



**Table of Contents**

**Introduction ................................................. 1**

**System Components .................................... 2**

**Application (User) Level ............................. 4**

**Implementation ........................................... 7**

**Testing ......................................................... 10**

**Queue Data Structure Operations ............. 12**

**Collaboration of ADTs and Algorithms ....... 14**

**Benefits of Using Independent ADTs .......... 15**

**Comparative Analysis of ADTs .................. 16**

**ADTs in Object-Oriented Programming ...... 17**

**Algorithms Performance and Efficiency .... 19**

**Experimental Comparison of Sorting Algorithms ... 21**

**Memory Stack Usage in Merge Sort ........... 23**

**Big-O Notation and Asymptotic Analysis .... 25**

**Alternative Methods for Measuring Efficiency ... 27**

**Graph-Based Task Recommendation Feature ... 29**

**Shortest Path Algorithms .......................... 31**

1.1

In my project the user can interact with the software via a console and menu system where the user can order the functionalities and the features from the code by reading and responding to what the code want to be able to complete and function as a task management system.

in the system users there are many functions, but they all start with adding tasks the user adds tasks by writing details like name due date priority and category.

user can mark any task as completed and the system will separate the completed tasks from the uncompleted tasks and when the user asks for them the program will retrieve the tasks based on the order that the user want, the available orders in the program are:

1. 1.view all tasks
2. 2.by due date
3. 3.urgency of the task
4. 4.by the category of the task
5. How do those components interact with each other compared to the rest of the normal tasks? It will get retrieved with the urgent ones assuming the user wants to retrieve tasks according to their priorities.
6. Deleting a task, deleting a task wouldn't sort the list till the user wants to.
7. 1.2
8. components of my software
9. the primary components of the software are:
10. 1.

task class: the purpose of this class is to provide all the necessary details for the project which are

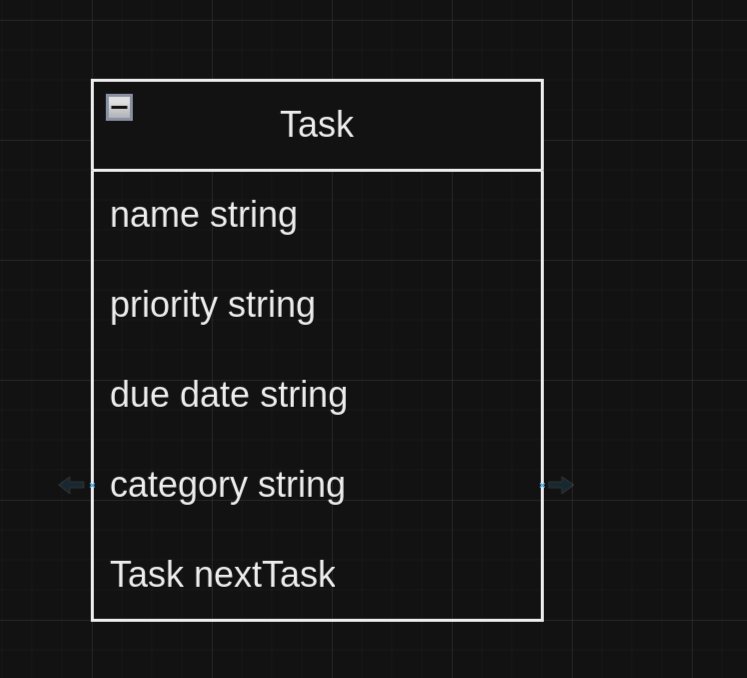
name: the name of the task you want,

* + - * 1. due date: when is this task due,

1. priority: is the task urgent or normal in terms of the priority,

category: the category of the task can help when the user wants to see the tasks ordered by category,

next task: this variable is there to point us to the next task to make the tasks in a list not an array to make it more flexible in terms of adding/deleting tasks.

1. ****
2. 2.

task list class: this class purpose is to manage the tasks using a linked list to be able to operate the functions more efficiently

1. the primary operations are:

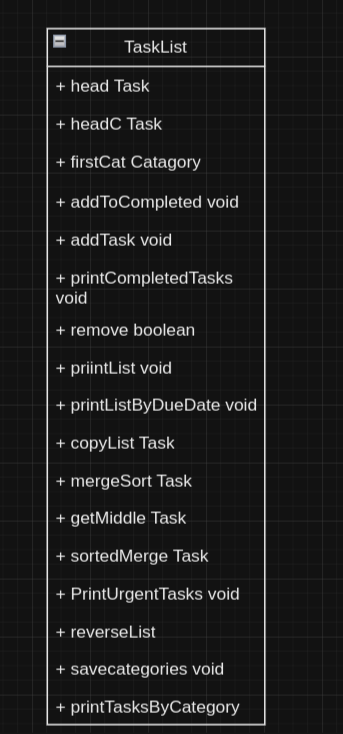
add task: this operation is responsible for adding tasks to the list and the formatting of the information.

remove: this operation is triggered in two cases: first case when the user wants to simply remove a task from the app, the second case is when the user wants to change the task from **to in this case, we remove the task from the primary list and add it to the completed task list.**

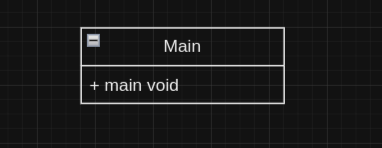
**print list: this operation is made up of 3 different operations the normal case is to retrieve the list with the order that they have been added to.**

**case 2 print it by the order of the due date in this case you can use it to know which tasks are coming up next.**

**case 3 separating the urgency from the urgency so the user can retrieve one of them without the other case 4 if the user wants to retrieve them according to their category to see one of the categories without the others.**

****

**3. main: in this class we execute what operation we want by choosing form a menu**

****

**task 1.2**

**stack:**

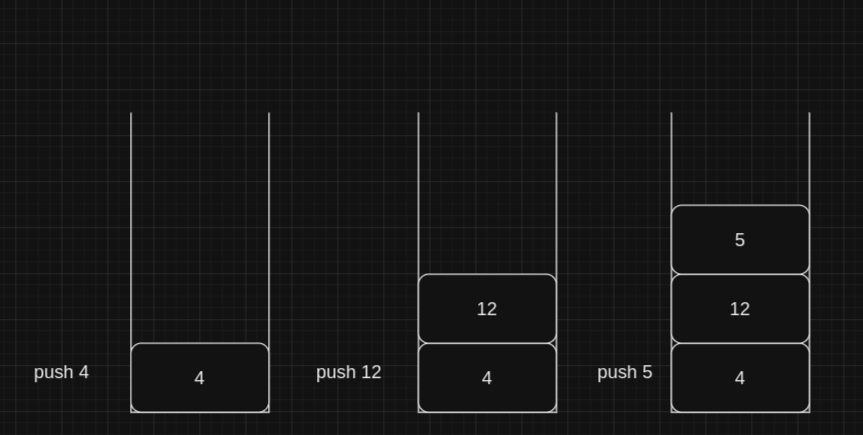
**a stack is one of the fundamental data structure used in CS and it is one of the famous ADTs, the use of s stack is to store and manage a collection of elements in a particular order, specifically stacks is known for its unique order LIFO(last in first out) which mean the last element to be added to the stack is the first one to be out(print and remove).**

**the core components of stack are**

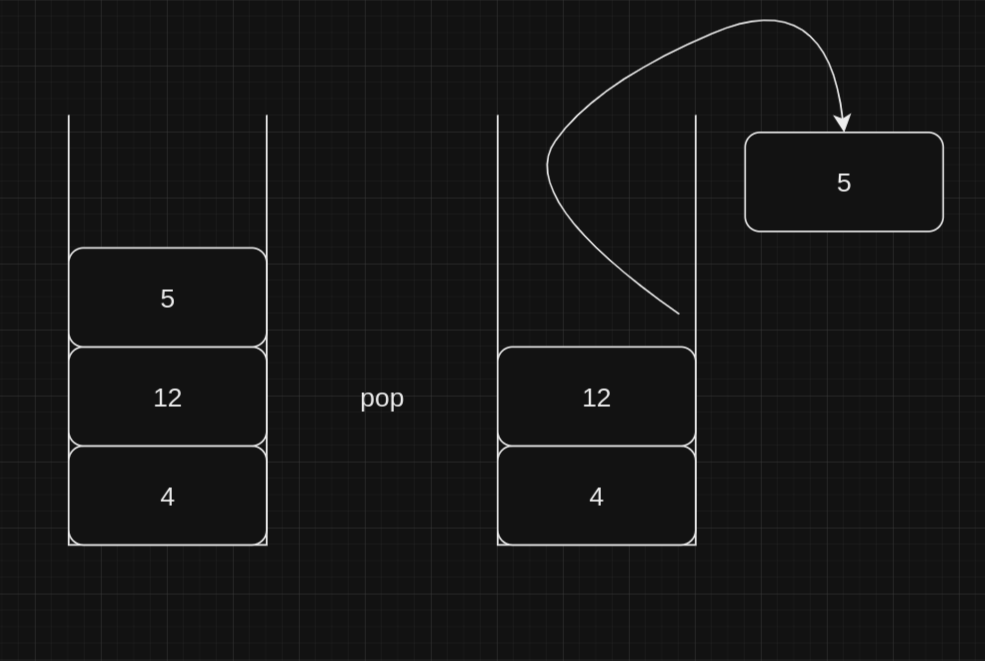
1. **element: the data items that the stack hold and stores these can be any data type int char even objects etc**
2. **container: the storge for the elements in the stack could be an array or linked list depending on the implementation**
3. **top: a pointer to point at the first element (the most recently added)**
4. **max: if the implementation was in an array there should be a maximum number of elements**

**there are 4 operations**

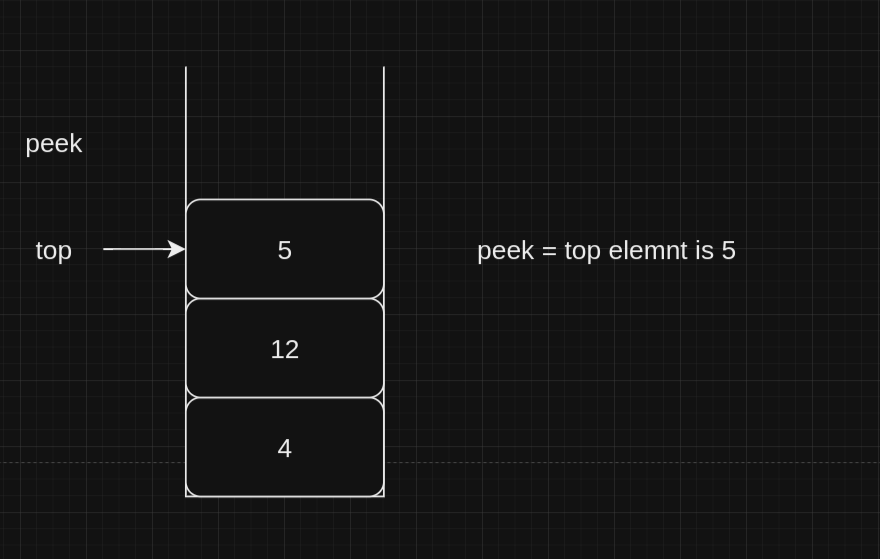
**push: adds an element to the top of the stack**

****

**pop: removes and return the top element from the stack**

****

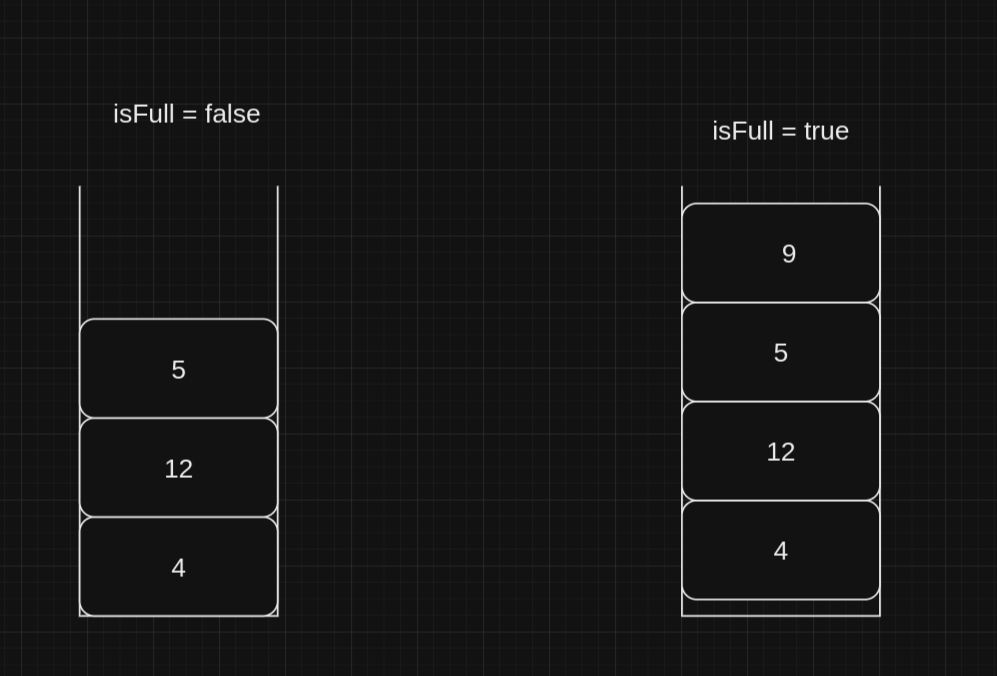
**peek: return the top element without removing it**

****

**isEmpty: checks if the stack is empty or has values**

****

**isFull(if the implementation has a full): check if the stack is full or there is still space available.**



1. **Task 2 - Application (User) Level**
2. **In my project, utilizing Abstract Data Types (ADTs) like the Task and TaskList classes is very important. These classes show the principles of encapsulation and information hiding, ensuring that the details of data management are kept away from the user's purview. These approaches come with several benefits to our task management system, enhancing both quality and user-friendliness. Here is an exploration of these benefits, showing how they contribute positively to the application:**
3. **1. Encapsulation to Preserve Data Integrity:**
4. **In object-oriented programming, encapsulation involves combining data and the methods that manipulate this data into a single unit, or class. In our application, this principle is exemplified by the TaskList class, which seamlessly manages tasks without revealing the internal mechanics of list management to the user.**
5. **Illustration:**
6. **- The method `addTask` in the TaskList class simplifies how a task is added. Users input task details such as name, due date, and priority, and the system efficiently places these tasks in the proper sequence in the list, abstracting away the underlying complexity.**
7. **2. Simplifying Interactions through Information Hiding:**
8. **The principle of information hiding supports encapsulation by limiting access to the internals of a class. This restriction reduces the likelihood of user errors and simplifies the operations that can be performed, enhancing user interaction with the system.**
9. **Illustration:**
10. **- When users want to view tasks sorted by due date or category, they need not understand the sorting mechanisms. They can utilize features like `printListByDueDate` or `printTasksByCategory`, which handle all sorting internally.**
11. **3. Enhanced System Maintainability:**
12. **With crucial functionalities encapsulated and internal details hidden, making updates to the system becomes a less daunting task. Changes can be made internally without affecting the users who rely on the high-level functions of the system.**
13. **Illustration:**
14. **- Should there be a need to alter how tasks are stored, say from a linked list to a different data structure, such modifications would be restricted to the internals of the TaskList class. The methods accessible to users would remain unchanged, thus not impacting their interaction with the system.**
15. **4. Security Through Controlled Access:**
16. **Encapsulation enhances security by controlling how data can be accessed or modified. This prevents accidental or unauthorized changes that could disrupt data integrity.**
17. **Illustration:**
18. **- Direct modification of task details in the list is blocked. Instead, tasks must be managed via defined methods like `addTask` or `remove`. This process not only ensures all changes are monitored but also maintains consistency and integrity in task management.**
19. **By employing encapsulation and information hiding, our task management system is not just efficient but also secure and easy to maintain. These principles effectively shield users from the complexities of the system while ensuring that the architecture remains flexible and robust. This thoughtful design significantly boosts the user experience and the reliability of the system.**
20. **Task3-implementation:**
21. **1. implementation**
22. **The implementation of the design took more effort than anticipated, in the original design there was 6 functionalities so my original line of thoughts was to have 8 classes and it was as follows:**
23. **1. class for the linked list**
24. **2. class add task**
25. **3. class complete task**
26. **4. view tasks class**
27. **5. view by due date class**
28. **6. view urgent class**
29. **7. view by category class**
30. **8. the main class**

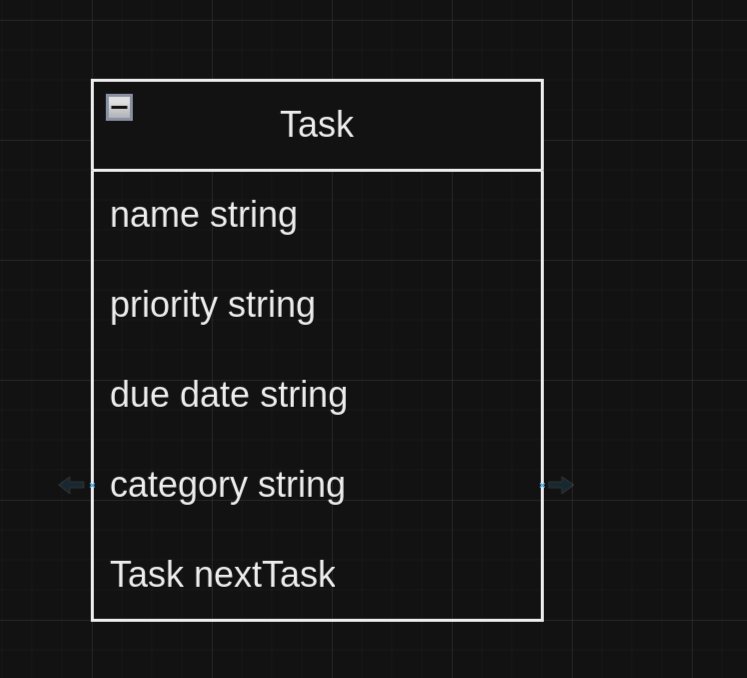
**After starting the implementation process that design quickly changed**

**The design to make each functionality in a separate class was extremely impractical and buggy, the reason behind these complications is separating each function in a different class would hide all variables and make them local to each functionality which would make the program more secure but my concern in writing this program was putting functionality above security so after this changes has been maid the design got changed to the following:**

**Classes**

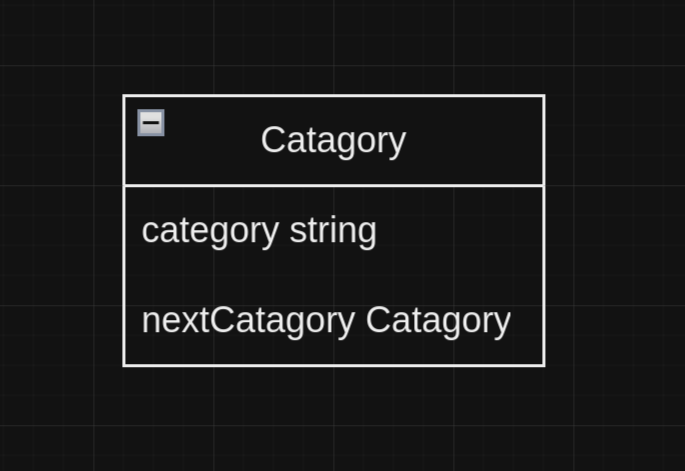
1. **Task**

**In this class we do the initialization process for the linked list and for the attributes in it.**

****

1. **Category**

**This class is difficult because I had to shift the categories to be in 2 lists: the normal task list and a special list to keep track of all unique categories for functionality in the code.**

****

1. **TaskList**

**This class has most of the methods in the program and they are:**

**1. AddToCompleted**

**2. addTask**

**3.printCompletedTasks**

**4. Remove**

**5. printList**

**6. printListByDueDate**

**7.copyList**

**8.mergeSort**

**9.getMiddle**

**10.soretedMerge**

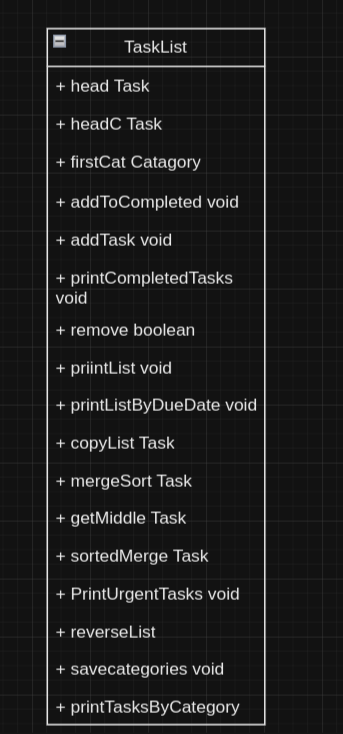
**11.printUrgentTasks**

**12.reverseList**

**13.saveCategories**

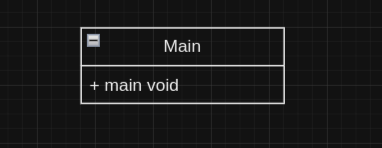
**14.printTasksByCategory**

**Each of those methods are very important for the program, some of them complete one full functionality and most of them needed to complete a step of a functionality and need other methods to complete and functionality either being called in main or in another Method.**

****

1. **Main**

**This class is the portal for the user to access all the methods indirectly. The user will choose the functionality that is needed for accomplishing his goal, and the program will call all relevant methods.**

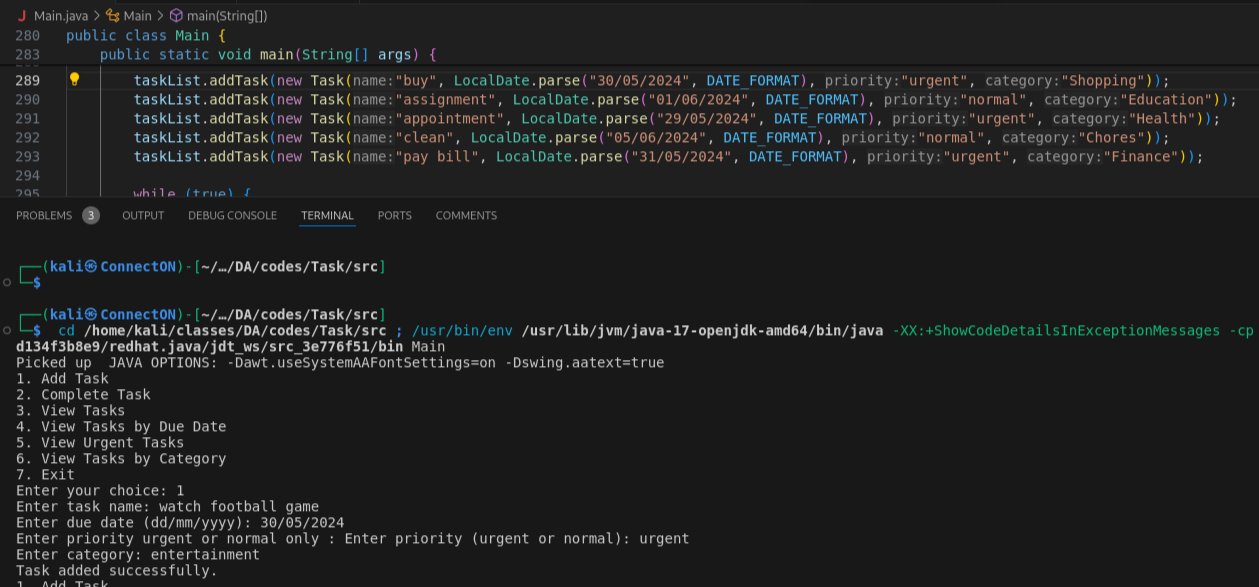
****

**2. testing**

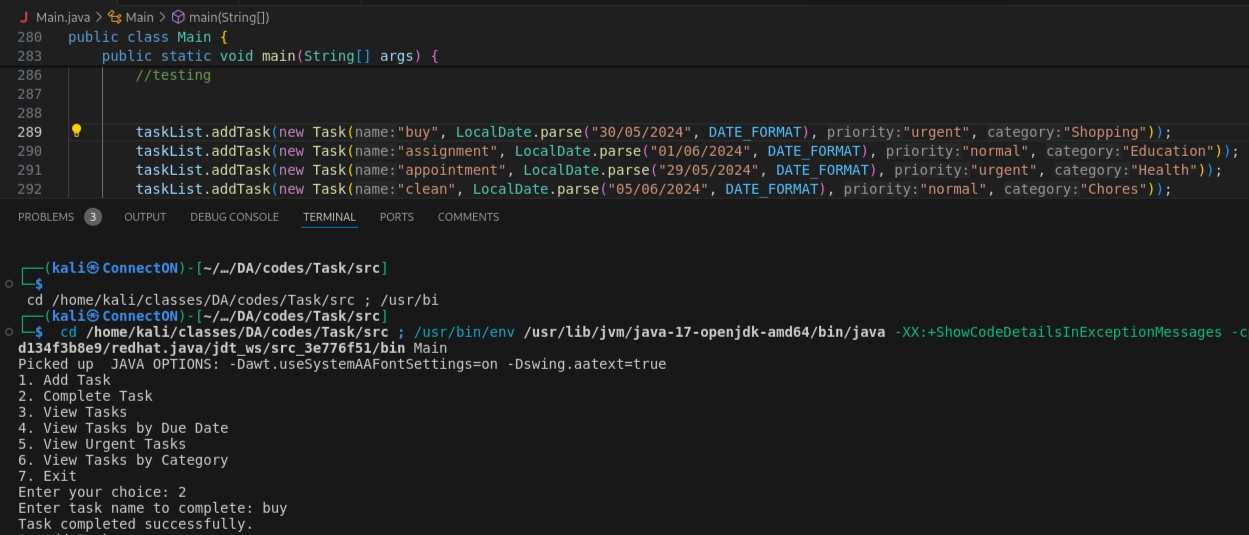
**We must separate the testing process into 2 sub testing test each functionality alone and then test a combination of functionalities back-to-back from each other and see if the results are what we intended**

**1. testing each functionality alone**

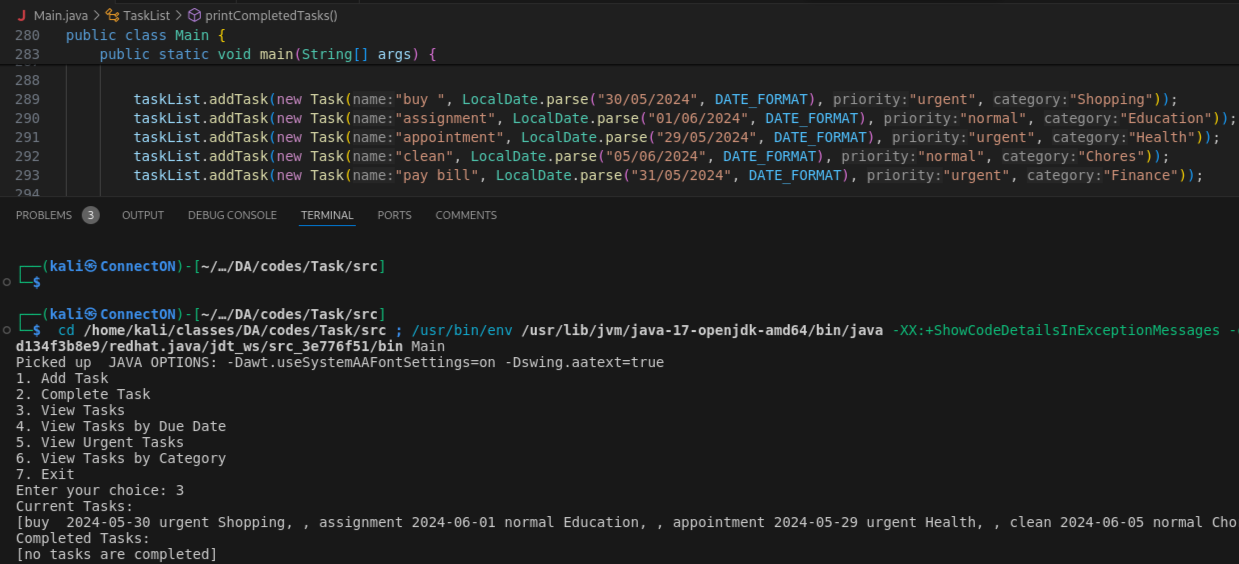
1. **Entering new task:**



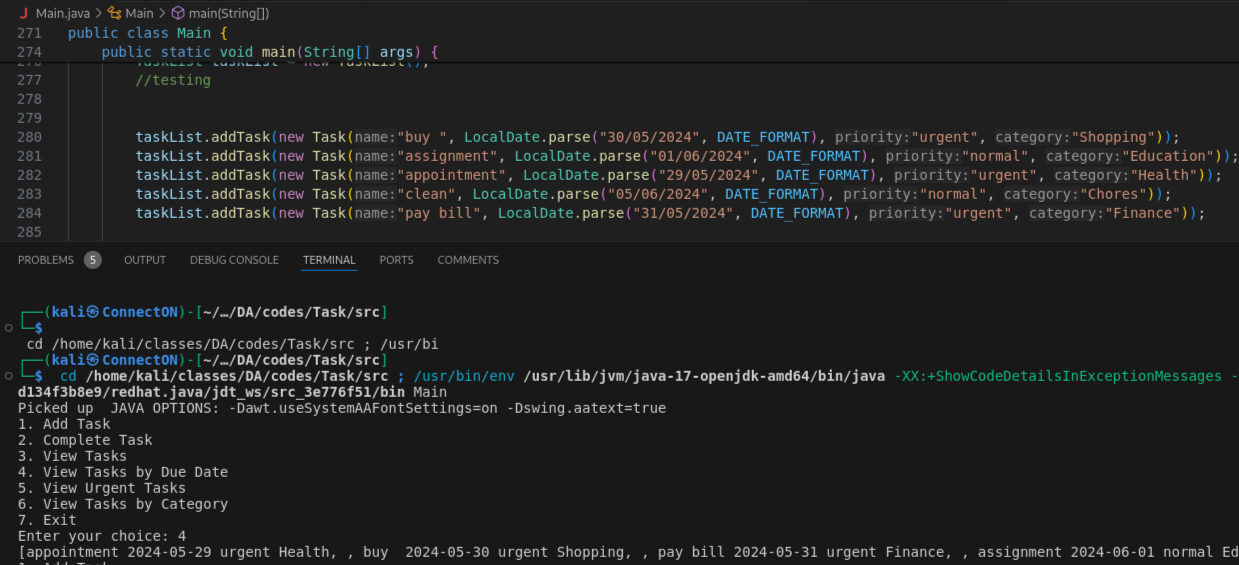
1. **Completing a task:**



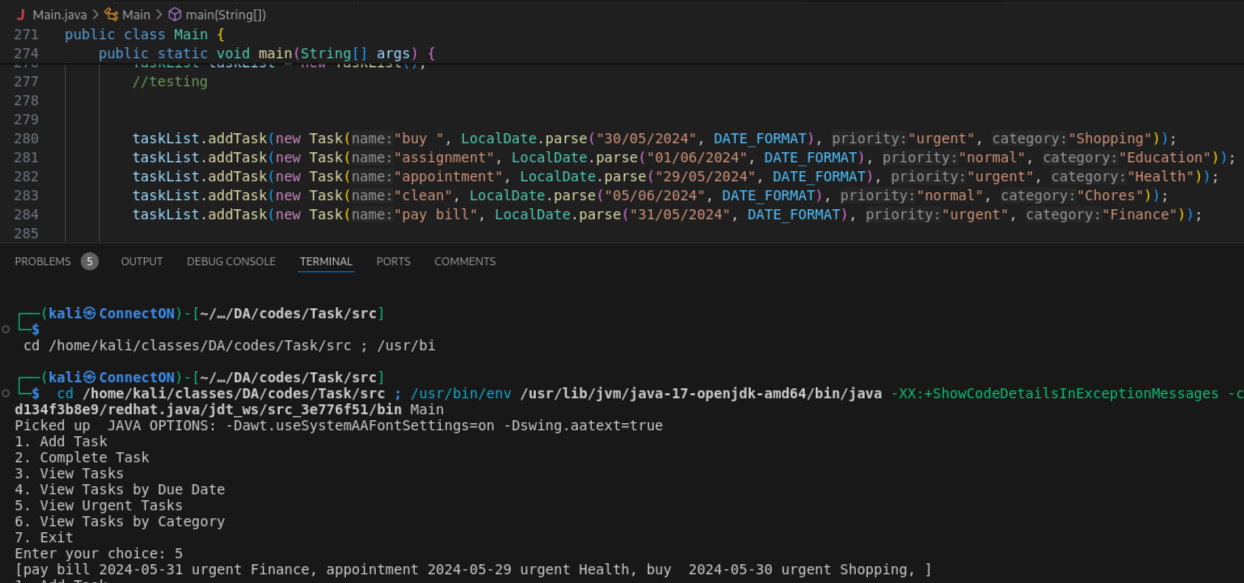
1. **View tasks:**



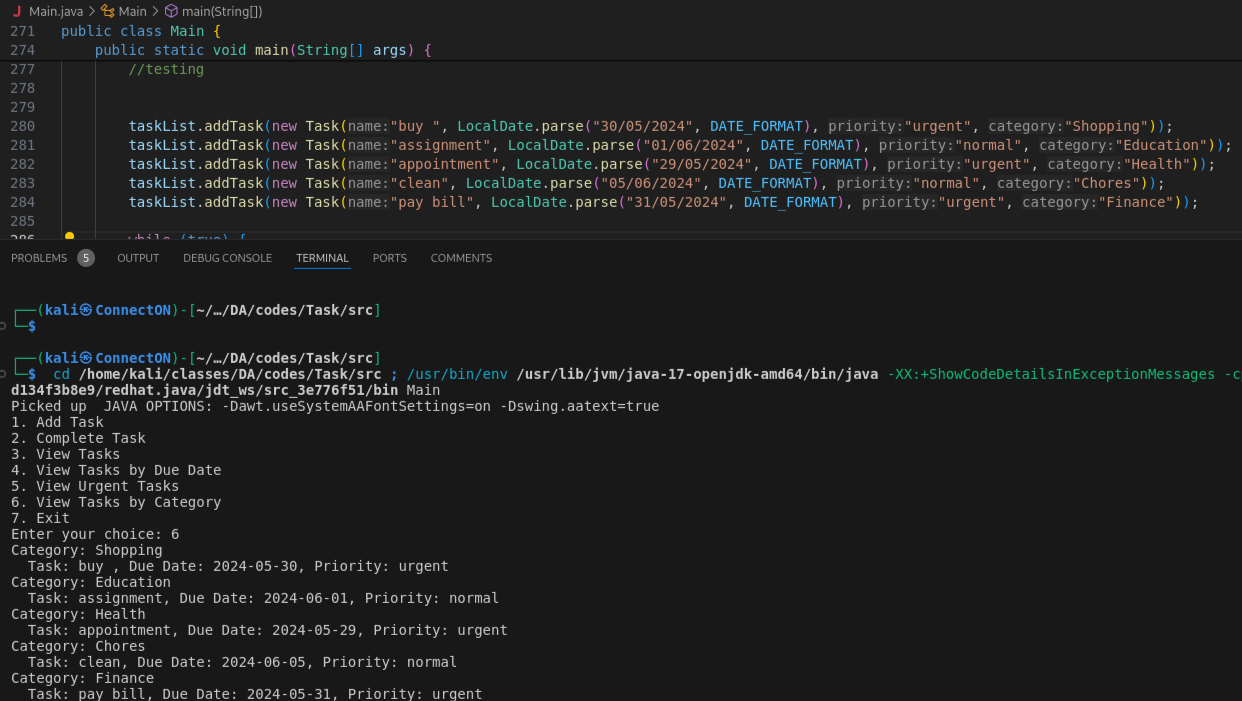
1. **View tasks by due date**



1. **View urgent tasks**



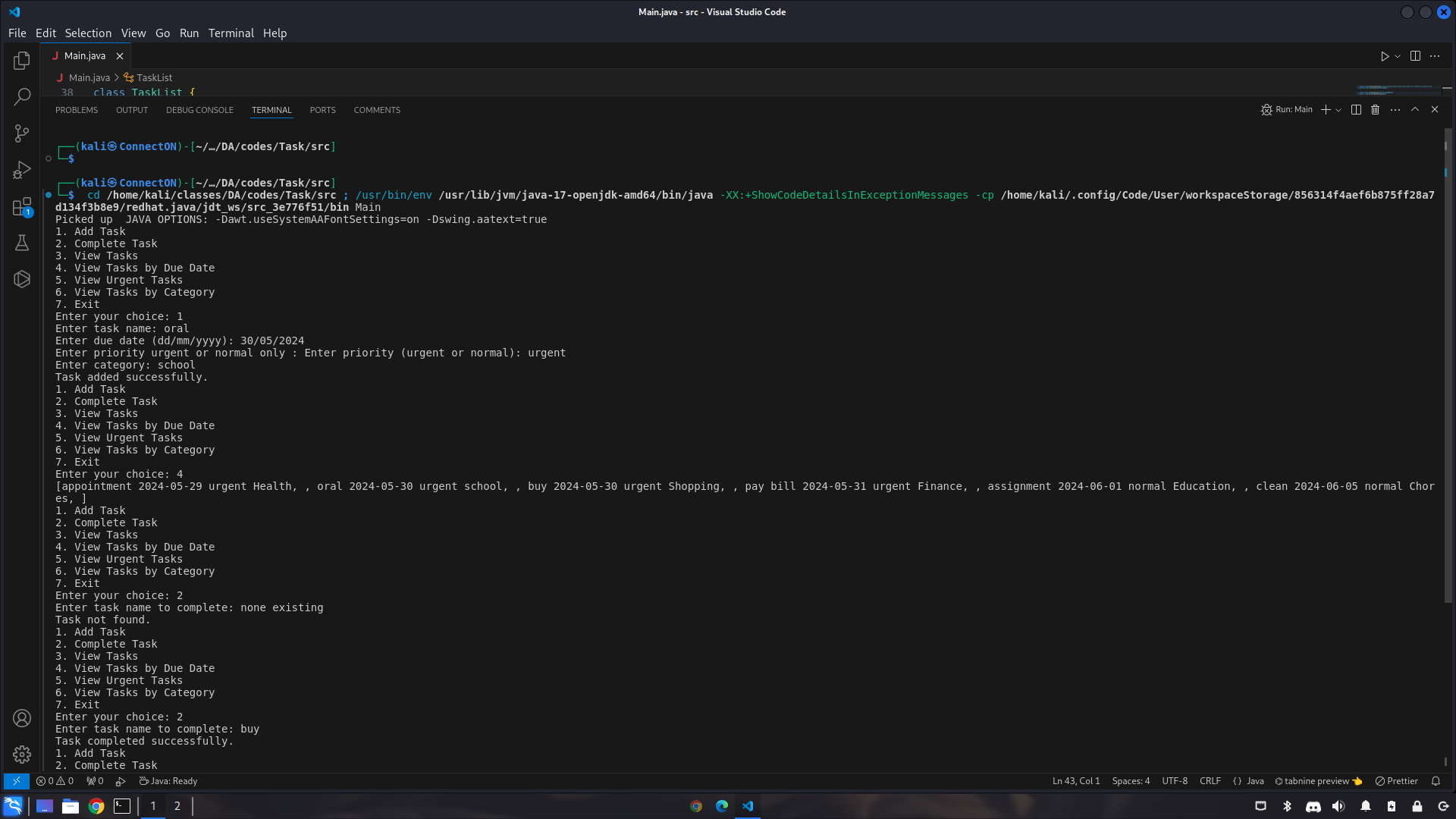
1. **View tasks by category**



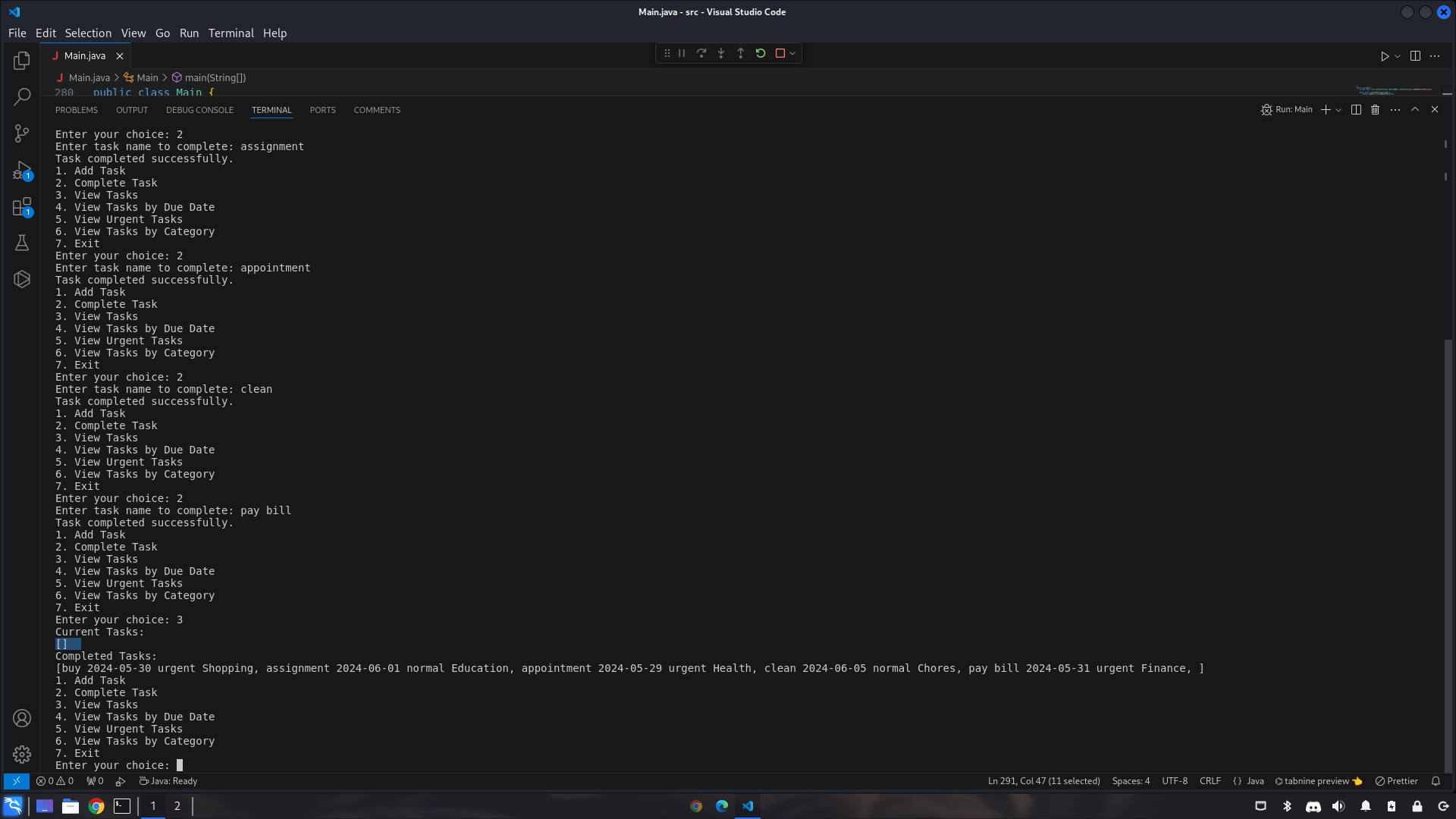
As you can see in the result this step was successful. After some editing of the code and rewriting where the code had holes/problems we came to the required result and now we need to start step 2 of testing.

**2. combination of functionalities back-to-back from each other**

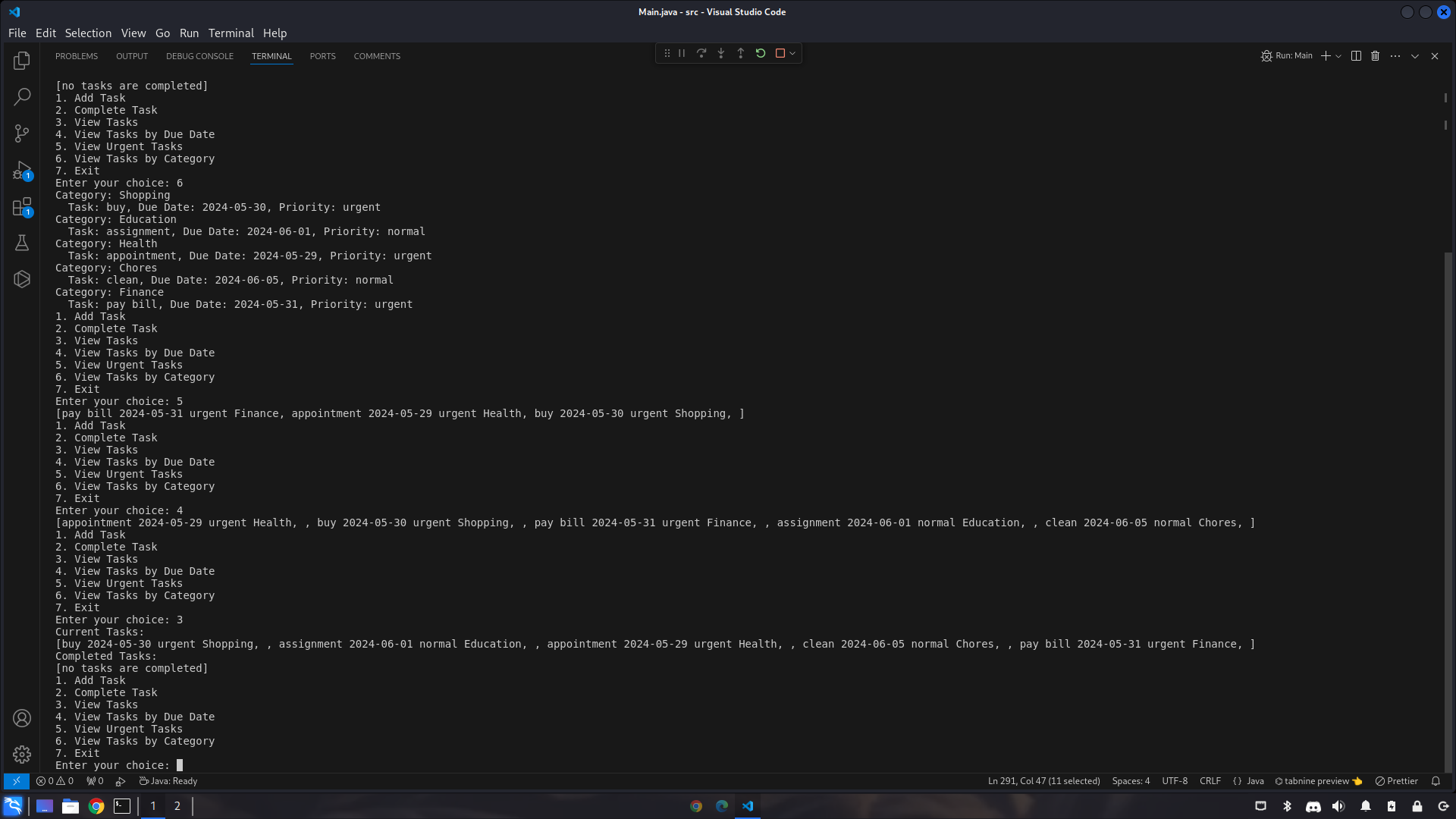
1. **Adding tasks with the same due date**
2. **Complete a task that doesn't exists**



1. **Print an empty list**



1. **Try normal functions after each other**



**Task 3.2:**

**Illustration and explanation of 4 queue data structure operation:**

**Lets break down those 4 queue operations, the common operation in a queue are:**

1. **Enqueue op: adding an element to the queue.**
2. **Dequeue op: removing an element from the queue.**
3. **Peek op: viewing the elements in a specific order.**
4. **IsEmpty op: check if the queue has any elements or is it empty.**

**1.enqueue**

**The enqueue operation is revolving about adding a new element to the queue, most people like to start adding elements at the end so it would make it easier to read/peek and dequeue/remove from the first but this is a requirement since you can add/enqueue from the start and read/peek and remove/dequeue from the end to achieve the FIFO property of the queue.**

**Process:**

**Check if the queue is empty**

**If empty the new task is the first task or the “head”**

**If it is not empty go to the end of the queue and add the element there.**

**Example:**

**If you want to enqueue 3 tasks task1 task2 task3 the queue would be empty at the beginning so task1 would be the head of the queue after that task2 would be added after task1 so the list or array would be [task1, task2] and the same process for task3 the result would be [task1, task2, task3] so the order of FIFO would be achieved.**

**2. Dequeue**

**Dequeue operation revolves around essentially about removing from the queue from the same end as it was added in so if you added it to first of the list it would be FIFO but if you added it to the last of the list it would be LILO**

**Process**

**Check if the queue is empty.**

**If not empty, remove the first task that has been added which in this example would be at the start of the queue at the “head’.**

**Update the head to the next task in the queue.**

**Example if the queue had the tasks in this order [task1, task2, task3] the first task added is task1, so we need to remove/dequeue from the beginning to achieve the queue FIFO property.**

**3. Peek**

**The peek operation is looking at the beginning of the queue ‘head’ without removing it, so it follows the same logic as FIFO but not exactly out as in out of the list, but it is out as printed and if there was a need to print more than one you must go through it.**

**Process;**

**Check if the queue is empty.**

**If not empty, return the task at the same end as we started saving in this example the head.**

**Example if this is the list [head: task1, task2, task3] the head would be printed first so we have achieved FIFO.**

**4. is empty**

**This operation checks if the queue has values or empty this usually used to avoid errors**

**Process**

**Simply check if there is any value in the head or first node.**

**Example**

**If the list [] is empty it would be true or have a value of 1, if the list [task1,----] it would be false or 0.**

**Task 3.3**

**ADTs and algorithms collaborating to solve the problem.**

1. **Completing a task**

**ADTs involved: ‘TaskList’ this class is using a linked lists of tasks to save active tasks and in this method, it removes the completed task, and a queue based on the linked list for the completed tasks.**

**Algorithm: remove from taskList and enqueue in completed tasks queue.**

**Description: when a task is completed, it is removed from taskList and enqueued in the queue, this involves finding the las task and updating its pointer then adding the completed task to the next node (end of the queue)**

1. **Viewing tasks:**

**ADT involved: taskList**

**Algorithms traverse the linked list.**

**Description: to view all the tasks, the system has to traverse all the taskList from head to end, printing each task details.**

1. **Viewing tasks by due date:**

**ADT: copy of Tasklist.**

**Algorithm: merge sort.**

**Description: tasks are sorted by due date using the algorithm merge sort, a copy of the list is made to avoid altering the original order of the list, and merge sort is applied to copy, and the copied list is printed.**

**Collaboration of ADTs and algorithms**

1. **Task insertion: adding a new task involves the simple algorithm of traversing the list and uprating pointers, the taskList ADT provides a structured way to manage this linked list.**
2. **Task completion: removing a task from the list and adding it to queue showcases how different ADTs (linked list and queue) can work together. The list structure allows efficient removal, and the queue structure ensures FIFO order for completed tasks.**
3. **Task sorting: sorting tasks by due date demonstrates how an algorithm (merge sort) can be applied to an ADT (linked list) to achieve a specific functionality, enhancing the utility of ADT.**

**Benefits of using independent ADTs**

**1. Security and modularity:**

**ADTs provide a separate approach where each type of data structure is encapsulated with its operations.**

**Benefits: this approach makes the system easier to understand, develop, and maintain for example, the linked list makes it so it's easier to modify a task without effecting other than the pointer in the last task.**

**2. reusability:**

**ADTs can be used multiple times in a project or across different projects.**

**Benefit: the taskList and queue implementation can be reusing wherever similar functionality is needed, reducing code duplication and improving efficiency.**

**3. Ease to enhance:**

**With a good interface, ADTs can be easily maintained and enhanced.**

**Benefit: if a more efficient algorithm is found for sorting it can be implemented**  **within the TaskList ADT without affecting other parts of the system. similarly,**  **changes to the queue implementation won't impact the task addition.**

**Evaluation of benefits:**

1. **Security and modularity:**

**The separation of the TaskList and the queue for completed tasks allows developers to focus on one part of the system at a time. If a bug is found in the task addition logic, it can be fixed without worrying about how tasks are completed or viewed.**

1. **Reusability: The same TaskList implementation logic can be used to manage different types of tasks or even different collections of items in other applications, promoting code reuse and saving development time.**
2. **Ease to enhance: If the task sorting needs to be optimized, developers can focus on improving the merge sort algorithm within the TaskList class. Other functionalities, such as task completion and viewing, remain unaffected by these changes.**

**Task3.4 Comparative analysis of ADTs:**

|  | **BST** | **Ordered LL** | **Sorted array** |
| --- | --- | --- | --- |
| **Time complexity** | **Search: avg O(n) worst case O(n)**  **Insert: avg O(log n) O(n) worst case**  **Remove: O(log n) avg O(n)worst case** | **O(n)** | **Search: O(log n)**  **Ins: O(n)**  **Rmv: O(n)** |
| **Space complexity** | **O(n)** | **O(n)** | **O(n)** |

**Binary search tree**

**Pros: efficient search, insert, and delete option on avg O(log n) time good when you want to delete and add a lot**

**Cons: can be costly in terms of time in a case of unbalance tree it becomes O(n) in worst case if the tree is unbalanced requires extra to keep the tree balanced**

**When to use: when you have smaller datasets or dynamic resizing are a priory**

**Ordered LL**

**Pros: dynamic resizing**

**Cons: inefficient in time complexity it can be O(n) worst case for all operations, because of traversal is a requirement**

**When to use: when it's more important to have a dynamic size more then effeint time complexity, especially in small datasets**

**Sorted array**

**Pros: efficient search operations like binary search with a time complexity O(log n)**

**Cons: insertion and deletion are costly with a O(n) time complexity**

**When to use: when the datasets are mostly static sized, and fast searching is important.**

**Task 3.5**

**ADTs in object-oriented programming:**

**Some people say that ADTs are the fundamentals that allowed us to build OOP programming and others say that the OOP was before the ADT was maid which tells us even if either of those claims was true that they are both connected to one another by many different aspects ADTs and OOP are implemented using classes and objects, which offer a clear separation between what operations can be performed and how these operations are preformed this separation is important to achieving abstraction, encapsulation, and polymorphism.**

**Encapsulation and abstraction**

**Encapsulation refers to the bundling of data with the methods that operate on that data. By using ADTs, we can hide the internal representations of data from the outside world, exposing only what is necessary. This led to something called abstraction, where complex implementation details are hidden and only can accessed by essential features that are presented**

**Example:**



**In this example, the task class encapsulates the properties of the task and the TaskList class manages a collection of tasks. The internal details of how the tasks are linked are hidden from the user, who can interact with the list only through the provided methods.**

**Reusability**

**ADTs promote reusability, by defining a clear interface for data structures, we can reuse these components in different parts of an application or different projects without worrying about their internal implementations.**

**Example**

**The taskList class can be used in any application that requires task management, not just in this project.**



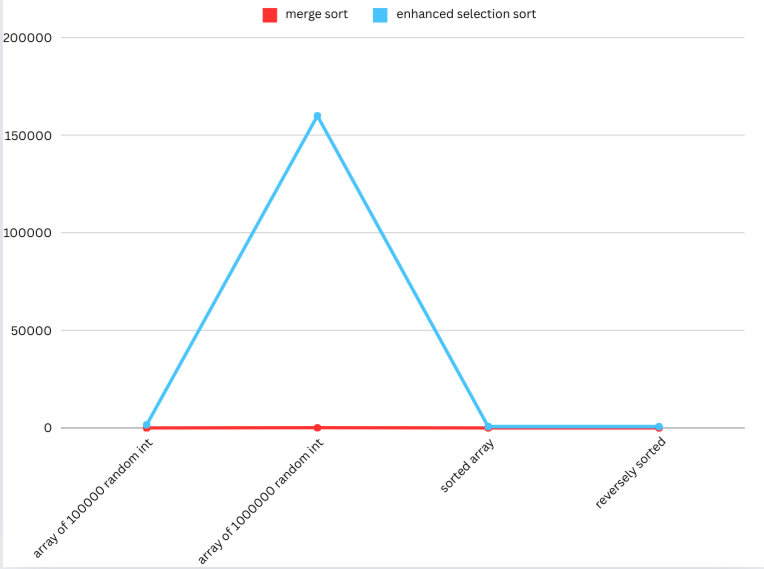
**ADTs and OOP are heavily related to one another, and they both promote the same principles encapsulation, abstraction, reusability, modularity, and polymorphism, in our application the use of ADTs creates a clear separation that makes the code easier to understand, maintain, extend, this enables us to build flexible software system.**

**Part 2- algorithms performance and efficiency.**

**2.1:**

**in this part we are going to test and compare 2 sorting algorithms merge sort and enhanced selection sort under different circumstances and we are going to record the results and compare the two algorithms according to speed**

|  | **merge sort** | **enhanced selection sort** |
| --- | --- | --- |
| **array of 100000 random int** | **17** | **1625** |
| **array of 1000000 random int** | **137** | **159889** |
| **sorted array like [1, 2, 3 etc] 100000 size** | **12** | **750** |
| **reversely sorted exp [10, 9, 8]100000 size** | **13** | **751** |
| **array of 10000000 random int** | **1397** | **i couldn't do this number because my computer couldn't handle it** |

****

**as we can see in the results sorting an array with merge sort is much better in all scenarios**

**then selection sort we can see the big difference when we ran a million random int in both sorting algorithms and the merge sort took less the 0.2 seconds compared to the selection sort it took approximately 160 seconds around 3 minutes this show that the difference between a logarithmic time complexity algorithm and a quadratic time complexity algorithm it can be heavily favored in large datasets as we can see from the results and that's why in java libraries they use a logarithmic approach for the array.sort(), because it's the best way to achieve sorting an array.**

**and we can see in the result that both sorting have a significant difference between sorted and unsorted(random) arrays: the merge sort with ⅔ of the time approximately and the selection sort with half of the time approximately.**

**But what intrigued my curiosity is that there isn't any significant difference in either of the algorithms when it comes to the sorted array in both algorithms whether the array was sorted in ascending or descending order. In both algorithms there was a 1 millisecond which I found interesting.**

**here is why those points happened:**

**time complexity: the time complexity of the selection sort is O(n^2) compared to the merge sort time complexity O(n log n), and this can be seen more effectively on large scale data.**

**logarithmic vs quadratic: as i mentioned, logarithmic time complexity grows much slower than quadratic with increasing data size, this why merge sort shines for large datasets.**

**java’s array.sort() this function uses quicksort and it is another algorithm with similar time complexity on average O(n log n) and this is why merge sort is a good choice for sorting.**

**reference 1 and 2**

**2.2:**

**merge sort stack usage with a size 6 array**

* **initial call:**

**the program start sorting by calling mergeSort(arr) where arr is a 6 element array**

**stack frame created on the memory stack, this frame keeps track of what functions are called and which are done which include the array and its memory location**

* **recursive call**

**the mergeSort divide the array into 2 sub arrays left and right and then 2 recursive calls are made mergeSort(right) mergeSort(left) let's assume that we started at 1 not 0 then it would be**

**left[1,2,3] right[4, 5, 6] and so on with the recursion calls**

* **base case**

**When a sub array has only one element the mergeSort function reaches the base case and returns the top frame of the stack will no longer be needed; the memory stack will pop this frame and free up space for the other function call.**

* **merging**

**after both left sub arr and right sub arr are done and returned sorted the merge function is called to reattach them together, the initial call to mergeSort is still active**

* **returning the memorie**

**after the merge function completes sorting the array and returns control to merge sort function that called it (recursive 2) the stack frame will pop the call and releasing its memory**

* **repeat for rest of the call**

**this whole process repeat itself for the remaining recursion, and for each stack frame is popped after the merge complets sorting the sub arr**

**After everything in the merge sort is done, the initial call for merge sort is popped and the program will resume its original objective and execute the rest of the program.**

**2.3**

**what is big-O notation**

**big O-notation is a tool that is used to describe an algorithm time complexity mathematically as the input size grows infinitely, it focuses on time growing and ignore constant factors**

**big-O types**

**O(1): constant time which means that the time doesn't get affected by the input (no matter the input time is the same)**

**O(log n): logarithmic running time grows matching tp the logarithm of input size**

**O(n): linear running time grows with a match to growing input size**

**O(n\*n): quadratic which is running time grows squared with the input size**

**this is used to assess the efficiency of an algorithm by comparing the input size and time of the algorithm to other algorithm equation(relationship between time and input size)**

**merge sort asymptotic analysis**

**merge sort use a divide and conquer approach it recursively divide itself to smaller and smaller sub arrays and then merges them back together**

**the divide and conquer steps contribute the logarithmic term (log n)**

**and merging them together takes a partially of the linear approach for each sub array**

**so as the recursive call is logarithmic the complexity becomes O(n log n)**

**selection sort:**

**this sorting algorithm goes through the array finding the maximum and minimum each time and putting them in their place (first or last) and this process repeat itself until the array is sorted , in each time it compares the element with the rest of them as the number of comparison grows linearly with the input size (n) the time complexity becomes O(n^2)**

**comparison as we have seen in the example above and the time complexity explained too, the merge sort is significantly better and more efficient than selection sort O(n log n) vs O(n^2)**

**2.4:**

**other methods to determine the efficiency of an algorithm**

1. **profiling:**

**profiling is analyzing the code to identify bottleneck areas(areas that take significant time) some tools can track functions calls, memory usage and execute times for a specific code**

**it can be used for merge sort and see how much time spent in the merge function compares to the recursive calls.**

1. **benchmarking:**

**benchmarking involves comparing the performance of different algorithms or implementations on the same data, this allows us to see which algo works better in our scenario**

**We can benchmark the merge sort and the selection sort and compare the 2 algorithms.**

**part 3 adding recommendation feature**

**the first step to accomplishing those tasks is making an example of tasks:**

**Pay bill 2024-05-31, urgent, finance (a)**

**Buy something 2024-06-25, urgent shopping(i)**

**Attend appointment 2024-06-25 urgent health(h)**

**Clean 2024-06-20 normal chores (g)**

**Exercise 2024-06-07 normal health(f)**

**Read book 2024-06-05 normal personal development (b)**

**Complete assignment 2024-06-10 normal education(c)**

**Write report 2024-06-13 normal work(e)**

**Prepare presentation 2024-06-15 normal work(d)**

**(a) → (c), weight 2**

**(a) → (b), weight 3**

**(a) →(i), weight 5**

**(b) →(i), weight 10**

**(b) → (c), weight 2**

**(b) → (f), weight 10**

**(b) → (g), weight 2**

**(c) → (d), weight 4**

**(c) → (e), weight 3**

**(e) → (d), weight 1**

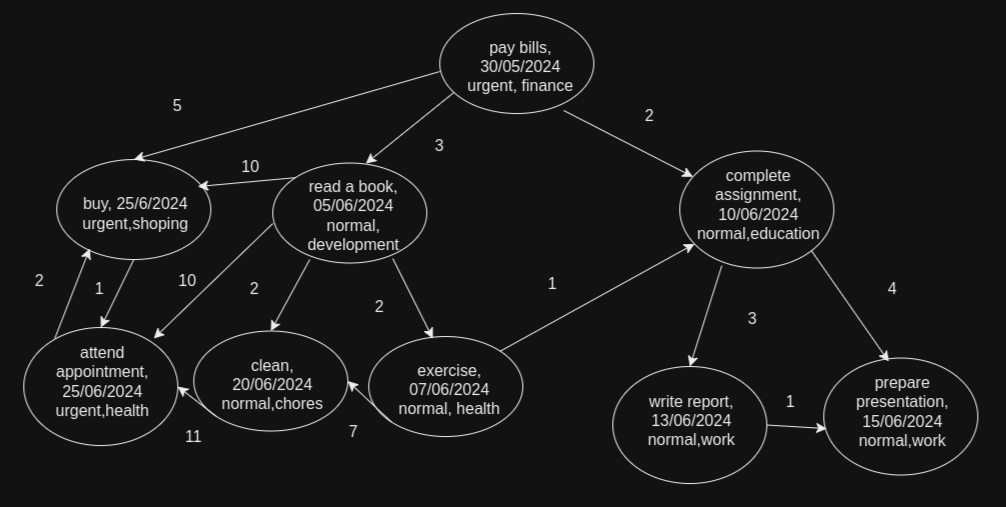
**(f) → (g), weight 7**

**(f) → (c), weight 1**

**(g) → (h), weight 11**

**(i) → (h), weight -1**

**this is the example i made for the graph**

****

**this example matches all the requirements of the graph**

* **Number of vertices at least 8**
* **Number of edges [15-30]**
* **Consider directed edges.**
* **Consider one single source vertex.**
* **Ensure a path exists from the source to each other vertex.**

**3.2 Dijkstra and bellman-ford**

1. **dijkstra’s algorithm**

**first step**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **∞** |  |
| **c** | **∞** |  |
| **d** | **∞** |  |
| **e** | **∞** |  |
| **f** | **∞** |  |
| **g** | **∞** |  |
| **h** | **∞** |  |
| **i** | **∞** |  |

**we start from a**

**update b and i and c**

**b distance 3**

**i distance 5**

**c distance 2**

**the new table would be :**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **3** | **a** |
| **c** | **2** | **a** |
| **d** | **∞** |  |
| **e** | **∞** |  |
| **f** | **∞** |  |
| **g** | **∞** |  |
| **h** | **∞** |  |
| **i** | **5** | **a** |

**then we move to b (smallest number) (3)**

**update c 5(already smaller)**

**update f 13**

**update g 5**

**update i 10 (5 is smaller no update)**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **3** | **a** |
| **c** | **2** | **a** |
| **d** | **∞** |  |
| **e** | **∞** |  |
| **f** | **13** | **b** |
| **g** | **5** | **b** |
| **h** | **∞** |  |
| **i** | **5** | **a** |

**move to i smallest**

**update h 6**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **3** | **a** |
| **c** | **2** | **a** |
| **d** | **∞** |  |
| **e** | **∞** |  |
| **f** | **10** | **b** |
| **g** | **5** | **b** |
| **h** | **6** | **i** |
| **i** | **5** | **a** |

**move to c**

**update d 6**

**update e 5**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **3** | **a** |
| **c** | **2** | **a** |
| **d** | **6** | **c** |
| **e** | **5** | **c** |
| **f** | **10** | **b** |
| **g** | **4** | **i** |
| **h** | **6** | **i** |
| **i** | **5** | **a** |

**we continue this process and continue updating as the following**

**move to e**

**no updates**

**move to h**

**no updates**

**move to d**

**no updates**

**move to f**

**no updates**

**move to g**

**no updates**

1. **bellman-ford algorithm**

**let's change the weight -1 of the edge(i) to (g) per requirements**

**(a) → (c), weight 2**

**(a) → (b), weight 3**

**(a) →(i), weight 5**

**(b) → (c), weight 2**

**(b) → (f), weight 10**

**(b) → (g), weight 2**

**(b) →(i), weight 10**

**(c) → (d), weight 4**

**(c) → (e), weight 3**

**(e) → (d), weight 1**

**(f) → (g), weight 7**

**(f) → (c), weight 1**

**(g) → (h), weight 11**

**(h) → (i), weight 2**

**(i) → (h), weight -1**

**step 1**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **∞** |  |
| **c** | **∞** |  |
| **d** | **∞** |  |
| **e** | **∞** |  |
| **f** | **∞** |  |
| **g** | **∞** |  |
| **h** | **∞** |  |
| **i** | **∞** |  |

**update from a**

**a to b 3**

**a to i 5**

**a to c 2**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **3** | **a** |
| **c** | **2** | **a** |
| **d** | **∞** |  |
| **e** | **∞** |  |
| **f** | **∞** |  |
| **g** | **∞** |  |
| **h** | **∞** |  |
| **i** | **5** | **a** |

**update from b**

**to c 5(no update)**

**to f 13**

**to g 5**

**to i 13(no update)**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **3** | **a** |
| **c** | **2** | **a** |
| **d** | **∞** |  |
| **e** | **∞** |  |
| **f** | **13** | **b** |
| **g** | **5** | **b** |
| **h** | **∞** |  |
| **i** | **5** | **a** |

**update from i**

**to h 6**

**to g 4**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **3** | **a** |
| **c** | **2** | **a** |
| **d** | **∞** |  |
| **e** | **∞** |  |
| **f** | **10** | **b** |
| **g** | **4** | **i** |
| **h** | **6** | **i** |
| **i** | **5** | **a** |

**update from c**

**to d 6**

**to e 5**

| **node** | **distance from a** | **last node** |
| --- | --- | --- |
| **a** | **0** | **-** |
| **b** | **3** | **a** |
| **c** | **2** | **b** |
| **d** | **6** | **c** |
| **e** | **5** | **c** |
| **f** | **10** | **b** |
| **g** | **4** | **i** |
| **h** | **6** | **i** |
| **i** | **5** | **a** |

**from d no changes**

**from e no changes**

**from f no changes**

**from g no changes**

**from h no changes**

**Update from a: No changes needed (b and i already have shortest distances).**

**Update from b: No changes needed (c, f, and g already have shortest distances).**

**Update from c: No changes needed (d and e already have shortest distances).**

**Update from d: No changes needed.**

**Update from e: No changes needed.**

**Update from f: No changes needed.**

**Update from g: No changes needed.**

**Update from h: No changes needed.**

**Update from i: No changes needed.**

**repeat the above steps but no updates are needed, all of them are in the shortest path.**

**3.3 critically evaluate both algorithms**

**after applying both algorithms and measuring the time in ms to see which is more efficient the results were as following:**

**Dijkstra's time: 10 ms**

**Bellman-Ford's time: 1 ms**

**With the run time calculated it indicates how much of a difference between the two algorithms and it's clear that bellman-ford’s algorithm is much better on a small datasets.**

1. **dijkstra’s algorithm**

**time complexity:**

**for the implementation that i did (a matrix) the time complexity is O(V^2) but there is a more efficient implementation using queue and the time complexity of that is O(V log V +E log V)**

**space complexity:**

**the space complexity for array is O(V) and for the shortest path tree is O(V) so ovrall O(V)**

1. **Bellman-Ford Algorithm:**

**The time complexity of my implementation is O(V.E) because it goes through all edges V-1 times.**

**space complexity**

**the space complexity is O(V)**

**critical evaluation:**

**Efficiency analysis:**

1. **small datasets**

**in my code bellman-ford outperformed dijkstra’s by 1:10 time this is the normal case since the time for standard implementation for dijkstra’s is better then bellman’s(if they didn't have a negative edge)**

1. **suitability**

**usually dijkstra’s algorithm is used for non-negative weights and bellman-ford’s is used when there is negative weights for edges**

**if implemented correctly(priority queue) with non-negative weights dijkstra’s is more efficient than bellman-ford’s but this isn't the case in my implementation because of the matrix not the queue implementation.**

1. **Reference list**
2. **1. i couldn't find an exact reference from java that says what algorithm is used in the sort.array but i found couple on unofficial sources: https://ioflood.com/blog/sort-array-java/**
3. **2. chapter 7 (“quicksort”) of the book Introduction to Algorithms, Third Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest**
4. **3.https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm**