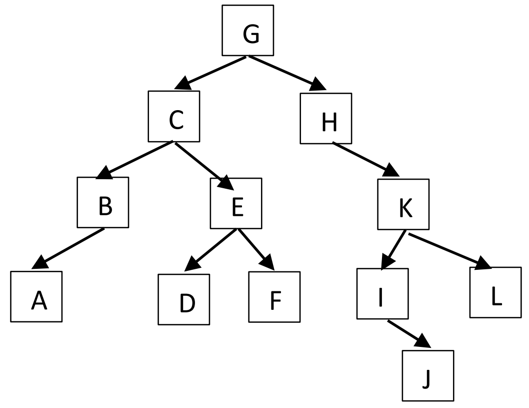
Exercise 1: **2.5 points**

Questions: 0.5 point for each correct answer; -0.25 for each incorrect answer.

For any question, CIRCLE the correct answer or leave empty.

1. In which order does a post-order traversal of the following tree visit the nodes?

	a) G C B A E D F H K I J L b) A B C D E F G H I J K L **c) A B D F E C J I L K H G** d) All of these are valid post-order traversals. e) None of these is a valid post-order traversal.
Suppose that we have numbers between 1 and 1000 in a Binary Search Tree and we want to search for the number 363. Which of the following sequences cannot be the sequence of nodes visited in the search?	(a) 2, 252, 401, 398, 330, 344, 397, 363 (b) 924, 220, 911, 244, 898, 258, 362, 363 (c) 2, 399, 387, 219, 266, 382, 381, 278, 363 **(d) 925, 202, 911, 240, 912, 245, 363** (e) 935, 278, 347, 621, 399, 392, 358, 363
Which of the following is the strongest valid statement made about AVL trees?	(a) They are minimal height trees (b) **They are height-balanced trees** (c) They are minimal height and height-balanced trees (d) They are full, minimal height, and height-balanced trees (e) None of the above are valid statements
Suppose an order m B-tree contains n items. In the worst-case, how many operations would be required to search the tree for a specific key?	(a) $O(\log_2 n)$ (b) **$O(\log_m n)$** (c) $O(m \log_2 n)$ (d) $O(m \log_2 m)$ (e) $O(m \log_m n)$
In an array-based implementation of a Heap, at what array location can the right-child of the right-child of the node at index i be found, if it exists?	**(a) 4i + 3** (b) 2i + 1 (c) 4i + 1 (d) 2i + 2 (e) 4i + 2

Exercise 2: **5.5 points**

1. Fill in the following table of best-case and worst-case runtime complexities of the indicating sorting algorithms. Use Big-Oh notation. **2.5 points**

Sorting Algorithm	**Best-case runtime**	**Briefly explain why and in which case best-case occurs**	**Worst-case runtime**	**Briefly explain why and in which case worst-case occurs**
Quick Sort	O(n logn)	Each time the pivot happens to partition each array (or subarray) intro equal-sized ones. In this case, it works like MergeSort.	$O(n^2)$	If the pivot is picked as the first element of the array and the data elements happen to be either sorted or reverse-sorted.
Merge Sort	O(n logn)	In any case. Mergesort partitions the data (array) elements into exactly two subarrays and recursively Mergesorts these, which is clearly done a logarithmic number of times. There is an additional constant-time for concatenating the resulting arrays at each step of the recursion.	O(n logn)	In any case. Mergesort partitions the data (array) elements into exactly two subarrays and recursively Mergesorts these, which is clearly done a logarithmic number of times. There is an additional constant-time for concatenating the resulting arrays at each step of the recursion.
Insertion Sort	O(n)	If the input is pre-sorted (or almost sorted), the running time is $O(N)$, because the test in the inner for loop always fails immediately	$O(n^2)$	It is the case when the data is rather random. A precise calculation shows that the number of tests in the inner loop is at most $p + 1$ for each value of p. Summing over all p gives a total of 2+3+4+…+N
Counting Sort	O(N+M) where N is the size of the *input* array and M is the size of the *count* array.	It is the case in all cases. The complexity is the same because no matter how the elements are placed in the array, the algorithm goes through them N+M times.	O(N+M) where N is the size of the *input* array and M is the size of the *count* array.	It is the case in all cases. The complexity is the same because no matter how the elements are placed in the array, the algorithm goes through them N+M times.
Radix Sort (which uses Counting Sort)	*O(d * (N+M))*, where d is the number of digits in the largest data item and hence the number of cycles; and *O(N+M)* is the time complexity of Counting Sort	In all cases, Radix Sort is repeated a number *d* of cycles and each time its cost is O(N+M) of Counting Sort	*O(d * (N+M))*, where d is the number of digits in the largest data item and hence the number of cycles; and *O(N+M)* is the time complexity of Counting Sort	In all cases, Radix Sort is repeated a number *d* of cycles and each time its cost is O(N+M) of Counting Sort

2. Consider the following array of data elements:

5	9	4	8	6	1	2	3	7	0

Apply Quick Sort to it, showing the various steps and details of the sorting process on the same array (in parallel) till the final output is reached. **2 points**

First Solution:

5	9	4	8	6	1	2	3	7	0

center = (0+9)/2 =4

Sorting (A{left] = 5 ; A[right}= 0 ; A[center] = 6) pivot =5

0	9	4	8	5	1	2	3	7	6

Swap (A[center], A[right)]

0	9	4	8	6	1	2	3	7	5

pivot =A[right] = 5

0	9	4	8	6	1	2	3	7	5
i								j	

1st swap (9,3)

0	3	4	8	6	1	2	9	7	5

2nd swap (8,2)

0	3	4	2	6	1	8	9	7	5

3nd swap (6,1)

0	3	4	2	1	6	8	9	7	5

Now j and i cross, so swap 6 with the pivot 5

0	3	4	2	1	5	8	9	7	6

Now QuickSort the left subarray containing 0,3,4,2,1,

A[Left]=0 ; Center = 0+4/2 =2 so A[center]= 4 ; A[right] = 1 sorting(0,1,4)

0	3	1	2	4	5	8	9	7	6

Swap (1,4)

0	3	4	2	1	5	8	9	7	6
i			j						

Pivot A[right] = 1

crossing i =1, j= 0 so so swap 3 with the pivot 1

0	1	4	2	3	5	8	9	7	6

Now QuickSort the right subarray containing 4,2,3

0	1	2	3	4	5	8	9	7	6

0	1	2	4	3	5	8	9	7	6

0	1	2	3	4	5	8	9	7	6

Now QuickSort the right subarray containing 8,9,7,6

A[Left]=8 ; Center = 6+9/2 =7 so A[center] = 9 ; A[right] = 6 sorting (8,9,6)

0	1	2	3	4	5	6	8	7	9

Swap(8,9)

0	1	2	3	4	5	6	9	7	8
						i		j	

pivot = 8 =A[right]

swap(9,7)

0	1	2	3	4	5	6	7	9	8

Crossing i , j : Swap (9,8)

0	1	2	3	4	5	6	7	8	9

Sorted

Second Solution:

5	9	4	8	6	1	2	3	7	0

center = (0+9)/2 =4

A[left] = 5 ; A[right]= 0 ; A[center] = 6) pivot =5

swap(pivot, A[right)]

0	9	4	8	6	1	2	3	7	5

Move the left bound to the right until it reaches a value greater than or equal to the pivot.
Move the right bound to the left until it reaches a value smaller than or equal to the pivot.

0	9	4	8	6	1	2	3	7	5
	i						j		

swap(9,3)

0	3	4	8	6	1	2	9	7	5
			i						j

swap(8,2)

0	3	4	2	6	1	8	9	7	5
				i	j				

swap(6,1)

0	3	4	2	1	6	8	9	7	5

Move the left bound to the right until it reaches a value greater than or equal to the pivot. i stops at 6
Move the right bound to the left until it reaches a value smaller than or equal to the pivot. j stops at 1
i and j have crossed so swap(6,5). Pivot is in its final position.

0	3	4	2	1	5	8	9	7	6

QuickSort the left subarray containing 0,3,4,2,1
Then QuickSort the right subarray containing 8,9,7,6
Let us start with QuickSort of 0,3,4,2,1

Select the pivot: center = (0+4)/2 =2 ; A[Left]=0 ; A[right] = 1; A[center]= 4 ➔ pivot = 1 .
pivot stays in its position

0	3	4	2	1	5	8	9	7	6

i j

Move the left bound to the right until it reaches a value greater than or equal to the pivot. i stops at 3
Move the right bound to the left until it reaches a value smaller than or equal to the pivot. j stops at 0
i and j have crossed, so swap(3,1). Pivot is in its final position.

0	1	4	2	3	5	8	9	7	6

Now QuickSort the left subarray containing 0.

So already sorted.

0	1	4	2	3	5	8	9	7	6

Now QuickSort the right subarray containing 4,2,3

pivot = 2

swap(2,3).

0	1	4	3	2	5	8	9	7	6

Move the left bound to the right until it reaches a value greater than or equal to the pivot. i stops at 4
Move the right bound to the left until it reaches a value smaller than or equal to the pivot. j stops at 1
i and j have crossed, so swap(4,2). Pivot is in its final position.

0	1	2	3	4	5	8	9	7	6

QuickSort the right subarray containing 4,3

pivot = 3

swap(3,4).

0	1	2	4	3	5	8	9	7	6

Move the left bound to the right until it reaches a value greater than or equal to the pivot. i stops at 4
Move the right bound to the left until it reaches a value smaller than or equal to the pivot. j stops at 2
i and j have crossed, so swap(4,3). Pivot is in its final position.

0	1	2	3	4	5	8	9	7	6

QuickSort the right subarray containing 4

So already sorted.

0	1	2	3	4	5	8	9	7	6

Let us now **QuickSort the right subarray containing 8,9,7,6** ➔ **pivot = 8**

swap(8,6)

0	1	2	3	4	5	6	9	7	8

Move the left bound to the right until it reaches a value greater than or equal to the pivot. i stops at 9
Move the right bound to the left until it reaches a value smaller than or equal to the pivot. j stops at 7
swap(9,7)

0	1	2	3	4	5	6	7	9	8

Move the left bound to the right until it reaches a value greater than or equal to the pivot. i stops at 9
Move the right bound to the left until it reaches a value smaller than or equal to the pivot. j stops at 7
i and j have crossed, so swap(9,8). Pivot is in its final position.

0	1	2	3	4	5	6	7	8	9

QuickSort the right subarray containing 6,7

pivot = 6

swap(6,7)

0	1	2	3	4	5	7	6	8	9

Move the left bound to the right until it reaches a value greater than or equal to the pivot. i stops at 7
Move the right bound to the left until it reaches a value smaller than or equal to the pivot. j stops at 5
i and j have crossed, so swap(7,6). Pivot is in its final position.

0	1	2	3	4	5	6	7	8	9

QuickSort the 1-element array containing 7, then QuickSort the 1-element array containing 9 yielding the sorted array

0	1	2	3	4	5	6	7	8	9

3. Apply Radix Sort (that uses Counting Sort) on the following array, **showing the details of the sorting process TILL THE OUTPUT OF THE FIRST CYCLE**. **1 point**

Input array:

34	421	12	25	132	5	601

Completing with left zeros, we see that 601 is the largest numbers with 3 digits. So Radix Sort would use 3 cycles. We will concentrate on the rightmost digit for the first cycle as stated in the exercise.

034	421	012	025	132	005	601

The max element is 5 so we will create a count array of size 5+1 = 6

The Cumulative Count array is:

0	1	2	3	4	5
0	2	4	4	5	7

After first decrements

0	1	2	3	4	5
0	1	3	4	4	6

After second decrements

0	1	2	3	4	5
0	0	2	4	4	5

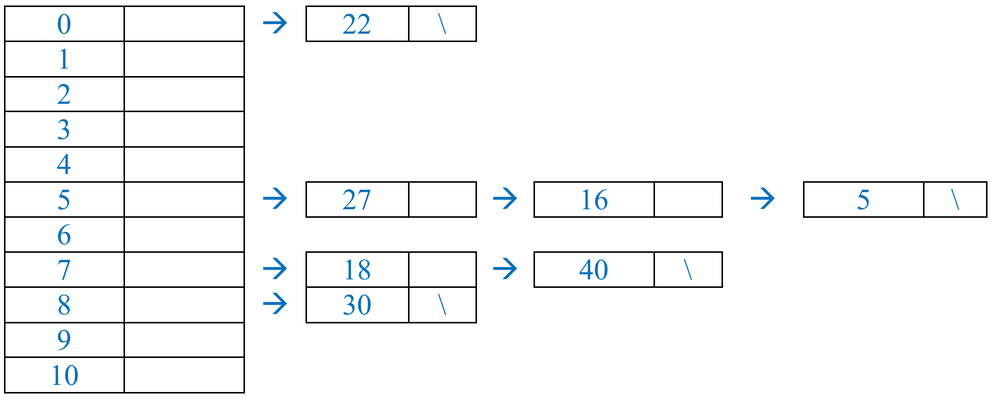
Output array after Cycle 1

601	421	132	012	034	005	025

<u>Exercise 3</u>: **3 points** **15 minutes**

Suppose you are given a table T of size 11 and a set S = {5, 40, 18, 22, 16, 30, 27} to hash into the table, using the hash function h(k) = k%11.

1. Show T after the values from S are entered into it using separate chaining. **0.5 points**



2. Show T after the values from S are entered into it using linear probing with the function h(k,i) = (h(k)+2i)%11. **0.5 points**

0	22
1	
2	16
3	
4	27
5	5
6	
7	40
8	30
9	18
10	

3. For the above two choices of T for S, which one is the better choice and why? Your answer should discuss the number of probes necessary to find a particular key. **1 point**

   Separate chaining is the better choice for this set S. The longest list in the separate chaining case contains 3 elements to search through, while linear probing requires as many as 6 probes to insert or find elements.

4. Give an example of a set S with 7 elements for which the other method (than what you answered in (c) is the better choice. **1 point**

   A good set for linear probing is one that has no collisions, e.g., {1, 2, 3, 4, 5, 6, 7}. In this case, the extra space required for the lists of separate chaining is a waste of memory, as a find is equally fast for either method.

Exercise 4: **5 points** **M-S HA: 40 minutes** **(AG: I would say at most 30 minutes)**

A priority queue implemented as a singly linked list is a data structure that allows elements to be added with associated priorities. We want to simulate a task scheduling system where tasks have different priorities: High (H), Medium (M), Low (L) using a priority queue implemented as a singly linked list, following specific rules:

i. Elements with higher priorities are served before those with lower priorities.
ii. Elements with the same priority follow the First-In-First-Out (FIFO) principle.
iii. Elements with higher priorities need to be at the head pointer of the list to be served first.

1- Consider six tasks with their associated priorities and order according to their time of arrival:
T1: L, T2: H, T3: M, T4: H, T5: M, T6: L
Here T1 came first and has "low" priority, then T2 came after it and has "high" priority, then T3 has "medium" priority, and so on.

- Draw the resulting linked list after adding all six tasks to the queue following the 3 specified rules for ordering tasks. The head of the list should be located on the left.

2- Provide the time complexity for the following operations in this singly linked list implementation:
a. Complexity of enqueue operation.
b. Complexity of dequeue operation.
c. If the queue is not ordered what is the complexity of ordering it (mention the used sorting algorithm).
d. Complexity of checking if a given priority queue is correctly ordered based on priority and the FIFO principle (respects the above three mentioned rules).

3- Is there a way to improve the enqueue operation complexity by using a different data structure? Justify your answer?

Solution:

1- Drawing the linked list: **1.5 points**

The linked list representation after adding the six tasks while following the three specified rules for ordering tasks is as follows, with the head on the left:

[T2:H] -> [T4:H] -> [T3:M] -> [T5:M] -> [T1:L] -> [T6:L]

2- Time Complexity Analysis:

a. Complexity of enqueue operation: The time complexity of the enqueue operation in this singly linked list implementation is O(n) in the worst case, where n is the number of elements in the linked list. This is because, in the worst case, you may need to traverse the entire list to find the correct position to insert the task. **0.5 points**

b. Complexity of dequeue operation: The time complexity of the dequeue operation is O(1) because it always involves removing the element at the head of the list, which is a constant-time operation. **0.5 points**

c. If the queue is not ordered, the complexity of ordering it depends on the sorting algorithm that you choose. In the worst case, algorithms such as bubble sort, selection sort, and insertion sort have a time complexity of O(n^2), while more efficient sorting algorithms like Heapsort or MergeSort would have a time complexity of O(n log n). **0.5 points**

d. The complexity of checking if a given priority queue is correctly ordered based on priority and the FIFO principle can be done in O(n), where n is the number of elements in the queue. You need to traverse the entire queue to ensure it adheres to the specified rules. **0.5 points**

3- Improving Complexity: **1.5 points**

Yes, there are ways to improve the enqueue operation complexity. Using a different data structure, such as a self-balancing binary search tree (e.g., AVL tree), it can be can done in O (log n) time complexity. These data structures are specifically designed for efficient priority queue operations and provide better performance compared to a singly linked list.

<u>Exercise 5:</u> **4 points** **20 minutes**

1. In this question you will implement the *rotateLeft* member function for the AVLTree class. Here is the partial AVLTree class defnition:

```
class AVLTree {
public:
    ...
private:

    class AVLTreeNode {
    public:
        ...
        int element;
        int height;
        AVLTreeNode* left;
        AVLTreeNode* right;
    };
     int height(AVLTreeNode const * x) const;
     ...
     // function declarations for rotations go here.
     …
};
```

The function *height( )* takes an AVLTreeNode pointer argument and returns the integer height of the subtree rooted at the parameter node. Each of the rotation functions takes a reference to an AVLTreeNode pointer, performs its rotation, and returns nothing.
Using this information, implement the *rotateLeft* member function. **2 points**

One solution is given below (other solutions are also possible):

```
void AVLTree::rotateLeft(AVLTreeNode * & node) {
// No need to check for conditions, as they are supposed to be checked before calling this function
        AVLTreeNode* y = node;
        node = node->right;
        y->right = node->left;
        node->left = y;
        y->height = height(y);
}
```

2. Prove by induction that the number of null pointers in an n-node AVL tree is n + 1. **2 points**

Let T be an arbitrary AVL tree with n nodes. Assume (Inductive Hypothesis) that any AVL tree with $j < n$ nodes has $j + 1$ null pointers.

<u>Base Case (n = 0):</u> Suppose T is a tree with zero nodes (an empty tree). Then, since we represent T using one null pointer, the number of null pointers is indeed one more than the number of nodes.

<u>Inductive Case (n > 0):</u> By definition of AVL trees, T's left and right children, TL and TR, are AVL trees. Denote by a and b the number of nodes in TL and TR respectively. Since $a < n$ and $b < n$ we can apply the inductive hypothesis to TL and TR, and thus TL has a+1 null pointers and TR has b+1 null pointers. The total number of null pointers in T is (a+1)+(b+1), and this is n+1 since the total number of nodes is n = a+b+1.