer the following questions. You have to answer all of them correctly to get the point.

1. Which node is the root?
2. What are the internal nodes?
3. How many descendants does node `cs016/` have?
4. How many ancestors does node `cs016/` have?
5. What are the sibilings of node `homeworks/`?
6. Which nodes are in the subtree rooted at node `projects/`?
7. What is the depth of node `papers/`?
8. What is the height of the tree?

A5-2 (1 point)

Draw a binary tree ***T*** that simultaneously satisfies the followings:

- Each internal node ***T*** stores a single character
- A *preorder* traversal of ***T*** yields **EXAMFUN**
- A *inorder* traversal of ***T*** yields **MAFXUEN**

A5-3 (2 points)

Let ***T*** be a binary tree with *n* positions that is realized with an array representation ***A***, and let *f()* be the level numbering function of the positions of ***T***.

A *level numbering* function is a function that numbers the positino on each level of ***T*** in increasing order from left to right. For example, check this figure.

Give pseudo-code descriptions of each of the methods `root(), parent(p), left(p), right(p), is_leaf(p), is_root(p)`, when *p* is given as the current position in the array. Assume the numbering starts from 0.

A5-4 (3 points)

All the required python files are in 'Ch8' directory. Find `LinkedBinaryTree` class in `linked_binary_tree.py`. Implement a new method, `_delete_subtree(p)`, that removes the entire subtree rooted at position *p*, making sure to maintain the count on the size of the tree. What is the running time of your implementation?

A6 - Priority Queues, Heaps (8+2 points)

A6-1 (1 point)

How long would it take to remove the $\lceil \log n \rceil$ smallest elements from a heap that contains *n* entries, using the `remove_min` operation?

A6-2 (1 point)

What does each `remove_min` call return within the following sequence of priority queue ADT methods: `add(5,A), add(4,B), add(7,F), add(1,D), remove_min(), add(3,J), add(6,L), remove_min(), remove_min(), add(8,G), remove_min(), add(2,H), remove_min(), remove_min()`?

Note: Use only a pen and paper. Don't cheat by executing the code in a computer.

A6-3 (1 point)

An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a time stamp that denotes the time when the event will occur. The simulation program needs to efficiently perform the following two fundamental operations:

- Insert an event with a given time stamp (that is, add a future event).
- Extract the event with smallest time stamp (that is, determine the next event to process).

Which data structure should be used for the above operations? Why?

A6-4 (1 point)

Illustrate the execution of the selection-sort algorithm on the following input sequence: (22, 15, 36, 44, 10, 3, 9, 13, 29, 25). ***Note:*** Display the internal data in the priority queue for each step.

A6-5 (1 point)

Repeat A6-3 with the insertion-sort algorithm.

A6-6 (3 points)

In 'Ch9' directory, find `HeapPriorityQueue` class in `heap_priority_queue.py`. Implement `heappushpop` method and `heapreplace` methods, with semantics akin to the described methods for the `heapq` Python module as follow:

- `heappushpop(e)`: Push element e on and then pop and return the smallest item. The time is O(logn), but it is slightly more efficient than separate calls to push and pop because the size of the list never changes. If the newly pushed element becomes the smallest, it is immediately returned. Otherwise, the new element takes the place of the popped element at the root and a down-heap is performed.
- `heapreplace(e)`: Similar to heappushpop, but equivalent to the pop being performed before the push (in other words, the

new element cannot be returned as the smallest). Again, the time is O(logn), but it is more efficient than two separate operations.

A6-Bonus (2 points)

Implement an in-place heap sort algorithm, `heapsort(a)`, where **a** is a list of unsorted numbers before running the method, and **a** becomes a sorted list after the method. Follow these steps for implementing it (textbook p389).

If the collection C to be sorted is implemented by means of an array-based sequence, most notably as a Python list, we can speed up heap-sort and reduce its space requirement by a constant factor using a portion of the list itself to store the heap, thus avoiding the use of an auxiliary heap data structure. This is accomplished by modifying the algorithm as follows:

1. We redefine the heap operations to be a maximum-oriented heap, with each position's key being at least as large as its children. This can be done by recoding the algorithm, or by adjusting the notion of keys to be negatively oriented. At any time during the execution of the algorithm, we use the left portion of C, up to a certain index i−1, to store the entries of the heap, and the right portion of C, from index i to n−1, to store the elements of the sequence. Thus, the first i elements of C (at indices 0, . . . , i−1) provide the array-list representation of the heap.
2. In the first phase of the algorithm, we start with an empty heap and move the boundary between the heap and the sequence from left to right, one step at a time. In step i, for i = 1, . . . ,n, we expand the heap by adding the element at index i−1.
3. In the second phase of the algorithm, we start with an empty sequence and move the boundary between the heap and the sequence from right to left, one step at a time. At step i, for i = 1, . . . ,n, we remove a maximum element from the heap and store it at index n−i.

In general, we say that a sorting algorithm is ***in-place*** if it uses only a small amount of memory in addition to the sequence storing the objects to be sorted. The variation of heap-sort above qualifies as in-place; instead of transferring elementsout of the sequence and then back in, we simply rearrange them. We illustrate the second phase of in-place heap-sort in the following figure.

In [ ]:

```

```