# CS-202
# Data Structures
# Assignment 4

Sorting and Priority Queues

**Due Date: April 9th, 2022**

## Learning Outcomes

In this assignment, you will:

- Practice implementation of two of the most important sorting algorithms: merge and insertion sort
- Create binary min-heap
- Apply the concepts of min-heap to a practical scenario and practice coding priority queues

## Prerequisites

For this assignment, you will require:

- Proper understanding of merge sort and insertion sort algorithms
- Proper understanding of binary min-heap
- Linux Machine to run the test cases

## Structure

This assignment has three parts. Part 1 is on Merge Sort and Insertion Sort. Part 2 is on Min-heap. Part 3 is a practical scenario of priority queues using min-heap.

The deadline for this assignment is strict. No extension will be provided. You can use your grace days if you are unable to finish your assignment on time. Refer to the end of this manual for the Grading Scheme and instructions for automated testing.

## Plagiarism Policy

The course policy about plagiarism is as follows:

1. Students must not share the actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Best of Luck!

## Task 1: Sorting

In this part, you will be implementing two of the sorting algorithms you have studied in class: Merge Sort and Insertion Sort. In **Part1_A.py**, write your implementation for **Merge Sort** and in **Part1_B.py**, code for **Insertion Sort**.

Make sure you follow the standard implementation of the sorting algorithms as the main aim is not just to sort, but aso the correct implementation. This will be evaluated during the checking of the assignment and *even if your tests pass* but the implementation is incorrect, marks will be deducted. Lastly, write the time complexities of your algorithm above each function.

**Parameters**: Unsorted List, **Return Value**: Sorted List

You can test your functionality in the provided main before running the test file in order to ensure correctness because the test file will take quite some time to execute.

**Side Note**: The TAs might check the runtime of your code. For this, you are encouraged to check the time your program takes yourself.
Hint: Try Googling ***'time'*** module in python to calculate the total runtime a piece of code takes.

# Task 2: Heaps

In this task, you'll be implementing a **min-heap** data structure by completing the function definitions that are mentioned below, in **Part2.py**.
The boiler code for the min-heap has also been provided in **Part2.py**.

## Member Functions:

Write the implementation for the following functions as described here. Feel free to make any helper functions that you deem necessary; however, do **not** change any of the given function declarations.

**minHeap** (self, cap)

- Simple constructor that creates a min-heap array (harr) of the given capacity, with all elements initialized to None

**def insert_node** (self, value, doc_title=None)

- Inserts a new node into the heap, where value is the integer key of the node
- doc_title will be used in Part 3, so it's to be set as None for Part 2
- Return "Error: The heap is at maximum capacity!" if the user tries to insert a node when the heap is full
- A successful insertion won't require you to return anything

**def delete_node** (self, i)

- Deletes the node stored at index i
- Return "Error: Index out of range!" if the user enters an index i that's greater than the size of the min-heap
- A successful deletion won't require you to return anything
- Remember to ensure the structure of the min-heap is maintained after deletion

**def parent** (self, i)

- Returns the index of the parent of the node at index i

**def get_left_child_index** (self, i)

- Returns the index of the left child of the node at index i

**def get_right_child_index** (self, i)

- Returns the index of the right child of the node at index i

**def get_min** (self)

- Returns the minimum node of the heap, and None if the heap is empty

**def extract_min** (self)

- Removes the minimum node from the heap
- Returns this minimum node
- Return “Error: The heap is empty!” if the min–heap contains no elements

**def decrease_node** (self, i, new_val)

- Decreases the value of the node at index i to the new value provided in new_val
- It’s safe to assume that new_val is always going to be less than the current value of the node at index i

## Task 3: Sorting + Heaps

In this part, you will be implementing a Printing Job Scheduler (similar to the ones found in various Operating Systems) with priority queues, using the min-heap functionality from Task 2, in **Part3.py**.

For setting up some context: imagine if you and your friend send a printing request to a printer at the same time, however you send a one-page document while your friend sends an entire 300-page novel to be printed. If the printer starts printing your friend's novel first before it prints your document, you will have to wait for an unnecessarily long time. So to save time (*and also for the sake of efficiency*), the printer will assign priority to both the jobs. As your document was just one page long, it will set higher priority to your printing job and lower priority to your friend's (novel) printing job. In this way, despite sending the printing jobs at the same time, you will receive your printed document first and then your friend's novel will start printing.

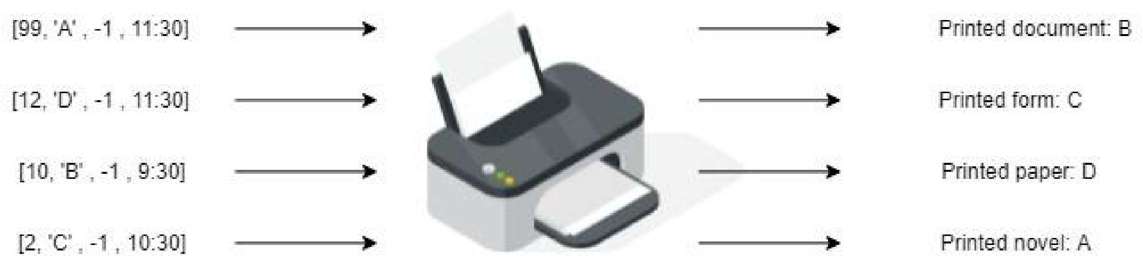
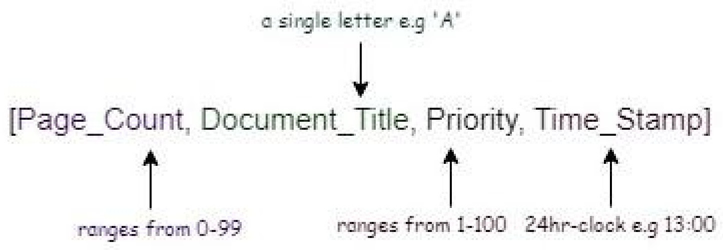


**Fig. 1: Printing jobs format**

Let's implement it!

Remember, we set self.doc_title to None in Part 2.py. We will be making use of it in this part.

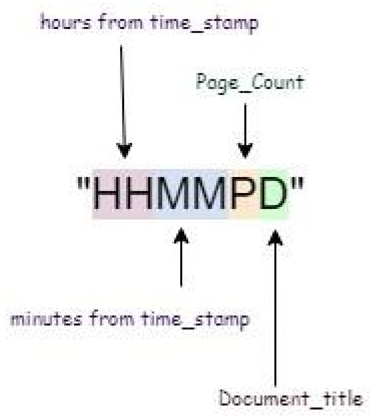
You are given a set of jobs in the form of a list in the following format:



The priority is initially set to -1 (as shown in Fig 1). You will be changing it at a later point in this task.

**Functions:**

**def job_string_maker(printing_jobs):**
Parameter: list of printing jobs
Return : list of printing job strings
In this function, you have to convert each printing job to a string in the following format and return it.



**def job_string_sorter(job_strings):**
Parameter: list of printing job strings
Return: list of sorted printing job strings
In this function, you have to sort the above printing job string with one of your previously implemented sorting algorithms (either merge_sort or insert_sort from

Part1). Also, state the reason for choosing a particular algorithm in the comments. Note that the sorting of your string should occur on the basis of time_stamp. If two or more jobs have the same time_stamps, sort them on the basis of page_count and if page count is also the same, sort them on the basis of their document title.

**def assign_priorities(sorted_job_strings, printing_jobs):**
Parameter: list of sorted printing job strings, list of printing jobs
Return: list of sorted printing jobs with priority
Once the jobs are sorted, you are required to set a priority to each printing job in-place i.e, inside each printing job (list). This priority is set on the basis of the time_stamp, however if two or more jobs have the same time_stamp, assign a higher priority to the job with less number of pages. Keep in mind that since we are implementing this scenario with a min-heap, so lower the priority value, higher is the priority.

**def create_heap(prioritized_printing_jobs):**
Parameter: list of sorted printing jobs
Return: list of documents' title
Here, you have to create a min-heap with printing jobs sorted above. For convenience, we are only adding priority (as self.value) and document name (as self.doc_title) to min-heap. Once the heap is created, you are required to print the documents based on its priority and return the names of documents in the correct sequence in the form of a list.

And *viola*, you are done!

## Grading Scheme

Run the test files on a Linux/MacOS environment to ensure that your code is taking optimal time to execute. If it takes longer than expected, try to see how you can improve it. You can make helper functions to streamline your implementation, however make sure the helper functions are not adding extra time complexity.

If you need any help, please do not resort to unfair means, and feel free to reach out to any of the teaching staff. All of us are here to help you learn and grow as budding computer scientists, and above all, as responsible professionals.

Best of luck! All of you are fully capable of attempting this assignment, and we are sure you will do great

**Part 1**

| **Task** | **Score** |
|---|---|
| Merge Sort (Implementation & Time Complexity) | 15 (=10+5) |
| Insertion Sort (Implementation & Time Complexity) | 15 (=10+5) |
| Total | 30 |

**Part 2**

| **Task** | **Score** |
|---|---|
| Insertion | 10 |
| Deletion | 10 |
| Extracting the minimum node | 4 |
| Decreasing the Key | 6 |
| Total | 30 |

**Part 3**

| **Task** | **Score** |
|---|---|
| Creating printing job strings | 10 |
| Sorting printing job strings | 5 |
| Assigning priority | 10 |
| Final Document Sequence | 15 |
| Total | 40 |

**Total Marks**: 100