TO-DO LISTS

ABSTRACT

My project is based on developing a tool that will help a student to prioritize their to-do list. This is to solve the problem where students are unable to keep track of deadlines of the many tasks they have to do as well as they face difficulty in objectively deciding which tasks are more important to complete before others.

The app should enable the student to initially enter categories and set their importance in terms of ranking. The student will then be able to enter tasks into the list and the app will be able to sort their to-do list in order of priority based on their category and deadline. This is helpful as it will only allow students to be objective than subjective in creating their schedules as well as completing tasks. Ultimately giving them better time management.

PROBLEM ANALYSIS

Description:

The main problem a student faces when managing their time is poor prioritization. This leads them to focus on low priority tasks more and end up either missing deadlines or having little time for high priority tasks. The existing to-do apps enable them to track items but do not prioritize the tasks for the students. Also, they are mostly accessible online which is inconvenient for a frequent task.

Breakdown:

These are the main constraints the tool should meet:

- The app should be easily accessible to the student.
- The app should be able to prioritize the student’s tasks.
- The app should be fast in updating the to-do list once an item is added or removed.

PROBLEM INTERPRETATION

Data Objects Needed: For this app, the objects used will be the task, category, and storage objects. This is because to-do lists are used by an individual and are specific to them, there is no need to store a user object.

The Task Object will be used to store attributes of the task to be tracked. These are:

Title: This is the name of the task to be tracked.

Category: This is the type of task that is to be tracked, for example, assignment, reading, hobby, leadership, or part-time.
Deadline: This is the date when the task is to be completed.

The Storage Object which has only the size attribute.

The Category Object will be used to store the different categories of the task being tracked. The attributes are:
Name: This is the title of the category
Rank: This will be the rank of the category based on the user input.

Data Object Details:

Relationships

- Each instance of the Task object and Category object will be added to the Storage object from where it will be accessed from.
- Instances of the task object with the same status will be stored together.
- Each task object will have an attribute of category which will be an instance of the category class.

Operations Done Often:

- **Add:** Inserts an item to the list.
- **Delete:** Removes an item from the list.
- **Print:** Print list in order of priority.

Collective Data Structures Needed:
A data structure will be needed for each of the objects.
The data structure should be able to meet the following considerations:

- It should be easy to implement.
- It should be able to hold relationships where needed.
- It should take into account the order of data.
- It should also allow the primitive data types to be used.
- It should be space-efficient.

SOLUTION

Description:
A desktop application that uses a GUI interface to enable the student to keep track of their to-do list. The app will use a CSV file for storage.

Justification
The app will be easily accessible to a student and does not require internet access to run.

Using a GUI will create a seamless user experience for the student and the app can easily be pinned as a favorite on the taskbar such that it's easily visible.

The storage enables the app to run quickly as well as optimize storage that is easily accessible on the computer.

Reading and storing data to the CSV will be O(n) time.

IMPLEMENTATION

Primitive Data Types:

Attribute	Data Type
Title	String
Category	String
Deadline	Integer in format MMDD
Name	String
Rank	Integer
Weight - Derived Attribute	Integer
Category	Object
Task	Tuple

Data Structures:

Based on the constraints, the best abstract data type would be a priority queue because:

- Items will be dequeued from the list based on their priority. In this case, we will create a metric weight that will be a method of the task class that will add the rank and weight to create the key for the priority queue.

The priority queue can be implemented using several underlying data structures that were analyzed as follows:

Data Structure	**Ordering**	**Sorting**	**Speed**	**Ease of implementation**
Binary Heap	Items are inserted and deleted from the heap while maintaining the heap invariant.	Uses a heap sort O(n log n). Takes in tuples	Insert and delete are O(log n) Get max O(1) Print O(n log n)	Easy to implement
Binary Search Tree (AVL)	Items are sorted as they are inserted and deleted.	O(n log n)	Insert and delete are O(log n) Get max O(1) if pointers are stored. Print O(n)	Complex to implement the self-balancing and pointers. Also, more constraints.

Based on the analysis above, the Binary Heap and AVL are quite similar in their implementation of a priority queue. I chose to use a binary heap making a trade-off for the time taken to print for the ease of implementation. Also, binary heaps allow tuples as inputs allowing the task weight and details to be added to the list for printing.
A min-heap is the most suitable as it will dequeue elements with a lower weight meaning they have a higher priority as they will have a sooner deadline as well as a higher rank. An example is shown below:

Task	**Deadline (MMDD)**	**Category.rank**	**Weight**
Dsa assignment	0512	1	512.1
Finish painting	0512	3	512.3
Read for quiz	0513	2	513.2

The priority of the tasks above will be in the order: dsa assignment, finish painting, read for quiz
Even though the rank for reading is higher than that of a hobby, the deadline is farther and thus the student can do the hobby before with ease.

Limitations
The limitation of using the binary heap is the sort stability. Items of the same priority may not necessarily be returned in order of the addition to the list.
Items can only be removed from the top of the queue. Therefore, to remove an item that is not at the top of the list, one has to complete all items before it on the list.
This, however, can be an advantage as it will restrict the student from completing tasks that are not urgent before those that are urgent.

Methods

Method	**Description**	**Speed**
`def isEmpty(self):`	Checks if the priority queue is empty	O(1)
`def isLeaf(self, pos):`	Checks if the node at the position passed has any children	O(1)
`def swap(self, a, b):`	Swaps the nodes at the positions passed	O(1)
`def heapify(self, pos):`	Takes in the index of an element and swaps with the lesser child if the node is not a leaf	O(log n)
`def insert(self, node):`	Inserts a node into the priority queue while maintaining the heap invariant	
`def delete(self):`	Removes a node from the priority queue while maintaining the heap invariant	O(log n)
`def print(self):`	Prints the priority queue in order of the highest priority.	

CONCLUSION

Does it solve the problem?
Yes, it does. A student can now objectively organize their time based on the priority of the tasks they have to do.

Does the code run smoothly and user friendly?
Yes, the code runs smoothly and the user interface allows the student to easily navigate through the terminal app. The user experience will only improve with the implementation of a GUI.

Would a user be happy with your solution?
The user will be happy with the solution especially as the prioritization is automated and quick.

Is it flexible enough to change if the problem scope changes?
Yes. The code has been broken down into different classes and well commented which will make any updating easy.