Designing basic data structures

CAA1: First continuous assessment test

Statement

A Technology and Telecommunications Center (CTT) provides its facilities to accommodate workers employed by various companies in the sector, allowing them to perform their duties effectively. We have been approached by CTT to collaborate on the development of an application that not only manages essential data structures but also offers the functionality required for seamless operations.

CTT has a diverse range of facilities, which it extends to different companies in the sector to enable their recruited personnel to perform their tasks efficiently. These facilities include individual offices, rooms suitable for up to ten or twelve individuals, and rooms equipped with video conferencing capabilities, among others. To secure the use of a facility, a company must confirm the availability of a facility with the specific attributes required.

These attributes cover the type of facility, the specific number of occupants, and the duration spanning the start and end dates. If CTT possesses a facility that aligns with these attributes, it results in a favorable evaluation for the request. Subsequently, the company specifies the number of job positions to be filled within that facility. To allocate these job positions, the company issues an offer for different workers to apply. Job access is granted on a first-come, first-served basis, provided the worker meets the minimum qualifications. No additional evaluation criteria are considered. In the course of executing the projects associated with their job positions, workers encounter a series of interrelated activities that must be carried out in a specific sequence to achieve the desired objectives of each project. Upon completion of their respective tasks, workers can rate their performance with a numerical score and provide comments. The overall job assessment is determined by the average of all these evaluations. This assessment holds significant importance as it enables the management company to make necessary adjustments for future hiring decisions

In addition to all of the above, there is information regarding the elements that are part of the application to be developed:

- The number of workers (W) is known and relatively small, a few hundred.
- The number of companies (C) is known and small, a few dozen.
- The number of applications for jobs (A) is small and indeterminate.
- The number of job offers (JO) is known and relatively small, a few hundred.
- The number of job offers from a company (CJO) is small and indeterminate.
- The number of job offers for which a worker is hired (WJO) is small and indeterminate.
- The number of evaluations of a job offer (VJO) is small and indeterminate.
- The number of registrations for a job offer (RJO) is relatively small, in the hundreds.

NOTE: In general, throughout the exercise, the efficiency of the query operations should be optimized, although this may penalize write operations.

Exercise 1 (2.5 points)

Specify a **CTTCompaniesJobs ADT** that allows the following operations:

1. Add a new worker to the system. For each worker, we gather information on their unique identifier, first and last names, date of birth, and qualifications, which are categorized from the lowest to the highest levels as follows: compulsory, high school, vocational training, university, master's, and doctorate. In case an employee with the same identifier already exists, their information will be updated
2. Add a company to the system. We have information regarding each company, including its identifier, name, and a descriptive overview. If the company with the same identifier is already present, its data will be modified.
3. Create a job application. From the application we know its identifier, the identifier of the job offer, the identifier of the company offering it, the description, the minimum qualification required, the number of jobs, the start date and the end date. It is known in advance that the job offer does not exist. If the company does not exist, an error is indicated. For a job offer to be finalized, the CTT is required to issue a favorable response based on the availability of its facilities. It is at this time that the request will be stored in the system. To do this, the following steps are necessary:

   Within the application, we have access to the identifier of the application itself, the job offer identifier, the identifier of the offering company, a description of the offer, the minimum qualification level required, the number of job positions available, the start date, and the end date. It is important to note that, by design, the job offer does not currently exist. If the associated company is not registered, an error is indicated. To finalize a job offer, the CTT must issue a positive response, contingent on the availability of its facilities. At this time, the request will be stored within the system. To achieve this, the following steps must be followed:
   a. A request is created for a job offer in the CTT by a company.
   b. CTT processes incoming requests in the order they are received and responds with either an affirmative or negative decision depending on the availability of its facilities
   c. When CTT provides a positive response to the application, it is stored in the system, and the corresponding job offer will be executed at their facilities.
   d. In the event of an unfavorable response, the request is dismissed and documented as a rejected application
4. Provide an outcome for a request, which may be either favorable or unfavorable. In both instances, the date of the decision is documented. If the decision is unfavorable, the reason is indicated. If there is no request, an error is reported.
5. Enroll a worker for a job offer. In cases where either the worker or the offer is nonexistent, an error will be indicated. Registrations are processed on a first-come, first-served basis, as long as the worker meets the minimum qualification criteria for the respective offer. If the maximum

number of available job positions has been reached, an error is triggered, but these additional registrations are stored as substitute workers.
6. Check the percentage of rejected applications.
7. Consult the job offers of a company. It is known in advance that the company exists. If the company has no job openings, an error is indicated.
8. Consult all job offers in the system. If they do not exist, an error is indicated.
9. Consult the job offers in which a worker has worked. It is known in advance that the worker exists. If there are no job offers, an error is indicated.
10. Add a rating in numerical format (1-10) and a comment to a job by a worker. If the worker or position does not exist, an error is indicated. If the worker has not been part of the job offer, an error is indicated.
11. Consult the rating of a job offer. If the job offer does not exist, an error is indicated. If there are no ratings, an error is also indicated.
12. Consult the most active worker. The worker who has been working the longest is returned. That is, the sum of all the periods of all the job offers in which you have worked. If none exist, an error is indicated.
13. Get about the job offer rated most favorably by workers. In the absence of such an offer, an error is reported

Section a) [1 point]

Specify the signature of the **CTTCompaniesJobs ADT**. That is, indicate the name of the operations associated with each of the previously mentioned functionalities. Indicate also, what are the input parameters and the type of data returned in each case.

1. addWorker(id, name, surname, dateOfBirth, qualification)
2. addCompany(id, name, description)
3. addRequest(id, jobOfferId, companyId, description, minQualification, maxWorkers, startDate, endDate)
4. updateRequest(status, date, description): Request
5. signUpJobOffer(workerId, jobOfferId)
6. getPercentageRejectedRequests(): number
7. getJobOffersByCompany(companyId): Iterator
8. getAllJobOffers(): Iterator
9. getJobOffersByWorker(workerId): Iterator
10. addRating(workerId, jobOfferId, value, message)
11. getRatingsByJobOffer(jobOfferId): Iterator
12. getMostActiveWorker(): Worker
13. getBestJobOffer(): JobOffer

Section b) [1.5 points]

Make the **contractual specification** of the operations of the ADT **CTTCompaniesJobs**. Specifically, you must define the initial conditions that the operations must have (@pre) and the conditions on the state (@post) that the system remains (variables, data structures, return values) after its execution. Take as a reference the Abstract data types (ADT) specifications in the learning resources. It will be valued:

- Concision (absence of redundant or unnecessary elements).
- Precision (correct definition of the result of operations).
- Completetion (consideration of all possible cases in which each operation can be executed).
- Not contain ambiguities (exact knowledge of how each operation behaves in all possible cases).

NOTE: It is important to use a conditional description and not a procedural one, and although it is not always easy to distinguish between the two descriptions, that is why this aspect is mentioned.

For example, the following is the conditional (correct) description of starting a vehicle:

> @pre The vehicle's engine is off.
>
> @post The vehicle's engine is running.

However, a procedural description (incorrect for this section) could be:

> @pre The vehicle's engine should be off, if it is not.
>
> @post The engine cranks until it starts to run.

In describing the ADT, it should also be noted that a contract should have an invariant if necessary.

ADT **CTTCompaniesJobs** {

@pre True.
@post If the worker code is new, the workers will be the same plus a new one.
If not, the worker's data will have been updated.

1. **addWorker(id, name, surname, dateOfBirth, qualification)**

@pre True.
@post If the company code is new, the companies will be the same plus one new one.
If not, the company data will have been updated.

2. **addCompany(id, name, description)**

@pre The job application and offer do not exist.
@post The requests will be the same plus a new one.
If the company does not exist, the error will be reported

3. **addRequest(id, jobOfferId, companyId, description, minQualification, maxWorkers, startDate, endDate)**

@pre True.
@post The status of the first request is modified
Returns the result request.
The count of pending requests awaiting a response will decrease by one. If the application receives a favorable response, the count of job offers will increase by one. If the response is unfavorable, the count of rejected applications will increase by one.
If there are no pending requests, an error will be reported.

4. **updateRequest(status, date, description)**

@pre True.
@post The count of workers enrolled for a job offer will increase by one. If the worker lacks the minimum qualifications, they will be excluded. If the worker was already registered for that job offer, an error will be reported. If the maximum capacity has been reached, an error will be reported, and it will be added as a substitute. If either the job offer or the worker does not exist, an error will be reported.

5. **signUpJobOffer(workerId, jobOfferId)**

@pre True.
@post Provides a real number representing the percentage of denied requests

6. **getPercentageRejectedRequests(): number**

@pre The company exists.
@post Returns an iterator over a company's job offers.
If they do not exist, the error will be reported.

7. **getJobOffersByCompany(companyId): Iterator**

@pre True.
@post Returns an iterator over all job offers.
If they do not exist, the error will be reported.

8. **getAllJobOffers(): Iterator**

@pre The worker exists.
@post Returns an iterator over the job offers in which a worker has enrolled or participated.
If they do not exist, an error is returned.

9. **getJobOffersByWorker(workerId): Iterator**

@pre True.
@post The ratings will be the same plus one.
If the worker or the job offer does not exist, the error will be reported.
If the worker has not been enrolled in the job offer, an error will be reported

10. **addRating(workerId, jobOfferId, value, message)**

@pre True.
@post Returns an iterator over the ratings of a job offer.
If the job offer does not exist, the error will be reported.
If there are no ratings, the error will be reported.

11. **getRatingsByJobOffer(jobOfferId): Iterator**

@pre True.
@post Returns the most active worker, the one who has been working the longest.
If it does not exist, the error will be reported.

12. **getMostActiveWorker(): Worker**

@pre True.
@post Returns the highest rated job offer.
If it does not exist, the error will be reported.

13. **getBestJobOffer(): JobOffer**

}

Exercise 2 (3.5 points)

Carry out the **design of the data structures** of the **CTTCompaniesJobs ADT**. It must be made as efficient as possible, both temporally and spatially.

NOTE: Only sequential structures will be used; therefore, neither trees, nor scatter tables, nor graphs can be used.

Section a) [0.5 points]

To store **workers** :

[0.25 points] Provide a rationale for determining the optimal choice among a vector, a linked list, or an ordered linked list.

Since the number of workers is known and relatively small, a few hundred, the best option is a **static vector**, since we can assign it a fixed initial capacity that allows us to store all the elements.

[0.25 points] What changes, if necessary, would have to be made in this structure so that the temporal efficiency in the worker consultation operation is constant?

Each worker would have to be stored in the position of the vector that marks its identifier. Consequently, the size of the vector would increase in size. It would go from a size equal to the number of workers to a size equal to the maximum existing identifier among the collection of workers.

Section b) [0.5 points]

To store **a company's job offers** :

[0.25 points] Justify which is the best option between a vector, a linked list, or an ordered linked list.

Since the number is small and indeterminate, the best option is a **linked list**. The ordered linked list is discarded since the elements don't need to be ordered a priori. A static structure like a vector has a predefined space and would not allow elements to be added beyond its initial capacity.

[0.25 points] Justify if it is possible to perform binary searches in the previous data structure.

Binary or dichotomous searches are not possible on linked lists. **Not even if its elements were ordered** according to some criteria. This is because you cannot access a specific element directly, through an index, as you could in a static vector, but rather you need to traverse the linked list until you reach the specific element.

Section c) [1 point]

Justify **every one of the data structures to store the rest of the elements,** taking into account everything commented on in the statement and in the description of the operations. To do this, it is convenient to use the following format:

"To save XXX we choose an ordered linked list since the number of elements is not known in advance, it is not very large and we need ordering traversals".

For **companies** we choose a **vector**, since its number is known and small, a few dozen.

For **applications (requests)** we choose a **queue**, since they are taken in order of arrival.

For **job offers,** we will use a **vector**, since its number is known and relatively small, a few hundred.

For **job offers in which a worker** is hired, we select a **linked list** since it is a small and indeterminate number.

For the **evaluations** of a job offer, we use a **linked list**, since it is a small and indeterminate number.

For worker **registrations for a job offer, we use a queue**, since they are done in order of arrival.

For the **total number of requests,** we use an **integer**.

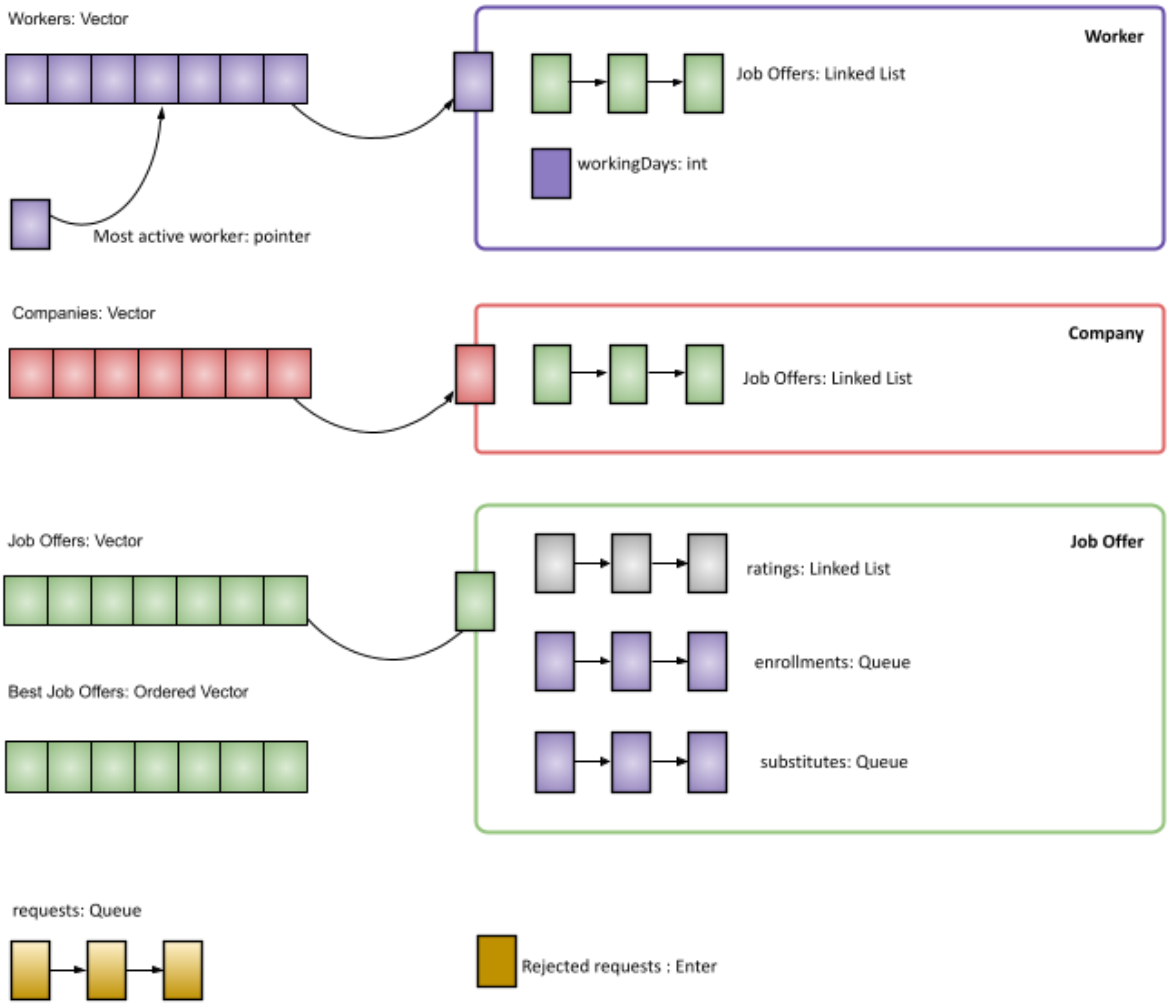
For the **total number of rejected requests,** we use an **integer**.

For the **most active worker,** we choose a **pointer**.

For the **highest rated job offer** an **ordered vector**.

Section d) [1 point]

Make a **graphical representation of the global data structure** of the **CTTCompaniesJobs ADT**. Every one of the data structures that have been chosen for each of the elements must be represented, as well as the existing relationships between them.



Section e) [0.5 points]

What differences might exist between the information presented in the statement and an actual scenario?

Some of the most relevant could be the following:

- In the statement all the information related to the sizes of the data structures is previously known: for example, known and relatively small, a few tens or hundreds, etc. This is usually complicated in a real case in which a priori there is no evidence of the necessary size for the structures, although there could be an estimate as to their magnitude.
- A real problem may require the design of data structures other than “traditional” ones (ad hoc structures) or involving a combination of them. Let's think, for example, of a group of people waiting at a ticket office to buy a ticket, but some of them have a preferential or VIP pass that allows them to make the purchase without having to wait for all the people in the group who arrived previously. Would its management be with a queue, a stack, a combination of both, or a new structure with that specific operation?
- The information related to the real problem is changing over time compared to a theoretical statement, which makes analysis and subsequent design difficult, often making it necessary to redo part of the work already prepared.
- Time estimation may not be appropriate in a real situation due to its changing nature and unforeseen difficulties, which could delay the release of the application. In a theoretical statement, the amount of work and its difficulty are adjusted to pre-established content and calendar.
- The need to work in a team when we are in a real environment, with the advantages and disadvantages that this entails.

With these points we wish to demonstrate the dimension involved in the development of an application in a real environment. Obviously we are in a training stage in which it is essential to acquire good skills and abilities. Among them, knowledge of data structures is crucial. And all this in order to build a good foundation that allows us to face the difficulties that we are undoubtedly going to encounter.

Exercise 3 (3 points)

This exercise involves specifying the **algorithm** to implement some of the operations that have been specified above and carrying out a **study of its efficiency**. It must be kept in mind that the implementation of each of the operations to be developed is closely related to the data structure selected in each case.

Section a) [2.5 points]

Specify the **algorithm** and study **the efficiency** of each of the following operations:

- Register a worker in a job offer.
- Add a new worker.
- Add a review by a worker to a job offer.

To specify each step of the **algorithm**, its behavior must be briefly described, for example: *"Insert into the vector, delete from the linked list, query the element of the ordered list, ..."* as well as the **asymptotic efficiency associated with such a step.** It is also necessary to specify the **global efficiency resulting** from each operation.

Register a worker in a job offer:
Find the worker in the vector of workers: O(W).
Search for the job offer in the vector of job offers: O(JO).
Check that the job offer is not in their list of job offers O(WJO)
Check that the maximum number of workers has not been exceeded: O(1).
Add an element to the worker queue of a job offer: O(1).
Add an element to the chained list of job offers for a worker: O(1).
If necessary, update the pointer of the most active worker: O(1)
Total: **O(W + JO + WJO)**.

Add a new worker:
Find the worker in the vector of workers: O(W).
If found, update the worker with the new data: O(1)
If not found,
    Create a new worker: O(1).
    Add the new worker to the vector of workers: O(1).
Total: **O(W)**.

Add a review by a worker to a job offer:
Find the worker in the vector of workers: O(W).
Search for the job offer in the vector of job offers: O(JO).
Add a review to the linked list of reviews: O(1).
Modify the ordered vector of top-rated job offers: O(JO).
Total: **O(W + JO + JO) = O(W + 2JO) = O(W + JO)** .

Section b) [0.5 points]

For various elements of the application, a linked list has been used to store them. This data structure, as we know it,

[0.25 points] is it as efficient as possible in terms of spatial efficiency?

It can be indicated that **its spatial efficiency is optimal** since it only maintains as many cells as the data you wish to store. However, **it is not maximum** , since it is necessary to have an extra pointer in each cell to be able to reference the next one. These pointers do not store data per se, but are used to make sense of the structure and be able to traverse all the data.

[0.25 points] And a double linked one?

If it were a doubly linked list, the number of extra pointers would be double, which would result in a worsening of spatial efficiency, although it would also mean the advantage of being able to perform routes in both directions.

Exercise 4 (1 point)

Indicate which **ADT from the library** https://eimtgit.uoc.edu/DS/DSLib would be used to implement each of the data structures defined in the **CTTCompaniesJobs ADT**. If there is no implementation already made in said library, indicate how it would be implemented.

Workers: Java Array: **Worker[]**.

Job offers from a company: Linked List: **LinkedList**.

Companies: Java Array: **Company[]**.

Requests : Queue: New ADT that implements an unbounded queue as an extension of a linked list and implements the Queue interface: **QueueLinkedList**.

Job Offers: Java Array: **JobOffer[].**

Job offers for a worker: Linked List: **LinkedList**.

Ratings: Linked List: **LinkedList**.

Enrollments : Queue: **QueueLinkedList**.

Substitutes: Queue: **QueueLinkedList**

Total requests : Integer: **Integer**.

Total rejected requests : Integer: **Integer**.

Most active worker : Pointer: **Worker**.

Top Rated Job Offer: New ADT OrderedVector that implements the *FiniteContainer* interface and keeps all elements of its internal container sorted according to a sorting criterion specified by a comparator. **OrderedVector.**